PART VIII

Advanced ColdFusion Development

41 More About SQL and Queries E3
42 Working with Stored Procedures E75
43 Using Regular Expressions E113
44 ColdFusion Scripting E161
45 Working with XML E181
46 Manipulating XML with XSLT and XPath E205
47 Using WDDX E231
48 Using JavaScript and ColdFusion Together E277
49 Using XForms E351
50 Internationalization and Localization E381
51 Error Handling E429
52 Using the Debugger E471
53 Managing Your Code E495
54 Development Methodologies E507

CHAPTER 41

More About SQL and Queries

IN THIS CHAPTER

Advanced SQL Topics E3
Additional <cfquery> Topics E40
Using Database Transactions E67

Advanced SQL Topics

You have already learned quite a bit about SQL in this book. In Chapter 6, "Introducing SQL," in *Adobe ColdFusion 8 Web Application Construction Kit*, *Volume 1: Getting Started*, you learned how to retrieve records using `SELECT`, `WHERE`, and `ORDER BY`. In Chapter 7, "SQL Data Manipulation," also in Vol. 1, *Getting Started*, you learned how to change the data in your database with `INSERT`, `UPDATE`, and `DELETE` and how to use the `AS` keyword to create aliases. You have also seen how the `<cfquery>` tag lets you create SQL statements dynamically, based on form parameters and other variables.

This chapter introduces you to some additional topics related to querying your databases with ColdFusion. This section explains certain aspects of SQL that haven't been discussed yet, and which will be particularly relevant when putting together report style pages.

Later in this chapter, the "Additional `<cfquery>` Topics" section explains how to use several advanced features of the `<cfquery>` tag, including ColdFusion's "query of queries" (also known as In Memory Queries) feature.

NOTE

The discussion in this section assumes that you have already read Chapters 6 and 7 or that you have used SQL on your own in the past. If not, it's recommended that you take a few minutes to look through those chapters (especially Chapter 6) before reading further.

Getting Unique Records with `DISTINCT`

In Chapter 6, you learned how to use the `SELECT` statement to retrieve records from your database tables, with or without `WHERE` or `ORDER BY` clauses. You can add `DISTINCT` to these `SELECT` statements in situations where you don't want to retrieve duplicate rows from your tables.

In general, all you have to do is add the `DISTINCT` keyword directly after the `SELECT` keyword. As an example, Listing 41.1 shows the results of two queries side by side. The first query retrieves the

descriptions for all expenses logged in the `Expenses` table. The second query is exactly the same as the first, except it uses the `DISTINCT` keyword to ensure that any repeats in the result set are eliminated before the query results are returned to ColdFusion. As you can see from Figure 41.1, the first query returns multiple entries for Costumes and Extras, whereas the second query does not.

Figure 41.1

The value Costumes, which appears several times in the `Expenses` table, appears only once when `DISTINCT` is used.

TIP

It's often appropriate to remove duplicated values when creating report-style pages. Even if duplicated entries don't yet exist in your database, keep `DISTINCT` in the back of your mind when putting together such pages.

Listing 41.1 `Distinct.cfm`—Using `DISTINCT` to Eliminate Duplicates

```
<!---
 Filename: Distinct.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates use of SQL DISTINCT keyword
--->

<!--- Retrieve Expense Descriptions from Expense table; --->
```

Listing 41.1 (continued)

```
<!--- This query may include any repeated descriptions --->
<cfquery name="getExp" datasource="#APPLICATION.DataSource#">
 SELECT Description
 FROM Expenses
 ORDER BY Description
</cfquery>

<!--- Retrieve Expense Descriptions from Expense table; --->
<!--- The DISTINCT means we won't get any repeated rows --->
<cfquery name="getUniqueExp" datasource="#APPLICATION.DataSource#">
 SELECT DISTINCT Description
 FROM Expenses
 ORDER BY Description
</cfquery>

<html>
<head><title>Film Expenses</title></head>
<body>
<h2>What Are We Spending Money On?</h2>

<table border="1">
  <tr valign="top">
    <!--- First, ordinary SELECT records --->
    <td>Ordinary SELECT statement:
    <ol>
    <cfoutput query="getExp">
    <li>#getExp.Description#</li>
    </cfoutput>
    </ol>
    </td>
    <!--- Now the SELECT DISTINCT records --->
    <td>SELECT DISTINCT statement:
    <ol>
    <cfoutput query="getUniqueExp">
    <li>#getUniqueExp.Description#</li>
    </cfoutput>
    </ol>
    </td>
  </tr>
</table>

</body>
</html>
```

NOTE

You could get the same unique effect in other ways. For instance, you could just run the first query and then add a `group="Description"` attribute to the `<cfoutput>` tag, causing ColdFusion to display a record only if its `Description` value were different from the previous row's (see Chapter 10, "Creating Data-Driven Pages," in Vol. 1, *Getting Started*, for more information about `group`). However, the `DISTINCT` approach would be more efficient because the duplicate records never leave the database in the first place.

NOTE

It's important to understand that the `DISTINCT` keyword applies to all columns in each row being returned by the query, not just the first column. So, if the `SELECT` statements in Listing 41.1 were selecting the `ExpenseAmount` column as well as the `Description` column, rows would be eliminated only in the unlikely event that the descriptions and expense amounts were exactly the same.

NOTE

This document uses the `Application.cfc` file found online, so be sure to copy it over. The `Application.cfc` file simply sets up the `APPLICATION.datasource` value used in this and later examples.

Summarizing Data with Aggregate Functions

SQL provides a number of aggregate functions for your use. Aggregate functions are used to crunch information from any number of rows into a summarized overview of your data. Again, you will find yourself using these functions most often when trying to present some type of big picture to your users, such as with report-style summaries or overviews in your ColdFusion applications.

NOTE

Aggregate functions are sometimes called set functions, or set-based functions.

Standard Aggregate Functions

There are five standard aggregate functions, which any SQL-compliant database system should support: `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`. Table 41.1 explains what each of these functions does and the types of columns they can generally be used with (although certain database systems might make exceptions).

Table 41.1 Standard SQL Aggregate Functions

FUNCTION	ON NUMERIC COLUMNS	ON DATE S COLUMN	ON CHARACTER COLUMN
`COUNT(*)`	Counts the number of rows in a table	(same)	(same)
`COUNT(column)`	Counts the number of values found in a column	(same)	(same)
`SUM(column)`	Adds the total of the values found in a column	(not allowed)	(not allowed)
`AVG(column)`	Computes the average of values found in a column	(not allowed)	(not allowed)
`MIN(column)`	Finds the smallest value in a column	Finds the earliest date	Finds the first value (alphabetically)
`MAX(column)`	Finds the largest value in a column	Finds the latest date	Finds the last value (alphabetically)

NOTE

Some database systems might support additional aggregate or aggregate-like functions, such as `FIRST` and `LAST`, and related keywords, such as `PIVOT`, `CUBE`, and `ROLLUP`. Consult your database documentation for details.

Listing 41.2 shows how aggregate functions can be used in a ColdFusion template. This example uses `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX` on numeric and date values in Orange Whip Studios' `Expenses` table. The query result set that is returned contains just one row of data (the computed, summarized information from the database) and seven columns (ExpenseCount, DateMin, DateMax, and so on). The results are shown in Figure 41.2.

Listing 41.2 `ExpenseReport1.cfm`—Using Aggregate Functions on All Records of a Table

```
<!---
 Filename: ExpenseReport1.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates use of SQL aggregate functions
--->

<!--- Retrieve summarized expense data from database --->
<cfquery name="getExp" datasource="#APPLICATION.dataSource#">
  SELECT
  COUNT(*) AS ExpenseCount,
  MIN(ExpenseDate) AS DateMin,
  MAX(ExpenseDate) AS DateMax,
  MIN(ExpenseAmount) AS ExpenseMin,
  MAX(ExpenseAmount) AS ExpenseMax,
  SUM(ExpenseAmount) AS ExpenseSum,
  AVG(ExpenseAmount) AS ExpenseAvg
  FROM Expenses
</cfquery>


<html>
<head><title>Film Expenses</title></head>
<body>
<h2>Expense Overview</h2>

<!--- Output high-level expense summary --->
<cfoutput>
<b>Number of expenses logged:</b>
#getExp.ExpenseCount#<br>

<b>Oldest expense on record:</b>
#lsDateFormat(getExp.DateMin, "mmmm d, yyyy")#
(#dateDiff("m", getExp.DateMin, now())# months ago)<br>

<b>Most recent expense on record:</b>
#lsDateFormat(getExp.DateMax, "mmmm d, yyyy (dddd)")#<br>

<b>Smallest expense made:</b>
#lsCurrencyFormat(getExp.ExpenseMin)#<br>

<b>Largest expense made:</b>
#lsCurrencyFormat(getExp.ExpenseMax)#<br>

<b>Average expense amount:</b>
#lsCurrencyFormat(getExp.ExpenseAvg)#<br>
```

Listing 41.2 (continued)

```
<b>Total of all expenses to date:</b>
#lsCurrencyFormat(getExp.ExpenseSum)#<br>
</cfoutput>

</body>
</html>
```

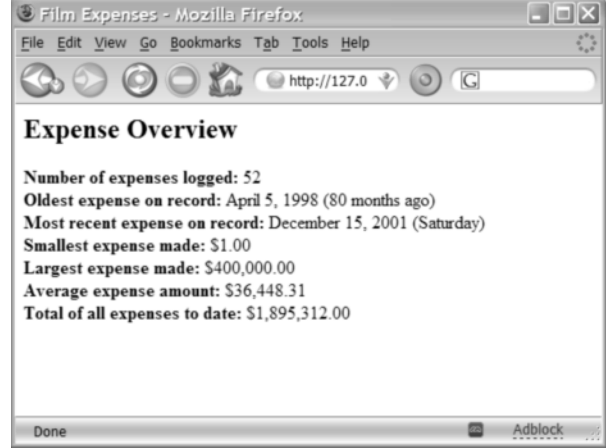


Figure 41.2

Aggregate functions can be used to report on trends in your data.

NOTE

Queries that use only aggregate functions in their `SELECT` lists, such as the query in Listing 41.2, will never return more than one row.

NOTE

It's important to note that no "source" columns are being selected on their own by the query in this listing. (The only columns the query returns are the temporary ones named with the AS keyword.) To mix ordinary column references and aggregate columns in the same `SELECT` list, you also must add a `GROUP BY` clause (discussed in the next section). Otherwise, you will get an error message from the database driver.

Getting More Selective with `WHERE`

You can use `WHERE` and aggregates in the same query to get summarized records about only certain rows in your database tables. For instance, if a parameter named `FilmID` might be passed to the page in the URL, you could show the summarized expense information for just that film by adding the following to the `<cfquery>` tag in Listing 41.2 (right after the `FROM` line of the query):

```
<!--- Limit expense summary to single film, if provided --->
<cfif isDefined("URL.filmID")>
 WHERE FilmID = #URL.filmID#
</cfif>
```

The results would look the same when viewed in a browser, except that the various totals and dates would apply only to the `filmID` (if any) specified in the URL. When no `filmID` parameter is provided, the data for all expenses would continue to be shown (as shown previously in Figure 41.2).

NOTE

Be careful not to confuse `WHERE` with `HAVING` (discussed later in this section). As a rule, you almost always should use `WHERE` when specifying criteria. `HAVING` is generally used for special cases only.

Breaking It Down with `GROUP BY`

So far, you have seen how aggregate functions can be used to run simple computations that return only one row of data, which reports the computed sums (or averages, or counts) of all matching records at once. You'll often want to break down the computations further, returning a separate set of summary data results for each value in a particular column. You can get this effect by adding a `GROUP BY` clause to your query.

To use `GROUP BY`, the columns for which you want summarized data are added in two places: the query's `SELECT` list and the new `GROUP BY` clause, which should come right after the `FROM` part of your query (but before any `ORDER BY` clause).

Behind the scenes, your database breaks down your data into groups, one for each value found in your `GROUP BY` columns. Each of your aggregate functions is computed separately for each group. When the results are returned to ColdFusion, one row will exist for each unique value in your `GROUP BY` column(s). That is, there will be one row for each group.

Listing 41.3 shows how `GROUP BY` can be used to apply aggregate functions, such as `SUM`, to each group in the `Expenses` table. Here, the results are broken down (or grouped) by `FilmID`. The results are shown in Figure 41.3.

Listing 41.3 `ExpenseReport2.cfm`—Using `GROUP BY` with Aggregate Functions

```
<!---
 Filename: ExpenseReport2.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates use of SQL aggregate functions
--->

<!--- Retrieve summarized expense data from database --->
<cfquery name="getExp" datasource="#APPLICATION.dataSource#">
  SELECT
  f.FilmID, f.MovieTitle,
  SUM(ExpenseAmount) AS ExpenseSum
  FROM Expenses e, Films f
  WHERE e.FilmID = f.FilmID
  GROUP BY f.FilmID, f.MovieTitle
  ORDER BY f.MovieTitle
</cfquery>

<html>
<head><title>Film Expenses</title></head>
<body>
<h2>Expense Overview</h2>

<!--- Output film-level expense summary --->
<cftable query="getExp" htmltable="Yes" border="Yes" colheaders="Yes">
```

Listing 41.3 (continued)

```
 <cfcol header="Film Title" text="#MovieTitle#">
 <cfcol header="Amount" text="#lsCurrencyFormat(ExpenseSum)#" align="right">

</cftable>

</body>
</html>
```

Film Title	Amount
Being Unbearably Light	$25,800.00
Charlie's Devils	$221,000.00
Closet Encounters of the Odd Kind	$80.00
Folded Laundry, Concealed Ticket	$28,500.00
Forrest Trump	$101,245.50
Four Bar-Mitzvah's and a Circumcision	$10,250.00
Geriatric Park	$10,451.00
Gladly Ate Her	$600,000.00
Ground Hog Day	$2,000.00
Hannah and Her Blisters	$39,900.00
Harry's Pottery	$20,500.00
It's a Wonderful Wife	$55,500.00
Kramer vs. George	$75,000.00
Mission Improbable	$609.00
Nightmare on Overwhelmed Street	$255,000.00
Raiders of the Lost Aardvark	$45,008.50
Silence of the Clams	$30,000.00
Starlet Wars	$230,000.00
Strangers on a Stain	$143,000.00
The Funeral Planner	$450.00
The Sixth Nonsense	$1.00
Use Your ColdFusion II	$17.00
West End Story	$1,000.00

Figure 41.3

Aggregate functions become more powerful when combined with `GROUP BY` to break down the results.

NOTE

If you specify more than one column in the `GROUP BY` part of your query, the results will include one record for each unique permutation of values among the specified columns. That is, each unique permutation of values in the `GROUP BY` columns is what defines a new group.

Filtering Summarized Records with HAVING

You've seen how you can use WHERE to cause only certain rows to be considered by aggregate functions such as SUM and MAX. That's fine for filtering totals based on something fixed, such as an ID number. But what if you wanted to apply a different type of filtering criteria, based on what the values are after the aggregate functions do their work, not before?

The HAVING keyword lets you do just that. You get to provide criteria using the same syntax you use in a WHERE clause, using the =, <, and > operators. The difference is that the HAVING criteria is applied to the final grouped rows just before they are returned to ColdFusion, whereas the WHERE criteria is applied before any of the values are considered for grouping in the first place.

To use HAVING, place it directly after the GROUP BY part of your query (but before the ORDER BY part, if any). For instance, if you wanted to keep films with a small amount of expenses—less than $1,000, say—from being shown in the report from Listing 41.3, you could change the query code from this

```
WHERE e.FilmID = f.FilmID
GROUP BY f.FilmID, f.MovieTitle
```

to this

```
WHERE e.FilmID = f.FilmID
GROUP BY f.FilmID, f.MovieTitle
HAVING SUM(ExpenseAmount) > 1000
```

Now only those groups that have a total expense of more than 1000 will be returned to ColdFusion. Note that this is very different from the following:

```
WHERE e.FilmID = f.FilmID
AND ExpenseAmount > 1000
GROUP BY f.FilmID, f.MovieTitle
```

This snippet simply means that no individual expenses that are less than 1,000 will be considered; the total amount for any group (film) could exceed 1,000. The HAVING snippet, by contrast, considers only the total for each film, with no regard to how big the individual payments are.

Of course, you can combine both concepts, like so:

```
WHERE e.FilmID = f.FilmID
AND ExpenseAmount < 50000
GROUP BY f.FilmID, f.MovieTitle
HAVING SUM(ExpenseAmount) > 1000
```

The previous snippet shows total spending only for films that have 1,000 or more in expenses. However, in coming up with these totals, it ignores any individual expenses that are very large (more than 50,000). This might be useful to an executive at Orange Whip Studios who probably doesn't care about movies that are spending very small amounts of money (less than 1,000 total) and who also probably already knows about the really big expenditures (more than 50,000 each). This report lets the executive focus on the middle ground, where movie executives seem to feel most at home.

Listing 41.4 is a simple revision of the template from Listing 41.3, which presents the data returned from this last version of the query. Figure 41.4 shows the results.

NOTE

Compared with everything else you're learning in this chapter, `HAVING` is relatively obscure, and you may not find yourself using it much. That said, it's still good to know everything that's available to you.

Listing 41.4 `ExpenseReport3.cfm`—Using `GROUP BY`, `HAVING`, `WHERE`, and Aggregate Functions

```
<!---
 Filename: ExpenseReport3.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates use of SQL aggregate functions
--->

<!--- These values will control the filtering --->
<cfset filterMinTotal = 1000>
<cfset filterMaxIndiv = 50000>

<!--- Retrieve summarized expense data from database --->
<cfquery name="getExp" datasource="#APPLICATION.dataSource#">
  SELECT
  f.FilmID, f.MovieTitle,
  SUM(ExpenseAmount) AS ExpenseSum
  FROM Expenses e, Films f
  WHERE e.FilmID = f.FilmID
  AND ExpenseAmount < #filterMaxIndiv#
  GROUP BY f.FilmID, f.MovieTitle
  HAVING SUM(ExpenseAmount) > #filterMinTotal#
  ORDER BY f.MovieTitle
</cfquery>


<html>
<head><title>Film Expenses</title></head>
<body>
<h2>Expense Overview</h2>

<cfoutput>
<p>
<!--- Note that we are filtering out small numbers --->
For clarity, films that haven't reported more than
#lsCurrencyFormat(filterMinTotal, "international")#
have been ommitted from this list.<br>
In addition, large payments
(over #lsCurrencyFormat(filterMaxIndiv, "international")#)
have not been considered in the totals.
</p>
</cfoutput>

<!--- Output film-level expense summary --->
<cftable query="getExp" htmltable="Yes" border="Yes" colheaders="Yes">

 <cfcol header="Film Title" text="#MovieTitle#">
 <cfcol header="Amount" text="#lsCurrencyFormat(ExpenseSum)#" align="right">

</cftable>

</body>
</html>
```

Figure 41.4

The `HAVING` keyword lets you filter records based on the results of your aggregate functions.

Selecting Related Data with Joins

The design of the tables in Orange Whip Studios' database calls for a number of relationships between the various tables. This section focuses on the relationships between the `Films`, `Actors`, and `FilmsActors` tables.

You will recall that the `Films` and `Actors` tables contain relatively straightforward information about the studio's films and actors (the title of each movie, the name of each actor, and so on). That is, the principal purpose of these two tables is to remember the attributes of each film and actor. In contrast, the principal purpose of the `FilmsActors` table is to connect each film with its actors (and vice versa). There are a couple of extra columns in `FilmsActors` (the `Salary` and `IsStarringRole` columns), but the really important columns are the `FilmID` and `ActorID` columns. Because of these two columns, the database is able to track the relationship between the other two tables (`Films` and `Actors`).

Because tables such as `FilmsActors` define the relationship, or connection, between other tables, they can be thought of as connector tables. Much of the SQL language's power comes from its capability to return records from several tables at once, using something called a *join*. A join is sim-

ply a way to describe the relationship between tables, right in a SQL query statement. This means you can use join syntax in a `SELECT` query, for example, to easily retrieve information about the actors who have appeared in each film.

NOTE

For clarity, this section refers to tables such as `Films` and `Actors` as data tables and tables such as `FilmsActors` as connector tables. These aren't official SQL terms; they are used here just to keep the discussion as clear as possible.

Joining Two Tables

When you need to join two tables in a query, you actually have your choice of several syntax forms. The simplest and most common method is to add a piece of criteria in the query's `WHERE` clause, using this basic form:

```
SELECT Columns
FROM TableA, TableB
WHERE TableA.SomeID = TableB.SomeID
```

Applying this to the `Films` and `FilmsActors` tables, say, might translate to

```
SELECT Films.FilmID, Films.MovieTitle, FilmsActors.ActorID
FROM Films, FilmsActors
WHERE Films.FilmID = FilmsActors.FilmID
```

The `WHERE` keyword is clearly being used in a new way here. Rather than merely specifying selection criteria, it is used here to bind the `Films` and `FilmsActors` tables together. Translated into plain English, this statement might read, "Using the `FilmID` as a guide, show me the `FilmID` and `MovieTitle` for each row in the `Films` table, along with the corresponding `ActorID` numbers from the `FilmsActors` table."

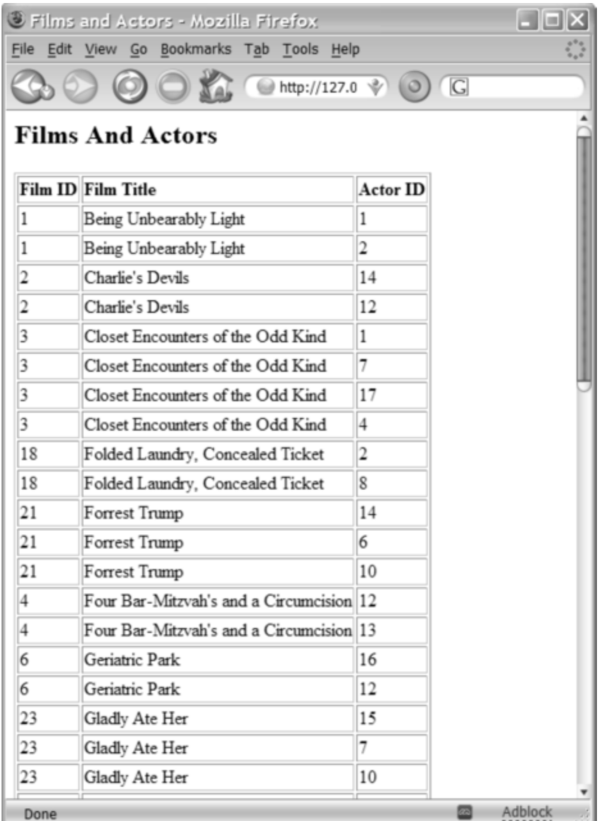
Listing 41.5 shows how the previous join query can be used in a ColdFusion template. For each film, the query results include one record for each actor in that film, as shown in Figure 41.5.

Listing 41.5 `FilmCast1.cfm`—Retrieving Data from Two Tables at Once, Using a Join

```
<!---
 Filename: FilmCast1.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates the use of SQL joins
--->

<!--- Retrieve Films and Related Actors from database --->
<cfquery name="getFilmInfo" datasource="#APPLICATION.DataSource#">
 SELECT
 Films.FilmID, Films.MovieTitle,
 FilmsActors.ActorID
 FROM Films, FilmsActors
 WHERE Films.FilmID = FilmsActors.FilmID
 ORDER BY Films.MovieTitle
</cfquery>

<html>
<head><title>Films and Actors</title></head>
```

Listing 41.5 (continued)

```
<body>
<h2>Films And Actors</h2>

<!--- Display retrieved information in HTML table --->
<cftable query="getFilmInfo" htmltable border colheaders>
 <cfcol header="Film ID" text="#FilmID#">
 <cfcol header="Film Title" text="#MovieTitle#">
 <cfcol header="Actor ID" text="#ActorID#">
</cftable>

</body>
</html>
```



Film ID	Film Title	Actor ID
1	Being Unbearably Light	1
1	Being Unbearably Light	2
2	Charlie's Devils	14
2	Charlie's Devils	12
3	Closet Encounters of the Odd Kind	1
3	Closet Encounters of the Odd Kind	7
3	Closet Encounters of the Odd Kind	17
3	Closet Encounters of the Odd Kind	4
18	Folded Laundry, Concealed Ticket	2
18	Folded Laundry, Concealed Ticket	8
21	Forrest Trump	14
21	Forrest Trump	6
21	Forrest Trump	10
4	Four Bar-Mitzvah's and a Circumcision	12
4	Four Bar-Mitzvah's and a Circumcision	13
6	Geriatric Park	16
6	Geriatric Park	12
23	Gladly Ate Her	15
23	Gladly Ate Her	7
23	Gladly Ate Her	10

Figure 41.5

Two tables can be queried together using simple join syntax.

NOTE

If a film has no associated actors (that is, if there are no rows in the `FilmsActors` table with the same `FilmID`), no records are returned for that film. Therefore, the query results return records only for those combinations of film and actor that can actually be found in both tables. If you wanted to get information about all films, including those with no corresponding actors, you would need to use an outer join (see the section "Using Outer Joins," later in this chapter).

Understanding the WHERE clause in this listing is absolutely critical. The conceptual relationship between the two tables hinges on the fact that the rows of the Films table in which the FilmID is 1 correspond to the rows in the FilmsActors table in which the FilmID is 1, and so on. The WHERE clause explains this fact to your database system.

NOTE

Notice the simple dot notation used to specify which columns are in which tables. Now that two tables are participating in the SELECT, this dot notation is necessary. Otherwise, your database system wouldn't know which table to retrieve the FilmID from (because a column called FilmID exists in each one).

TIP

You actually have to use the dot notation only for columns that exist in both tables (that is, where the column name alone would be ambiguous). However, it is recommended that you use the dot notation for all columns in any query that involves more than one table, simply to make your code clearer and remove any possibility of (gasp!) human error.

Joining Three Tables

Joining three tables isn't much different from joining two tables. You simply specify two join conditions in your WHERE clause, using AND between them. When querying against a relationship established by a connector table, you often need to join three tables together in order to provide user-friendly pages.

To retrieve each actor's last name instead of just the actor's ID number, you would join the Films table to the FilmsActors table (as you have already seen) and then join the FilmsActors table to the Actors table (to pick up the last name associated with each actor's ID number). The resulting query would look like this:

```
SELECT Films.FilmID, Films.MovieTitle, Actors.NameLast
FROM Films, FilmsActors, Actors
WHERE Films.FilmID = FilmsActors.FilmID
 AND FilmsActors.ActorID = Actors.ActorID
```

Listing 41.6 shows how this three-table join can be used to revise the previous example (refer to Listing 41.5) to create a much friendlier display for your users. Now, instead of just seeing each actor's ID number, your users can see each actor's first and last names, as shown in Figure 41.6. If you wanted to display other information from the Actors table, you could just add them to the query's SELECT list.

Listing 41.6 FilmCast2.cfm—Joining Three Tables at Once

```
<!---
 Filename: FilmCast2.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates the use of SQL joins
--->


<!--- Retrieve Films and Related Actors from database --->
<cfquery name="getFilmInfo" datasource="#APPLICATION.dataSource#">
 SELECT
 Films.FilmID, Films.MovieTitle,
```

Listing 41.6 (CONTINUED)

```
     Actors.NameFirst, Actors.NameLast
     FROM Films, FilmsActors, Actors
     WHERE Films.FilmID = FilmsActors.FilmID
     AND FilmsActors.ActorID = Actors.ActorID
     ORDER BY Films.MovieTitle, Actors.NameLast
    </cfquery>

    <html>
    <head><title>Films and Actors</title></head>
    <body>
    <h2>Films And Actors</h2>

    <!--- Display retrieved information in HTML table --->
    <cftable query="getFilmInfo" htmltable border colheaders>
     <cfcol header="Film ID" text="#FilmID#">
     <cfcol header="Film Title" text="#MovieTitle#">
     <cfcol header="Actor (First Name)" text="#NameFirst#">
     <cfcol header="Actor (Last Name)" text="#NameLast#">
    </cftable>

    </body>
    </html>
```

Films And Actors

Film ID	Film Title	Actor (First Name)	Actor (Last Name)
1	Being Unbearably Light	Sean	Conway
1	Being Unbearably Light	Harry	Fjord
2	Charlie's Devils	Nick	Nutty
2	Charlie's Devils	L.L.	P.J.
3	Closet Encounters of the Odd Kind	Sean	Conway
3	Closet Encounters of the Odd Kind	Nicole	Kidmen
3	Closet Encounters of the Odd Kind	Kevin	Space
3	Closet Encounters of the Odd Kind	Patrik	Sway
18	Folded Laundry, Concealed Ticket	Fabio	
18	Folded Laundry, Concealed Ticket	Harry	Fjord
21	Forrest Trump	Sam	Gold
21	Forrest Trump	Nick	Nutty
21	Forrest Trump	Sharon	Stoned
4	Four Bar-Mitzvah's and a Circumcision	Ferris	Beuller
4	Four Bar-Mitzvah's and a Circumcision	L.L.	P.J.
6	Geriatric Park	Halle	Beary
6	Geriatric Park	L.L.	P.J.
23	Gladly Ate Her	Dolly	Pardon
23	Gladly Ate Her	Sharon	Stoned
23	Gladly Ate Her	Patrik	Sway
7	Ground Hog Day	Sean	Conway
7	Ground Hog Day	Harry	Fjord
7	Ground Hog Day	Patrik	Sway
20	Hannah and Her Blisters	Fabio	

Figure 41.6

Three-table joins are common when you need to display names, labels, or titles from related data tables.

TIP

To create queries that join four or more tables together, continue to add additional lines of criteria in your WHERE clause, using the AND keyword between each one.

Improving Readability with Table Aliases

When you start using table joins more frequently, you will probably find it helpful to use table aliases to make the queries easier to read (and easier to type!). A table alias is a temporary nickname for a table, good only within the context of the current query. You can define a very short table alias that is only one or two letters long, allowing you to remove clutter from the remainder of the query statement.

To define a table alias, provide a short nickname right after the table's real name in the FROM part of your query. Now you can use the short alias in the other parts of the query, in place of the actual table name. Many developers use the first letter or two of the table name (in lowercase) for an alias. For instance, this query

```
SELECT Films.FilmID, Films.MovieTitle, Actors.NameLast
FROM Films, FilmsActors, Actors
WHERE Films.FilmID = FilmsActors.FilmID
 AND FilmsActors.ActorID = Actors.ActorID
```

could become this, which is relatively easy on the eyes:

```
SELECT f.FilmID, f.MovieTitle, a.NameLast
FROM Films f, FilmsActors fa, Actors a
WHERE f.FilmID = fa.FilmID
 AND fa.ActorID = a.ActorID
```

If you prefer, you can use the AS keyword between the table's real name and the alias, like so:

```
SELECT f.FilmID, f.MovieTitle, a.NameLast
FROM Films AS f, FilmsActors AS fa, Actors AS a
WHERE f.FilmID = fa.FilmID
 AND fa.ActorID = a.ActorID
```

The Two Types of Join Syntax

So far, you have seen how tables can be joined using the WHERE part of a query. This is the simplest and most common method, but there is another. The two types of join syntax generate the same results (except in very rare circumstances that are well beyond the scope of this book, and which you are unlikely to encounter). You can use whichever syntax you prefer.

The two forms of join syntax are

- WHERE JOIN syntax—You have already learned about this simple syntax.
- INNER JOIN syntax—Some SQL experts prefer this somewhat more complex-looking method mainly because it keeps your table relationships separate from any filter criteria; that is, it gets the join description out of the WHERE clause.

The first join syntax describes the join as part of the query's `WHERE` criteria, as you have already seen:

```
SELECT f.FilmID, f.MovieTitle, fa.ActorID
FROM Films f,
 FilmsActors fa
WHERE f.FilmID = fa.FilmID
```

The `INNER JOIN` syntax describes the join in the `FROM` part of the query, instead of the `WHERE` part. Instead of separating the table names with commas, you separate them with the words `INNER JOIN`. Each pair of table names is followed by the word `ON`, followed by the equality condition that describes the join. For instance, the `WHERE`-style join shown above would translate into the following:

```
SELECT Films.FilmID, Films.MovieTitle, FilmsActors.ActorID
FROM Films INNER JOIN
 FilmsActors ON Films.FilmID = FilmsActors.FilmID
```

Adding table aliases to improve readability, it looks like this:

```
SELECT f.FilmID, f.MovieTitle, fa.ActorID
FROM Films f INNER JOIN
 FilmsActors fa ON f.FilmID = fa.FilmID
```

To join three or more tables using `INNER JOIN` syntax, use parentheses to isolate the individual `INNER JOIN` pieces from one another, like so:

```
SELECT f.FilmID, f.MovieTitle, a.NameFirst, a.NameLast
FROM (Films f INNER JOIN
 FilmsActors fa ON f.FilmID = fa.FilmID) INNER JOIN
 Actors a ON fa.ActorID = a.ActorID
```

Listings 41.7 and 41.8 are revisions of Listings 41.5 and 41.6, respectively, using `INNER JOIN` syntax instead of the `WHERE`-style of join syntax. Use whichever syntax you find more straightforward.

Listing 41.7 `FilmCast1a.cfm`—Using `INNER JOIN` Syntax to Join Two Tables

```
<!---
 Filename: FilmCast1a.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates the use of SQL joins
--->

<!--- Retrieve Films and Related Actors from database --->
<cfquery name="getFilmInfo" datasource="#APPLICATION.dataSource#">
 SELECT f.FilmID, f.MovieTitle, fa.ActorID
 FROM Films f INNER JOIN
 FilmsActors fa ON f.FilmID = fa.FilmID
 ORDER BY f.MovieTitle
</cfquery>

<html>
<head><title>Films and Actors</title></head>
<body>
<h2>Films And Actors</h2>

<!--- Display retrieved information in HTML table --->
<cftable query="getFilmInfo" htmltable border colheaders>
```

Listing 41.7 (continued)

```
 <cfcol header="Film ID" text="#FilmID#">
 <cfcol header="Film Title" text="#MovieTitle#">
 <cfcol header="Actor ID" text="#ActorID#">
</cftable>

</body>
</html>
```

Listing 41.8 `FilmCast2a.cfm`—Using `INNER JOIN` Syntax to Join Three Tables

```
<!---
 Filename: FilmCast2a.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates the use of SQL joins
--->

<!--- Retrieve Films and Related Actors from database --->
<cfquery name="getFilmInfo" datasource="#APPLICATION.dataSource#">
 SELECT
 f.FilmID, f.MovieTitle,
 a.NameFirst, a.NameLast
 FROM (Films f INNER JOIN
 FilmsActors fa ON f.FilmID = fa.FilmID) INNER JOIN
 Actors a ON fa.ActorID = a.ActorID
 ORDER BY f.MovieTitle, a.NameLast
</cfquery>

<html>
<head><title>Films and Actors</title></head>
<body>
<h2>Films And Actors</h2>

<!--- Display retrieved information in HTML table --->
<cftable query="getFilmInfo" htmltable border colheaders>
 <cfcol header="Film ID" text="#FilmID#">
 <cfcol header="Film Title" text="#MovieTitle#">
 <cfcol header="Actor (First Name)" text="#NameFirst#">
 <cfcol header="Actor (Last Name)" text="#NameLast#">
</cftable>

</body>
</html>
```

Using Outer Joins

So far, all of the joins in this chapter have been inner joins, which means that only those records that match up in both tables are included in the results. From time to time, you will run into situations in which you need to use an *outer* join, which means that all records in one of the tables are returned, regardless of whether any matching records exist in the other.

For instance, say you need to create a simple report page—similar to Listing 41.5 or 41.6—but which shows each film and the rating it has been given. You decide to join the `Films` table to the

Ratings table, using the RatingID column as the join criteria. Putting your new join skills to work, you come up with something like this:

```
SELECT f.FilmID, f.MovieTitle, r.Rating
FROM Films f INNER JOIN
 FilmsRatings r ON f.RatingID = r.RatingID
ORDER BY f.MovieTitle
```

The query seems to work fine, and no one notices any problems for the first few weeks your application is up and running. But after a while, people start complaining that some movies never show up on the list. After a bit of trial and error, you realize that whenever a movie doesn't have a rating (when the RatingID column is null, or blank), it gets excluded from the query results. This is because inner joins return records only when matching rows exist in both tables.

To solve the problem, you need to change the query to use an outer join instead of an inner join. To do so, you simply change the words INNER JOIN to OUTER JOIN, preceded by either LEFT or RIGHT, depending on whether the outer table (the table that should always get included in the query results) is on the left or the right side of the OUTER JOIN statement.

So, the query statement shown previously could become this:

```
SELECT f.FilmID, f.MovieTitle, r.Rating
FROM Films f LEFT OUTER JOIN
 FilmsRatings r ON f.RatingID = r.RatingID
```

or this:

```
SELECT f.FilmID, f.MovieTitle, r.Rating
FROM FilmsRatings r RIGHT OUTER JOIN
 Films f ON f.RatingID = r.RatingID
```

These two queries behave in exactly the same way. They will always return all records from the Films table, regardless of whether any corresponding rows exist in the Ratings table. When matching records exist in Ratings, a row gets added to the query results for each combination of film and rating, just like with a normal inner join.

But because these are outer joins, the film is included in the results even when no matching rating exists for it (when the film's RatingID is null or is set to some number that doesn't exist in the Ratings table). Any columns from the Ratings table—in this case, just the Rating column—are simply returned as blank columns. in the above examples, the Rating column will be set to an empty string for any films that don't have ratings.

NOTE

Technically, the Rating column isn't returned to ColdFusion as an empty string; it's returned as a null value. However, ColdFusion converts null values to empty strings as it receives them, so as far as your CFML code is concerned, the column is an empty string. You would need to test for it as such in any <cfif> statements. See "Working with NULL Values," later in this chapter, for details.

Listing 41.9 shows how this outer join query can be used in an actual template. To test this template, add a few new films to the Films table, being sure to leave the RatingID as null, 0, or some other unknown rating ID number. Then, visit Listing 41.9 with your browser. The new films will appear, but their ratings will be blank, as shown in Figure 41.7. If you then change Listing 41.9 so

that it uses ordinary inner join syntax, you will see that the new films disappear altogether when you revisit the page.

Film Ratings

Film ID	Film Title	Rating
1	Being Unbearably Light	Adults
2	Charlie's Devils	General
3	Closet Encounters of the Odd Kind	Adults
18	Folded Laundry, Concealed Ticket	Kids
21	Forrest Trump	Accompanied Minors
4	Four Bar-Mitzvah's and a Circumcision	General
6	Geriatric Park	Mature Audiences
23	Gladly Ate Her	Accompanied Minors
7	Ground Hog Day	Teens
20	Hannah and Her Blisters	General
5	Harry's Pottery	General
8	It's a Wonderful Wife	General
9	Kramer vs. George	Mature Audiences
10	Mission Improbable	Kids
11	Nightmare on Overwhelmed Street	Mature Audiences
17	Raiders of the Lost Aardvark	Teens
27	Ray's Movie	General
26	Ray's Movie	General
12	Silence of the Clams	Accompanied Minors
13	Starlet Wars	Accompanied Minors
22	Strangers on a Stain	Kids
14	The Funeral Planner	Adults
15	The Sixth Nonsense	Teens
19	Use Your ColdFusion II	General

Figure 41.7

Because an outer join is used, unrated movies show up with a blank rating, instead of being eliminated from the list altogether.

Listing 41.9 `FilmRatings.cfm`—Using an Outer Join to Show All Films

```
<!---
 Filename: FilmRatings.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates use of SQL outer joins
--->

<!--- Retrieve Films and Related Actors from database --->
<cfquery name="getFilmInfo" datasource="#APPLICATION.dataSource#">
 SELECT
 f.FilmID, f.MovieTitle,
 r.Rating
```

Listing 41.9 (continued)

```
 FROM Films f LEFT OUTER JOIN
 FilmsRatings r ON f.RatingID = r.RatingID
 ORDER BY f.MovieTitle
</cfquery>

<html>
<head><title>Film Ratings</title></head>
<body>
<h2>Film Ratings</h2>

<!--- Display retrieved information in HTML table --->
<cftable query="getFilmInfo" htmltable border colheaders>
 <cfcol header="Film ID" text="#FilmID#">
 <cfcol header="Film Title" text="#MovieTitle#">
 <cfcol header="Rating" text="#Rating#">
</cftable>

</body>
</html>
```

NOTE

If you want, you can leave the word `OUTER` out of your outer join queries. That is, `LEFT JOIN` and `LEFT OUTER JOIN` are synonyms; so are `RIGHT JOIN` and `RIGHT OUTER JOIN`. However, we recommend that you leave the word `OUTER` in there to emphasize what is going on.

NOTE

Some database systems place restrictions on whether you can use inner and outer joins together in the same `SELECT` statement. See your database documentation for details.

Subqueries

SQL allows you to nest complete `SELECT` statements within other statements for various purposes. These nested queries are known as *subqueries*. Although subqueries cover similar conceptual ground as joins, you need to be aware of both, because subqueries provide great flexibility and allow your queries to get into hard-to-reach places.

Subqueries can be included in your queries in a number of ways. The most common are

- In the `WHERE` part of a SQL statement to correlate data in various tables.
- In the `SELECT` part of a SQL statement to return an additional column.

Either way, the subquery itself is put inside parentheses and can contain just about any valid `SELECT` statement. The subquery can use dot notation to refer to tables outside the parentheses, but not vice versa. (The main statement can't reach in and refer to tables inside the parentheses, but the subquery can reach out and refer to tables in the main statement.)

NOTE

Most database systems support subqueries, but some place certain restrictions on their use. If you run into problems, consult your database system's documentation.

Subqueries in `WHERE` Criteria

The simplest way to include a subquery in your SQL statements is as part of your `WHERE` criteria, using the = operator. In general, this type of subquery statement is best for quickly and easily looking up information based on some type of ID number.

Say you were constructing a set of pages that will enable the folks in Orange Whip Studios' accounting department to view a list of merchandise orders. On the first page, the user sees a list of all orders, retrieved straightforwardly from the `MerchandiseOrders` table. Each order can be clicked to view the name and other information about the person who placed the order. This link (for each order's contact information) passes the `OrderID` in the URL. Therefore, the code in the contact details page must retrieve information from the `Contacts` table, based on the given `OrderID`.

The following statement, which uses a subquery, does this job nicely:

```
<cfquery name="GetContact" datasource="ows">
 SELECT * FROM Contacts
 WHERE ContactID =
 (SELECT ContactID FROM MerchandiseOrders
 WHERE OrderID = #URL.orderID#)
</cfquery>
```

NOTE

This example uses the = operator to include the subquery in the larger query statement, but you also could use <>, >, <, >=, or <= in place of =.

When this query is executed, your database system works on it from the inside out, starting with the subquery. Say the `URL.orderID` value is passed as `2`. The subquery looks in the `MerchandiseOrders` table and finds that the corresponding `ContactID` number is `4`. Now the outer query statement is run, using the value of `4` in place of the subquery, almost as if the outer query had used `WHERE ContactID = 4`.

The result is that all contact information from the `Contacts` table is returned to ColdFusion, based only on the `OrderID`. This type of subquery technique is great for situations such as this, in which there is a conceptual step-by-step process to go through (first, get the `ContactID` from the `MerchandiseOrders` table; then use that to retrieve the correct record from the `Contacts` table).

NOTE

When you include a subquery in a SQL statement with the = operator, be sure that only one record will ever be found by the subquery. If the subquery part of the statement shown previously were to return more than one record, a database error message would be displayed. If the subquery might return more than one record, you should use the `IN` operator (discussed shortly) instead of =.

Subqueries Are Often Just Alternatives

Take another look at the previous subquery code snippet. There are other ways that the same information could be retrieved from the database. For instance, you could simply use two separate

<cfquery> tags, which represent the two steps mentioned previously. First, you could get the correct ContactID, like so:

```
<cfquery name="getID" datasource="ows">
 SELECT ContactID FROM MerchandiseOrders
 WHERE OrderID = #URL.orderID#
</cfquery>
```

Then, you would pass the retrieved ContactID to a second query, like so:

```
<cfquery name="GetContact" datasource="ows">
 SELECT * FROM Contacts
 WHERE ContactID = #getID.ContactID#
</cfquery>
```

This method returns the same information to ColdFusion; you are free to use either method. However, you can expect slightly better performance from the subquery approach, because only one database operation is necessary. Also, if you conceptually ask your questions all at once, your database system often can make optimizations based on indexes and other tuning algorithms, which are usually bypassed if you run two separate queries. As a general rule, the fewer times ColdFusion must interact with the database, the better.

You also could get the same information by using a join, which you learned about in the last section. The following query (which uses a join) returns the same results as the subquery statement shown earlier or the two <cfquery> tags shown most recently:

```
<cfquery name="GetContact" datasource="ows">
 SELECT c.*
 FROM Contacts c INNER JOIN MerchandiseOrders o
 ON c.ContactID = o.ContactID
 WHERE o.OrderID = #URL.orderID#
</cfquery>
```

Again, you are free to use the join syntax instead of the subquery syntax. They will both return the same data to ColdFusion. In this case, you could theoretically expect slightly better performance from the subquery because it implies that there is only one relevant ContactID and only one matching OrderID (whereas the database system would potentially need to scan all rows of the tables to see how many matching records might exist). Whether any real-world performance difference exists between the two, however, depends on the database system being used, how the columns are indexed, and other factors too numerous to discuss here. In fact, many database systems will decide to treat a subquery and a join in exactly the same manner internally.

Deciding whether to use a subquery or a join in any given situation is often a matter of personal choice. Joins are more powerful, because they can return multiple columns from all tables at once. Subqueries are easier to write and understand, and can sometimes be more efficient. The best rule of thumb is to use the method that springs to mind first as you are thinking about the query you need to write. You can always make changes later.

NOTE

As a rule, joins are more likely to run faster than subqueries, so when in doubt, use a join. But feel free to experiment and use whichever seems more straightforward for a given situation.

Using Subqueries with IN

You just saw how a subquery can be introduced into a larger query statement by including it on the right side of the = operator in the WHERE clause. That type of subquery is useful when you know the subquery will never return more than one record. If the subquery can return any number of records, you must introduce the subquery using the IN keyword. Conceptually, the values found by the subquery will be turned into a comma-separated list. Then, the comma-separated list is used to complete the outer query statement.

For instance, to get information from the Merchandise table about the items included in a particular order, you could do the following:

```
SELECT * FROM Merchandise
WHERE MerchID IN
 (SELECT ItemID FROM MerchandiseOrdersItems
 WHERE OrderID = #URL.orderID# )
```

You also can nest subqueries within other subqueries. For instance, this statement first retrieves all the OrderID numbers from the MerchandiseOrders table that a particular contact has made. The list of order numbers is passed to the middle subquery, which retrieves a list of corresponding ItemID numbers from the MerchandiseOrdersItems table. Finally, the list of item numbers is passed to the outermost query, which retrieves the list of merchandise. The result is a set of records that provide information about all of the merchandise items a particular customer has ordered:

```
SELECT * FROM Merchandise WHERE MerchID IN
 (SELECT ItemID FROM MerchandiseOrdersItems
 WHERE OrderID IN
 (SELECT OrderID FROM MerchandiseOrders
 WHERE ContactID = #URL.contactID#))
```

Again, you could get this same information using join syntax. The following join returns the same information that the previous snippet's nested subquery syntax does:

```
SELECT m.*
FROM Merchandise m, MerchandiseOrders o, MerchandiseOrdersItems oi
WHERE m.MerchID = oi.ItemID AND o.OrderID = oi.OrderID
AND o.ContactID = #URL.contactID#
```

The different syntaxes will feel more natural to different developers. The thought process to get to the first query is more step by step, whereas the second requires a more relational mindset. Use the approach that seems more intuitive to you.

NOTE

The IN keyword isn't just for subqueries. It's for any situation in which you need to provide a comma-separated list of values as criteria. For instance, you could use `WHERE OrderID IN (4,7,9)` to get information about order numbers 4, 7, and 9. Or, if you have several check boxes named OrderID on a form, you could use `WHERE OrderID IN (#FORM.orderID#)` to retrieve information about the orders the user checked.

Using Subqueries with NOT IN

One interesting way to use subqueries is with the NOT IN operator. You can use NOT IN just like IN, as discussed previously. The difference is that the outer query will retrieve records that don't correspond

to the values found by the subquery. In other words, the query as a whole will return the opposite set of records.

So, to select the merchandise items a user hasn't ordered in the past, you could use `NOT IN` as follows. This is the same as the first example in the previous section, "Using Subqueries with `IN`," except that it adds the word `NOT` to invert the results:

```
SELECT * FROM Merchandise
WHERE MerchID NOT IN
 (SELECT ItemID FROM MerchandiseOrdersItems
 WHERE OrderID = #URL.orderID# )
```

Listing 41.10 shows how to use several nested `IN` and `NOT IN` subqueries together to create a lightweight collaborative filtering effect. The term collaborative filtering is sometimes used to describe the online merchandising technique of presenting a user with a list of suggested items based on what similar people have ordered in the past. The technique can be seen most visibly at Amazon.com ("People who bought *Raiders of the Lost Aardvark* also bought *The Mommy Returns*"). The listing that follows is relatively simple and might not count as true collaborative filtering, bit it shows how subqueries can be used to create something fairly similar.

The idea behind Listing 41.10 is simple: It updates the `StoreCart.cfm` template from Chapter 22, "Online Commerce," online. When the user prepares to check out by visiting the version of `StoreCart.cfm` shown in Listing 41.10, the `getSimilar` query is run. `getSimilar` looks at the items in the user's cart and finds the completed orders in which the items have appeared. In addition, if the user has logged in, the query uses a `NOT IN` subquery to ensure that the user's own orders are not included in the list of completed orders being considered.

Then, for each of the other completed orders, the query retrieves all items that were included in those orders (not counting the items in the current user's cart). These items are then shown to the user with the message "People who have purchased items in your cart have also bought the following," as shown in Figure 41.8.

Listing 41.10 `StoreCart3.cfm`—Using Subqueries to Mimic a Collaborative Filtering Feature

```
<!---
 Filename: StoreCart.cfm (save in chapter 41s folder)
 Created by: Nate Weiss (NMW)
 Purpose: Displays the current user's shopping cart
 Please Note Depends on the <cf_ShoppingCart> custom tag
--->

<!--- Show header images, etc., for Online Store --->
<cfinclude template="StoreHeader.cfm">

<!--- If MerchID was passed in URL --->
<cfif isDefined("URL.addMerchID")>
  <!--- Add item to user's cart data, via custom tag --->
  <cf_ShoppingCart action="Add" merchID="#URL.addMerchID#">

<!--- If user is submitting cart form --->
<cfelseif isDefined("FORM.merchID")>
```

Listing 41.10 (continued)

```
    <!--- For each MerchID on Form, Update Quantity --->
    <cfloop list="#FORM.merchID#" index="thisMerchID">
    <!--- Update Quantity, via Custom Tag --->
    <cf_ShoppingCart action="Update" merchID="#thisMerchID#"
    quantity="#FORM['Quant_#ThisMerchID#']#">
    </cfloop>

    <!--- If user submitted form via "Checkout" button, --->
    <!--- send on to Checkout page after updating cart. --->
    <cfif isDefined("FORM.isCheckingOut")>
      <cflocation url="../22/StoreCheckout.cfm">
    </cfif>
  </cfif>

  <!--- Get current cart contents, via Custom Tag --->
  <cf_ShoppingCart action="List" returnVariable="getCart">


  <!--- Stop here if user's cart is empty --->
  <cfif getCart.recordCount eq 0>
    There is nothing in your cart.
    <cfabort>
  </cfif>

  <!--- Create form that submits to this template --->
  <cfform action="#CGI.SCRIPT_NAME#">
   <table>
   <tr>
   <th colspan="2" bgcolor="silver">Your Shopping Cart</th>
   </tr>
   <!--- For each piece of merchandise --->
   <cfloop query="getCart">
     <tr>
     <td>
     <!--- Show this piece of merchandise --->
     <cf_MerchDisplay merchID="#getCart.MerchID#" showAddLink="No">
     </td>
     <td>
     <!--- Display Quantity in Text entry field --->
     <cfoutput>
     Quantity:
     <cfinput type="hidden" name="merchID" value="#getCart.merchID#">
     <cfinput type="text" size="3" name="quant_#getCart.MerchID#"
     value="#getCart.Quantity#"><br>
     </cfoutput>
     </td>
     </tr>
   </cfloop>

   </table>
```

Listing 41.10 (continued)

```
 <!--- Submit button to update quantities --->
 <cfinput type="submit" name="submit" value="Update Quantities">

 <!--- Submit button to Check out --->
 <cfinput type="submit" value="Checkout" name="isCheckingOut">
</cfform>


<!--- Convert current cart contents to comma-sep list --->
<cfset currentMerchList = valueList(getCart.MerchID)>

<!--- Run query to suggest other items for user to buy --->
<cfquery name="getSimilar" datasource="#APPLICATION.dataSource#"
 cachedwithin="#createTimeSpan(0,0,5,0)#" maxrows="3">
 <!--- We want all items NOT in user's cart... --->
 SELECT ItemID
 FROM MerchandiseOrdersItems
 WHERE ItemID NOT IN (#currentMerchList#)
 <!--- ...but that *were* included in other orders --->
 <!--- along with items now in the user's cart... --->
 AND OrderID IN
 (SELECT OrderID FROM MerchandiseOrdersItems
 WHERE ItemID IN (#currentMerchList#)
 <!--- ...not including this user's past orders! --->
 <cfif isDefined("SESSION.auth.contactID")>
 AND OrderID NOT IN
 (SELECT OrderID FROM MerchandiseOrders
 WHERE ContactID = #SESSION.auth.contactID#)
 </cfif> )
</cfquery>

<!--- If at least one "similar" item was found --->
<cfif getSimilar.RecordCount gt 0>
 <p>People who have purchased items in your
 cart have also bought the following:<br>

 <!--- For each similar item, display it, via Custom Tag --->
 <!--- (show five suggestions at most) --->
 <cfloop query="getSimilar">
  <cf_MerchDisplay merchID="#getSimilar.ItemID#">
 </cfloop>
</cfif>
```

NOTE

This `StoreCart.cfm` listing is meant to work with the other listings from Chapter 22. For all the links to work properly, you should save Listing 41.10 as `StoreCart.cfm` in the same folder with Chapter 22's listings. We have also included most of the relevant files with this chapter's listings online.

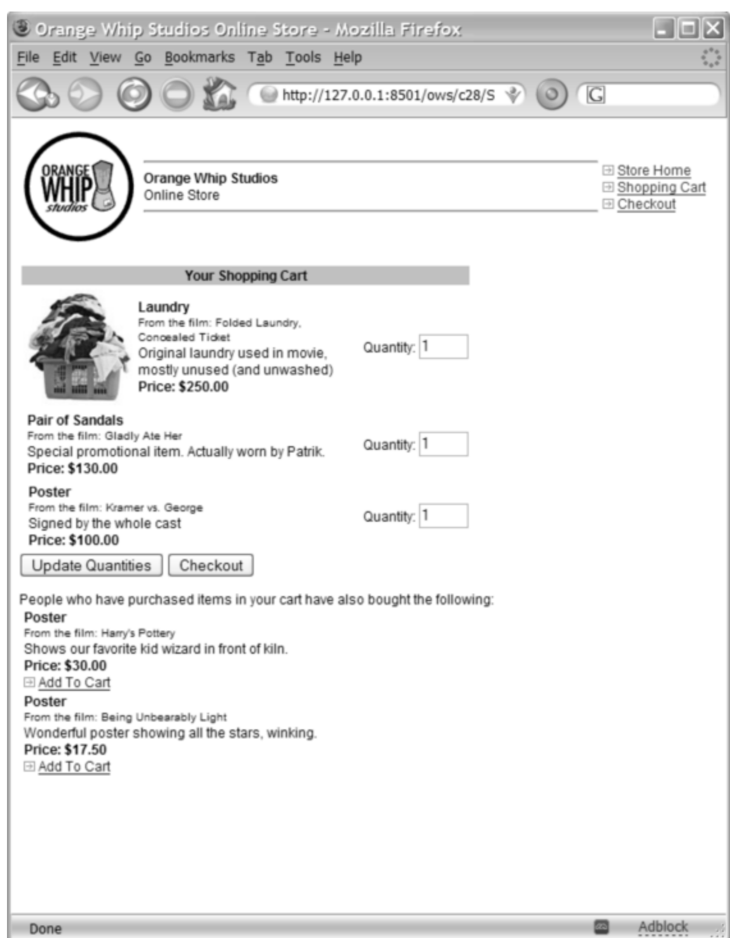


Figure 41.8

You can use `IN` and `NOT IN` subqueries to build interesting takes on your data, such as this lightweight collaborative filtering example.

Calculating New Columns with Subqueries

Another convenient place to use subqueries is in the `SELECT` part of an ordinary query statement. Simply include the subquery in the `SELECT` list, as if it were a column. Within the subquery, add criteria to its `WHERE` clause using dot notation, similar to a join. The idea is to somehow connect the subquery to the outer query, usually based on a common ID number of some type.

For instance, to get a list of all contacts in the `Contacts` table, along with a simple count of the number of orders that each contact has made, you could use the following query:

```
SELECT ContactID, FirstName, LastName,
 (SELECT COUNT(*) FROM MerchandiseOrders o
 WHERE c.ContactID = o.ContactID) AS OrderCount
FROM Contacts c
```

You could get this same information using an outer join, as discussed earlier in this chapter, as shown below:

```
SELECT c.ContactID, c.FirstName, c.LastName,
 COUNT(o.OrderID) AS OrderCount
FROM Contacts c LEFT OUTER JOIN MerchandiseOrders o
ON c.ContactID = o.ContactID
GROUP BY c.ContactID, c.FirstName, c.LastName
```

NOTE

Because an aggregate function is in this join query's `SELECT` list, all other columns must be repeated in the `GROUP BY` list. Depending on the situation and the database system being used, queries that have a long `GROUP BY` list can start to perform somewhat poorly when the number of rows in the tables gets to be large.

Combining Record Sets with `UNION`

You can use SQL's `UNION` operator to combine the results from two different queries. Compared with joins, subqueries, aggregates, and the like, `UNION` is pretty simple. `UNION` is simply about combining the results of two different `SELECT` statements. Each `SELECT` can be based on the same table or different tables.

Say someone at Orange Whip Studios has asked you to come up with a simple report page that shows which actors and directors are involved with each of the studio's films. For each film, the people's last names should be sorted alphabetically.

Listing 41.11 shows how the `UNION` operator makes this task easy. The `<cfquery>` in this template contains two `SELECT` statements. The first retrieves information about the actors in each film. If it were run on its own, it would return four columns: `MovieTitle`, `NameFirst`, `NameLast`, and `Credit`. The second `SELECT` retrieves information about the directors of each film. Run on its own, it would also return four columns, with slightly different names (`MovieTitle`, `FirstName`, `LastName`, and `Credit`). The `UNION` statement causes the records from both statements to be returned together, in the order specified by the `ORDER BY` at the end of the query (first by `MovieTitle` and then by the last names of each actor or director involved with that movie). The results are shown in Figure 41.9.

NOTE

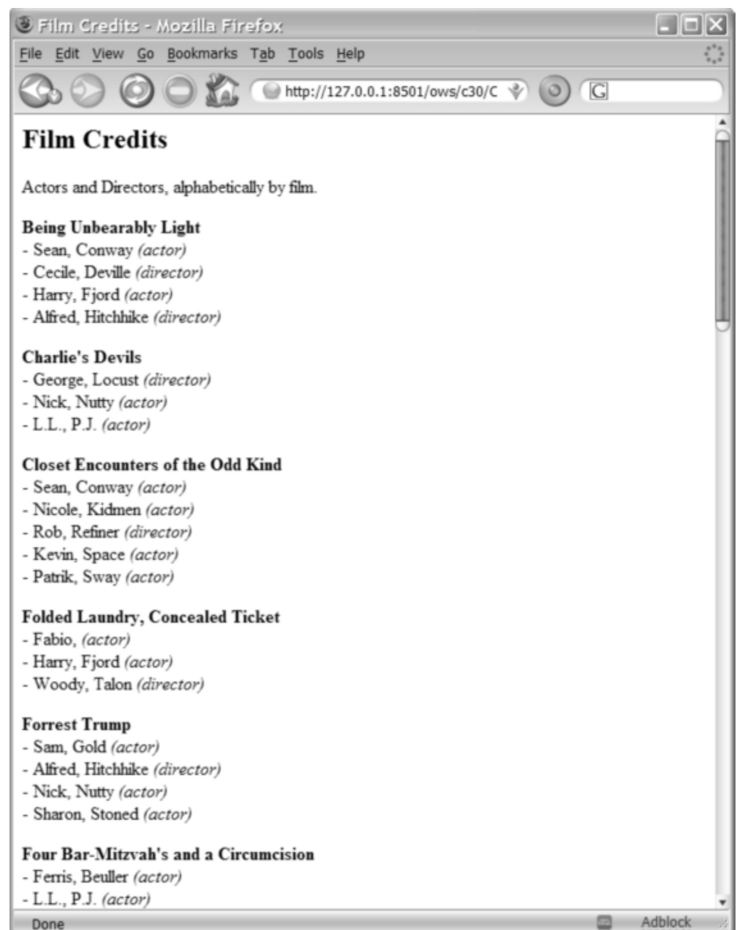
ColdFusion's query of queries feature enables you to perform `UNION`s between queries that come from different data sources or database systems. See the section "Queries of Queries," later in this chapter.

NOTE

Even though the column names aren't exactly the same, this query is still valid (see the following list of rules and considerations). Use the column names in the first `SELECT` statement in your ColdFusion code.

Figure 41.9

`UNION` statements allow you to combine the results of several `SELECT` statements.



Listing 41.11 `CreditReport.cfm`—Using `UNION` to Combine Two Result Sets

```
<!---
 Filename: CreditReport.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Demonstrates use of SQL UNION statements
--->

<html>
<head><title>Film Credits</title></head>
<body>
<h2>Film Credits</h2>
Actors and Directors, alphabetically by film.

<!--- Retrieve Films and Actors, then --->
<!--- Retrieve Films and Directors, then order everyone --->
<!--- by related Film Title and the person's last name. --->
<cfquery name="getExp" datasource="#APPLICATION.dataSource#">
```

Listing 41.11 (continued)

```
 SELECT
 f.FilmID, f.MovieTitle,
 a.NameFirst, a.NameLast, 'actor' AS Credit
 FROM
 Films f, Actors a, FilmsActors fa
 WHERE
 f.FilmID = fa.FilmID AND
 a.ActorID = fa.ActorID
 UNION
 SELECT
 f.FilmID, f.MovieTitle,
 d.FirstName as NameFirst, d.LastName as NameLast, 'director' AS Credit
 FROM
 Films f, Directors d, FilmsDirectors fd
 WHERE
 f.FilmID = fd.FilmID AND
 d.DirectorID = fd.DirectorID
 ORDER BY
 MovieTitle, NameLast
</cfquery>

<!--- For each film, show the title --->
<cfoutput query="getExp" group="FilmID">
 <p><b>#MovieTitle#</b><br>

 <!--- Within each film, show each person involved --->
 <cfoutput>
 - #NameFirst#, #NameLast# <i>(#Credit#)</i><br>
 </cfoutput>
</cfoutput>

</body>
</html>
```

NOTE

The `Credit` column for the first `SELECT` statement has a constant value of `actor`. Every row returned by that statement will hold the actual word `actor` in the `Credit` column. The same column for the second `SELECT` has a constant value of `director`. This allows the user to see whether each person is an actor or a director (otherwise, there would be no way to tell, because they are all mixed together in alphabetical order). It isn't done very often, but you can actually return constant values from any query. Just provide the constant value (a string in single quotes or a number without any quotes) where you would normally provide the column name; then give the new pseudo-column an alias using the `AS` keyword.

TIP

You can use more than one `UNION` in a SQL statement–which means that you can combine records from more than two `SELECT` statements.

When you are using `UNION` in your queries, the following rules and considerations apply:

- All of the `SELECT` statements must specify the same number of columns.
- The columns' data types—`Text`, `Numeric`, `Date`, and so on—must match. However, the text width or numeric precision of the columns might not need to, depending on the type of database you are using. See your database documentation for details.

- The column names in the SELECT statements *don't* have to match. In fact, the column names from the first SELECT are the only ones that matter and are the only ones that will be returned to ColdFusion.
- Duplicate rows (rows that are the same from more than one of the SELECT statements) are automatically eliminated. This is true unless you use UNION ALL instead of UNION.
- There might be only one ORDER BY statement, at the very end, and it refers only to column names from the first SELECT statement. It arranges all the rows from all the queries involved in the UNION, intermingling the rows with one another.
- With most database systems, you may not use DISTINCT and UNION in the same query.
- So-called BLOB (Binary Large Object) columns, which allow unlimited amounts of data to be stored (such as Memo columns in Access or Image columns in SQL Server), generally cannot be used in UNION statements. Some database systems might make an exception to this rule if you are using UNION ALL instead of UNION.

Working with NULL Values

One of the most confusing concepts we encounter when starting out with databases is the idea of a null value. A null value indicates that there is literally nothing recorded in the table for that row and column (in layman's terms, the value was left blank). For instance, consider the MerchandiseOrders table's ShipDate column. When the order is shipped, the date is recorded here. Until that point, there is nothing in the column; its value is NULL.

Setting Values to NULL

To indicate that you want to set a column to a null value, use the keyword NULL where you would normally provide a value. Don't surround the NULL keyword with quotation marks; that would set the column to the four-character string NULL rather than the single, special value of NULL.

If you need to record a null value when a form field has been left blank, ColdFusion's <cfif> and <cfelse> tags come in handy. For instance, imagine a form with OrderID and ShipDate fields on it. The following code snippet would set the ShipDate column to whatever the user provides, as long as it is a valid date. If the form field has been left blank, or if ColdFusion doesn't recognize it as a date value, it will be set to NULL to indicate that no date is known:

```
UPDATE MerchandiseOrders
SET ShipDate = <cfif isDate(FORM.shipDate)>
 '#createODBCDate(FORM.ShipDate, "m/d/yyyy")#'
 <cfelse> NULL </cfif>
WHERE OrderID = #FORM.orderID#
```

Testing for NULL Values in SQL Code

SQL statements that deal with null values can produce unexpected results if you don't know what to watch out for. Without getting into the formal theory behind null values, the basic idea is that a null

value by its very definition is not equal or unequal to anything—not even another null value. Therefore, the following statement should not return any records, even if null values exist for `ShipDate` in the table:

```
SELECT * FROM MerchandiseOrders
WHERE ShipDate = NULL
```

Instead, to retrieve the orders that haven't yet shipped, you must use the special `IS NULL` operator, which is used only for testing for `NULL` values. In addition, a special `IS NOT NULL` operator is available, which can be used to return values that do not have `NULL` values.

So, the following could be used to find orders that have not yet shipped:

```
SELECT * FROM MerchandiseOrders
WHERE ShipDate IS NULL
```

Conversely, to find records that have already been shipped, you would use this:

```
SELECT * FROM MerchandiseOrders
WHERE ShipDate IS NOT NULL
```

NOTE

Some database systems, especially older ones, don't make a distinction between `= NULL` and `IS NULL`. But you should still avoid using = to test for null values in your tables, to ensure that your applications will continue to work while the database system is brought up to date.

It's also important to note that any comparison against a null value (not just equality comparisons done with the = operator) must, by definition, not result in a match. Therefore, inequality tests (with the <> operator) and other comparisons (such as < or >=) often produce what end users might consider to be counterintuitive results. Take a bit of time to think about how any null values might affect your application's queries. For instance, at first glance, you would probably expect the following snippet to return all films except for one (*The Lice Storm*):

```
SELECT * FROM MerchandiseOrders
WHERE MovieTitle <> 'The Lice Storm'
```

Strange as it might seem, if any of the films have a null value in the `MovieTitle` column (perhaps the studio's focus groups haven't yet determined a title), those films will not be returned by the previous snippet, because null values are not allowed to pass an inequality test. To get all rows except *The Lice Storm*, including ones that have `NULL` titles, you would need to use the following:

```
SELECT * FROM MerchandiseOrders
WHERE (MovieTitle <> 'The Lice Storm' OR MovieTitle IS NULL)
```

Testing for `NULL` Values in CFML Code

The situation is further complicated by the fact that CFML doesn't include the concept of a null value (which is surprising, given the database-centric nature of the product). When ColdFusion receives a `NULL` value from a database query, it converts the value into an empty string.

NOTE

SQL purists wince when they hear this, because it means that there is no way to tell the difference between a `NULL` value and a value that has actually been set to an empty string. In practice, though, this isn't usually much of a problem, because such a distinction rarely is significant in a real-world application. Would there ever be a need, for instance, to distinguish between a film title that hasn't yet been named (a null value) and a film that has actually been named `""` (an empty string)? Probably not. So it would be nice if ColdFusion maintained null values in query objects, but the fact that it doesn't isn't a deal breaker.

If you want to display a date only when it is has not been set to a null value in the database, you can test for an empty string in your CFML code, like so:

```
<cfif getOrders.ShipDate eq "">
 (this order has not shipped yet)
<cfelse>
 Shipped on: #lsDateFormat(getOrders.ShipDate)#
</cfif>
```

Working with Date Values

Working with date values in databases and ColdFusion can be confusing. The confusion often stems from the fact that a date value in a database is recorded conceptually as a moment in history, rather than as a simple string (such as `4/8/01`). How exactly the date is stored internally is up to the database system. Often it's stored as a very long number that represents the number of milliseconds before or after some fixed reference point. Only when the figure is returned to ColdFusion and then output using a CFML function such as `dateFormat()` or `lsDateFormat()` does it look like a date to us humans.

NOTE

This discussion assumes that you are storing your dates in actual date-type columns in your database tables. Depending on the database system you're using, these columns can be called date columns, date/time columns, or something similar.

Specifying Dates in `<cfquery>` Tags

First, the bad news: Not all database systems agree on the syntax used to specify a date in a query. Some are willing to parse the date based on a number of formats; others have a strict format that must be used at all times. The good news is that the ODBC specification defines a common format that can be used for any ODBC data source. The JDBC specification uses the same format, and ColdFusion conveniently provides a `createODBCDate()` function that can be used to automatically put dates into this format in your `<cfquery>` tags. Since all database connections in ColdFusion use JDBC under the hood, you can nearly always use `createODBCDate()` to specify dates in queries.

NOTE

It's admittedly a bit strange that you use a function with ODBC in the name to specify dates, even when you're not using an ODBC data source. Chalk it up to history; it's just the way things have evolved over time.

So, for almost all data sources, you can use `createODBCDate()` to correctly format the date in your `INSERT` and `UPDATE` queries. Here's a sample snippet that updates a column based on a form field called `dateInTheaters`:

```
<cfquery datasource="ows">
 UPDATE Films
```

```
 SET DateInTheaters = #createODBCDate(FORM.dateInTheaters)#
 WHERE FilmID = #FORM.filmID#
</cfquery>
```

NOTE

If your situation is such that you can use `<cfupdate>` or `<cfinsert>` to update your database, you don't even have to worry about this part. ColdFusion takes care of writing the correct query syntax for you.

Alternatively, you can substitute the `createODBCDate()` function with a `<cfqueryparam>` tag that uses a `CFSQLTYPE` attribute of `CF_SQL_DATE`. ColdFusion will take care of adapting your SQL statement so the date is sent to the database in a format it will understand.

You will learn more about `<cfqueryparam>` later in this chapter. For now, just think of it as a way to tell ColdFusion to convert the `value` you supply into whatever format is appropriate for the type of database you are using (ODBC or otherwise). The query shown previously, then, would become this:

```
<cfquery datasource="ows">
 UPDATE Films
 SET DateInTheaters =
 <cfqueryparam cfsqltype="CF_SQL_DATE" value="#FORM.dateInTheaters#">
 WHERE FilmID = #FORM.filmID#
</cfquery>
```

As of ColdFusion MX, the decision to use `createODBCDate()` or `<cfqueryparam>` as shown above is up to you; both will do the same thing. The distinction was more important in previous versions of ColdFusion, which didn't use JDBC as a common ground for all data sources. For more about `<cfqueryparam>`, see the section "Parameterized Queries," later in this chapter.

Dates in Query Criteria

Developers often get confused about how to work with columns that contain both date and time information. For instance, consider the `ShipDate` column of the `MerchandiseOrders` table in the Orange Whip Studios database. This column contains the date and time the order was shipped. The date and time are not stored separately. They are stored as one value, which you can think of as a moment in history.

If, when a date is being recorded in a database, a time isn't supplied, the database will store the time as midnight at the beginning of that day. If, later, you retrieve that date value with a `SELECT` statement and output the time portion of it with CFML's `timeFormat()` function, the time will display as `12:00 AM`.

All of this makes perfect sense. But you will often need to query the database based on a date value, where the time portions of the date values can feel like they are working against you. For instance, say you are allowing the user to see all orders made on a particular day. You decide to allow the user to specify the date in a form field called `searchDate`, using a `<cfinput>` tag, like this:

```
<cfinput
 name="searchDate"
 validate="date"
 required="Yes" message="You must provide a date first!">
```

The user expects to be able to type a simple date into this field (perhaps `3/18/2001`) to see all orders placed on that date. You decide, quite sensibly, to write a query such as the following, which executes when the form is submitted:

```
<cfquery name="GetOrders" database="ows">
 SELECT * FROM MerchandiseOrders
 WHERE OrderDate = #createODBCDate(FORM.searchDate)#
</cfquery>
```

When you test the template, though, you find that it doesn't work as planned. Users expect to see all orders placed on the date they specify, no matter what time the order was placed during that day. However, your code never seems to find any records. This is because no time is specified in the query criteria, so the database assumes a time of midnight. If the user types `3/18/2001` in the search field, your query is sent as something such as the following to your database system (conceptually):

```
 SELECT * FROM MerchandiseOrders
 WHERE OrderDate = '3/18/2001 12:00 AM'
```

Once you see it this way, it becomes clear what's going on: Your query returns only records that were shipped at exactly midnight on the specified day. You need to translate the user-specified date into a *range* between two moments in time (the beginning of that day and the end).

The following will work as expected, returning all orders placed on or after midnight at the beginning of the specified day, all the way up to (but not including) midnight at the end of that same day:

```
<cfquery name="getOrders" database="ows">
 SELECT * FROM MerchandiseOrders
 WHERE OrderDate >= #createODBCDate(FORM.searchDate)#
 AND OrderDate < #createOBDCDate(dateAdd("d", FORM.searchDate, 1))#
</cfquery>
```

The following query will also work as expected and can be used interchangeably with the one shown previously. It uses a special keyword provided by SQL, called `BETWEEN`:

```
<cfquery name="getOrders" database="ows">
 SELECT * FROM MerchandiseOrders
 WHERE OrderDate BETWEEN
 #createODBCDate(FORM.searchDate)# AND
 #createOBDCDate(dateAdd("d", FORM.searchDate, 1))#
</cfquery>
```

NOTE

You could substitute all the `createODBCDate()` functions shown in this section with a `<cfqueryparam>` tag, as explained earlier in the section "Specifying Dates in `<cfquery>` Tags."

Understanding Views

Many database systems support something called a *view*. The idea behind a view is to save the SQL code for a particular `SELECT` query as a permanent part of the database. From that point on, the name of the view can be used like a table name in other queries.

Creating a View

Again, the exact syntax needed to create a view might vary depending on your database system, but you usually can create one by typing the words `CREATE VIEW`, then a name for the view, then the word `AS`, and then the actual `SELECT` statement you want to use to create the view, like so:

```
CREATE VIEW OrdersPending AS
SELECT * FROM MerchandiseOrders
WHERE ShipDate IS NULL
```

After you execute this SQL statement once (either via a `<cfquery>` tag; by executing it with ColdFusion Studio's Query Builder; or via whatever command-line, graphical, or other query tools your database system provides), it becomes a part of your database. You can now refer to the `OrdersPending` view in your queries as if it were a table. It's as if you have created a virtual or shadow table that always contains the same data as the `MerchandiseOrders` table, except that all the orders that have already shipped (where the `ShipDate` is not a null value) are excluded.

For instance, the following would return all pending orders:

```
SELECT * FROM OrdersPending
ORDER BY OrderDate
```

NOTE

The exact definition of what a view is called, how you create it, and what it can do varies depending on the database system you use. For instance, if you use `CREATE VIEW` with an Access database, Access will create what it normally calls a query.

Advantages of Using Views

You almost never need to create a view when building a ColdFusion application, but they can be helpful. Here, the `OrdersPending` view makes it easier to separate the abstract idea of a pending order from its implementation in the database (the `NULL` value). If you're working on a part of the application that deals only with pending orders (a section of the company intranet for the shipping department, say), you can use the `OrdersPending` view in each of your queries, instead of having to remember to use `ShipDate IS NULL` each time.

Also, if in the future you decide to keep pending orders in a different table, you would need to change only the definition of the view, rather than having to alter each individual query. In this respect, views become a method of abstraction, somewhat comparable to CFML's custom tags. They become a tool for separating your application's logic from the physical details of how the data is stored behind the scenes.

In fact, if your database is being designed by a different person on your team, or if you are connecting ColdFusion to some type of legacy database already in place, your database administrator may decide to give you access to only the `OrdersPending` view rather than the underlying `MerchandiseOrders` table. This way, the administrator knows that your ColdFusion application won't be able to display inappropriate records (in this case, orders that have already shipped).

You can also create views that involve more than one table by using ordinary join syntax in the `SELECT` statement that follows `CREATE VIEW`. For instance, you could create a view called `ActorsInFilms`, which

uses a join to select the `FilmID` and `MovieTitle` of each film, plus the `ActorID` and name of each actor in that film. The `CREATE VIEW` statement would look like this:

```
CREATE VIEW ActorsInFilms AS
SELECT f.FilmID, f.MovieTitle, a.ActorID, a.NameFirst, a.NameLast
FROM Actors a, Films f, FilmsActors fa
WHERE a.ActorID = fa.ActorID AND f.FilmID = fa.FilmID
```

After the previous snippet is executed once, you can retrieve the title and actors for any film without having to use any joins at all. For instance:

```
SELECT * FROM ActorsInFilms
WHERE FilmID = #URL.FilmID#
```

NOTE

Views that involve more than one table are considered read-only by most database systems. You can `SELECT` from them but not make changes via `INSERT`, `UPDATE`, or `DELETE`.

Additional `<cfquery>` Topics

The rest of this chapter discusses several advanced features provided by the `<cfquery>` tag. Each of these features provides you with finer-grained control over how ColdFusion interacts with your database or database server.

Query of Queries (In Memory Queries)

One of the most interesting features in ColdFusion is its improved *query of queries* (QofQ) capability. As the name implies, this feature lets you retrieve information from queries that have already been run, using standard SQL syntax. The feature is simple but has many uses. This section introduces you to the query of queries feature and suggests several ways to put it to use in your ColdFusion applications.

NOTE

The ColdFusion documentation refers to this feature as Query of Queries and also as In-Memory Queries (IMQ). I will use the term Query of Queries in this discussion because that is what the feature has been called in the past.

The Basics

Considering all the problems it can solve, the actual process of using the query of queries feature is surprisingly straightforward. Best of all, it lets you get new benefits from the SQL skills you already have.

To use the QofQ feature, follow these steps:

1. Run one or more ordinary queries using the `<cfquery>` tag in the way that you are already familiar with. These are the queries you will be able to query further in a moment. You can think of these as source queries.

2. Create a new `<cfquery>` tag, this time with `dbtype="query"`. This tells ColdFusion you don't intend to contact a traditional database (via a database driver). Instead, you will query the results returned by the queries from Step 1.
3. Within this new `<cfquery>` tag, write SQL code that retrieves the records you want, using the names of the source queries as if they were table names. You can't do everything you would be able to within a traditional database query, but the most important SQL concepts and keywords are supported.
4. Now you can use the results of the query normally, just as you would any other query. You can output its records in a `<cfoutput>` block, loop through them using `<cfloop>`, and so on. You can even query the records again, using yet another query of queries.

Using QofQ for Heterogeneous Data Analysis

Say Orange Whip Studios has some type of legacy database in place that predates your ColdFusion application. Perhaps the studio is experimenting with direct phone sales, trying to get people to buy collector's editions of coins (complete with certificates of authenticity) that commemorate the studio's classic films.

This database has just one table, called `Calls`, which contains the `ContactID` of each person called and columns named `CallID`, `CallDate`, `Status`, and `Comments`. Using the ColdFusion Administrator, you set up a new ODBC data source called `DirectSales` and then write a query that retrieves all the records from the `Calls` table, like so:

```
<cfquery datasource="DirectSales" name="GetCalls">
 SELECT ContactID, CallID, CallDate, Status, Comments
 FROM Calls
</cfquery>
```

Next, you run a second query against the `Contacts` table, using the usual `ows` data source:

```
<cfquery datasource="ows" name="GetContacts">
 SELECT ContactID, FirstName, LastName
 FROM Contacts
</cfquery>
```

Now you can use the QofQ feature to join the results of the `GetCalls` and `GetContacts` queries, like so:

```
<cfquery dbtype="query" name="GetJoined">
 SELECT *
 FROM GetCalls, GetContacts
 WHERE GetCalls.ContactID = GetContacts.ContactID
</cfquery>
```

That's it. Now you can use the `GetJoined` query just like the results of any other `<cfquery>` tag. Its results will contain one row for each row of the `GetCalls` and `GetContacts` queries that contain the same `ContactID`. All columns from the original query will be included.

NOTE

What makes this particularly interesting is the fact that the records from the two source queries need not come from the same database or even the same type of database system. In fact, as you will see shortly, the source queries don't even have to come from databases at all. Any ColdFusion tag or function that returns a query object, such as the `<cfdirectory>`, `<cfpop>`, or `<cfsearch>` tags, can be requeried using the query of queries feature.

SQL Statements Supported by Query of Queries

In general, you can use just about any type of SQL statement in a QofQ query. There are a few exceptions, though. The absolute nitty-gritty can be found in the ColdFusion documentation, but the important things to keep in mind are all listed in Table 41.2.

Table 41.2 SQL Functionality Supported by Query of Queries

SQL CONCEPT	QUERY OF QUERIES NOTES
Table Joins	Fully supported, as long as you use basic `WHERE` syntax for joining the tables (as opposed to the alternate `INNER JOIN` syntax). For details, see "The Two Types of Join Syntax," earlier in this chapter.
Outer Joins	Unfortunately, outer joins are not supported by ColdFusion's query of queries feature. Only normal table joins (inner joins) are supported. Of course, this doesn't restrict you from using outer joins in ordinary `<cfquery>` tags; the restriction applies only when using query-of-queries. Also, you are free to query a database directly using an outer join and then re-query it with QofQ. The only thing you can't do is to actually use the `OUTER JOIN` keywords inside a `<cfquery>` of `dbtype="query"`.
Unions	Fully supported. For information about unions, see "Combining Recordsets Sets with `UNION`," earlier in this chapter.
Aggregate functions, `GROUP BY` and `HAVING`	Supported. For more information, see the "Summarizing Data with Aggregate Functions" section, earlier in this chapter.
Case Sensitivity	Most database systems consider `WHERE` conditions without regard to upper- or lower case. However, `WHERE` conditions are case sensitive when using QofQ. To get non-case-sensitive behavior, you need to use the `UPPER()` or `LOWER()` functions. For details, see the "Case Sensitivity and Query of Queries" section, later in this chapter.
Reserved Words	ColdFusion's QofQ feature defines a list of reserved words—including `CHECK`, `ADD`, and `ZONE`—that can't be used as column or table names without escaping them with square brackets. See "Reserved Words and Query of Queries," later in this chapter.
Null values	Works normally, except that null values will pass inequality tests. For example, if you include `WHERE RatingID <> 3` in a QofQ, all records except for the ones with a `RatingID` of 3 will be returned, even rows where the `RatingID` was `NULL`. The proper, SQL-compliant behavior would be for the null rows to be omitted from the results. For more information about null values, see "Working with `NULL` Values" earlier in this chapter.

Using QofQ to Reduce Database Interaction

In Chapter 31, "Improving Performance," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 2: Application Development*, you learned about ColdFusion's query-caching mechanism, which allows you to transparently share query results between page requests, thereby cutting down on the interaction with the database system. You can use the QofQ feature in combination with query caching to reduce the actual communication with the database system even further.

You can use QofQ and query caching together in many ways; most are variations on the following basic idea. First, you write a source query using a normal `<cfquery>` tag that queries your database in the usual way and uses the `cachedWithin` attribute to cache the query's results. For instance, you might retrieve all records from the `Films` table and cache them for 30 minutes at a time, like so:

```
<cfquery datasource="ows" name="getFilms"
 cachedWithin="#createTimeSpan(0,0,30,0)#">
 SELECT * FROM Films
 ORDER BY MovieTitle
</cfquery>
```

Now you can get information about individual film records by querying the `GetFilms` query, instead of running a separate query. For instance, if you wanted to retrieve a film record based on a URL parameter called `filmID`, you could use code such as the following, which uses the query of queries feature to fetch a single row from the `GetFilms` query:

```
<cfquery dbtype="query" name="getThisFilm">
 SELECT * FROM GetFilms
 WHERE FilmID = #URL.filmID#
</cfquery>
```

The `getThisFilm` query can now be used to output the film record. The net effect is that you have access to all the data you need, even though the database is hit only once every 30 minutes at most. Since cutting down on database communication typically has a positive impact on overall system performance, using QofQ in this way makes sense.

But why is this necessarily better than simply letting ColdFusion run separate cached queries for each individual record? That is, why not just use another version of `getThisFilm`, rather than the combination of the `getFilms` and `getThisFilm` shown previously? Consider:

```
<cfquery datasource="ows" name="getThisFilm"
 cachedWithin="#createTimeSpan(0,0,30,0)#">
 SELECT * FROM Films
 WHERE FilmID = #URL.filmID#
</cfquery>
```

The answer is that neither approach is inherently better than the other; the choice depends on the situation. For the moment, let's suppose that 100 films exist in the `Films` table and that the individual film records are all being accessed about five times in any 30-minute period. The first approach retrieves all 100 records at the beginning of the 30 minutes and serves the individual queries from the cached source query; however, all 100 records must be remembered in ColdFusion's RAM. The second approach contacts the database more often (up to 100 times during a 30-minute period,

once for each film), but each query is very small and takes up only a tiny amount of memory in ColdFusion's RAM.

All things being equal, the first approach probably makes the most sense when a significant number of the individual records will actually be accessed during the 30-minute period. If it turns out that only 3 out of the 100 records are accessed during the 30 minutes, the memory being used by the other 97 is essentially wasted, so the second approach is probably more efficient. On the other hand, if almost all of the individual records are accessed during the 30 minutes, the first approach is likely to be more efficient.

NOTE

> Also, depending on the situation, the cached `GetFilms` query might be capable of being used by a large number of other QofQ queries in the application. Some pages might retrieve individual records using `WHERE`, as shown previously. Other pages might need to obtain counts or summaries from the data using aggregate functions and `GROUP BY` statements in QofQ queries. Still other pages might simply need to display the records as is, using the cached query results directly. Using the query of queries feature, you could tell all these pages to use the same cached version of the `GetFilms` query, so there is still only one access to the database every 30 minutes, even though the query is being massaged and reinterpreted by the various pages in various ways.

ColdFusion Is Not a Database Server

It is critical to understand that ColdFusion's query of queries feature, while very useful, probably isn't better at searching through large sets of records than a dedicated database system. Whereas database systems can rely on indexes, active query optimizers, and other tools to find the correct record in a large database table, ColdFusion's query of queries feature just iterates through the rows, looking for the correct values. This is what database server products call an iterative table scan, which is unlikely to scale well in extreme conditions.

If the number of records in the cached `getFilms` query becomes very large over time (say, 100,000 records instead of 100), the performance of the first approach will start to degrade because ColdFusion will have to look through all of them each time it needs to find an individual record. In contrast, a decent database system should be capable of returning the correct record very quickly, regardless of the size of the table (especially if a well-tuned index exists on the `FilmID` column), making the second approach more efficient.

To put it another way, the existence of the query of queries feature in ColdFusion doesn't mean that ColdFusion should be considered a database server, competing directly with high-performance database products such as MySQL, Oracle, and Microsoft SQL Server. The fact that query records are maintained in ColdFusion's RAM doesn't automatically mean ColdFusion will be capable of requerying the records more efficiently than your database server would. Today's database servers are very sophisticated animals and hard for ColdFusion to outperform.

As a rule of thumb, a practice of caching queries of 1,000 records or less (and then requerying those records using QofQ) is likely to serve you very well. When the record count starts to climb toward five digits, the benefit of having the records already in RAM will eventually be overcome by the sheer weight of the large record set and will probably be further hampered by ColdFusion's iterative approach to requerying the data.

A Real-World Example

In Chapter 29, "Improving the User Experience," in Vol. 2, *Application Development*, several versions of a next-n interface were created, allowing the user to click through records on pages ten rows at a time. Then, in Chapter 31, the next-n interface was revised to use ColdFusion's query-caching feature via the `cachedWithin` attribute to improve performance.

The code in Listing 41.12 revises the next-n interface again, this time using ColdFusion's query of queries feature to allow the user to re-sort the records by clicking the column names. Upon the user's first visit to the page, the records are sorted by date (with the most recent expense first), just as in the previous versions of the template. Then, if the user clicks the `Film` column, the page reloads with the records sorted alphabetically by film title. If the user clicks the `Film` column again, the sort order is reversed. A small triangle image indicates which column the records are sorted by, and in what order.

??AOThis listing uses `<cfinclude>` tags to include the `NextNIncludePageLinks.cfm` and `NextNIncludeBackNext.cfm` templates from Chapter 29. Either save this template in the same directory as Chapter 29's listings or copy the included templates into the folder you are using for this chapter. For your convenience, the included files have been duplicated in the folder for this chapter, online.

Listing 41.12 `NextN6.cfm`—Allowing Users to Re-sort Cached Query Results

```
<!---
 Filename: NextN6.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Displays Next N record-navigation interface
 Please Note Includes NextNIncludeBackNext.cfm and NextNIncludePageLinks.cfm
--->

<!--- Maintain ExpenseReport filtering variables at session level --->
<cfparam name="SESSION.expenseReport.userFilter" type="string" default="">
<cfparam name="SESSION.expenseReport.dateFrom" type="string" default="">
<cfparam name="SESSION.expenseReport.dateThru" type="string" default="">
<!--- Also which column is being sorted on, and in which direction --->
<cfparam name="SESSION.expenseReport.sortCol" type="string" default="d">
<cfparam name="SESSION.expenseReport.sortAsc" type="boolean" default="No">

<!--- If user is asking to change sort order --->
<cfif isDefined("URL.sortCol") and isDefined("URL.sortDir")>
 <!--- Save new order column/order in SESSION scope --->
 <cfset SESSION.expenseReport.sortCol = URL.sortCol>
 <cfset SESSION.expenseReport.sortAsc = URL.sortDir>
 <!--- Send user back to first row of query results --->
 <cfset URL.startRow = 1>
</cfif>

<!--- If the user is submitting the "filter" form, --->
<!--- we'll make their submission be the filter for rest of session --->
<cfif isDefined("FORM.userFilter")>
 <cfset SESSION.expenseReport.userFilter = FORM.userFilter>
 <cfset SESSION.expenseReport.dateFrom = FORM.dateFrom>
 <cfset SESSION.expenseReport.dateThru = FORM.dateThru>
</cfif>
<!--- Retrieve expense records from database --->
```

Listing 41.12 (continued)

```
<cfquery name="getExp" datasource="#APPLICATION.DataSource#"
 cachedwithin="#createTimeSpan(0,0,15,0)#">
 SELECT
 f.FilmID, f.MovieTitle,
 e.Description, e.ExpenseAmount, e.ExpenseDate
 FROM
 Expenses e INNER JOIN Films f
 ON e.FilmID = f.FilmID
 WHERE
 0=0
 <!--- If the user provided a filter string, --->
 <!--- show only matching films and/or expenses --->
 <cfif SESSION.expenseReport.userFilter is not "">
 AND (f.MovieTitle LIKE '%#SESSION.expenseReport.userFilter#%' OR
 e.Description LIKE '%#SESSION.expenseReport.userFilter#%')
 </cfif>
 <!--- Also filter on From date, if provided --->
 <cfif isDate(SESSION.expenseReport.dateFrom)>
 AND e.ExpenseDate >= #createODBCDate(SESSION.expenseReport.dateFrom)#
 </cfif>
 <!--- Also filter on Through date, if provided --->
 <cfif isDate(SESSION.expenseReport.dateThru)>
 AND e.ExpenseDate <= #createODBCDate(SESSION.expenseReport.dateThru)#
 </cfif>
 ORDER BY
 e.ExpenseDate DESC
</cfquery>


<!--- If user's current sort order differs from the --->
<!--- default, use Q-of-Q to re-sort original query --->
<cfif not
 (SESSION.expenseReport.sortCol eq "d" and SESSION.expenseReport.sortAsc eq "No")>
 <!--- Re-query "GetExp" with appropriate ORDER BY --->
 <cfquery name="getExp" dbtype="query">
 SELECT * FROM GetExp
 ORDER BY
 <!--- Use appropriate sort column --->
 <cfswitch expression="#SESSION.expenseReport.sortCol#">
 <cfcase value="f">MovieTitle</cfcase>
 <cfcase value="e">Description</cfcase>
 <cfcase value="a">ExpenseAmount</cfcase>
 <cfdefaultcase> ExpenseDate</cfdefaultcase>
 </cfswitch>
 <!--- Appropriate sort direction --->
 <cfif SESSION.expenseReport.sortAsc>ASC<cfelse>DESC</cfif>
 </cfquery>
</cfif>

<!--- Number of rows to display per Next/Back page --->
<cfset rowsPerPage = 10>
<!--- What row to start at? Assume first by default --->
<cfparam name="URL.startRow" default="1" type="numeric">
<!--- Allow for Show All parameter in the URL --->
<cfparam name="URL.showAll" type="boolean" default="No">
```

Listing 41.12 (continued)

```
<!--- We know the total number of rows from query --->
<cfset totalRows = getExp.RecordCount>
<!--- Show all on page if ShowAll passed in URL --->
<cfif URL.showAll>
 <cfset rowsPerPage = totalRows>
</cfif>
<!--- Last row is 10 rows past the starting row, or --->
<!--- total number of query rows, whichever is less --->
<cfset endRow = min(URL.startRow + rowsPerPage - 1, totalRows)>
<!--- Next button goes to 1 past current end row --->
<cfset startRowNext = endRow + 1>
<!--- Back button goes back N rows from start row --->
<cfset startRowBack = URL.startRow - rowsPerPage>

<!--- Page Title --->
<html>
<head><title>Expense Browser</title></head>
<body>
<cfoutput><h2>#APPLICATION.companyName# Expense Report</h2></cfoutput>

<!--- Simple style sheet for formatting --->
<style>
 FORM { font-family:sans-serif;font-size:smaller;}
 TH { font-family:sans-serif;font-size:smaller;
 background:navy;color:white}
 TD { font-family:sans-serif;font-size:smaller}
 TD.DataA { background:silver;color:black}
 TD.DataB { background:lightgrey;color:black}
 A.Head { color:white}
 A.Head:visited { color:white}
</style>

<!--- Simple form to allow user to filter results --->
<cfform action="#CGI.script_name#" method="post">
 <!--- Filter string --->
 <b>Filter:</b>
 <cfinput type="Text" name="userFilter"
 value="#SESSION.expenseReport.userFilter#" size="15">

 <!--- From date --->
  
 <b>Dates:</b> from
 <cfinput type="Text" name="dateFrom" value="#SESSION.expenseReport.dateFrom#"
 size="9" validate="date" message="Please enter a valid date, or leave it blank.">

 <!--- Through date --->
 through
 <cfinput type="Text" name="dateThru" value="#SESSION.expenseReport.dateThru#"
 size="9" validate="date" message="Please enter a valid date, or leave it blank.">

 <!--- Submit button to activate/change/clear filter --->
 <cfinput type="submit" name="submit" value="Apply">
</cfform>

<table width="600" border="0" cellSpacing="0" cellPadding="1">
```

Listing 41.12 (continued)

```
<!--- Row at top of table, above column headers --->
<tr>
<td width="500" colspan="3">
<!--- Message about which rows are being displayed --->
<cfoutput>
Displaying <b>#URL.startRow#</b> to <b>#endRow#</b>
of <b>#TotalRows#</b> Records<br>
</cfoutput>
</td>
<td width="100" align="right">
<cfif not URL.showAll>
<!--- Provide Next/Back links --->
<cfinclude template="NextNIncludeBackNext.cfm">
</cfif>
</td>
</tr>

<!--- Row for Column Headers --->
<tr>
<cfoutput>
<!--- For each of the four columns... --->
<cfloop list="Date,Film,Expense,Amount" index="col">
<!--- Use 1st letter of column as "Alias" to pass in URL --->
<cfset alias = lCase(left(col, 1))>
<!--- If user already viewing by this col, link should --->
<!--- reverse order; otherwise, order should be ASC for --->
<!--- all columns except Date, when order should be DESC --->
<cfif SESSION.expenseReport.sortCol eq alias>
 <cfset sortDir = not SESSION.expenseReport.sortAsc>
<cfelse>
  <cfset sortDir = col neq "Date">
</cfif>
<!--- URL for when user clicks on column name --->
<cfset sortLink = "#CGI.script_name#?sortCol=#alias#&sortDir=#sortDir#">
<th>
<!--- Show column heading as link that changes order --->
<a href="#sortLink#" class="Head">#col#</a>
<!--- If this is current column, show icon to indicate current sort --->
<cfif SESSION.expenseReport.sortCol eq alias>
<cfset sortImgSrc = iif(SESSION.expenseReport.sortAsc, "'SortA.gif'",
 "'SortD.gif'")>
<img src="../images/#sortImgSrc#" width="12" height="12" alt="">
</cfif>
</th>
</cfloop>
</cfoutput>
</tr>

<!--- For each query row that should be shown now --->
<cfloop query="getExp" startRow="#URL.startRow#" endRow="#endRow#">
<!--- Use class "DataA" or "DataB" for alternate rows --->
<cfset class = iif(getExp.CurrentRow mod 2 eq 0, "'DataA'", "'DataB'")>

<cfoutput>
<tr valign="baseline">
```

Listing 41.12 (continued)

```
    <td class="#class#" width="100">#lsDateFormat(ExpenseDate)#</td>
    <td class="#class#" width="250">#MovieTitle#</td>
    <td class="#class#" width="150"><i>#Description#</i></td>
    <td class="#class#" width="100">#lsCurrencyFormat(ExpenseAmount)#</td>
    </tr>
    </cfoutput>
    </cfloop>

    <!--- Row at bottom of table, after rows of data --->
    <tr>
    <td width="500" colspan="3">
    <cfif not URL.showAll AND totalRows gt rowsPerPage>
    <!--- Shortcut links for "Pages" of search results --->
    Page <cfinclude template="NextNIncludePageLinks.cfm">
    <!--- Show All link --->
    <cfoutput>
    <a href="#CGI.script_name#?&showAll=Yes">Show All</a>
    </cfoutput>
    </cfif>
    </td>
    <td width="100" align="right">
    <cfif not URL.showAll>
    <!--- Provide Next/Back links --->
    <cfinclude template="NextNIncludeBackNext.cfm">
    </cfif>
    </td>
    </tr>
   </table>

   </body>
   </html>
```

Most of the code in this template is identical to the versions presented in Chapters 29 and 31. The most important additions are discussed here.

First, two `<cfparam>` tags have been added, which define `SESSION.expenseReport.sortCol` and `SESSION.expenseReport.sortAsc` variables to track the user's current sort column and sort direction. The `nextNSortCol` value will be `d`, `f`, `e`, or `a` to signify the `ExpenseDate`, `MovieTitle`, expense `Description`, and `ExpenseAmount` columns, respectively. The Boolean `nextNSortAsc` value will be `True` if the column is being sorted in ascending order, and will be `False` if it is being sorted in descending order.

A `<cfif>` block at the top of the template changes the sort order and sort direction for the user's session if URL parameters called `sortCol` and `sortDir` are given. If provided, the values provided in the URL are copied to the two new `SESSION` variables, which causes the template to remember to show the records in the new sort order for the rest of the session. In addition, the `startRow` variable is set to `1`, so the user is always shown the first page of records whenever the sort is changed.

After the cached `GetExp` query, a second `<cfquery>` tag that uses the query of queries feature is used to re-sort the cached query results, depending on the current values of the two `SESSION` variables. For instance, if `SESSION.expenseReport.sortCol` is `f` and `SESSION.expenseReport.sortAsc` is `True`, the resulting `ORDER BY` statement would be `MovieTitle ASC`, thereby providing the user with the

desired sorting effect. Simple `<cfif>` logic skips this requerying step if the user is viewing the template with the default sort options.

The rest of the code is largely unchanged from prior versions. The only other major change is in the middle of the template, where the column headings are displayed.

A `<cfloop>` is used to output the four columns. For each column, the `alias` variable is set to the first letter of the column name; this is passed in the URL when the user clicks a column heading. Then, a `sortDir` variable is set to `True` or `False`, depending on whether the results are already being sorted by that column. The `sortDir` determines the direction in which the records should be sorted if the user clicks the column heading. For instance, for the `Date` column, if the user is already viewing the records by date, the `sortDir` is the opposite of the current `SESSION.expenseReport.sortAsc` value. If the user is currently viewing records by some other column, the `sortDir` is `NO`, meaning the sort direction is changed to a descending sort if the user clicks the `Date` column.

The column name is then displayed as a link, passing the appropriate values as URL parameters called `sortCol` and `sortDir`. Plus, if the column heading being output corresponds with the current sort order, a small triangle icon is displayed (`SortA.gif` or `SortD.gif`), depending on whether the column is currently sorted in ascending or descending order, respectively. The results are shown in Figure 41.10.

Figure 41.10

When users click a column header, the results are re-sorted using a QofQ query.

NOTE

In this example, the query of queries feature is used to reorder the original, cached result set. The same basic technique could be used for other types of requerying needs. For instance, the second QofQ query might retrieve a subset of the original records, using some type of `WHERE` filter criteria.

Using QofQ with Nondatabase Queries

The query of queries feature can be used on any ColdFusion query object. Most commonly, this is the result of a `<cfquery>` tag that retrieves records from a database. However, a number of other CFML tags return information as ColdFusion query objects, which means they can be requeried using QofQ.

Table 41.3 provides a short list of tags that return query objects.

Table 41.3 CFML Tags That Return Query Objects, Which Can Be Requeried Using QofQ

CFML TAG	RETURNS QUERY OBJECT THAT CONTAINS
`<cfdirectory>`	A listing of files in a particular directory (when used with `action="List"`). See Chapter 70, "Interacting with the Operating System," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 3: Advanced Application Development.*
`<cfftp>`	A listing of files on a remote FTP server (when used with `action="ListDir"`).
`<cfldap>`	Directory listings retrieved from a remote LDAP server (when used with `action="Query"`).
`<cfpop>`	Incoming mail messages, as retrieved from a remote email mailbox (when used with `action="GetHeaderOnly"` or `action="GetAll"`). See Chapter 21, "Interacting with Email," online.
`<cfquery>`	Any database query.
`<cfsearch>`	Search results found by a Verity full-text search operation. See Chapter 39, "Full-Text Searching," in Vol. 2, *Application Development.*
`<cffeed>`	RSS entries from a remote RSS feed. See Chapter 69, "Working with Feeds," in Vol. 3, *Advanced Application Development*.
`<cfprocresult>`	A result set returned by a stored procedure in a database. See Chapter 42, "Working with Stored Procedures," online.
Custom Tags	CFML custom tags can use the `queryNew()` function to create and return query objects, filled with rows and columns of whatever data is appropriate. For more about CFML custom tags, see Chapter 26, "Building Reusable Components," in Vol. 2, *Application Development.*
User-Defined Functions	UDFs can query objects by specifying `returnType="query"` in a `<cffunction>` tag. For details, see Chapter 24, "Building User-Defined Functions," in Vol. 2, *Application Development.*
ColdFusion Components	CFC methods can also return query objects, also by specifying `returnType="query"` in the corresponding `<cffunction>` tag. For details, see Chapter 26.
CFX Tags	Many CFX tags return query objects, filled with whatever specialized data is appropriate. For information about writing your own CFX tags (using Java or C++), see ColdFusion Web Application Construction Kit—Volume 3.

For instance, Listing 41.13 shows how ColdFusion's query of queries feature can be used to combine result sets from two different query objects. Here, QofQ is used to get around a limitation of the `<cfdirectory>` tag. It's a good example of how QofQ can add flexibility and power to CFML tags that return query objects (refer to Table 41.3).

The purpose of this listing is to display all the GIF and JPEG images in a particular directory. As will be explained in Chapter 70, "Interacting with the Operating System," in Vol. 3, *Advanced Application Development*, the `<cfdirectory>` tag can be used to easily obtain a list of files, and it even provides a `filter` attribute that enables you to filter the files based on DOS-style wildcards. For instance, a `filter` of `*.jpg` returns all JPEG files in a directory, and a `filter` of `*.gif` lists all the GIF files. Unfortunately, `<cfdirectory>` doesn't allow you to provide a single filter that returns all GIF and JPEG files together as a single, sorted query object.

Listing 41.13 overcomes this limitation by using separate `<cfdirectory>` tags to retrieve the GIF and JPEG directory listings. Then, the query of queries feature is used to combine the two result sets as a single query object, sorted by file name.

Listing 41.13 MultiDirectory1.cfm—Results of Two `<cfdirectory>` Tags with UNION

```
<!---
 Filename: MultiDirectory1.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Displays a list of all GIF and JPEG files in images folder
--->

<!--- Directory to scan for images --->
<cfset dir = expandPath("../images")>

<!--- First, get query object of JPEG files --->
<cfdirectory action="list" directory="#dir#" filter="*.jpg" name="getJPG">

<!--- Next, get query object of GIF files --->
<cfdirectory action="list" directory="#dir#" filter="*.gif" name="getGIF">

<!--- Use CF's Query-of-Queries feature to --->
<!--- combine the queries with a SQL UNION --->
<!--- Sort the resulting query by filename --->
<cfquery dbtype="query" name="getAll">
 SELECT * FROM getJPG
 UNION
 SELECT * FROM getGIF
 ORDER BY Name
</cfquery>


<html>
<head><title>Images Folder</title></head>
<body>
<h2>Images Folder</h2>

<!--- Display all files together as a list --->
<cfoutput>
 <p>There are #getAll.recordCount# images in #dir#:<br>
```

Listing 41.13 (CONTINUED)

```
     <cfloop query="getAll">
     <p>
     #currentRow#. #getAll.Name#<br>
     <img src="../images/#Name#"><br>
     </p>
     </cfloop>
    </cfoutput>

    </body>
    </html>
```

First, the `expandPath()` function is used to get the absolute file system path of Orange Whip Studios' `images` folder. It is assumed that the `images` folder and the current folder are both contained within the same parent folder.

Next, two `<cfdirectory>` tags are used to retrieve listings of all JPEG and GIF files in the `images` folder. The query objects returned by the two tags are named `getJPG` and `getGIF`, respectively.

Now the two query objects can be combined in the QofQ query named `getAll`. The SQL syntax used here is extremely simple. All that's needed is a standard `UNION` statement that retrieves all records from both source queries and sorts the combined result set by file name (see the section "Combining Record Sets with `UNION`," earlier in this chapter). The result is a single ColdFusion query object that can be used just like any other. It contains a row for each GIF and JPEG file. The images and file names are then displayed using a standard `<cfloop>` block that loops over the `getAll` query, as shown in Figure 41.11.

Figure 41.11

The query of queries feature makes obtaining flexible directory listings easier.

You can wrap such QofQ operations in CFML custom tags. Listing 41.14 takes the code from Listing 41.13 and turns it into a custom tag called `<cf_GetDirectoryContents>`. The custom tag takes `directory`, `filter`, `sort`, and `name` attributes, which correspond to the same attributes of ColdFusion's native `<cfdirectory>` tag. The advantage of this custom tag is that its `filter` attribute accepts multiple wildcard filters. The individual filters are separated with semicolons, like so:

```
<cf_GetDirectoryContents
 directory="c:\ inetpub\ wwwroot\ ows\ images\ "
 filter="*.gif;*.jpg"
 name="getAll">
```

NOTE

Because this is a custom tag template, it should be saved in the special `CustomTags` folder. As an alternative, you can just save a copy of it in the folder where you intend to use it (the same folder you are using for the other code listings in this chapter).

Listing 41.14 `GetDirectoryContents.cfm`—Creating `<cf_GetDirectoryContents>` Custom Tag

```
<!---
 Filename: GetDirectoryContents.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Creates the <CF_GetDirectoryContents> custom tag
 Please Note This example uses Query-of-Queries to combine query objects
--->

<!--- Tag Parameters --->
<cfparam name="ATTRIBUTES.directory">
<cfparam name="ATTRIBUTES.filter" type="string" default="*.*">
<cfparam name="ATTRIBUTES.sort" type="string" default="name">
<cfparam name="ATTRIBUTES.name" type="variableName">

<!--- For each filter (filters separated by semicolons) --->
<cfloop list="#ATTRIBUTES.filter#" index="thisFilter" delimiters=";">

 <!--- Get query object of matching files --->
 <cfdirectory action="list" directory="#ATTRIBUTES.directory#"
 filter="#thisFilter#" name="getFiles">

 <!--- If this is first time through loop, --->
 <!--- Save GetFiles as our "ResultQuery". --->
 <cfif not isDefined("resultQuery")>
   <cfset resultQuery = getFiles>
 <!--- If ResultQuery already exists, add --->
 <!--- GetFiles's records to it via UNION --->
 <cfelse>
   <cfquery dbtype="query" name="resultQuery">
   SELECT * FROM ResultQuery
   UNION
   SELECT * FROM GetFiles
   ORDER BY #ATTRIBUTES.sort#
   </cfquery>
 </cfif>

</cfloop>
```

Listing 41.14 (continued)

```
<!--- Return completed resultset to calling template --->

<cfset "Caller.#ATTRIBUTES.name#" = resultQuery>
```

First, four `<cfparam>` tags are used to establish the tag's parameters. Then, a `<cfloop>` tag is used to loop through the semicolon-delimited list provided to the tag's `filter` attribute. If the `filter` is provided as `*.gif;*.jpg`, the loop runs twice. The first time through the loop, `thisFilter` is `*.gif`, and so on.

Within the loop, the `<cfdirectory>` tag called `getFiles` is used to retrieve the files in the directory that correspond to the current `thisFilter` value. Then, if this is the first time through the loop, the `getFiles` object is set to a new variable named `resultQuery`, which is what gets passed back to the calling template when the tag is finished with its work. The next time through the loop, the results of the `getFiles` query are added to the `resultsQuery`, using the `UNION` technique from Listing 41.13.

When the loop is finished, the `resultsQuery` variable contains the complete file listing. It is returned to the calling template using quoted `<cfset>` syntax. See Chapter 25 for details. Listing 41.15 shows how this custom tag can be used in a normal ColdFusion template. It is the same as Listing 41.13, except that the `<cfdirectory>` and `<cfquery>` tags have been replaced with the `<cf_GetDirectoryContents>` custom tag. Refer to Figure 41.11 to see what the results will look like.

Listing 41.15 `MultiDirectory2.cfm`—Executing a QofQ Operation

```
<!---
 Filename: MultiDirectory2.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Displays a list of all GIF and JPEG files in images folder
--->

<!--- Directory to scan for images --->
<cfset dir = expandPath("../images")>

<!--- Get GIF and JPG file listing, via custom tag --->
<cf_GetDirectoryContents directory="#Dir#" filter="*.gif;*.jpg" name="getAll">

<html>
<head><title>Images Folder</title></head>
<body>
<h2>Images Folder</h2>

<!--- Display all files together as a list --->
<cfoutput>
 <p>There are #getAll.recordCount# images in #dir#:<br>

 <cfloop query="getAll">
 <p>
 #currentRow#. #getAll.Name#<br>
 <img src="../images/#Name#"><br>
 </p>
 </cfloop>
</cfoutput>

</body>
</html>
```

Joining Database and Nondatabase Queries

You have seen how ColdFusion's QofQ feature can be used to join or combine query objects returned by database queries or nondatabase tags, such as `<cfdirectory>`. You also can use QofQ to merge the results of database queries with nondatabase query objects, using `UNION` or join syntax.

Listing 41.16 is another revision of the `StoreCart.cfm` templates created in Chapter 22. Similar to the version in Listing 41.10, this template displays suggestions to the user using a pseudo-collaborative filtering technique. In addition, this version displays a subtotal for each item in the user's cart, along with a grand total of all the items. Versions provided up to this point did not display totals.

??A0Because this listing relies on other templates from Chapter 22, you should save this template in the same folder you use for Chapter 22's listings.

Listing 41.16 `StoreCart4.cfm`—Using Several QofQ Queries Together

```
<!---
 Filename: StoreCart.cfm (save as StoreCart.cfm)
 Created by: Nate Weiss (NMW)
 Purpose: Displays the current user's shopping cart
 Please Note Depends on the <CF_ShoppingCart> custom tag
--->

<!--- Show header images, etc., for Online Store --->
<cfinclude template="StoreHeader.cfm">

<!--- If MerchID was passed in URL --->
<cfif isDefined("URL.addMerchID")>
  <!--- Add item to user's cart data, via custom tag --->
  <cf_ShoppingCart
  action="Add"
  merchID="#URL.addMerchID#">

<!--- If user is submitting cart form --->
<cfelseif isDefined("FORM.merchID")>
  <!--- For each MerchID on Form, Update Quantity --->
  <cfloop list="#FORM.merchID#" index="thisMerchID">
    <!--- Update Quantity, via Custom Tag --->
    <cf_ShoppingCart
    action="Update"
    merchID="#thisMerchID#"
    quantity="#FORM['Quant_#thisMerchID#']#">
  </cfloop>

  <!--- If user submitted form via "Checkout" button, --->
  <!--- send on to Checkout page after updating cart. --->
  <cfif isDefined("FORM.isCheckingOut")>
    <cflocation url="../41/StoreCheckout.cfm">
  </cfif>
</cfif>

<!--- Get current cart contents, via Custom Tag --->
<cf_ShoppingCart
 action="List"
```

Listing 41.16 (continued)

```
 returnVariable="getCart">


<!--- Stop here if user's cart is empty --->
<cfif getCart.recordCount eq 0>
 There is nothing in your cart.
 <cfabort>
</cfif>


<!--- Retrieve items in user's cart from database --->
<cfquery name="getMerch" datasource="ows">
 SELECT MerchID, MerchPrice
 FROM Merchandise
 WHERE MerchID IN (#valueList(getCart.MerchID)#)
</cfquery>

<!--- Use QofQ to join queried records against cart data --->
<!--- The SubTotal column is Quantity times MerchPrice --->
<cfquery name="getPrices" dbtype="query">
 SELECT
 GetMerch.MerchID AS MerchID,
 GetCart.Quantity AS Quantity,
 (GetCart.Quantity * GetMerch.MerchPrice) AS SubTotal
 FROM GetMerch, GetCart
 WHERE GetMerch.MerchID = GetCart.MerchID
</cfquery>

<!--- Use QofQ again to get a grand total of all items --->
<cfquery name="getTotal" dbtype="query">
 SELECT SUM(SubTotal) AS GrandTotal
 FROM GetPrices
</cfquery>

<!--- Create form that submits to this template --->
<cfform action="#CGI.script_name#">
 <table>
 <tr>
 <th colspan="2" bgcolor="silver">Your Shopping Cart</th>
 </tr>
 <!--- For each piece of merchandise --->
 <cfloop query="getPrices">
 <tr>
 <td>
 <!--- Show this piece of merchandise --->
 <cf_MerchDisplay
 merchID="#getPrices.MerchID#"
 showAddLink="No">
 </td>
 <td>
 <!--- Display Quantity in Text entry field --->
 <cfoutput>
 Quantity:
 <cfinput type="Hidden" name="merchID"
```

Listing 41.16 (continued)

```
 value="#GetPrices.MerchID#">
 <cfinput type="Text" size="3"
 name="Quant_#getPrices.MerchID#"
 value="#getPrices.Quantity#"><br>
 Subtotal: #lsCurrencyFormat(subTotal)#
 </cfoutput>
 </td>
 </tr>
 </cfloop>

 <tr>
 <td></td>
 <td>
 <!--- Display Grand Total for all items in cart --->
 <cfoutput>
 <b>Total: #lsCurrencyFormat(getTotal.GrandTotal)#</b>
 </cfoutput>
 </td>
 </tr>
 </table>

 <!--- Submit button to update quantities --->
 <cfinput type="submit" name="submit" value="Update Quantities">

 <!--- Submit button to Check out --->
 <cfinput type="submit" value="Checkout" name="isCheckingOut">
</cfform>



<!--- Convert current cart contents to comma-sep list --->
<cfset currentMerchList = valueList(getCart.MerchID)>

<!--- Run query to suggest other items for user to buy --->
<cfquery name="getSimilar" datasource="#APPLICATION.dataSource#"
 cachedWithin="#createTimeSpan(0,0,5,0)#" maxrows="3">
 <!--- We want all items NOT in user's cart... --->
 SELECT ItemID
 FROM MerchandiseOrdersItems
 WHERE ItemID NOT IN (#currentMerchList#)
 <!--- ...but that *were* included in other orders --->
 <!--- along with items now in the user's cart... --->
 AND OrderID IN
 (SELECT OrderID FROM MerchandiseOrdersItems
 WHERE ItemID IN (#currentMerchList#)
 <!--- ...not including this user's past orders! --->
 <cfif isDefined("SESSION.auth.contactID")>
 AND OrderID NOT IN
 (SELECT OrderID FROM MerchandiseOrders
 WHERE ContactID = #SESSION.auth.contactID#)
 </cfif> )
</cfquery>

<!--- If at least one "similar" item was found --->
<cfif getSimilar.recordCount GT 0>
```

Listing 41.16 (CONTINUED)

```
     <p>People who have purchased items in your
     cart have also bought the following:<br>

     <!--- For each similar item, display it, via Custom Tag --->
     <!--- (show five suggestions at most) --->
     <cfloop query="getSimilar">
       <cf_MerchDisplay merchID="#getSimilar.ItemID#">
     </cfloop>
    </cfif>
```

Listing 41.16 is identical to Listing 41.10 in most respects. The important additions are the `getMerch`, `getPrices`, and `getTotal` prices near the middle of the template.

First, for each item in the user's cart the `getMerch` query retrieves the `MerchID` and `MerchPrice` from the database. This works because the `valueList` function returns a comma-separated list of values in the `MerchID` column of the `getCart` query object that was returned by the `<cf_ShoppingCart>` custom tag. Therefore, the `IN` criteria causes the database to return just the records that are actually in the user's cart at the moment.

Next, a QofQ query named `getPrices` is used to calculate the subtotals for each item in the user's cart by joining the `MerchID` columns from the `getMerch` query, which came from a database, and the `getCart` query, which did not. The resulting query contains three columns for each item in the cart: `MerchID`, `Quantity`, and `SubTotal`.

Finally, a second QofQ query called `getTotal` calculates the grand total of all items in the user's cart by applying the aggregate `SUM()` function against the `SubTotal` column from the `getPrices` query. The result is a query with one value, `GrandTotal`, which represents the total value of all cart items. This proves that you can use QofQ to retrieve information from query objects that were themselves created with QofQ.

Now outputting the rest of the page, including the subtotals and totals, is a simple matter. Note that the `<cfloop>` that displays each item now iterates over the `getPrices` query created by the first QofQ. The results are shown in Figure 41.12.

Case Sensitivity and Query of Queries

Most database systems consider `WHERE` conditions without regard to upper- or lowercase. However, `WHERE` conditions are case sensitive when using QofQ. For instance, consider the following QofQ query:

```
<cfquery dbtype="query" name="getThisFilm">
 SELECT * FROM GetFilms
 WHERE MovieTitle LIKE '%#FORM.searchString#%'
</cfquery>
```

This snippet will return all films with titles that include the words provided by the user in the form field called `#searchString#`. However, the search will be case sensitive; if the user types `encounters` as the search criteria, movies with *Encounters* in the title won't be found (because of the capital *E*). In most situations, you probably don't want case-sensitive behavior from QofQ.

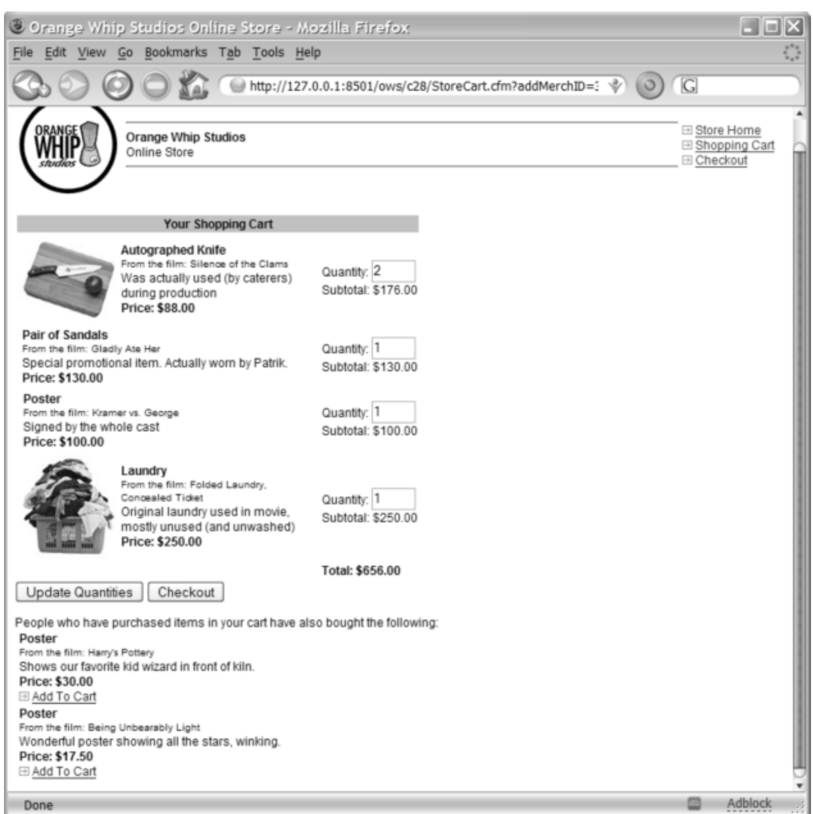


Figure 41.12

Using ColdFusion's query of queries feature to `SUM` data that lives both inside and outside a database.

QofQ provides functions called `UPPER()` and `LOWER()`, which can be used to subvert the default case-sensitive behavior. Just use `UPPER()` or `LOWER()` around the column name that you want to query, and then use CFML's `uCase()` or `lCase()`, respectively, around the actual query criteria. (Whether you use the combination of `UPPER()` and `uCase()` or the combination of `LOWER()` and `lCase()` doesn't matter.) So, to make the previous snippet behave as expected, you would do the following:

```
<cfquery dbtype="query" name="GetThisFilm">
 SELECT * FROM GetFilms
 WHERE LOWER(MovieTitle) LIKE '%#lCase(FORM.searchString)#%'
</cfquery>
```

Now, when ColdFusion examines each row of the source query, it compares the lowercase version of each title with the lowercase version of the user's search criteria. This results in the expected, non-case-sensitive behavior.

Working with Column Types

When you perform a query against a database, the database table has explicit definitions for what each column contains. One column may only contain numbers, or dates, or binary data. There are cases, however, when you need to treat the columns as other types. Query of query will examine the

query to see if any information exists for the existing data. If you performed a query against a database, this information exists in the metadata, and can now be examined by using the `getMetaData` function:

```
<cfset queryInformation = getMetaData(someQuery)>
```

This function will return an array for each column in the query. Each item in the array is a structure containing information on the columns name, it's type, and if it is case sensitive or not.

As we said, when working with a query from a database, it is easy to determine each column type. However, queries created by the `queryNew` function do not automatically have meta information about the columns. As another example, if you use the `<cfhttp>` tag to retrieve a remote URL automatically into a query, all the columns will be marked as a `VARCHAR` column. Query of query will attempt to guess what a column is if no metadata exists. It does this by parsing the first 50 rows of data and making a best guess. You can also provide metadata for a query when using `queryNew()` by supplying a second argument that contains a list of column types. As an example, this line:

```
<cfset x = queryNew("data")>
```

Could become more specific like so:

```
<cfset x = queryNew("data","varchar")>
```

So what do you when a column is specified as a type but you want to treat it as something else? Or perhaps the query of query "guess" is incorrect? You can now tell query of query to specifically treat a column as a certain type. This is done using the CAST function. This lets you convert a column from it's native type into one of the following types: BINARY, BIGINT, BIT, DATE, DECIMAL, DOUBLE, INTEGER, TIME, TIMESTAMP, VARCHAR. Listing 30.17 shows a simple example of how CAST can be used.

Listing 41.17 `QoQCast.cfm`—Using Cast in Query of Query

```
<!---
 Filename: qofqcast.cfm
 Created by: Raymond Camden (ray@camdenfamily.com)
 Purpose: Demonstrates using the cast function.
--->

<cfset x = queryNew("data","varchar")>
<cfset queryAddRow(x)>
<cfset querySetCell(x,"data",200)>
<cfset queryAddRow(x)>
<cfset querySetCell(x,"data",300)>
<cfset queryAddRow(x)>
<cfset querySetCell(x,"data",400)>
<cfset queryAddRow(x)>
<cfset querySetCell(x,"data",100)>

<cfquery name="test" dbtype="query">
  SELECT SUM(CAST(data AS INTEGER)) AS total
  FROM x
</cfquery>

<cfoutput>The sum was #test.total#</cfoutput>
```

This listing begins with the `queryNew()` function. The first attribute defines a list of columns. In our example, we only have one column, data. The second attribute defines a list of column types. The `varchar` type simply means text data. We then created a set of new rows and set the data to numerical data. Since numbers can be treated like string data, this doesn't throw an error.

Next we have a query of query. As we showed before, we can use the aggregate function sum to find the total of a column, but since we specified the column was using string data, this will throw an error. By using the CAST function, however, we tell ColdFusion's query of query parser to treat the column as a numerical column. Since the data was all numbers, this conversion works just fine.

Reserved Words and Query of Queries

ColdFusion's QofQ feature defines a list of reserved words that can't be used as column or table names without taking special steps. For instance, QofQ considers `BEGIN` a reserved word. If you have a query object that you want to requery with QofQ, and one of its columns is named `BEGIN`, you will receive an error message unless you escape the column name with square brackets.

For instance, instead of:

```
<cfquery dbtype="query" name="getThisFilm">
 SELECT Name, Begin
 FROM getFilms
 WHERE Begin > 25
</cfquery>
```

you would need to use this:

```
<cfquery dbtype="query" name="getThisFilm">
 SELECT Name, [Begin]
 FROM getFilms
 WHERE [Begin] > 25
</cfquery>
```

The following are the reserved words around which you need to use square brackets within a QofQ query: `ABSOLUTE`, `ACTION`, `ADD`, `ALL`, `ALLOCATE`, `ALTER`, `AND`, `ANY`, `ARE`, `AS`, `ASC`, `ASSERTION`, `AT`, `AUTHORIZATION`, `AVG`, `BEGIN`, `BETWEEN`, `BIT`, `BIT_LENGTH`, `BOTH`, `BY`, `CASCADE`, `CASCADED`, `CASE`, `CAST`, `CATALOG`, `CHAR`, `CHAR_LENGTH`, `CHARACTER`, `CHARACTER_LENGTH`, `CHECK`, `CLOSE`, `COALESCE`, `COLLATE`, `COLLATION`, `COLUMN`, `COMMIT`, `CONNECT`, `CONNECTION`, `CONSTRAINT`, `CONSTRAINTS`, `CONTINUE`, `CONVERT`, `CORRESPONDING`, `COUNT`, `CREATE`, `CROSS`, `CURRENT`, `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `CURRENT_USER`, `CURSOR`, `DATE`, `DAY`, `DEALLOCATE`, `DEC`, `DECIMAL`, `DECLARE`, `DEFAULT`, `DEFERRABLE`, `DEFERRED`, `DELETE`, `DESC`, `DESCRIBE`, `DESCRIPTOR`, `DIAGNOSTICS`, `DISCONNECT`, `DISTINCT`, `DOMAIN`, `DOUBLE`, `DROP`, `ELSE`, `END`, `END-EXEC`, `ESCAPE`, `EXCEPT`, `EXCEPTION`, `EXEC`, `EXECUTE`, `EXISTS`, `EXTERNAL`, `EXTRACT`, `FALSE`, `FETCH`, `FIRST`, `FLOAT`, `FOR`, `FOREIGN`, `FOUND`, `FROM`, `FULL`, `GET`, `GLOBAL`, `GO`, `GOTO`, `GRANT`, `GROUP`, `HAVING`, `HOUR`, `IDENTITY`, `IMMEDIATE`, `IN`, `INDICATOR`, `INITIALLY`, `INNER`, `INPUT`, `INSENSITIVE`, `INSERT`, `INT`, `INTEGER`, `INTERSECT`, `INTERVAL`, `INTO`, `IS`, `ISOLATION`, `JOIN`, `KEY`, `LANGUAGE`, `LAST`, `LEADING`, `LEFT`, `LEVEL`, `LIKE`, `LOCAL`, `LOWER`, `MATCH`, `MAX`, `MIN`, `MINUTE`, `MODULE`, `MONTH`, `NAMES`, `NATIONAL`, `NATURAL`, `NCHAR`, `NEXT`, `NO`, `NOT`, `NULL`, `NULLIF`, `NUMERIC`, `OCTET_LENGTH`, `OF`, `ON`, `ONLY`, `OPEN`, `OPTION`, `OR`, `ORDER`, `OUTER`, `OUTPUT`, `OVERLAPS`, `PAD`, `PARTIAL`, `POSITION`, `PRECISION`, `PREPARE`, `PRESERVE`, `PRIMARY`, `PRIOR`, `PRIVILEGES`, `PROCEDURE`, `PUBLIC`, `READ`, `REAL`, `REFERENCES`, `RELATIVE`, `RESTRICT`, `REVOKE`, `RIGHT`, `ROLLBACK`, `ROWS`, `SCHEMA`, `SCROLL`, `SECOND`, `SECTION`,

SELECT, SESSION, SESSION_USER, SET, SMALLINT, SOME, SPACE, SQL, SQLCODE, SQLERROR, SQLSTATE, SUBSTRING, SUM, SYSTEM_USER, TABLE, TEMPORARY, THEN, TIME, TIMESTAMP, TIMEZONE_HOUR, TIMEZONE_MINUTE, TO, TRAILING, TRANSACTION, TRANSLATE, TRANSLATION, TRIM, TRUE, UNION, UNIQUE, UNKNOWN, UPDATE, UPPER, USAGE, USER, USING, VALUE, VALUES, VARCHAR, VARYING, VIEW, WHEN, WHENEVER, WHERE, WITH, WORK, WRITE, YEAR, ZONE.

Parameterized Queries

ColdFusion enables you to create parameterized queries by placing `<cfqueryparam>` tags within the SQL code you supply to `<cfquery>`. The value of each `<cfqueryparam>` tag is sent to the database.

Behind the scenes, as ColdFusion sends your query to the database, it substitutes a SQL bind parameter for each `<cfqueryparam>` tag you supply. Unfortunately, a proper definition of a SQL bind parameter would be beyond the scope of this book. In a nutshell, bind parameters enable a database client (here, ColdFusion) to send a SQL statement to the database with placeholders in it, followed by the actual values for each placeholder. This method is a bit more formalized than the usual method of sending completed, ad hoc query statements to the database.

NOTE

Most database systems support bind parameters. If the database you are using doesn't, ColdFusion simply inserts the correct value into your query for you.

Theoretically, depending on the database system, this could result in slightly faster performance, because the system might be capable of compiling a parameterized query for later reuse. If your knowledge of your database system is such that you believe the use of SQL bind parameters would result in a performance boost, `<cfqueryparam>` is the way to implement them in your ColdFusion applications. In practice, you won't see much of a performance boost.

There are other reasons to use parameterized queries in your templates, though. In some cases, the tag can prevent a certain type of hack and can make your query code more database independent. This section introduces the `<cfqueryparam>` tag and explains how to use it in your code and why.

Introducing <cfqueryparam>

At its simplest, you can parameterize a query by adding a `<cfqueryparam>` tag to a query wherever you would normally provide a variable name. For instance, suppose you want to run a query called `getFilm`, which retrieves a record from the `Films` table, based on a `FilmID` value passed in the URL.

Normally, you would do this:

```
<cfquery name="getFilm" datasource="ows">
 SELECT * FROM Films
 WHERE FilmID = #URL.filmID#
</cfquery>
```

Instead, you could do the following, which supplies the dynamic part of the query (the value of `URL.filmID`) as a SQL bind parameter:

```
<cfquery name="getFilm" datasource="ows">
```

```
 SELECT * FROM Films
 WHERE FilmID = <cfqueryparam value="#URL.filmID#">
</cfquery>
```

In general, you should provide ColdFusion with the data type of the corresponding column in your database via the `cfsqltype` attribute. This lets ColdFusion send the data to the database system correctly (rather than relying on the database to convert the parameter on the fly), which reduces overhead. Because the `FilmID` column contains integers, you would specify a `cfsqltype` of `CF_SQL_INTEGER`:

```
<cfquery name="getFilm" datasource="ows">
 SELECT * FROM Films
 WHERE FilmID = <cfqueryparam value="#URL.filmID#" cfsqltype="CF_SQL_INTEGER">
</cfquery>
```

In most situations, you can use `<cfqueryparam>` with just the `value` and `cfsqltype` attributes, as shown previously. A number of other attributes can be used to deal with special situations, as listed in Table 41.4. Table 41.5 shows the values you can supply to the `cfsqltype` attribute and helps you understand which `cfsqltype` to use for corresponding column data types in Access, SQL Server, and Oracle databases.

Table 41.4 Attributes for the `<cfqueryparam>` Tag

ATTRIBUTE	PURPOSE
`value`	Required. The value you want ColdFusion to send to your database system in place of the `<cfqueryparam>` tag.
`cfsqltype`	Optional. One of the `cfsqltype` values listed in Table 30.6. If you don't provide this attribute, the parameter is treated as a `CFSQLCHAR`.
`maxLength`	Optional. The maximum number of characters to allow for the `value`. If you provide this attribute, it should reflect the maximum number of characters (width) allowed by the corresponding column in your database.
`scale`	Optional. The number of decimal places to allow. If you provide this attribute, it should reflect the numeric scale defined for the corresponding column in your database (see your database documentation about the concepts of scale and precision for numeric columns). Applicable only if the `cfsqltype` is `CF_SQL_NUMERIC` or `CF_SQL_DECIMAL`.
`list`	Optional. If set to `Yes`, indicates that the value you are supplying to `VALUE` should be treated as a comma-separated list of values. Typically, you use this if the `<cfqueryparam>` tag is being placed between the parentheses of SQL's `IN` keyword. ColdFusion takes care of quoting each element in the list for you, if appropriate for the data type of the column. If you want to use a different delimiter for the list (instead of a comma), provide the delimiter as the `separator` attribute (see the following).
`separator`	The delimiter character to use to separate the `VALUE` into separate values. Relevant only if `list="Yes"`. The default is a comma.
`null`	Optional. Set this attribute to `Yes` to send a null value to the database, instead of the `value`. The default, of course, is `No`.

Table 41.5 Which CFSQLTYPE to Use with Which Native Data Type

CFSQLTYPE	USE WITH ACCESS	USE WITH SQL SERVER	USE WITH ORACLE
CF_SQL_BIGINT		bigint	
CF_SQL_BIT	Yes/No	bit	
CF_SQL_CHAR		char, nchar	CHAR, NCHAR
CF_SQL_DATE			
CF_SQL_DECIMAL		numeric, decimal	
CF_SQL_DOUBLE		double, float	
CF_SQL_FLOAT		double, float	
CF_SQL_IDSTAMP		timestamp	
CF_SQL_INTEGER	AutoNumber	int	
CF_SQL_LONGVARCHAR	Memo	text	LONG, CLOB, NCLOB
CF_SQL_MONEY	Currency	money	
CF_SQL_MONEY4		smallmoney	
CF_SQL_NUMERIC	Number	numeric, decimal	NUMBER
CF_SQL_REAL		real	
CF_SQL_REFCURSOR		cursor	
CF_SQL_SMALLINT		smallint	
CF_SQL_TIME			
CF_SQL_TIMESTAMP	Date/Time	datetime,	DATE_smalldatetime
CF_SQL_TINYINT		tinyint	
CF_SQL_VARCHAR	Text	varchar, nvarchar, uniqueidentifier	VARCHAR2, NVAR CHAR2, VARCHAR

NOTE

ColdFusion also supports two additional SQL types: CF_SQL_BLOB and CF_SQL_CLOB. See the ColdFusion documentation for details.

Using Parameterized Queries for Database Independence

In the section "Specifying Dates in <cfquery> Tags," earlier in this chapter, you learned how the <cfqueryparam> tag can be used to avoid having to provide dates in the format required for the particular database system you are using. This approach can be especially helpful if you are creating custom tags or complete ColdFusion applications that need to interact with different types of databases. As long as your queries use standard SQL statements, and as long as tricky data types such as dates are handled with <cfqueryparam>, you can feel reasonably confident that the queries will work on just about any database ColdFusion encounters.

Using Parameterized Queries for Security

Parameterized queries can also help you prevent a certain type of database hack. The hack depends on the fact that many database systems enable you to execute multiple SQL statements in the same `<cfquery>` tag. If your queries are being built dynamically, using information being passed via URL or form parameters, your database could be subject to this form of attack.

For instance, take another look at this unparameterized query:

```
<cfquery name="getFilm" datasource="ows">
 SELECT * FROM Films
 WHERE FilmID = #URL.filmID#
</cfquery>
```

Normally, you would expect that the value of `URL.filmID` to be a number. But what if it's not? What if the user changes the URL parameter from `2`, say, to a value that includes actual SQL code? For instance, consider what would happen if some pesky user changed a template's URL from this:

```
ShowFilm.cfm?filmID=2
```

to this:

```
ShowFilm.cfm?filmID=2%3BDELETE%20FROM%20Films
```

You guessed it. The actual SQL sent to the database would be:

```
SELECT * FROM Films
WHERE FilmID = 2;DELETE FROM Expenses
```

Most database systems would consider the semicolon to indicate the start of a new SQL statement, and would thus dutifully carry out the `DELETE` statement, removing all records from your `Expenses` table. Try explaining this one to your colleagues in the accounting department!

NOTE

This isn't a bug or security hole in ColdFusion itself, or in your database or database driver. Both ColdFusion and your database are only doing what they are being told to do.

If you use the `<cfqueryparam>` tag in your query, specifying the appropriate data type with `CFSQLTYPE`, it becomes impossible for users to violate your database in this way because the `<cfqueryparam>` tag displays an error message if the URL parameter doesn't contain the expected type of information.

So, the unsafe, unparameterized query shown previously would become the following, which should defeat the type of attack described in this section:

```
<cfquery name="getFilm" datasource="ows">
 SELECT * FROM Films
 WHERE FilmID = <cfqueryparam value="#URL.filmID#" cfsqltype="CF_SQL_INTEGER">
</cfquery>
```

NOTE

There are other ways to deal with this problem. For instance, you could put a `<cfparam>` tag at the top of the template, with `name="URL.filmID"` and `type="numeric"`. This would prevent users from causing damage, because an error message would be generated by the `<cfparam>` tag if the value is not a simple number, stopping all further template execution. Another option is to use `#val(URL.filmID)#` instead of just `#URL.filmID#` in the original `<cfquery>` tag.

NOTE

Or you could just use a simple `<cfif>` test at the top of the template that uses the `isNumeric()` function to ensure that `URL.filmID` is valid; if not, you could skip the query or abort all processing with `<cfabort>`.

Building Query Results Programmatically

ColdFusion provides functions that enable you to create new query objects programmatically. These functions are called `queryNew()`, `queryAddRow()`, and `querySetCell()`. For instance, the `<cf_ShoppingCart>` custom tag created in Chapter 22 uses these functions to return a query object that represents the items currently in a user's shopping cart.

Furthermore, Listing 41.5 showed you how query results that are built using these functions can be requeried further, using ColdFusion's query of queries feature.

Using Database Transactions

ColdFusion provides a tag called `<cftransaction>`, which can be used to explicitly specify the beginning and end of a database transaction. A database transaction is a way of telling your database system that several SQL statements should be thought of as representing a single unit of work.

For instance, in the Orange Whip Studios project, the series of related inserts that need to occur to record a merchandise order should be thought of as a single transaction. An order really hasn't been properly recorded unless the appropriate records have been added to both the `MerchandiseOrders` and `MerchandiseOrdersItems` tables. During the moments between those inserts, the database is in what's called an *inconsistent state*, meaning that the data doesn't yet properly represent the real-world facts. Database transactions ensure that your database doesn't expose this inconsistent state to other connections.

Using `<cftransaction>` is simple:

- Whenever you need to make several related changes to your database, you should use a `<cftransaction>` tag around the `<cfquery>` tags that perform all the various steps. This tells your database system that the changes made by the queries should not be considered a permanent part of the database until the transaction has completed all its work.
- Until the transaction has finished its work, any changes made to the database during the transaction won't be visible to any other connections that might be querying the database at the same time. This means that queries made by other ColdFusion page requests will be incapable of interrupting your database transaction or catching the database in an inconsistent state (for instance, between the moments during which two related changes are made).
- Because changes made to the database during the transaction are not considered a permanent part of the database until the transaction is finished, you are free to undo all the changes at any time during the transaction. This is called rolling back the transaction and is supported by ColdFusion via `action="Rollback"` (see Table 41.6). Or, you can permanently commit the transaction using `action="Commit"`. After a transaction is committed, it can't be rolled back.

Table 41.6 shows the attributes supported by <cftransaction> in ColdFusion.

Table 41.6 <cftransaction> Tag Attributes

ATTRIBUTE	DESCRIPTION
action	Can be set to BEGIN, COMMIT, SETSAVEPOINT, or ROLLBACK. If you do not provide an ACTION, it defaults to BEGIN.
isolation	Can be set to Read_Uncommitted, Read_Committed, Repeatable_Read, or Serializable. These values allow you to control the appropriate balance between concurrency and consistency. Consult your database documentation to find out which isolation levels are actually supported by your database system and how your database vendor has chosen to implement the various isolation levels.
savepoint	The name of a savepoint. Allows you to roll back to a specific point in the transaction.

Database scholars generally define the concept of a database transaction as having a number of properties, often referred to as the so-called *ACID* properties (atomicity, consistency, isolation, and durability). If you're interested in the theory behind database transactions and how they are implemented in your database software, consult a dedicated SQL text and your database's documentation.

Using the <cftransaction> Tag

You have already seen <cftransaction> used in several of this book's listings. In general, our examples use <cftransaction> to explicitly mark the beginning and end of database transactions to accomplish one of two things:

- To correctly retrieve an automatically generated ID number after a new record is inserted into the database
- To be able to commit or roll back changes within the transaction, based on the success or failure of some type of external processing

Using Transactions for ID Number Safety

In Chapter 14, "Using Forms to Add or Change Data," in Vol. 1, *Getting Started*, you learned how the <cftransaction> tag should be used around the two-step process that's often necessary when inserting a new record into a database. First, the actual insert is performed (via <cfinsert> or a SQL INSERT statement), then the SQL MAX() function is used to retrieve the ID number of the just-inserted record.

The examples in this book always use <cftransaction> around such a multistep process. Without <cftransaction>, the wrong ID number could occasionally be retrieved if several users were visiting the ColdFusion template at the same time. With <cftransaction>, you are assured that you will get back the correct number because no other database operations are allowed to affect the state of

each transaction until it is finished. Conceptually, you are asking your database to freeze just before the changes are made and unfreeze only after the changes are complete.

You generally end up with code that follows this basic pattern:

```
<cftransaction>
 <!--- 1) insert record, via <cfinsert> or SQL INSERT --->
 <!--- 2) get new id number, via a SELECT MAX() query --->
</cftransaction>
```

➜ See Chapter 14 for a complete example that uses `<cftransaction>` in this manner.

NOTE

The mechanics of how the database actually preserves the integrity of the transaction varies somewhat from database to database. Some of them implement transactions by simply blocking all other access to the relevant database (or table or portions of the table) until your transaction is finished. Others use a more sophisticated approach, in which simultaneous transactions each affect their own version of the data. Consult your database documentation for details.

Using Transactions for the Ability to Undo

In Chapter 22, a CFML custom tag called `<cf_PlaceOrder>` was created, which takes care of all the various steps that must be completed each time a user decides to buy merchandise from Orange Whip Studios. The custom tag inserts a new record into the `MerchandiseOrders` table and inserts several new records into the `MerchandiseOrdersItems` table. It also attempts to charge the user's credit card and is smart enough to undo all the changes to the database if for whatever reason the user's credit card can't be successfully charged at the time. The `<cftransaction>` tag makes this possible.

When using `<cftransaction>` to enable rollback processing, you usually end up with code that follows this basic pattern:

```
<cftransaction action="Begin">
 ... database changes here ...
 <cfif Everything Goes Well>
 <cftransaction action="Commit"/>
 <cfelse>
 <cftransaction action="Rollback"/>
 </cfif>
</cftransaction>
```

NOTE

The trailing forward slashes at the end of the previous `action="Commit"` and the `action="Rollback"` are important. Be sure to include the slashes when performing a `Commit` or `Rollback` in your own code.

If you take a look at the code for the `<cf_PlaceOrder>` tag in Chapter 22, you will see that it follows this pattern. First, the transaction is begun, using the opening `<cftransaction>` tag. Within the transaction, the various `INSERT` queries needed to record the order are executed. Because the queries are being executed within the safe space of the transaction, you don't need to worry about what might happen if only part of the database changes were able to take place.

Then, after the records have been inserted and there has been an attempt to charge the user's credit card, a `<cfif>` statement is used to determine whether the credit card charge was successful. If so,

the transaction is committed via an `action="Commit"`. If not, the transaction is rolled back via an `action="Rollback"`.

NOTE

With two important exceptions, you can't use more than one data source within a single `<cftransaction>` block. The first exception is when you're using ColdFusion's query of queries feature; queries of `dbtype="query"` are always allowed, even in transactions that involve other data sources. The second exception is after an explicit `COMMIT`; you can run queries that refer to other data sources after a transaction has been committed. Otherwise, any queries that need to use other data sources must be placed outside the `<cftransaction>` block.

Transactions and CFML Error Handling

Database transactions commonly are committed or rolled back based on the results of ColdFusion's structured exception-handling mechanisms. In general, this means using `<cftransaction>` inside a `<cftry>` block. Within the `<cftry>` block, any database errors are caught using the `<cfcatch>` tag, which causes the transaction to be rolled back with an `action="Rollback"`. For details, see Chapter 51, "Error Handling," online.

Transactions and Stored Procedures

It often makes the most sense to create stored procedures that encapsulate an entire database transaction. Rather than creating several `<cfquery>` tags and wrapping `<cftransaction>` around them, you would just make a single stored procedure that takes care of performing all the steps. Within the stored procedure, you would use whatever syntax your database requires to ensure the database server considers the whole process to be a single database transaction.

This way, everything about the transaction is owned conceptually by the database server and occurs entirely under its watch. Also, the stored procedure can then be used by other systems within your company, and not just ColdFusion.

You learned about stored procedures in Chapter 42, "Working with Stored Procedures."

NOTE

With Oracle systems, you use the `SET TRANSACTION`, `COMMIT`, and `ROLLBACK` statements to declare the beginning and end of a transaction within a stored procedure. With SQL Server, you use `BEGIN TRANSACTION`, `COMMIT TRANSACTION`, and `ROLLBACK`.

Using `<cfdbinfo>`

Normally you will design your database either before or during your application development process, but what about times when you have no control over the database. What about times when you don't even know what is in the database? While this seems unlikely, there are times when this exact scenario will occur. You may be working on an application that others have to install and therefore need to check that the proper tables were installed. You may be working on application

that works with other application data sources and introspect what is inside the data source. All of this is now possible with the new ColdFusion 8 tag, `<cfdbinfo>`. Table 41.7 defines the attributes of this tag.

Table 41.7 `<cfdbinfo>` Tag Attributes

ATTRIBUTE	DESCRIPTION
`datasource`	The data source to inspect.
`name`	The name of the result.
`type`	The type of information to retrieve from the data source. Valid values are: dbnames, tables, columns, version, procedures, foreignkeys, index.
`dbname`	Use this attribute to override the database used in the datasource.
`username/password`	A username and password necessary to connect to the data source.
`pattern`	A pattern used to filter results. Used only when type is tables, columns, or procedures. The pattern can contain underlines to represent any one character or a % to represent 0 or more characters.
`table`	The table to use when using types values of columns, foreignkeys, or index.
`type`	The type of information to retrieve from the data source. Valid values are: dbnames, tables, columns, version, procedures, foreignkeys, index.

When `<cfdbinfo>` is used, a query object is always returned. The values in the query are dependant on the type used. We won't cover all the forms of results here, but will instead focus on the most used. Listing 41.18 demonstrates a simple example of `<cfdbinfo>` to get information about the tables in the OWS data source.

Listing 41.18 `dbinfo1.cfm`—Using `<cfdbinfo>`

```
<cfdbinfo datasource="ows" type="tables" name="tables">

<table border="1">
  <tr>
    <th>Name</th>
    <th>Type</th>
  </tr>
  <cfoutput query="tables">
  <tr>
    <td>#table_name#</td>
    <td>#table_type#</td>
  </tr>
  </cfoutput>
</table>
```

This listing is relatively simple. We use the `<cfdbinfo>` tag to retrieve information about the tables in the OWS data source. This returns a query with `columns: remarks`, `table_name`, and `table_type`. We then output a simple table (a table of tables!) and display the name and type. The first thing you may notice after running this template is that quite a few system tables exist. These are tables the database use to maintain information about itself. Normally you don't care about these tables. Listing 41.19 modifies the previous listing to hide system tables.

Listing 41.19 `dbinfo2.cfm`—Using `<cfdbinfo>`

```
<cfdbinfo datasource="ows" type="tables" name="tables">

<table border="1">
  <tr>
    <th>Name</th>
    <th>Type</th>
  </tr>
  <cfoutput query="tables">
  <cfif table_type is not "SYSTEM TABLE">
  <tr>
    <td>#table_name#</td>
    <td>#table_type#</td>
  </tr>
  </cfif>
  </cfoutput>
</table>
```

Listing 41.19 simply adds a `<cfif>` tag to check the value of `table_type`. If the type is not SYSTEM TABLE, the table is displayed. This shows a list of tables that will be closer to what you expect. Now let's take it a bit further. Listing 41.20 builds upon the previous two examples by displaying information about a particular table.

Listing 41.20 `dbinfo3.cfm`—Using `<cfdbinfo>`

```
<cfdbinfo datasource="ows" type="tables" name="tables">

<table border="1">
  <tr>
    <th>Name</th>
    <th>Type</th>
  </tr>
  <cfoutput query="tables">
  <cfif table_type is not "SYSTEM TABLE">
  <tr>
    <td>
<a href =
"dbinfo3.cfm?table=#urlEncodedFormat(table_name)#"
    >#table_name#</a>
    </td>
    <td>#table_type#</td>
  </tr>
```

Listing 41.20 (continued)

```
      </cfif>
      </cfoutput>
    </table>

    <cfif isDefined("url.table")>

      <cfdbinfo datasource="ows" type="columns"
            table="#url.table#" name="cols">

      <cfoutput>
      <p>
      The table #url.table# has the following columns:
      </p>
      </cfoutput>

      <table border="1">
        <tr>
          <th>Name</th>
          <th>Type</th>
          <th>Primary</th>
          <th>Nullable</th>
        </tr>
        <cfoutput query="cols">
        <tr>
          <td>#column_name#</td>
          <td>#type_name#</td>
          <td>#is_primarykey#</td>
          <td>#is_nullable#</td>
        </tr>
        </cfoutput>
      </table>

    </cfif>
```

The first part of listing 41.20 is the same as 41.19, except for the addition of a link that passes the name of the table back to the template. If we see that a table was passed in the URL, we then use `<cfdbinfo>` to retrieve information about the table. As stated above, all `<cfdbinfo>` calls return a query. The columns returned for each type may be found in the reference. For this example we worked with four columns: `column_name`, `type_name`, `is_primarykey` and `is_nullable`.

The `column_name` and `type_name` columns refer to the name of the column and the type of column respectively. The `is_primarykey` and `is_nullable` values will be true if the column is either a primary key or allows for null values. All together this creates a table of information about the table clicked on from the original list. Again – more information then this is returned – but the display gives us a good set of information about the table. Figure 41.13 shows an example of one of the tables.

Figure 41.13
Example of information returned from <cfdbinfo>.

Name	Type
ACTORS	TABLE
CONTACTS	TABLE
DIRECTORS	TABLE
EXPENSES	TABLE
FILMS	TABLE
FILMSACTORS	TABLE
FILMSDIRECTORS	TABLE
FILMSRATINGS	TABLE
MERCHANDISE	TABLE
MERCHANDISEORDERS	TABLE
MERCHANDISEORDERSITEMS	TABLE
USERROLES	TABLE

The table FILMS has the following columns:

Name	Type	Primary	Nullable
FILMID	INTEGER	YES	NO
MOVIETITLE	CHAR	NO	NO
PITCHTEXT	CHAR	NO	NO
AMOUNTBUDGETED	REAL	NO	YES
RATINGID	INTEGER	NO	YES
SUMMARY	VARCHAR	NO	YES
IMAGENAME	CHAR	NO	YES
DATEINTHEATERS	DATE	NO	YES

Other Information Returned By `<cfdbinfo>`

Our previous examples focused on tables and columns, but what else can you do with `<cfdbinfo>`? `<cfdbinfo>` can return:

A list of all the stored procedures in a database.

All foreign keys. The columns results used in listing 41.20 also contain the foreign key information specific to one table, but `<cfdbinfo>` can return a complete list of foreign keys.

A list of all indexes in a database.

The version and type of driver used to connect to the database. This is important as some drivers will have an impact on the type of SQL you use in your application.

CHAPTER 42

Working with Stored Procedures

IN THIS CHAPTER

Why Use Stored Procedures? E75
Calling Stored Procedures from ColdFusion Templates E79
Creating Stored Procedures E100

Most server-based database systems—SQL Server, Oracle, and Sybase—support *stored procedures*. A stored procedure is a chunk of SQL code that's given a name and stored as a part of your database, along with your actual data tables. After the stored procedure has been created, you can invoke it in your ColdFusion templates using the `<cfstoredproc>` tag.

NOTE

The use of stored procedures in database applications is a relatively advanced topic. It's not rocket science, but you will be more comfortable working with stored procedures if you are familiar with the SQL concepts introduced in the previous chapter or have a specific reason for using stored procedures in a particular ColdFusion application.

NOTE

At the time of this writing, stored procedures are supported by most server-based database systems (such as SQL Server, Oracle, and Sybase) but are not generally supported by file-based databases (such as Access, FoxPro, Paradox, dBASE, and so on). If you don't plan on using a server-based database system, you can skip this chapter without missing out on anything essential.

Why Use Stored Procedures?

Stored procedures provide a way to consolidate any number of SQL statements—such as `SELECT`, `INSERT`, `UPDATE`, and so on—into a little package that encapsulates a complete operation to be carried out in your application.

For instance, consider Orange Whip Studio's online store. When a customer places an order, the application needs to verify that the customer's account is in good standing and then carry out `INSERT` statements to the `MerchandiseOrders` and `MerchandiseOrdersItems` tables to record the actual order. In the future, the application might be expanded to first ensure that the selected merchandise is in stock. If not, the application would need to display some type of "Sorry, out of stock" message for the user, and it might need to update another table somewhere else to indicate that the merchandise needs to be reordered from the supplier.

This type of complex, causally related set of checks and record keeping is often referred to as a *business rule* or *business process*. Using the techniques you've learned so far in this book, you already know that you could accomplish the steps with several `<cfquery>` tags and some conditional processing using `<cfif>` and `<cfelse>` tags. In fact, in Chapter 22, "Online Commerce," online, you learned how all the relevant processing can be bundled up into a CFML custom tag called `<cf_PlaceOrder>`.

In this chapter, you will see that you could wrap up all the database-related actions required to place an order into a single stored procedure called, say, `PlaceOrder`, thus encapsulating the entire business process into one smart routine your ColdFusion templates need only refer to.

This shifts the responsibility for ensuring that the steps are followed properly away from your ColdFusion code, and hands it to the database server itself. This shift generally requires a little bit of extra work on your part up front but offers some real advantages later.

Advantages of Using Stored Procedures

Depending on the situation, using a stored procedure in your application (rather than a series of `<cfquery>` and `<cfif>` tags) can offer a number of significant advantages. I will introduce you to the most common advantages in this section. You might want to consult your database server documentation for further details and remarks on the advantages of using stored procedures.

Modularity Is a Good Thing

When developing any type of application, it generally pays to keep pieces of code—whether it be CFML code, SQL statements, or something else—broken into small, self-explanatory modules that can perform a specific task on their own. This keeps your code readable, easier to maintain, and easier to reuse in other applications you might write later. You learned how to do this for CFML code in Chapter 24, "Building User-Defined Functions," and Chapter 25, "Creating Custom Tags," both in *Adobe ColdFusion 8 Web Application Construction Kit*, *Volume 2: Application Development*.

Keeping your code in separate chunks can also let several people more easily work on the same project at the same time without stepping on each other's toes. With respect to stored procedures specifically, you might have a scenario where Developer A says to Developer B, "Make me a stored procedure that does such-and-such. Meanwhile, I'll be putting together the ColdFusion templates that use the procedure." Because the stored procedure runs independently of the templates and vice versa, neither developer needs to wait for the other to get started. The result can be an application that gets up and running more quickly.

Sharing Code Between Applications

After a stored procedure has been created, it enables you to share code between different types of applications, even applications written with different development tools. For instance, a ColdFusion template, a Visual Basic program, and a Java application might all refer to the same stored procedure on your server. Clearly, this cuts down on development time and ensures that all three applications enforce the various business rules in exactly the same way.

This type of consistent enforcement of business rules can be particularly important if the three applications are being developed by different teams or developers. In addition, if the business rules change for order taking in the future, for example, it is likely that only the stored procedure would need to be adapted, rather than the code in all three applications needing to be revised.

Increasing Performance

Depending on the situation, using stored procedures can often cause an application to perform better. There are two ways in which stored procedures can help speed your application.

First, most database systems do some type of precompilation of the stored procedure so it runs more quickly when it's actually used (this is analogous to how ColdFusion compiles your ColdFusion pages into Java classes for you). For instance, Microsoft SQL Server makes all its performance-optimizing decisions (such as which indexes and which join algorithms to use) the first time a stored procedure is run. Subsequent executions of the stored procedure don't need to be parsed and analyzed, which causes the procedure to run somewhat faster than if you executed its SQL statements in an ad hoc fashion every time. Generally, the more steps the procedure represents, the more of a difference this precompilation makes. Oracle servers do something very similar.

Second, if you compare the idea of having one stored procedure versus several `<cfquery>` and `<cfif>` tags in a template, the stored procedure approach is often more efficient because much less communication is necessary between ColdFusion and the database server. Instead of sending the information about the number of books in stock and customer account status back and forth between the ColdFusion server and the database server, all the querying and decision-making steps are kept in one place. Because less data needs to move between the two systems, and because the database drivers aren't really involved at all until the very end, you are likely to have a slightly faster process if you use the stored-procedure approach. Generally, the more data that needs to be examined to enforce the various business rules, the more of a difference the use of stored procedures is likely to have on overall application performance.

NOTE

In general, the performance increases mentioned here will be relatively modest. Your mileage will vary from application to application and from database to database, but you should expect speed increases of about 10 percent. Don't expect your application to work 20 times faster just because you put all your SQL code into stored procedures.

Making Table Schemas Irrelevant

Because a stored procedure is invoked by referring to it by name, a database administrator can shield whoever is actually using a stored procedure from having to be familiar with the database tables' underlying structure. In fact, in a team environment, it's easy to imagine a situation in which you are developing a ColdFusion application but don't even know the names of the tables in which the data is being stored.

You can get your work done more quickly if whoever designed the database provides you with a number of stored procedures you can use without having to learn the various table and column names involved. In return, the database designer can be comforted by the knowledge that if the

relationships between the tables change at some point in the future, only the stored procedure will need to be changed, rather than the ColdFusion code you're working on.

Addressing Security Concerns

Depending on the type of database server you're using, stored procedures can provide extra security. For instance, you might have some confidential data in some of your database tables, so the database administrator might not have granted `SELECT` or `INSERT` privileges to whatever database username ColdFusion uses to interact with the database server. By creating a few stored procedures, your administrator could provide the needed information to ColdFusion without granting more general privileges than are necessary.

By granting privileges only to the stored procedures rather than enabling the actual tables to be queried or updated, the database administrator can be confident that the data in the tables won't be compromised or lose its real-world meaning. Because all database servers enable the administrator to grant `SELECT`, `UPDATE`, `DELETE`, and `INSERT` privileges separately, the administrator could enable your ColdFusion application to retrieve data but make changes only via stored procedures. That gives the administrator a lot of control and keeps you from worrying about harming the organization's data because of some mistake in your code.

Such a policy would enable the ColdFusion developer, the database administrator, and the company in general to feel very confident about the development of a ColdFusion application. Everyone involved can see that nothing crazy is going to happen as the company shifts toward Web-based applications.

Comparing Stored Procedures to CFML Custom Tags

Stored procedures are conceptually quite similar to CFML custom tags (also known as modules), which you learned about in Chapter 26, "Building Reusable Components," in Vol. 2, *Application Development*. Think of a stored procedure as the database server equivalent of a custom tag; this might help you get a handle on when you should consider writing a stored procedure.

Here are some of the ways stored procedures are like custom tags:

- Both stored procedures and custom tags let you take a bunch of code, wrap it up, slap a name on it, and use it as you develop your applications almost as if it were part of the language all along.
- Both encourage code reuse, cutting development times and costs in the long run.
- Both can accept parameters as input.
- Both can set ColdFusion variables in the calling template that can be used in subsequent CFML code.
- Both can generate one or more query result sets for the calling template.
- Both make integrating other people's work into your own easier.

Calling Stored Procedures from ColdFusion Templates

Now that you have an idea about what kinds of things stored procedures can be used for, this is a good time to see how to integrate them into your ColdFusion templates. For the moment, assume that several stored procedures have already been created and are ready for use within your ColdFusion application. Maybe you put the stored procedures together yourself, or maybe they were created by another developer or by the database administrator, the DBA.

At this point, all you care about is what the name of the stored procedure is and what it does. You will learn how to actually create stored procedures later in this chapter.

Two Ways to Execute Stored Procedures

There are two ways to execute a stored procedure from your ColdFusion templates. You can use the `<cfstoredproc>` tag, which is obviously designed specifically for stored procedures, or can use the `<cfquery>` tag that you already know and love.

With the `<cfstoredproc>` Tag

The `<cfstoredproc>` tag is the formal, recommended way to call stored procedures. You can pass input parameters to the stored procedure, collect return codes and output parameters passed back by the procedure, and use any recordsets (queries) the procedure returns. In theory, `<cfstoredproc>` should also result in the fastest performance.

- However, using `<cfstoredproc>` has one disadvantage. You can't use ColdFusion's query-caching feature with the recordsets the procedure returns. You can, of course, store the results in a persistent scope.

With the `<cfquery>` Tag

As an alternative to `<cfstoredproc>`, you can execute stored procedures via ordinary `<cfquery>` tags. Recordsets returned by stored procedures called in this way can be cached via ColdFusion's `cachedWithin` attribute (see Chapter 31, "Improving Performance," in Vol. 2, *Application Development*).

Unfortunately, this method has two big disadvantages:

- You can't capture any output parameters or result codes returned by the procedure (only query-style output). You'll learn what these items are later in this chapter, in the "Stored Procedures That Take Parameters and Return Status Codes" section.
- If the procedure returns multiple recordsets, you can capture only one of them for use in your ColdFusion code. See the section, "Calling Procedures with `<cfquery>` Instead of `<cfstoredproc>`," later in this chapter.

Using the `<cfstoredproc>` Tag

To call an existing stored procedure from a ColdFusion template, you refer to the procedure by name using the `<cfstoredproc>` tag. The `<cfstoredproc>` tag is similar to the `<cfquery>` tag in that

it knows how to interact with data sources you've defined in the ColdFusion Administrator. However, rather than accepting ad hoc SQL query statements (such as `SELECT` and `DELETE`), `<cfstoredproc>` is very structured, optimized specifically for dealing with stored procedures.

The `<cfstoredproc>` tag takes a number of relatively simple parameters, as listed in Table 42.1.

Table 42.1 Important `<cfstoredproc>` Attributes

ATTRIBUTE	PURPOSE
`procedure`	The name of the stored procedure you want to execute. You usually can just provide the procedure name directly, as in `procedure="MyProcedure"`. Depending on the database system you are using, however, you might need to qualify the procedure name further using dot notation.
`datasource`	The appropriate data source name. Just like the `datasource` attribute for `<cfquery>`, this can be the name of any data source listed in the ColdFusion Administrator.
`returnCode`	Optional. `Yes` or `No`. This attribute determines whether ColdFusion should capture the status code (sometimes called the return code or return value) reported by the stored procedure after it executes. If you set this attribute to `Yes`, the status code will be available to you in a special variable called `CFSTOREDPROC.StatusCode` unless specified as something else with the `result` attribute. See "Stored Procedures That Take Parameters and Return Status Codes," later in this chapter.
`result`	Optional. By default, the status code value returned when `returnCode` is set to Yes will be available in a variable called `CFSTOREDPROC`. The result attribute lets you specify another variable name to create.

NOTE

The `<cfstoredproc>` tag also supports `username`, `password`, `blockfactor`, and `debug` attributes. All these attributes work similarly to the corresponding attributes of the `<cfquery>` tag.

For simple stored procedures, you can just use its `procedure` and `datasource` attributes. When the template is executed in a Web browser, the stored procedure executes on the database server, accomplishing whatever it was designed to accomplish as i t goes.

For instance, suppose you have access to an imaginary stored procedure called `PerformInventoryMaintenance`. It has been explained to you that this stored procedure performs some type of internal maintenance on the data in the `Merchandise` table, and that people need some way to execute the procedure via the company intranet. Listing 42.1 shows a ColdFusion template called `InventoryMaintenanceRun.cfm`, which does exactly that.

NOTE

The code listings in this chapter refer to a data source called `owsSqlServer` to indicate a copy of the `ows` sample database sitting on a Microsoft SQLServer or Sybase server, and a data source called `owsOracle` to indicate a version of the database sitting on an Oracle server. The examples in this chapter are for illustration purposes of the ColdFusion tags used to interact with stored procedures.

Listing 42.1 `InventoryMaintenanceRun.cfm`—Calling a Stored Procedure with `<cfstoredproc>`

```
<!---
 Filename: InventoryMaintenanceRun.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates use of the <cfstoredproc> tag
--->

<html>
<head><title>Inventory Maintenance</title></head>
<body>
<h2>Inventory Maintenance</h2>

<!--- If the submit button was not just pressed, display form --->
<cfif not isDefined("FORM.executeNow")>

  <!--- Provide button to start stored procedure --->
  <cfform action="#CGI.script_name#" method="post">
  <cfinput type="submit" name="executeNow"
           value="Perform Inventory Maintenance">
  </cfform>

<!--- If the user just clicked the submit button --->
<cfelse>

  <p>Executing stored procedure...</p>
  <!--- Go ahead and execute the stored procedure --->
  <cfstoredproc procedure="PerformInventoryMaintenance"
                datasource="owsSqlServer">
  <p>Done executing stored procedure!</p>

</cfif>
</body>
</html>
```

As you can see, the code to execute the stored procedure is extremely simple. You'll see in a moment how to use stored procedures that receive input and respond by providing output back to you. This particular stored procedure doesn't require any information to be provided to it, so the only things you must specify are the `procedure` and `datasource` parameters.

The `procedure` parameter tells ColdFusion what the name of the stored procedure is. The `datasource` parameter works just like it does for the `<cfquery>` tag; it must be the name of the appropriate data source as defined in the ColdFusion Administrator.

NOTE

Listing 42.1 uses a simple piece of `<cfif>` logic that tests to see whether a form parameter called `executeNow` exists. Assuming that it does not exist, the template puts a simple form–with a single submit button–on the page. Because the submit button is named `executeNow`, the `<cfif>` logic executes the second part of the template when the form is submitted; the second part contains the `<cfstoredproc>` tag that executes the stored procedure. The template is put together this way so you can see the form code and the form-processing code in one listing. See Chapter 14, "Using Forms to Add or Change Data," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 1: Getting Started*, for further discussion of this type of single-template technique.

When the `InventoryMaintenanceRun.cfm` template from Listing 42.1 is first brought up in a Web browser, it displays a simple form (Figure 42.1). When the Perform Inventory Maintenance button

is clicked, the procedure is executed and a simple message is displayed to the user to indicate that the work has been done (Figure 42.2).

Figure 42.1

The `InventoryMaintenanceRun.cfm` template first displays a simple form.

Figure 42.2

The stored procedure executes when the form is submitted.

Stored Procedures That Return Recordsets

Stored procedures also can return recordsets to your ColdFusion templates. As far as your ColdFusion code is concerned, recordsets are just like query results. They contain columns and rows of data—usually from one or more of the database's tables—that you can then use the same way you would use rows of data fetched by a `<cfquery>` tag.

For instance, consider a stored procedure called `FetchRatingsList` that returns rating information straight from the `FilmsRatings` table in the `owsSqlServer` database. This stored procedure sends back its information as a recordset, just as if you had performed a simple `SELECT` type of query with a `<cfquery>` tag.

The person who created the stored procedure has told you that that the recordset will contain two columns: `RatingID` and `Rating`, which correspond to the columns of the `FilmsRatings` table. In other words, executing this particular stored procedure is similar to running an ordinary `SELECT * FROM FilmsRating` query (you'll see more complex examples shortly).

The `<cfprocresult>` Tag

To use a recordset returned by a stored procedure, you must use the `<cfprocresult>` tag, which tells ColdFusion to capture the recordset as the stored procedure is executed. The `<cfprocresult>` tag takes just a few attributes and is easy to use.

Table 42.2 lists the attributes supported by `<cfprocresult>`. Most of the time, you will need to use only the `name` attribute.

Table 42.2 `<cfprocresult>` Tag Attributes

ATTRIBUTE	PURPOSE
`name`	A name for the recordset. The recordset will become available as a query object with whatever name you specify; the query object can be used just like any other (like the results of a `<cfquery>` or `<cfpop>` tag).
`resultSet`	An optional number that indicates which recordset you are referring to. This attribute is important only for stored procedures that return several recordsets. Defaults to `1`. See the section, "Multiple Recordsets Are Fully Supported," later in this chapter.
`maxRows`	Optional; similar to the `maxRows` attribute of the `<cfquery>` tag. The maximum number of rows ColdFusion should retrieve from the database server as the stored procedure executes. If not provided, ColdFusion defaults to -1, which is the same as all rows.

NOTE

Because a recordset captured from a stored procedure is made available to your ColdFusion templates as an ordinary CFML query object, the query object has the same `RecordCount`, `ColumnList`, and `CurrentRow` properties that the result of a normal `<cfquery>` tag would. You can use these to find out how much data the recordset contains.

NOTE

The `<cfprocresult>` tag must be used between opening and closing `<cfstoredproc>` tags, as shown in Listing 42.2.

Using `<cfprocresult>`

The `FilmEntry1.cfm` template in Listing 42.2 shows how to retrieve and use a recordset returned from a stored procedure. As you can see, a `<cfstoredproc>` tag is used to refer to the stored procedure itself. Then the `<cfprocresult>` tag is used to capture the recordset returned by the procedure.

NOTE

Remember, it's assumed that you have created a version of the Orange Whip Studios sample database for SQLServer (or whatever database server you are using), and that the database is accessible via the data source name `owsSqlServer`.

NOTE

In addition, for this template to work, you must have created the `FetchRatingsList` stored procedure. See the "Creating Stored Procedures" section in this chapter for details.

Listing 42.2 `FilmEntry1.cfm`—Retrieving a Recordset from a Stored Procedure

```
<!---
 Filename: FilmEntry1.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates use of the <CFSTOREDPROC> tag
```

Listing 42.2 (continued)

```
--->

<!--- Get list of ratings from database --->
<cfstoredproc procedure="FetchRatingsList" datasource="owsSqlServer">
 <cfprocresult name="getRatings">
</cfstoredproc>

<html>
<head><title>Film Entry Form</title></head>
<body>
<h2>Film Entry Form</h2>

<!--- Data entry form --->
<cfform action="#CGI.script_name#" method="post" preserveData="Yes">

 <!--- Text entry field for film title --->
 <p><b>Title for New Film:</b><br>
 <cfinput name="movieTitle" size="50" maxlength="50" required="yes"
  message="Please don't leave the film's title blank."><br>

<!--- Text entry field for pitch text --->
 <p><b>Short Description / One-Liner:</b><br>
 <cfinput name="pitchText" size="50" maxlength="100" required="Yes"
  message="Please don't leave the one-liner blank."><br>

 <!--- Text entry field for expense description --->
 <p><b>New Film Budget:</b><br>
 <cfinput name="amountBudgeted" size="15" required="Yes"
  message="Please enter a valid number for the film's budget."
  validate="float"><br>

 <!--- Drop-down list of ratings --->
 <p><b>Rating:</b><br>
 <cfselect name="ratingID" query="getRatings" value="RatingID" display="Rating"/>
 <!--- Text areas for summary --->
 <p><b>Summary:</b><br>
 <cftextarea name="summary" cols="40" rows="3" wrap="soft"></cftextarea>

 <!--- Submit button for form --->
 <p><cfinput name="submit" type="submit" value="Submit New Film">
</cfform>

</body>
</html>
```

NOTE

The syntax shown in Listing 42.2 should work with most database server systems, including Microsoft SQLServer and Sybase. If you are using Oracle, you will need to make a minor change.

Because this listing specifies `name="getRatings"` in the `<cfprocresult>` tag, the rest of the template is free to refer to `getRatings` just like the results of a `<cfquery>`. Here, the `getRatings` query object is provided to a `<cfselect>` tag to populate a drop-down list (Figure 42.3).

Figure 42.3

The <cfprocresult> tag captures a recordset returned by a stored procedure.

NOTE

If you don't know the column names that the procedure outputs, you could run the procedure and output the value of the automatic `#getRatings.ColumnList#` variable, just as you could do for any normal query resultset.

Using <cfprocresult> with Oracle

With Oracle databases, stored procedures can't return recordsets in the traditional sense. They can return recordset-like data, but they do so via something called a *reference cursor*. You'll see an example of how to create a stored procedure that returns a reference cursor later in this chapter, but you need to consult your Oracle documentation if you want to learn all the ins and outs about reference cursors and the various ways in which they can be used.

ColdFusion makes using the data returned by reference cursors in Oracle stored procedures easy. Basically, you just add a `<cfprocresult>` tag to capture the data from each reference cursor.

Listing 42.3 is almost the same as the previous version of this template (refer to Listing 42.2), except that the `datasource` and the name of the procedure has been adjusted appropriately. Assuming that a procedure called `owsWeb.FetchRatingsList` exists on your Oracle server, and assuming that it has a single output parameter that exposes information from the `FilmsRatings` table as a reference cursor, this template will behave just like the previous listing (refer to Figure 42.3).

NOTE

Remember, it is assumed that you have created a version of the Orange Whip Studios sample database for Oracle, and that the database is accessible via a native Oracle data source named `owsOracle`.

NOTE

In addition, for this template to work, you must have created the `owsWeb.FetchRatingsList` stored procedure. See the "Creating Stored Procedures with Oracle" section in this chapter for details.

Listing 42.3 `FilmEntry1Oracle.cfm`—Retrieving Data from a Reference Cursor

```
<!---
 Filename: FilmEntry1Oracle.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates use of stored procedures
--->

<!--- Get list of ratings from database --->
<cfstoredproc procedure="OWSWEB.FetchRatingsList" datasource="owsOracle">
 <cfprocresult name="getRatings">
</cfstoredproc>

<html>
<head><title>Film Entry Form</title></head>
<body>
<h2>Film Entry Form</h2>

<!--- Data entry form --->
<cfform action="#CGI.script_name#" method="post" preserveData="Yes">

 <!--- Text entry field for film title --->
 <p><b>Title for New Film:</b><br>
 <cfinput name="movieTitle" size="50" maxlength="50" required="yes"
  message="Please don't leave the film's title blank."><br>

<!--- Text entry field for pitch text --->
 <p><b>Short Description / One-Liner:</b><br>
 <cfinput name="pitchText" size="50" maxlength="100" required="Yes"
  message="Please don't leave the one-liner blank."><br>

 <!--- Text entry field for expense description --->
 <p><b>New Film Budget:</b><br>
 <cfinput name="amountBudgeted" size="15" required="Yes"
  message="Please enter a valid number for the film's budget."
  validate="float"><br>

 <!--- Drop-down list of ratings --->
 <p><b>Rating:</b><br>
 <cfselect name="ratingID" query="getRatings" value="RatingID" display="Rating"/>
 <!--- Text areas for summary --->
 <p><b>Summary:</b><br>
 <cftextarea name="summary" cols="40" rows="3" wrap="soft"></cftextarea>

 <!--- Submit button for form --->
 <p><cfinput name="submit" type="submit" value="Submit New Film">
</cfform>

</body>
</html>
```

NOTE

In the stored procedure name `owsWeb.FetchRatingsList`, the `owsWeb` part refers to the name of the package that the stored procedure is in, and `FetchRatingsList` is the name of the procedure itself. For more information about packages, see your Oracle documentation.

NOTE

To see the Oracle code required to create the `owsWeb.FetchRatingsList` Procedures referred to in this template, see the section, "Creating Stored Procedures with Oracle," later in this chapter.

Stored Procedures That Take Parameters and Return Status Codes

So far you've been working with stored procedures that always work the same exact way each time they are called. The `PerformInventoryMaintenance` and `FetchRatingsList` procedures used in the first two examples don't accept any input from the calling application (in this case, a ColdFusion template) to do their work.

Most stored procedures, however, take input parameters or output parameters and also can return status codes. Here's what each of these terms means:

- **Input parameters.** Values you supply by name when you execute a stored procedure, similar to providing parameters to a ColdFusion tag. The stored procedure can then use the values of the input parameters internally, similar to variables. The stored procedure can take as many input parameters as needed for the task at hand. Similar to attributes for a ColdFusion tag, some input parameters are required, and others are optional. To supply an input parameter to a stored procedure, you use the `<cfprocparam>` tag (see Table 42.3 in the next section).
- **Output parameters.** Values the procedure can pass back to ColdFusion or to whatever other program might call the stored procedure. Each output parameter has a name, and the stored procedure can return as many output parameters as necessary. To capture the value of an output parameter from a stored procedure, use the `<cfprocparam>` tag with `type="OUT"` (see Table 42.3).
- **Status codes.** Values returned by a stored procedure after it executes. Each stored procedure can return only one status code. The code is usually used to indicate whether the stored procedure was capable of successfully carrying out its work. With most database systems, the status code must be numeric, and defaults to `0`. To capture a procedure's status code, use `returnCode="Yes"` in the `<cfstoredproc>` tag (refer to Table 42.1, earlier in this chapter).

For instance, consider a new stored procedure called `InsertFilm` that enables the user to add a new film to Orange Whip Studio's `Films` table. Whoever created the stored procedure set it up to require five input parameters called `@MovieTitle`, `@PitchText`, `@AmountBudgeted`, `@RatingID`, and `@Summary`, which supply information about the new film. The procedure uses these values to insert a new record into the `Films` table.

The procedure also has one output parameter called `@NewFilmID`, which passes the ID number of the newly inserted film record back to ColdFusion (or whatever program is executing the stored procedure).

NOTE

Stored procedure parameter names start with an @ sign with Microsoft SQLServer and Sybase databases. Other databases systems, such as Oracle, don't use the @ sign in this way.

In addition, the stored procedure has been designed to perform a few sanity checks before blindly recording the new film:

- First, it ensures that the film doesn't already exist in the Films table. If a film with the same title is already in the table, the procedure stops with a return value of `-1`.
- It ensures that the rating specified by the `@RatingID` parameter is valid. If the rating number does not exist in the `Ratings` table, the procedure stops with a return value of `-2`.

Provided both of the tests are okay, the procedure inserts a new row in the `Inventory` table using the values of the supplied parameters. Finally, it sends a return value of `1` to indicate success.

Providing Parameters with `<cfprocparam>`

ColdFusion makes supplying input and output parameters to a stored procedure easy, via the `<cfprocparam>` tag. Table 42.3 shows the attributes supported by the `<cfprocparam>` tag.

Table 42.3 `<cfproparam>` Tag Attributes

ATTRIBUTE	PURPOSE
`type`	`In`, `Out`, or `InOut`. Use `type="In"` (the default) to supply an input parameter to the stored procedure. Use `type="Out"` to capture the value of an output parameter. Use `typw="InOut"` if the parameter behaves as both an input and output parameter (such parameters are rather rare).
`value`	For input parameters only. The actual value you want to provide to the procedure.
`dbvarname`	Name of parameter passed to stored procedure.
`null`	For input parameters only. `Yes` or `No`. If `null="Yes"`, the `value` attribute is ignored; instead, a null value is supplied as the parameter's value. For more information about null values, see the section, "Working with NULL Values" in Chapter 41, "More About SQL and Queries," online.
`variable`	For output parameters only. A variable name you want ColdFusion to place the value of the output parameter into. For instance, if you provide `variable= "InsertedFilmID"`, you can output the value using `#InsertedFilmID#` after the `<cfstoredproc>` tag executes.
`cfsqltype`	Required. The parameter's data type. Unlike ColdFusion variables, stored procedure parameters are strongly typed, so you have to specify whether the parameter expects a numeric, string, date, or other type of value. The list of data types that you can supply to this attribute is the same as for the `<cfqueryparam>` tag; see Table 41.4 in Chapter 41 for the list of data types.
`maxLength`	Optional. The maximum length of the parameter's value.
`scale`	Optional. The number of significant decimal places for the parameter. Relevant only for numeric parameters.

NOTE

If you're using Oracle, don't include a `<cfprocparam>` tag for output parameters that return reference cursors. Use the `<cfprocresult>` tag to capture that type of output, as discussed in the previous section.

`FilmEntry2.cfm`, shown in Listing 42.4, builds on the previous version of the data entry template from Listing 42.3. It creates a simple form a user can use to fill in the title, description, budget, rating, and summary for a new book. The form collects the information from the user and posts it back to the same template, which executes the stored procedure, feeding the user's entries to the procedure's input parameters.

Listing 42.4 `FilmEntry2.cfm`—A Simple Form for Collecting Input Parameters

```
<!---
 Filename: FilmEntry2.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates use of stored procedures
--->

<!--- Is the form being submitted? --->
<cfset wasFormSubmitted = isDefined("FORM.ratingID")>

<!--- Insert film into database when form is submitted --->
<cfif wasFormSubmitted>
 <cfstoredproc procedure="InsertFilm" datasource="owsSqlServer" returncode="Yes">
  <!--- Provide form values to the procedure's input parameters --->
  <cfprocparam type="In" maxlength="50" cfsqltype="CF_SQL_VARCHAR"
   value="#FORM.movieTitle#">
  <cfprocparam type="In" maxlength="100" cfsqltype="CF_SQL_VARCHAR"
   value="#FORM.pitchText#">
  <cfprocparam type="In" maxlength="100" cfsqltype="CF_SQL_MONEY"
   value="#FORM.amountBudgeted#"
   null="#yesNoFormat(FORM.amountBudgeted eq '')#">
  <cfprocparam type="In" maxlength="100" cfsqltype="CF_SQL_INTEGER"
   value="#FORM.ratingID#">
  <cfprocparam type="In" cfsqltype="CF_SQL_LONGVARCHAR" value="#FORM.summary#">
  <!--- Capture @NewFilmID output parameter --->
  <!--- Value will be available in CFML variable named #InsertedFilmID# --->
  <cfprocparam type="Out" cfsqltype="CF_SQL_INTEGER" variable="InsertedFilmID">
 </cfstoredproc>

 <!--- Remember the status code returned by the stored procedure --->
 <cfset insertStatus = CFSTOREDPROC.StatusCode>
</cfif>
<!--- Get list of ratings from database --->
<cfstoredproc procedure="FetchRatingsList" datasource="owsSqlServer">
 <cfprocresult name="getRatings">
</cfstoredproc>

<html>
<head><title>Film Entry Form</title></head>
<body>
<h2>Film Entry Form</h2>

<!--- Data entry form --->
```

Listing 42.4 (continued)

```
<cfform action="#CGI.script_name#" method="post" preserveData="Yes">

 <!--- Text entry field for film title --->
 <p><b>Title for New Film:</b><br>
 <cfinput name="movieTitle" size="50" maxlength="50" required="Yes"
  message="Please don't leave the film's title blank."><br>

 <!--- Text entry field for pitch text --->
 <p><b>Short Description / One-Liner:</b><br>
 <cfinput name="pitchText" size="50" maxlength="100" required="Yes"
  message="Please don't leave the one-liner blank."><br>

 <!--- Text entry field for expense description --->
 <p><b>New Film Budget:</b><br>
 <cfinput name="amountBudgeted" size="15" required="No"
  message="Only numbers may be provided for the film's budget."
  validate="float"> (leave blank if unknown)<br>

 <!--- Drop-down list of ratings --->
 <p><b>Rating:</b><br>
 <cfselect name="ratingID" query="getRatings" value="RatingID" display="Rating"/>

 <!--- Text areas for summary --->
 <p><b>Summary:</b><br>
 <cftextarea name="summary" cols="40" rows="3" wrap="soft"></cftextarea>

<!--- Submit button for form --->
 <p><cfinput type="submit" name="submit" value="Submit New Film">
</cfform>

<!--- If we executed the stored procedure --->
<cfif wasFormSubmitted>
 <!--- Display message based on status code reported by stored procedure --->
 <cfswitch expression="#insertStatus#">
  <!--- If the stored procedure returned a "success" status --->
  <cfcase value="1">
   <cfoutput>
    <p>Film "#FORM.movieTitle#" was inserted as Film ID #insertedFilmID#.<br>
   </cfoutput>
  </cfcase>
  <!--- If the status code was -1 --->
  <cfcase value="-1">
   <cfoutput>
    <p>Film "#Form.MovieTitle#" already exists in the database.<br>
   </cfoutput>
  </cfcase>
  <!--- If the status code was -2 --->
  <cfcase value="-2">
   <p>An invalid rating was provided.<br>
  </cfcase>
  <!--- If any other status code was returned --->
  <cfdefaultcase>
   <p>The procedure returned an unknown status code.<br>
  </cfdefaultcase>
 </cfswitch>
```

Listing 42.4 (continued)

```
</cfif>

</body>
</html>
```

When the form is submitted, the `<cfif>` block at the top of the template is executed, which executes the `InsertFilm` stored procedure via the `<cfstoredproc>` tag. Within the `<cfstoredproc>` tag, six `<cfprocparam>` tags are used. The first five provide values for the `@MovieTitle`, `@PitchText`, and other input parameters. The last `<cfprocparam>` captures the value of the output parameter called `@NewFilmID`.

Note that the `cfsqltype` for each of the parameters has been set to the correct value for the type of information being passed. Also, the `null` attribute is used for the fifth `<cfprocparam>`, so that the film's budget will be recorded as a null value if the user leaves the budget blank on the form. After the procedure executes, the status code reported by the procedure is placed into the `insertStatus` variable.

At the bottom of the template, a `<cfswitch>` block is used to output a message to the user depending on the status code reported by the stored procedure. If the procedure returns a value of `1`, a success message is displayed, along with the new film's `FilmID` number—which was captured by the first `<cfprocparam>` tag at the top of the template (Figure 42.4). If the procedure returns some other status code, it is assumed that something went wrong, so the status code is shown to the user.

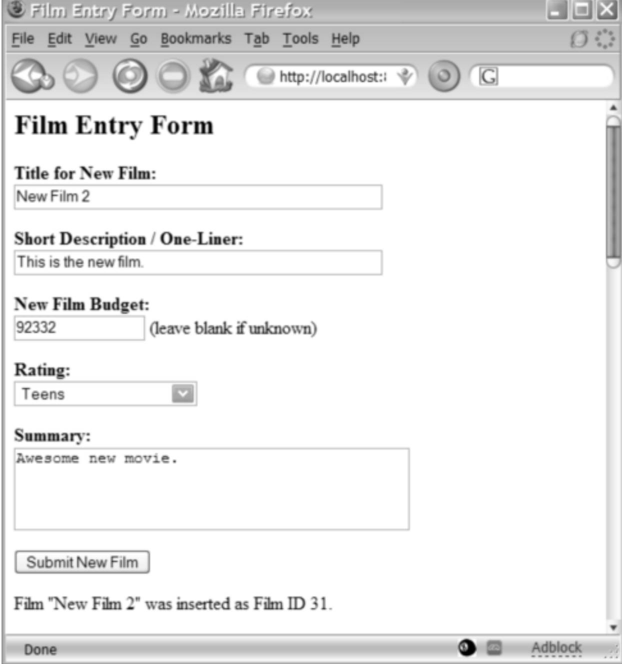


Figure 42.4

Parameters can be passed in and out of stored procedures.

NOTE

In an actual application, you would probably do more than just display the new `FilmID`. For instance, you might provide some type of Click Here link to a screen on which the user could do further data entry about the film. The point is that a stored procedure can give back whatever information its creator desires as output parameters, and you can use that information in just about any way you please.

NOTE

The template in Listing 42.4 uses ColdFusion's `<cfswitch>`, `<cfcase>`, and `<cfdefaultcase>` tags to analyze the return code from the stored procedure. Making decisions based on return codes is a perfect situation to use these case-switching tags because a single expression (the return code) exists that will have one of several predetermined values.

Ordinal Versus Named Parameter Positioning

When you use the `<cfprocparam>` tag to provide parameters to a stored procedure, you need to provide the `<cfprocparam>` tags in the correct order. That is, the first `<cfprocparam>` tag needs to correspond to the first parameter in the stored procedure's definition on the database server, the second `<cfprocparam>` tag needs to correspond to the second parameter, and so on. This is called *ordinal positioning*, meaning that the order of parameters is significant.

If you want to supply the parameters in another order, you must use the `dbVarName` attribute to specify the name of the parameter.

Parameter Data Types

As you saw in Listing 41.4, when you provide parameters to a stored procedure with the `<cfprocparam>` tag, you must specify the data type of the parameter, as defined by whoever created the procedure. ColdFusion requires that you provide the data type for each parameter you refer to in your templates, so it doesn't have to determine the data type itself on the fly each time the template runs. That would require a number of extra steps for ColdFusion, which in turn would slow your application.

It's important to specify the correct data type for each parameter. If you use the wrong data type, you might run into problems. The data type you provide for `CFSQLTYPE` in a `<cfprocparam>` tag must be one of ColdFusion's SQL data types, as listed in Table 41.5 in Chapter 41. These data types are based on the data types defined by the ODBC standard—one of them will map to each of the database-specific data types used when your stored procedure was created.

For instance, if you have a stored procedure sitting on a Microsoft SQL Server that takes a parameter of SQL Server data type `int`, you should specify the `CF_SQL_INTEGER` data type in the corresponding `<cfprocparam>` tag.

Wrapping a Stored Procedure Call in a Custom Tag

If you have a stored procedure that you plan to use often in your ColdFusion applications, you may want to consider creating a CFML Custom Tag that wraps all the needed `<cfstoredproc>` and related code into an easier-to-use module. For instance, you might create a custom tag called `<cf_spInsertFilm>` that calls the `InsertFilm` stored procedure.

The `spInsertFilm.cfm` template included online for this chapter creates such a custom tag, which could be used like this:

```
<!--- Insert film via Stored Procedure (via custom tag) --->
<cf_spInsertFilm
 movieTitle="#FORM.MovieTitle#"
 pitchText="#FORM.PitchText#"
 ratingID="#FORM.RatingID#"
 amountBudgeted="#FORM.AmountBudgeted#"
 summary="#FORM.Summary#"
 returnFilmID="InsertedFilmID"
 returnStatusCode="InsertStatus">
```

The `FilmEntry3.cfm` template (also online) is a revision of Listing 42.4. Instead of calling the `<cfstoredprc>` tag directly, the template simply calls the custom tag when the form is submitted. For more information about how to create and use CFML custom tags, see Chapter 25.

Multiple Recordsets Are Fully Supported

Some stored procedures return more than one recordset. For instance, consider a stored procedure called `FetchFilmInfo`. You are told that the procedure accepts one input parameter called `@FilmID`, and the procedure responds by returning five recordsets of information related to the specified film. The first recordset contains information about the film record itself; the second returns related records from the `Expenses` table; the third returns related records from the `Actors` table; the fourth returns related records from `Directors`; and the fifth returns information from the `Merchandise` table.

As you can see in Listing 42.5, the key to receiving more than one recordset from a stored procedure is to include one `<cfprocresult>` tag for each recordset, specifying `resultset="1"` for the first recordset, `resultset="2"` for the second, and so on.

Listing 42.5 `ShowFilmExpenses.cfm`—Dealing with Multiple Recordsets

```
<!---
 Filename: ShowFilmExpenses.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates use of stored procedures
--->

<!--- Execute stored procedure to fetch film information --->
<cfstoredproc procedure="FetchFilmInfo" datasource="owsSqlServer">
  <!--- Provide the FilmID parameter --->
  <cfprocparam type="In" cfsqltype="CF_SQL_INTEGER" value="#URL.filmID#">
  <!--- Film information --->
  <cfprocresult name="getFilm" resultset="1">
  <!--- Expense information --->
  <cfprocresult name="getExpenses" resultset="2">
  <!--- Actor information --->
  <cfprocresult name="getActors" resultset="3">
  <!--- Director information --->
  <cfprocresult name="getDirectors" resultset="4">
  <!--- Director information --->
  <cfprocresult name="getMerch" resultset="5">
```

Listing 42.5 (continued)

```
</cfstoredproc>

<!--- Get subtotals from the recordsets returned by stored procedure --->
<cfset expenseSum = arraySum(listToArray(valueList(getExpenses.ExpenseAmount)))>
<cfset actorSum = arraySum(listToArray(valueList(GetActors.Salary)))>
<cfset directorSum = arraySum(listToArray(valueList(GetDirectors.Salary)))>
<cfset merchSum = arraySum(listToArray(valueList(GetMerch.TotalSales)))>
<!--- Add up all expenses --->
<cfset totalExpenses = expenseSum + actorSum + directorSum - merchSum>
<!--- Determine how much money is left in the budget --->
<cfset leftInBudget = getFilm.AmountBudgeted - totalExpenses>

<html>
<head><title>Film Expenses</title></head>
<body>
<!--- Company logo and page title --->
<img src="../images/logo_b.gif" width="73" height="73" align="absmiddle">
<font size="+2"><b>Film Expenses</b></font><br clear="all">

<cfoutput>
 <!--- Show film title--->
 <p><b>Film:</b> #getFilm.MovieTitle#<br>
 <!--- Film budget, expense total, and amount left in budget --->
 <p><b>Budget:</b> #lsCurrencyFormat(getFilm.AmountBudgeted)#<br>
 <b>Expenses:</b> #lsCurrencyFormat(totalExpenses)#<br>
 <b>Currently:</b> #lsCurrencyFormat(leftInBudget)#
 <!--- Are we currently over or under budget? --->
 #iif(leftInBudget lt 0, "'over budget'", "'under budget'")#<br>
 <!--- Output information about actors --->
 <p><b>Actors:</b>
 <cfloop query="getActors">
  <li>#NameFirst# #NameLast# (Salary: #lsCurrencyFormat(Salary)#)
 </cfloop>
 <!--- Output information about directors --->
 <p><b>Directors:</b>
 <cfloop query="getDirectors">
  <li>#FirstName# #LastName# (Salary: #lsCurrencyFormat(Salary)#)
 </cfloop>
 <!--- Output information about expenses --->
 <p><b>Other Expenses:</b>
 <cfloop query="GetExpenses">
  <li>#Description# (#lsCurrencyFormat(ExpenseAmount)#)
 </cfloop>
 <!--- Output information about merchandise --->
 <p><b>Income from merchandise:</b>
 <cfloop query="getMerch">
  <li>#MerchName# (Sales: #lsCurrencyFormat(TotalSales)#)
 </cfloop>
</cfoutput>
</body>
</html>
```

After the <cfstoredproc> tag, the ColdFusion template is free to refer to the five recordsets named in the <cfprocresult> tags as if they were the results of five separate <cfquery>-type queries (Figure 42.5). But because only one communication between ColdFusion and the database server needed to take place, you can expect performance to be faster using the single stored procedure.

Figure 42.5

ColdFusion makes capturing multiple recordsets from a single stored procedure easy.

NOTE

Your template doesn't have to handle or receive all the recordsets a stored procedure spits out. For instance, if you weren't interested in the second recordset from the `FetchFilmInfo` procedure, you could simply leave out the second <cfprocresult> tag. Neither ColdFusion nor your database server will mind.

If you're using Oracle, then `resultset="1"` refers to the first output parameter of type `REF CURSOR`, `resultset="2"` refers to the second such parameter, and so on. So if you had a procedure called `owsWeb.FetchFilmInfo` on your Oracle server that exposed five reference cursors (such as the five recordsets returned by the SQLServer used in the previous listing), hardly anything needs to change. You would simply use the same <cfprocresult> tags to capture the data from the reference cursors, as shown in Listing 42.6.

Listing 42.6 `ShowFilmExpensesOracle.cfm`—Dealing with Multiple Reference Cursors

```
<!---
 Filename: ShowFilmExpensesOracle.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates use of stored procedures
--->

<!--- Execute stored procedure to fetch film information --->
<cfstoredproc procedure="OWSWEB.FetchFilmInfo" datasource="owsOracle">
   <!--- Provide the FilmID parameter --->
  <cfprocparam type="In" cfsqltype="CF_SQL_INTEGER" value="#URL.filmID#">
  <!--- Film information --->
  <cfprocresult name="getFilm" resultset="1">
  <!--- Expense information --->
  <cfprocresult name="getExpenses" resultset="2">
  <!--- Actor information --->
  <cfprocresult name="getActors" resultset="3">
  <!--- Director information --->
  <cfprocresult name="getDirectors" resultset="4">
  <!--- Director information --->
  <cfprocresult name="getMerch" resultset="5">
</cfstoredproc>

<!--- Get subtotals from the recordsets returned by stored procedure --->
<cfset expenseSum = arraySum(listToArray(valueList(getExpenses.ExpenseAmount)))>
<cfset actorSum = arraySum(listToArray(valueList(GetActors.Salary)))>
<cfset directorSum = arraySum(listToArray(valueList(GetDirectors.Salary)))>
<cfset merchSum = arraySum(listToArray(valueList(GetMerch.TotalSales)))>
<!--- Add up all expenses --->
<cfset totalExpenses = expenseSum + actorSum + directorSum - merchSum>
<!--- Determine how much money is left in the budget --->
<cfset leftInBudget = getFilm.AmountBudgeted - totalExpenses>

<html>
<head><title>Film Expenses</title></head>
<body>
<!--- Company logo and page title --->
<img src="../images/logo_b.gif" width="73" height="73" align="absmiddle">
<font size="+2"><b>Film Expenses</b></font><br clear="all">

<cfoutput>
 <!--- Show film title--->
 <p><b>Film:</b> #getFilm.MovieTitle#<br>
 <!--- Film budget, expense total, and amount left in budget --->
 <p><b>Budget:</b> #lsCurrencyFormat(getFilm.AmountBudgeted)#<br>
 <b>Expenses:</b> #lsCurrencyFormat(totalExpenses)#<br>
 <b>Currently:</b> #lsCurrencyFormat(leftInBudget)#
 <!--- Are we currently over or under budget? --->
 #iif(leftInBudget lt 0, "'over budget'", "'under budget'")#<br>
 <!--- Output information about actors --->
 <p><b>Actors:</b>
 <cfloop query="getActors">
  <li>#NameFirst# #NameLast# (Salary: #lsCurrencyFormat(Salary)#)
 </cfloop>
 <!--- Output information about directors --->
 <p><b>Directors:</b>
```

Listing 42.6 (continued)

```
    <cfloop query="getDirectors">
     <li>#FirstName# #LastName# (Salary: #lsCurrencyFormat(Salary)#)
    </cfloop>
    <!--- Output information about expenses --->
    <p><b>Other Expenses:</b>
    <cfloop QUERY="GetExpenses">
     <li>#Description# (#lsCurrencyFormat(ExpenseAmount)#)
    </cfloop>
    <!--- Output information about merchandise --->
    <p><b>Income from merchandise:</b>
    <cfloop query="getMerch">
     <li>#MerchName# (Sales: #lsCurrencyFormat(TotalSales)#)
    </cfloop>
   </cfoutput>
   </body>
   </html>
```

NOTE

This code assumes that the procedure parameter named `getFilm` is a reference cursor that exposes records from the `Films` table, `getExpenses` exposes data from the `Expenses` table, and so on.

Calling Procedures with `<cfquery>` Instead of `<cfstoredproc>`

The `<cfstoredproc>` and related tags were added back in ColdFusion 4. Before that, the only way to call a stored procedure from a ColdFusion template was to use special procedure-calling syntax in a normal `cfquery` tag, where you would normally provide a SQL statement. You can still call stored procedures this way, and you actually might prefer to do so in some situations.

When you execute a stored procedure with `<cfquery>`, ColdFusion treats the response from the database server the way it would treat the response from an ordinary `SELECT` query. In fact, ColdFusion doesn't even realize that it's executing a stored procedure exactly; it's just passing on what it assumes to be a valid SQL statement and hopes to get some rows of table-style data in return.

The fact that ColdFusion is treating the stored procedure the same way it treats a `<select>` statement brings with it several important limitations:

- ColdFusion can't directly access the return code generated by the stored procedure.
- ColdFusion can't directly access any output parameters generated by the stored procedure.
- If the stored procedure returns more than one recordset, only one of the recordsets will be captured by ColdFusion and be available for your use in your templates. Details about this limitation will vary between types of database servers. With some database servers, only the first recordset will be available for your use; with others, only the last will be available. The point is that stored procedures that return multiple recordsets are not fully supported when using `cfquery`.

However, using <cfquery> also has an important advantage over <cfstoredproc>:

- With <cfquery>, you can pass a stored procedure's parameters by name, instead of only by position. For details, see "Using Your Database's Native Syntax" in this section.

Using the ODBC/JDBC CALL Command

With most types of database systems (including SQLServer and Oracle), you can use the CALL command defined by the ODBC standard to execute a stored procedure. Listing 42.7 demonstrates how the FetchRatingsList stored procedure can be called using this method.

Note that this template is almost exactly the same as the FilmEntry1.cfm template shown in Listing 42.2. The only change is that <cfquery> is being used instead of <cfstoredproc>. When this template is brought up in a browser, it should display its results exactly the same way.

Listing 42.7 FilmEntry1a.cfm—Calling a Stored Procedure

```
<!---
 Filename: FilmEntry1a.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates use of stored procedures
--->

<!--- Get list of ratings from database --->
<cfquery name="getRatings" datasource="owsSqlServer">
 {  CALL FetchRatingsList }
</cfquery>

<html>
<head><title>Film Entry Form</title></head>
<body>
<h2>Film Entry Form</h2>

<!--- Data entry form --->
<cfform action="#CGI.script_name#" method="post" preservedata="Yes">
 <!--- Text entry field for film title --->
 <p><b>Title for New Film:</b><br>
 <cfinput name="movieTitle" size="50" maxlength="50" required="Yes"
  message="Please don't leave the film's title blank."><br>
 <!--- Text entry field for pitch text --->
 <p><b>Short Description / One-Liner:</b><br>
 <cfinput name="pitchText" size="50" maxlength="100" required="Yes"
  message="Please don't leave the one-liner blank."><br>

 <!--- Text entry field for expense description --->
 <p><b>New Film Budget:</b><br>
 <cfinput name="amountBudgeted" size="15" required="Yes"
  message="Please enter a valid number for the film's budget."
  validate="float"><br>

 <!--- Drop-down list of ratings --->
 <p><b>Rating:</b><br>
 <cfselect name="ratingID" query="getRatings" value="RatingID" display="Rating"/>
 <!--- Text areas for summary --->
```

Listing 42.7 (continued)

```
  <p><b>Summary:</b><br>
  <cftextarea name="summary" cols="40" rows="3" wrap="soft"></cftextarea>

  <!--- Submit button for form --->
  <p><cfinput type="submit" name="submit" value="Submit New Film">
 </cfform>

 </body>
 </html>
```

As you can see in Listing 42.7, the syntax for using the `CALL` command is simply the word `CALL` and then the procedure name. The entire command is surrounded by a set of curly braces, which indicate that the command must be interpreted by JDBC before it gets sent on to the database server.

Input parameters can be supplied in parentheses after the procedure name, separated by commas. If no input parameters exist for the procedure, leave the parentheses off. Because you aren't referring to them by name, input parameters must be supplied in the proper order (as defined by whomever wrote the stored procedure). If an input parameter is of a character type (such as `char`, `varchar`, or `text`), enclose the parameter's value in single quotation marks. For instance, to call the `InsertFilm` stored procedure from Listing 42.4, you could replace the `<cfstoredproc>` block in that template with the following snippet (see `FilmEntry2a.cfm` online to see this snippet in a complete template):

```
<cfquery datasource="owsSqlServer">
 { CALL InsertFilm (
  '#FORM.movieTitle#',
  '#FORM.pitchText#',
   #FORM.amountBudgeted#,
   #FORM.ratingID#,
  '#FORM.summary#') }
</cfquery>
```

But remember that ColdFusion isn't aware of the return code or output parameters returned by the procedure, so the code in the rest of the listing would fail. You would need to have the stored procedure rewritten so that the information provided by the return code or output parameters instead get returned as a recordset.

Using Your Database's Native Syntax

In addition to using the `CALL` command, most database drivers also let you use whatever native syntax you would use normally with that database system. All the same limitations (regarding return codes, output parameters, and so on) listed at the beginning of this section apply.

The native syntax to use varies according to the database server you're using; consult your database server documentation for details. Just as an example, if you were using Microsoft SQL Server, you could replace the `<cfquery>` shown in Listing 42.7 with the following code; the results would be the same:

```
<cfquery name="getRatings" datasource="owsSqlServer">
  EXEC FetchRatingsList
</cfquery>
```

One advantage of using the native syntax over the `CALL` syntax is that you may be able to refer to the input parameters by name, which leads to cleaner and more readable code. So, if you were using Microsoft SQL Server, the `InsertFilm` procedure could be called with the following. Your database server documentation has specific details:

```
<cfquery datasource="owsSqlServer">
 EXEC InsertFilm
  @MovieTitle = '#FORM.movieTitle#',
  @PitchText = '#FORM.pitchText#',
  @AmountBudgeted = #FORM.amountBudgeted#,
  @RatingID = #FORM.ratingID#,
  @Summary = '#FORM.summary#'
</cfquery>
```

Depending on your database server, you might be able to add more code to the `<cfquery>` tag to be able to capture the status code and output parameters from the stored procedure. Again, just as an example, if you were using Microsoft SQL Server, you would be able to use something similar to the following:

```
<cfquery datasource="owsSqlServer" name="ExecProc">
 -- Declare T-SQL variables to hold values returned by stored procedure
 DECLARE @StatusCode INT, @NewFilmID INT
 -- Execute the stored procedure, assigning values to T-SQL variables
 EXEC @StatusCode = InsertFilm
  @MovieTitle = '#FORM.movieTitle#',
  @PitchText = '#FORM.pitchText#',
  @AmountBudgeted = #FORM.amountBudgeted#,
  @RatingID = #FORM.ratingID#,
  @Summary = '#FORM.summary#',
  @NewFilmID = @NewFilmID OUTPUT
 -- Select the T-SQL variables as a one-row recordset
 SELECT @StatusCode AS StatusCode, @NewFilmID AS NewFilmID
</cfquery>
```

You then could refer to `ExecProc.StatusCode` and `ExecProc.NewFilmID` in your CFML template code. The `FilmEntry2b.cfm` template, online, is a revised version of Listing 42.4 that uses this `<cfquery>` snippet instead of `<cfstoredproc>` to execute the `InsertFilm` stored procedure.

NOTE

You will need to consult your database documentation for more information about how to use this type of syntax. In general, it's really best to use `<cfstoredproc>` instead of `<cfquery>` if the stored procedure you want to use generates important status codes or output parameters.

Creating Stored Procedures

Now that you've seen how to use stored procedures in your ColdFusion templates, you're probably curious about how you can create some of your own. We can't cover everything about creating stored procedures in this book, but you will learn enough here to hit the ground running.

Before you begin, a short note: The examples you see in this section—and the procedure-creating code you're likely to use as you write your own stored procedures—are not standard SQL. Stored

procedures are implemented slightly differently by different database servers, and you often will want to use code within the stored procedures that take advantage of various extensions proprietary to the database system you are using. In other words, when you get to the point of writing your own stored procedures, you likely will be tying yourself to the database system you're using to some extent. The procedures probably will not be portable to other database systems (like ordinary SQL generally is) without reworking the procedures to some extent.

Creating Stored Procedures with Microsoft SQL Server

To create a stored procedure with Microsoft SQL Server, you use the SQL Server Enterprise Manager tool, which helps you submit a `CREATE PROCEDURE` statement to the SQL Server itself. After it's created, the procedure is available to whoever has appropriate permissions.

NOTE

This chapter assumes that you are using the latest version of SQL Server. If you are using an older version (such as SQLServer 6.5 or SQLServer 7.0), the specific steps might be slightly different, but the basic concepts will be the same.

NOTE

To follow along with this chapter, you need to have a database on your SQL Server machine that has the same tables and columns as the OWS sample database.

Creating Your First Stored Procedure

As a simple example, you can create the `FetchRatingsList` stored procedure that was used in Listing 42.2. This procedure is a good one to start with because it's very simple and doesn't use any input or output parameters.

To create the stored procedure, follow these simple steps:

1. Start the SQL Server Enterprise Manager. Click your server, expand the databases tree, and click your database.
2. Click the Stored Procedures folder; right-click to select New Stored Procedure from the pop-up menu.
3. Type the actual code for the procedure—provided in Listing 42.8—in the window that appears (Figure 42.6).
4. Click the OK button to create the stored procedure.

Listing 42.8 shows the actual code for the `FetchRatingsList` stored procedure. This is the code you should enter after selecting New Procedure from the pop-up menu (refer to Figure 42.6). As you see, it's fairly simple. First, a `CREATE PROCEDURE` statement is used to provide a name for the new stored procedure, followed by the `AS` keyword. Everything after `AS` is the code for the stored procedure itself—the actual SQL statements you want to execute each time the procedure is called.

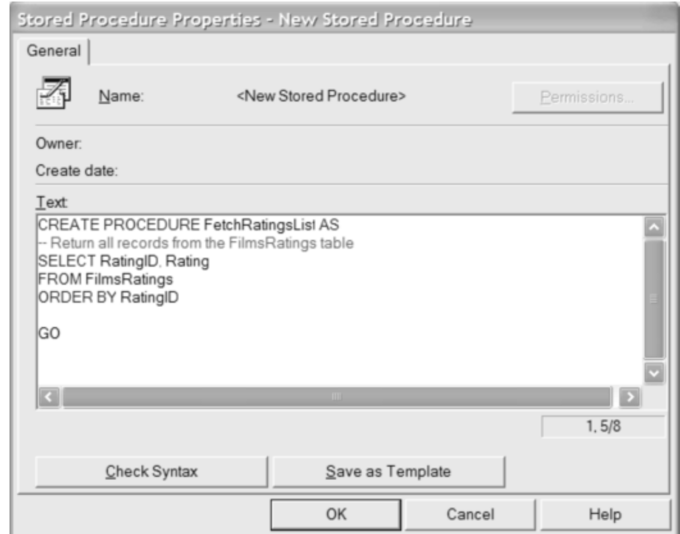


Figure 42.6
Enter the procedure's code in the New Stored Procedure window.

Listing 42.8 The `FetchRatingsList`—Stored Procedure

```
CREATE PROCEDURE FetchRatingsList AS
-- Return all records from the FilmsRatings table
SELECT RatingID, Rating
FROM FilmsRatings
ORDER BY RatingID
```

TIP

You also can execute the `CREATE PROCEDURE` statements shown in this section with the Query Analyzer (called iSQL/w in SQL Server 6.5 and earlier) or the `isql.exe` command-line utility that ships with SQL Server. You could even execute these `CREATE PROCEDURE` statements in a tag within a ColdFusion template.

This procedure's actual body is simple. It just uses a simple `SELECT` statement to return all the records from the `FilmsRatings` table. The records can be captured and accessed in your ColdFusion code via the `<cfstoredproc> tag` (refer to Listing 42.2) or `<cfquery>` tag (refer to Listing 42.7).

Defining Status Codes, Input Parameters, and Output Parameters

As demonstrated in Listing 42.4, stored procedures can accept input parameters, return values via output parameters, and return a status code.

With SQL Server, you define input parameters in the `CREATE PROCEDURE` part of the procedure code, between the procedure name and the `AS` keyword. Each input parameter is given a name, which must begin with the `@` symbol and can't contain any spaces or other unusual characters. The SQL Server data type for each parameter is provided after the parameter's name. If more than one parameter exists, separate them with commas.

If you want the parameter to be optional, type an = sign after the data type and then type the default value. To create an output parameter, provide an initial value for the parameter (usually NULL is most appropriate) using the = sign and then type the word OUTPUT.

For instance, Listing 42.9 shows the code to create the InsertFilm stored procedure used in Listing 42.4. As you can see, each of the parameters provided in <cfprocparam> tags in that template matches the corresponding parameters in the CREATE PROCEDURE statement here. Then, in the SQL statements that follow, the procedure is free to refer to the parameters by name—similar to how you refer to ColdFusion variables in your CFML templates. The values of the parameters are automatically plugged in with the appropriate values each time the procedure is actually used.

Listing 42.9 The InsertFilm Procedure Accepts Input Parameters

```
CREATE PROCEDURE InsertFilm
 @MovieTitle varchar(100),
 @PitchText varchar(100),
 @AmountBudgeted money,
 @RatingID int,
 @Summary text,
 @NewFilmID int = null OUTPUT
AS
-- Make sure there isn't a film by this name in the database already
IF EXISTS (SELECT * FROM Films WHERE MovieTitle = @MovieTitle)
-- If film exists already, return status of -1 to ColdFusion (or other program)
 RETURN -1
-- Make sure the specified rating is valid
ELSE IF NOT EXISTS (SELECT * FROM FilmsRatings WHERE RatingID = @RatingID)
 -- If specified rating code does not exist, return status of -2
 RETURN -2
-- Assuming that the film is not i nthe database already
ELSE
 BEGIN
  -- Insert the new film into the Films table
  INSERT INTO Films (MovieTitle, PitchText, AmountBudgeted, RatingID, Summary)
  VALUES (@MovieTitle, @PitchText, @AmountBudgeted, @RatingID, @Summary)
  -- Set output parameter called @NewFilmID to the ID of the just-inserted film
  SET @NewFilmID = @@IDENTITY
  -- Return a status of 1 to ColdFusion (or other program) to indicate success
  RETURN 1
 END
```

Note that the InsertFilm procedure uses SQL Server's IF keyword to do some conditional processing as the template executes. Similar conceptually to the CFIF tag in CFML, the IF keyword as used here lets you execute certain chunks of SQL code depending on conditions you define. You'll find IF extremely helpful when you need to write a stored procedure that must perform some kind of sanity check before committing changes to the database. For instance, the first IF line in Listing 42.11 halts processing and sends a return code of -1 to the calling application if the SELECT subquery inside the parentheses returns any rows.

Note also that the @NewFilmID output parameter is set to the value of SQL Server's built-in @@IDENTITY variable, which always contains the newly generated identity value for the last INSERT to a table that contained an identity-type column. (Identity columns are similar to AutoNumber columns in

Access tables.) Thus, the parameter gets set to the FilmID that was just assigned to the row just inserted into the Films table.

NOTE

IF is not part of SQL as most people would define it, but rather it is part of SQL Server's extensions to SQL, which Microsoft calls Transact-SQL. Transact-SQL provides many other control-of-flow keywords you might want to become familiar with, such as BEGIN, END, ELSE, DECLARE, and WHILE. Sybase also supports most of the same Transact-SQL extensions to standard SQL. Consult your database documentation about these keywords.

Returning Multiple Recordsets to ColdFusion

As you saw in Listing 42.5, stored procedures can return multiple recordsets to ColdFusion (or whatever program is calling the procedures). Again, think of a recordset as a set of rows and columns generated by a SELECT statement that outputs data from a database table.

To return recordsets, simply use the appropriate SELECT statements as you would normally. Whatever would be returned by the SELECT statement outside a stored procedure is what will be returned to the calling application (in this case, your ColdFusion template) as a recordset each time the procedure executes.

Your stored procedure can output as many recordsets as it pleases. The calling ColdFusion template need only provide a <cfprocresult> tag for each recordset that it wants to be aware of after the procedure runs. The second SELECT statement in the procedure that outputs rows can be captured by using recordset="2" in the <cfprocresult> tag, and so on.

For instance, Listing 42.10 provides the code to create the FetchFilmInfo procedure that is used by the ShowFilmExpenses.cfm template from Listing 42.5. As you can see, each <cfprocresult> tag used in that template matches with a corresponding SELECT statement in the procedure code itself.

Listing 42.10 Returning Multiple Recordsets from a Stored Procedure

```
CREATE PROCEDURE FetchFilmInfo
 @FilmID int
AS
-- Recommended setting; see SET in your SQLServer documentation
SET NOCOUNT ON
-- Return information about the film itself
SELECT * FROM Films
WHERE FilmID = @FilmID
-- Return all expense records related to the film
SELECT * FROM Expenses
WHERE FilmID = @FilmID
ORDER BY ExpenseDate DESC
-- Return all actor records related to the film
SELECT a.ActorID, a.NameFirst, a.NameLast, fa.Salary
FROM Actors a, FilmsActors fa
WHERE a.ActorID = fa.ActorID
AND fa.FilmID = @FilmID
ORDER BY NameFirst, NameLast
-- Return all director records related to the film
SELECT d.DirectorID, d.FirstName, d.LastName, fd.Salary
```

Listing 42.10 (continued)

```
FROM Directors d, FilmsDirectors fd
WHERE d.DirectorID = fd.DirectorID
AND fd.FilmID = @FilmID
ORDER BY LastName, FirstName
-- Return all merchandise records related to the film
SELECT m.MerchID, m.MerchName, SUM(ItemPrice) AS TotalSales
FROM Merchandise m, MerchandiseOrdersItems oi
WHERE m.MerchID = oi.ItemID
AND m.FilmID = @FilmID
GROUP BY m.MerchID, m.MerchName
ORDER BY m.MerchName
```

Creating Stored Procedures with Oracle

Stored procedures work somewhat differently with Oracle than they do with SQL Server and Sybase. But the basic concepts are the same, and many of the stored procedure examples in this chapter can be implemented on an Oracle server quite easily.

NOTE

This section assumes you are using Oracle 8i for Windows NT or 2000. If you're using a different version of Oracle (such as Oracle 9i or 10i), some of the figures shown in this chapter may look a little different from what you see on your screen.

NOTE

With Oracle, procedures that send back a return value are actually called stored functions, rather than stored procedures. Stored functions are not supported by the `<cfstoredproc>` tag (even if you set `returnCode="Yes"`). Therefore, you need to write your procedures so they send back any status information as an output parameter instead of a return value.

NOTE

Creating the `ImportCustomers` Stored Procedure

The easiest way to create a stored procedure is to use the Oracle DBA Studio, which is installed as part of the standard Oracle 8i installation. For instance, follow these steps to create the `InsertFilm` stored procedure:

1. Using the Windows Start menu, start the Oracle DBA Studio application and enter a valid username and password when prompted.
2. Navigate to the Schema item within your database, right-click the Procedure folder, and then select Create from the pop-up menu (Figure 42.7). The Create Procedure dialog box appears.
3. For the procedure's Name, type `INSERTFILM`; then select the correct Schema from the drop-down list.
4. Type the code shown in Listing 42.11 into the Source box (Figure 42.8). Don't include the first line of the listing (the `CREATE OR REPLACE PROCEDURE` line). The dialog box includes this line for you.

5. Click the Create button. Your new procedure is created and the dialog box disappears. You can now expand the Procedures folder along the left side of the screen and click the new procedure to confirm that it was created correctly.

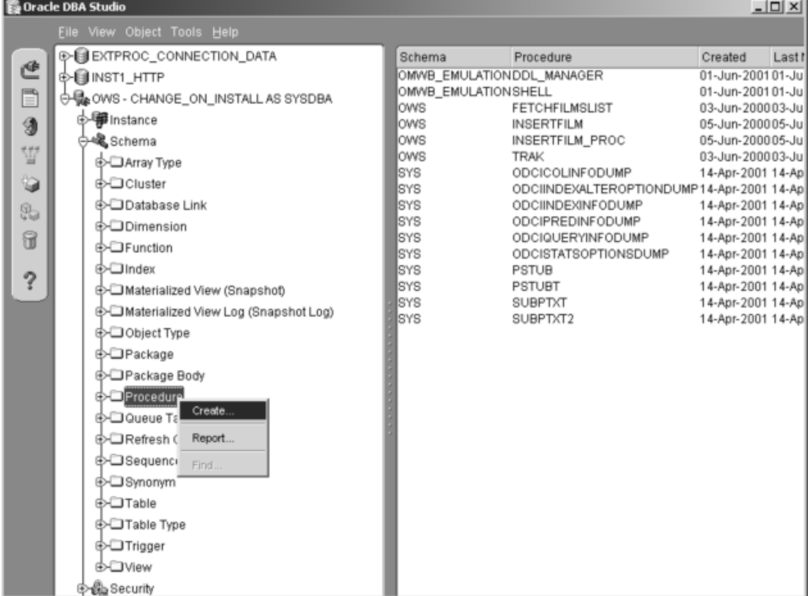


Figure 42.7

Creating a stored procedure is simple with the Oracle DBA Studio.

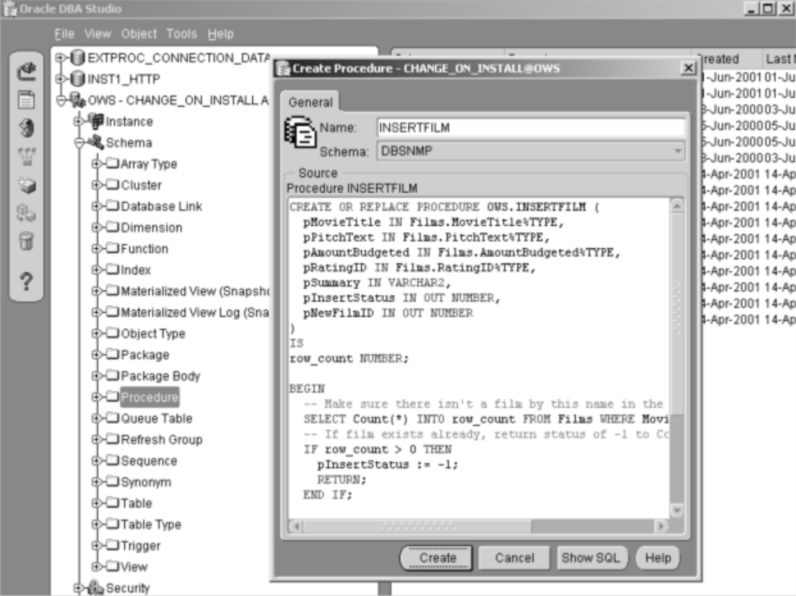


Figure 42.8

Enter the actual SQL code for the procedure into the Create Procedure dialog box.

NOTE

Instead of following the steps, you can just execute the code in Listing 42.11 directly into the Oracle SQL*Plus Worksheet utility. See your Oracle documentation for details.

Listing 42.11 Creating the `InsertFilm`—Stored Procedure for Oracle

```
CREATE OR REPLACE PROCEDURE OWS.INSERTFILM (
 pMovieTitle IN Films.MovieTitle%TYPE,
 pPitchText IN Films.PitchText%TYPE,
 pAmountBudgeted IN Films.AmountBudgeted%TYPE,
 pRatingID IN Films.RatingID%TYPE,
 pSummary IN VARCHAR2,
 pInsertStatus IN OUT NUMBER,
 pNewFilmID IN OUT NUMBER
)
IS
row_count NUMBER;
BEGIN
 -- Make sure there isn't a film by this name in the database already
 SELECT Count(*) INTO row_count FROM Films WHERE MovieTitle = pMovieTitle;
 -- If film exists already, return status of -1 to ColdFusion (or other program)
 IF row_count > 0 THEN
  pInsertStatus := -1;
  RETURN;
 END IF;
 -- Make sure the specified rating is valid
 SELECT Count(*) INTO row_count FROM FilmsRatings WHERE RatingID = pRatingID;
 -- If specified rating code does not exist, return status of -2
 IF row_count = 0 THEN
  pInsertStatus := -2;
  RETURN;
 END IF;
 -- Insert the new film into the Films table
 INSERT INTO Films (
  FilmID, MovieTitle, PitchText,
  AmountBudgeted, RatingID, Summary)
 VALUES (
  FilmsSeq.NEXTVAL, pMovieTitle, pPitchText,
  pAmountBudgeted, pRatingID, pSummary);

 -- Set output parameter called NewFilmID to the ID of the just-inserted film
 SELECT FilmsSeq.CURRVAL INTO pNewFilmID FROM DUAL;
 -- Return a status of 1 to ColdFusion (or other program) to indicate success
 pInsertStatus := 1;
END;
```

NOTE

Listing 42.11 assumes that an Oracle sequence called `FilmsSeq` is being used to autogenerate `FilmID` values. See your Oracle documentation for details about sequences.

Note that the SQL syntax for creating the stored procedure is slightly different from what it is for SQL Server, but the basic idea is the same. There's nothing here that's conceptually different from the SQL Server version of the procedure (refer to Listing 42.9).

The procedure name is provided first and follows the words CREATE OR REPLACE PROCEDURE. Then the procedure's parameters, if any, are provided within a pair of parentheses. The parentheses are followed by the IS keyword, and then the procedure's actual SQL code is provided between BEGIN and END keywords. Within the code, the actual value for the output parameter is set using Oracle's assignment operator, which is a colon followed by an equals sign (:=).

NOTE

Oracle requires that you place a semicolon after each statement in your SQL code. If you've done any work with C or C++ in the past, that semicolon is a familiar friend. See your Oracle documentation for details.

Because the Oracle version of the InsertFilm stored procedure created in Listing 42.11 returns the insert status as an output parameter rather than as the status code, the ColdFusion code that uses the stored procedure must be changed slightly. Listing 42.12 is a revised version of Listing 42.4, which adds a <cfprocparam> tag to capture the value of pNewFilmID. It also eliminates the return-Code="Yes" attribute from the first <cfstoredproc> tag.

Listing 42.12 FilmEntry2Oracle.cfm—Using the Oracle Version of the InsertFilm Stored Procedure

```
<!---
 Filename: FilmEntry2Oracle.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates use of stored procedures
--->

<!--- Is the form being submitted? --->
<cfset wasFormSubmitted = isDefined("FORM.ratingID")>

<!--- Insert film into database when form is submitted --->
<cfif wasFormSubmitted>
 <cfstoredproc procedure="insertFilm" datasource="owsOracle" returncode="No">
  <!--- Provide form values to the procedure's input parameters --->
  <cfprocparam type="In" maxlength="50" cfsqltype="CF_SQL_VARCHAR"
  value="#FORM.movieTitle#">
  <cfprocparam type="In" maxlength="100" cfsqltype="CF_SQL_VARCHAR"
   value="#FORM.pitchText#">
  <cfprocparam type="In" maxlength="100" cfsqltype="CF_SQL_INTEGER"
   value="#FORM.amountBudgeted#"
   null="#yesNoFormat(FORM.amountBudgeted eq '')#">
  <cfprocparam type="In" maxlength="100" cfsqltype="CF_SQL_INTEGER"
   value="#FORM.ratingID#">
  <cfprocparam type="In" cfsqltype="CF_SQL_LONGVARCHAR" value="#FORM.summary#">
  <!--- Capture pInsertStatus output parameter --->
  <!--- Value will be available in CFML variable named #InsertStatus# --->
  <cfprocparam type="Out" cfsqltype="CF_SQL_INTEGER" variable="InsertStatus">
  <!--- Capture pNewFilmID output parameter --->
  <!--- Value will be available in CFML variable named #InsertedFilmID# --->
  <cfprocparam type="Out" cfsqltype="CF_SQL_INTEGER" variable="InsertedFilmID">
 </cfstoredproc>

</cfif>

<!--- Get list of ratings from database --->
<cfstoredproc procedure="owsWeb.FetchRatingsList" datasource="owsOracle">
```

Listing 42.12 (continued)

```
 <!--- Make the data from reference cursor available as CFML query object --->
 <cfprocresult name="getRatings">
</cfstoredproc>

<html>
<head><title>Film Entry Form</title></head>
<body>
<h2>Film Entry Form</h2>

<!--- Data entry form --->
<cfform action="#CGI.script_name#" method="post" preserveData="Yes">

 <!--- Text entry field for film title --->
 <p><b>Title for New Film:</b><br>
 <cfinput name="movieTitle" size="50" maxlength="50" required="Yes"
  message="Please don't leave the film's title blank."><br>

 <!--- Text entry field for pitch text --->
 <p><b>Short Description / One-Liner:</b><br>
 <cfinput name="pitchText" size="50" maxlength="100" required="Yes"
  message="Please don't leave the one-liner blank."><br>

 <!--- Text entry field for expense description --->
 <p><b>New Film Budget:</b><br>
 <cfinput name="amountBudgeted" size="15" required="No"
  message="Only numbers may be provided for the film's budget."
  validate="float"> (leave blank if unknown)<br>

 <!--- Drop-down list of ratings --->
 <p><b>Rating:</b><br>
 <cfselect name="ratingID" query="getRatings" value="RatingID" display="Rating"/>

 <!--- Text areas for summary --->
 <p><b>Summary:</b><br>
 <cftextarea name="summary" cols="40" rows="3" wrap="soft"></cftextarea>

<!--- Submit button for form --->
 <p><cfinput type="submit" name="submit" value="Submit New Film">
</cfform>

<!--- If we executed the stored procedure --->
<cfif wasFormSubmitted>
 <!--- Display message based on status code reported by stored procedure --->
 <cfswitch expression="#insertStatus#">
  <!--- If the stored procedure returned a "success" status --->
  <cfcase value="1">
   <cfoutput>
    <p>Film "#FORM.movieTitle#" was inserted as Film ID #insertedFilmID#.<br>
   </cfoutput>
  </cfcase>
  <!--- If the status code was -1 --->
  <cfcase value="-1">
   <cfoutput>
    <p>Film "#Form.MovieTitle#" already exists in the database.<br>
   </cfoutput>
```

Listing 42.12 (continued)

```
    </cfcase>
    <!--- If the status code was -2 --->
    <cfcase value="-2">
     <p>An invalid rating was provided.<br>
    </cfcase>
    <!--- If any other status code was returned --->
    <cfdefaultcase>
     <p>The procedure returned an unknown status code.<br>
    </cfdefaultcase>
   </cfswitch>
  </cfif>

  </body>
  </html>
```

Creating the `FetchRatingsList` Stored Procedure

As you learned in Listing 42.3, Oracle stored procedures can't return recordsets per se; instead, they can expose reference cursors as output parameters. Although this is an important distinction within the Oracle universe, you can treat reference cursors just like traditional recordsets in your ColdFusion template code, by adding a `<cfprocresult>` tag to capture the data in each reference cursor.

Creating a stored procedure that exposes a reference cursor requires a bit more code than the equivalent SQLServer procedure, but the concepts are very similar. See your Oracle documentation for all the details. That said, we can provide you with enough information here to help get you started.

The first step is to create a package to hold the cursor definition and your procedure. Within the package, define a cursor type that contains the columns you want to send back to ColdFusion (if you are going to select all columns from a single table, consider using the special `%ROWTYPE` attribute, as shown in the next listing). Also within the package, provide a declaration for the stored procedure itself, which is basically the first line of the procedure without the words `CREATE OR REPLACE`.

Within the package body, create the stored procedure itself, using the `OPEN`, `FOR`, and `SELECT` keywords to open the cursor and fill it with the appropriate data.

Listing 42.13 shows how the `FetchRatingsList` stored procedure can be created on an Oracle server. You could issue this code from the Oracle SQL*Plus Worksheet application. Or you could provide the first part of the code (starting with the first `AS`) to the Oracle DBA Studio application in the Create Package dialog box, and the second part (starting with the second `AS`) to the Create Procedure dialog box.

Listing 42.13 Exposing Reference Cursors from Oracle Stored Procedures

```
CREATE OR REPLACE PACKAGE OWS.OWSWEB AS
 TYPE RatingsCurType IS REF CURSOR RETURN FilmsRatings%ROWTYPE;
 PROCEDURE FetchRatingsList(RatingsCur OUT RatingsCurType);
END owsWeb;
/
CREATE OR REPLACE PACKAGE BODY OWS.OWSWEB AS
```

Listing 42.13 (CONTINUED)

```
 PROCEDURE FetchRatingsList(RatingsCur OUT RatingsCurType) IS
 BEGIN
  OPEN RatingsCur FOR
  SELECT * FROM FilmsRatings ORDER BY RatingID;
 END FetchRatingsList;
END owsWeb;
/
```

NOTE

An Oracle stored procedure that returns multiple resultsets would simply include additional `IS REF CURSOR` lines in the package declaration and then include output parameters for each cursor type between the parentheses that follow the procedure name.

Creating Stored Procedures with Sybase

Creating stored procedures with Sybase is extremely similar to creating them with Microsoft SQL Server. This has a lot to do with the fact that SQL Server originally was developed as a joint effort between Microsoft and Sybase. Both products supported almost the same functionality until relatively recently. See your Sybase documentation for details.

CHAPTER 43

Using Regular Expressions

IN THIS CHAPTER

Introducing Regular Expressions E113
RegEx Support in ColdFusion E116
Using Regular Expressions in ColdFusion E117
Some Convenient RegEx UDFs E136
Building a RegEx Testing Page E138
Crafting Your Own Regular Expressions E144
Concluding Remarks E159

Introducing Regular Expressions

ColdFusion includes support for *regular expressions*. If you've worked at all with Perl, you probably know all about regular expressions because they are such a central part of Perl's string handling and manipulation capabilities, and generally walk hand in hand with the Perl language itself. As a rule, they aren't nearly as important to ColdFusion coders as they are to Perl coders, but regular expressions are still incredibly useful in ColdFusion development.

This chapter introduced you to regular expressions and explains how they can be used in ColdFusion applications.

What Are Regular Expressions?

Regular expressions are a way of looking for characters within chunks of text, using special wildcards to describe exactly what you're looking for. There are a lot of different wildcards you can use, from the simple `*` and `?` characters that you probably recognize from the DOS or Unix command line, to less common, more powerful wildcards that really only apply to regular expressions.

What Are Regular Expressions Similar To?

The analogy isn't perfect, but you can think of regular expressions as being kind of like `WHERE` statements in SQL, except that regular expressions are for querying plain text rather than database tables. Instead of specifying what records you want to find with a `WHERE` clause, you specify which characters you want to find using regular expressions.

Actually, the analogy works better if you think of regular expressions as being specifically analogous to a `SELECT` query that uses the `LIKE` keyword to search the database based on wildcards. You remember the `LIKE` keyword from SQL, don't you? It lets you select records using syntax such as:

```
SELECT * FROM Films WHERE Summary LIKE '%color%'
```

As you probably know, the database would respond to this query with all films that contain the word `color` in the summary. The `%` characters are behaving as wildcards; you can think of each `%` as being shorthand for saying "any amount of text." So you are asking the database to return all records where `Summary` includes any amount of text, followed by the word `color`, followed by any amount of text. SQL also lets you use sets of characters as wildcards, like this:

```
SELECT * FROM Films WHERE Summary LIKE '%[Pp]ress [0-9]%'
```

To this second query, the database would respond with all films where the summary contains the phrase `Press 1` or `Press 2` (or `Press 3`, and so on), using either a lowercase or uppercase `P`.

Even if you're not familiar with these SQL wildcards, you can see the basic idea. The various wildcard characters are used to describe what you're looking for. Regular expressions are really no different conceptually, except that there are lots of wildcards instead of only a few.

NOTE

Regular expression purists may shudder at the way I'm using the term "wildcard" here. Bear with me. We'll get to the nitty-gritty later.

At the risk of belaboring this introduction, and as I hinted in the first paragraph, you can also think of regular expressions as similar to the `*` and `?` wildcards that you use on the command line to find files. Again, as you probably know, MS-DOS and Windows command prompt lets you use commands like this:

```
c:\>dir P*.txt
```

This command finds all files in the current directory that start with `P` and that have a `.txt` extension. The `*` wildcard does the same thing here as the `%` wildcard does in SQL: It stands in for the idea of *any number of characters*.

So, you're already familiar with a couple of regular expression–like ways of using wildcards to find information. Now you just need to learn the specific wildcards you can use with regular expressions, and how to use them in your ColdFusion applications. That's what the rest of this chapter is all about.

What Are Regular Expressions Used For?

Within the context of ColdFusion applications, regular expressions are generally used for these purposes:

- **Pattern searching**. Regular expressions can be used as a kind of search utility that finds one or more *exact* occurrences of a pattern. By *pattern*, I mean a word, number, entire phrase, or any combination of characters, both printable and not. A match is successful when one or more occurrences of the pattern exist. You might use pattern searching to find all telephone numbers in a given paragraph of text, or all hyperlinks in a chunk of HTML.
- **Pattern testing**. Testing a pattern is a form of data validation, and an excellent one at that. The regular expression in this context is the rule, or set of rules, that your data conforms to in order to pass the test. You might use pattern testing to validate a user's form entries.

- **Pattern removal**. Pattern removal ensures data integrity by allowing you to search and remove unwanted or hazardous patterns within a block of text. Any string that causes complications within your application is hazardous. You might use pattern removal to remove all curse words, email addresses, or telephone numbers from a chunk of text, leaving the rest of the text alone.
- **Pattern replacement**. Functioning as a search-and-replace mechanism, pattern replacement allows you to find one or more occurrences of a pattern within a block of text and then replace it with a new pattern, parts of the original pattern, or a mixture of both. You might use pattern replacement to surround all email addresses in a block of text with a `mailto:` hyperlink so the user can click the address to send a message.

You'll see regular expressions being used for each of these purposes in this chapter's example listings.

What Do Regular Expressions Look Like?

Just so you can get a quick sense of what they look like, I'll show you some regular expressions now. Unless you've used regular expressions before, don't expect to understand these examples at this point. I'm showing them to you now just so you'll get an idea of how powerful the various wildcards are.

This regular expression matches the abbreviation CFML (each letter can be in upper- or lowercase, and each letter may or may not have a period after it):

```
[Cc]\.?[Ff]\.?[Mm]\.?[Ll]\.?
```

This regular expression matches any HTML tag (or, for that matter, a CFML, XML, or any other type of angle-bracketed tag):

```
<[^>]*>
```

This regular expression is one way of matching an email address:

```
([\w._]+)\@([\w_]+(\.[\w_]+)+)
```

Do Regular Expressions Differ Among Languages?

Yes. There are many tools and programming languages that provide regular expression functionality of one sort or another. Perl, JavaScript, grep/egrep, POSIX, and ColdFusion are just a few; there are plenty more. Over the years, some of the tools and languages have added their own extensions or improvements. Most of the basic regular expression wildcards will work in any of these tools, but other wildcards might work in Tool A but not in Tool B, or might have a slightly different meaning in Tool C. People often refer to the various levels of compatibility as "flavors" (the Perl flavor, the POSIX flavor, and so on).

You can think of these tweaks and flavors as resembling the various changes and improvements that have been made over the years to SQL, to the point where queries written for Access, Oracle, and Sybase databases might look considerably different (especially if the queries are doing something complicated). But that doesn't change the fact that they are all based on the same basic syntax; if you've learned one, you've basically learned them all.

NOTE

The term "regular expression" gets a bit tedious to read over and over again, so I will often use the term RegEx instead. It's a customary way to shorten the term.

RegEx Support in ColdFusion

Now that you have an idea of what regular expressions are, you need to understand what kind of support ColdFusion provides for them. The basic facts are these:

- The syntax you can use in your regular expressions (that is, the wildcards and such) is nearly identical to the syntax supported in Perl.
- In ColdFusion, you use the `reFind(), reMatch()` and `reReplace()` functions to perform a RegEx operation. This is in contrast to the way regular expressions are invoked in Perl or JavaScript, which allow you to sprinkle them throughout your code almost as if the expressions were ordinary strings.

Where Can You Use Regular Expressions?

You still haven't learned how to construct these strange-looking regular expression things, but assuming you have one of them already (such as the `<[^>]*>` or `([\w._]+)\@([\w_]+(\.[\w_]+)+)` expressions that I mentioned earlier), you might be wondering where you can use them. In Perl, you tell the engine that a string should be interpreted as a regular expression by delimiting it with `/` characters, optionally adding additional "switches" to control options such as case sensitivity. That wouldn't work so well in CFML, due to its tag-based nature. Instead, you use the special set of RegEx functions, listed in Table 43.1.

Table 43.1 ColdFusion's RegEx Functions

FUNCTION	DESCRIPTION
`reFind()`	Attempts to find a match for a regular expression within a block of text. It's similar conceptually to the normal `find()` function, except that the string you're looking for can include regular expression wildcards.
`reFindNoCase()`	Same as `reFind()`, except that the matching ignores capitalization.
`reMatch()`	Attempts to find as many matches in a string as possible. All matches are returned in an array.
`reMatchNoCase()`	Same as `reMatchNoCase()`, except that the matching ignores capitalization.
`reReplace()`	Finds matches within a block of text, replacing the matches with whatever replacement string you specify. You can use special characters in the replacement string to pull off all sorts of fancy replacement tricks.
`reReplaceNoCase()`	Same as `reReplace()`, except performing the matching without respect to capitalization.

Using Regular Expressions in ColdFusion

The next two portions of this chapter will teach you about two concepts:

- How to use CFML's RegEx functions (`reFind()` and the others listed in Table 43.1) to actually perform regular expression operations within your ColdFusion pages.
- How to craft the regular expression for a particular task, using the various RegEx wildcards available to you.

This is a kind of chicken-and-egg scenario for me. How can I explain how to incorporate regular expressions like `([\w._]+)\@([\w_]+(\.[\w_]+)+)` in your CFML code if you don't yet understand what all those wildcards mean? On the other hand, wouldn't it be pretty boring to learn about all the wildcards before knowing how to put them to use?

To put it another way, it's hard for me to guess what kind of learner you are, or how much you already know about regular expressions. If you don't know anything at all about them, you might want to learn about the various wildcards first. If you've already used them in other tools, you probably just want to know how to use them in ColdFusion. So feel free to skip ahead to the "Crafting Your Own Regular Expressions" section if you don't like looking at all these wildcards without understanding what they mean.

Finding Matches with `reFind()`

Assuming you have already crafted the wildcard-laden RegEx criteria you want, you can use the `reFind()` function to tell ColdFusion to search a chunk of text with the criteria, like this:

```
reFind(regex, string [, start] [, returnSubExpressions] )
```

Table 43.2 describes each of the `reFind()` arguments.

Table 43.2 `reFind()` Function Syntax

ARGUMENT	DESCRIPTION
`regex`	Required. The regular expression that describes the text that you want to find.
`string`	Required. The text that you want to search.
`start`	Optional. The starting position for the search. The default is `1`, meaning that the entire string is searched. If you provide a start value of `50`, then only the portion of the string after the first 49 characters is searched.
`returnSubExpressions`	Optional. A Boolean value indicating whether you want to obtain information about the position and length of the actual text that was found by the various portions of the regular expression. The default is False. You will learn more about this topic in the section "Getting the Matched Text Using `returnSubExpressions`" later in this chapter.

The `reFind()` function returns one of two things, depending on whether the `returnSubExpressions` argument is True or False:

- Assuming that `returnSubExpressions` is False (the default), the function returns the character position of the text that's found (that is, the first substring that matches the search criteria). If no match is found in the text, the function returns 0 (zero). This behavior is consistent with the ordinary, non-RegEx `find()` function.
- If `returnSubExpressions` is True, the function returns a CFML structure composed of two arrays called `pos` and `len`. These arrays contain the position and length of the first substring that matches the search criteria. The first value in the arrays (that is, `pos[1]` and `len[1]`) correspond to the match as a whole. The remaining values in the arrays correspond to any subexpressions defined by the regular expression.

The bit about the subexpressions might be confusing at this point, since you haven't learned what subexpressions actually are. Don't worry about it for the moment. Just think of the subexpressions argument as something you should set to True if you need to get the actual text that was found.

A Simple Example

For the moment, accept it on faith that the following regular expression will find a sensibly formed Internet email address (such as `nate@nateweiss.com` or `nate@nateweiss.co.uk`):

```
([\w._]+)\@([\w_]+(\.[\w_]+)+)
```

Listing 43.1 shows how to use this regular expression to find an email address within a chunk of text.

Listing 43.1 `RegExFindEmail1.cfm`—A Simple Regular Expression Example

```
<!---
 Filename: RegExFindEmail1.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates basic use of reFind()
--->

<html>
<head><title>Using a Regular Expression</title></head>
<body>

<!--- The text to search --->
<cfset text = "My email address is nate@nateweiss.com. Write to me anytime.">

<!--- Attempt to find a match --->
<cfset foundPos = reFind("([\w._]+)@([\w_]+(\.[\w_]+)+)", text)>

<!--- Display the result --->
<cfif foundPos gt 0>
 <cfoutput>
 <p>A match was found at position #foundPos#.</p>
 </cfoutput>
<cfelse>
 <p>No matches were found.</p>
```

Listing 43.1 (continued)

```
</cfif>

</body>
</html>
```

If you visit this page with your browser, the character position of the email address is displayed (Figure 43.1). If you change the `text` variable so that it no longer contains an Internet-style email address, the listing displays “No matches were found.”

Figure 43.1

Regular expressions can search for email addresses, phone numbers, and the like.

Ignoring Capitalization with `reFindNoCase()`

Internet email addresses aren’t generally considered to be case-sensitive, so you might want to tell ColdFusion to perform the match without respect to case. To do so, use `reFindNoCase()` instead of `reFind()`. Both functions take the same arguments and are used in exactly the same way, so there’s no need to provide a separate example listing for `reFindNoCase()`.

In short, anywhere you see `reFind()` in this chapter, you could use `reFindNoCase()` instead, and vice-versa. Just use the one that’s appropriate for the task at hand. Also, note that it is possible to use case-insensitive regular expressions, making `reFindNoCase()` unnecessary.

Getting the Matched Text Using the Found Position

Sometimes you just want to find out whether a match exists within a chunk of text. In such a case, you would use the `reFind()` function as it was used in Listing 43.1.

You can also use that form of `reFind()` if the nature of the RegEx is such that the actual match will always have the same length. For instance, if you were searching specifically for a U.S. telephone number in the form (999)999-9999 (where each of the 9s represents a number), you could use the following regular expression:

```
\([0-9]{3}\)[0-9]{3}-[0-9]{4}
```

Because the length of a matched phone number will always be the same due to the nature of phone numbers, it’s a simple matter to extract the actual phone number that was found. You use ColdFusion’s

built-in `mid()` function, feeding it the position returned by the `reFind()` function (as shown in Figure 13.1) as the start position, and the number `13` as the length.

Listing 43.2 puts these concepts together, displaying the actual phone number found in `text` (Figure 43.2).

Figure 43.2
If you know its length ahead of time, it's easy to display the matched text.

Listing 43.2 `RegExFindPhone1.cfm`—Using `mid()` to Extract the Matched Text

```
<!---
 Filename: RegExFindPhone1.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates basic use of reFind()
--->

<html>
<head><title>Using a Regular Expression</title></head>
<body>

<!--- The text to search --->
<cfset text = "My phone number is (718)555-1212. Call me anytime.">
<!--- Attempt to find a match --->
<cfset matchPos = reFind("(\([0-9]{3}\))([0-9]{3}-[0-9]{4})", text)>

<!--- Display the result --->
<cfif matchPos gt 0>
 <cfset foundString = mid(text, matchPos, 13)>

 <cfoutput>
 <p>A match was found at position #matchPos#.</p>
 <p>The actual match is: #foundString#</p>
 </cfoutput>
<cfelse>
 <p>No matches were found.</p>
</cfif>

</body>
</html>
```

Getting the Matched Text Using `returnSubExpressions`

If you want to adjust the email address example in Listing 43.1 so that it displays the actual email address found, the task is a bit more complicated because not all email addresses are the same length. What would you supply to the third argument of the `mid()` function? You can't use a constant number in the manner shown in Listing 43.2. Clearly, you need some way of telling `reFind()` to return the length, in addition to the position, of the match.

This is when the `returnSubExpressions` argument comes into play. If you set this argument to True when you use `reFind()`, the function will return a structure that contains the position and length of the match. (The structure also includes the position and length that correspond to any subexpressions in the structure, but don't worry about that right now.)

Listing 43.3 shows how to use this parameter of the `reFind()` function. It uses the first element in `pos` and `len` arrays to determine the position and length of the matched text and then displays the match (Figure 43.3).

Figure 43.3

It's easy to display a matched substring, even if its length will vary at run time.

Listing 43.3 `RegExFindEmail2.cfm`—Using `reFind()`'s `returnSubExpressions` Argument

```
<!---
 Filename: RegExFindEmail2.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates basic use of REFind()
--->

<html>
<head><title>Using a Regular Expression</title></head>
<body>

<!--- The text to search --->
<cfset text = "My email address is nate@nateweiss.com. Write to me anytime.">

<!--- Attempt to find a match --->
<cfset matchStruct = reFind("([\w._]+)\@([\w_]+(\.[\w_]+)+)", text, 1, True)>

<!--- Display the result --->
<cfif matchStruct.pos[1] gt 0>
 <cfset foundString = mid(text, matchStruct.pos[1], matchStruct.len[1])>
```

Listing 43.3 (continued)

```
     <cfoutput>
     <p>A match was found at position #matchStruct.pos[1]#.</p>
     <p>The actual match is: #foundString#</p>
     </cfoutput>
    <cfelse>
     <p>No matches were found.</p>
    </cfif>

    </body>
    </html>
```

Working with Subexpressions

As exhibited by the last example, the first values in the `pos` and `len` arrays correspond to the position and length of the match found by the `reFind()` function. Those values (`pos[1]` and `len[1]`) will always exist. So why are `pos` and `len` implemented as arrays if the first value in each is the only interesting value? What other information do they hold?

The answer is this: If your regular expression contains any *subexpressions*, there will be an additional value in the `pos` and `len` arrays that corresponds to the actual text matched by the subexpression. If your regular expression has two subexpressions, `pos[2]` and `len[2]` are the position and length of the first subexpression's match, and `pos[3]` and `len[3]` are the position and length for the second subexpression.

So, what's a subexpression? When you are using regular expressions to solve specific problems (such as finding email addresses or phone numbers in a chunk of text), you are often looking for several different patterns of text, one after another. That is, the nature of the problem is often such that the regular expression is made up of several *parts* ("look for this, followed by that"), where all of the parts must be found in order for the whole regular expression to be satisfied. If you place parentheses around each of the parts, the parts become subexpressions.

Subexpressions do two things:

- They make the overall RegEx criteria more flexible, because you can use many regular expression wildcards on each subexpression. This capability allows you to say that some subexpressions must be found while others are optional, or that a particular subexpression can be repeated multiple times, and so on. To put it another way, the parentheses allow you to work with the enclosed characters or wildcards as an isolated group. This isn't so different conceptually from the way parentheses work in `<cfif>` statements or SQL criteria.
- The match for each subexpression is included in the `len` and `pos` arrays, so you can easily find out what specific text was actually matched by each part of your RegEx criteria. You get position and length information not only for the match as a whole, but for each of its constituent parts.

TIP

If you don't want a particular set of parentheses, or subexpressions, to be included in the `len` and `pos` arrays (that is, if you are only interested in the grouping properties of the parentheses and not in their returning-the-match properties), you can put a `?:` right after the opening parenthesis. See Table 43.11 near the end of this chapter for details.

In real-world use, most regular expressions contain subexpressions—it's the nature of the beast. In fact, each of the regular expressions in the example listings shown so far has included subexpressions because the problems they are trying to solve (finding email addresses and phone numbers) require that they look for strings that consist of a few different parts.

Take a look at the regular expression used in Listing 43.3, which matches email addresses:

```
([\w._]+)@([\w_]+(\.[\w_]+)+)
```

I know you haven't learned what all the wildcards mean yet; for now, just concentrate on the parentheses. It may help you to keep in mind that the plain-English meaning of each of the `[\w_]+` sequences is "match one or more letters, numbers, or underscores."

By concentrating on the parentheses, you can easily recognize the three subexpressions in this RegEx. The first is at the beginning and matches the portion of the email address up to the `@` sign. The second subexpression begins after the `@` sign and continues to the end of the RegEx; it matches the "domain name" portion of the email address. Within this second subexpression is a third one, which says that the domain name portion of the email address can contain any number of subparts (but at least one), where each subpart is made up of a dot and some letters (such as `.com` or `.uk`).

Now take a look at the RegEx from Listing 43.2, which matches phone numbers:

```
(\([0-9]{3}\))([0-9]{3}-[0-9]{4})
```

This one has two subexpressions. You might have thought it has three because there appear to be three sets of parentheses. But the parentheses characters that are preceded by backslash characters don't count, because the backslash is a special escape character that tells the RegEx engine to treat the next character literally. Here, the backslashes tell ColdFusion to look for actual parentheses in the text, rather than treating those parentheses as delimiters for subexpressions.

So the phone number example includes just two subexpressions. The first subexpression starts at the very beginning and ends just after the `\)` characters and it matches the area code portion of the phone number. The second subexpression contains the remainder of the phone number (three numbers followed by a hyphen, then four more numbers). See Listing 43.4.

Listing 43.4 `RegExFindEmail3.cfm`—Getting the Matched Text for Each Subexpression

```
<!---
 Filename: RegExFindEmail3.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates basic use of REFind()
--->

<html>
<head><title>Using a Regular Expression</title></head>
```

Listing 43.4 (continued)

```
<body>

<!--- The text to search --->
<cfset text = "My email address is nate@nateweiss.com. Write to me anytime.">

<!--- Attempt to find a match --->
<cfset matchStruct = reFind("([\w._]+)@([\w_]+(\.[\w_]+)+)", text, 1, True)>

<!--- Display the result --->
<cfif matchStruct.pos[1] gt 0>

 <!--- The first elements of the arrays represent the overall match --->
 <cfset foundString = mid(text, matchStruct.pos[1], matchStruct.len[1])>
 <!--- The subsequent elements represent each of the subexpressions --->
 <cfset userNamePart = mid(text, matchStruct.pos[2], matchStruct.len[2])>
 <cfset domainPart = mid(text, matchStruct.pos[3], matchStruct.len[3])>
 <cfset suffixPart = mid(text, matchStruct.pos[4], matchStruct.len[4])>

 <cfoutput>
 <p>A match was found at position #matchStruct.pos[1]#.<br>
 The actual email address is: <b>#foundString#</b><br>
 The username part of the address is: #userNamePart#<br>
 The domain part of the address is: #domainPart#<br>
 The suffix part of the address is: #suffixPart#<br>
 </p>
 </cfoutput>
<cfelse>
 <p>No matches were found.</p>
</cfif>

</body>
</html>
```

This listing is similar to the preceding one (Listing 43.3), except that instead of working with only the first values in the `pos` and `len` arrays, Listing 43.4 also works with the second, third, and fourth values. It displays the username, domain name, and domain suffix portions of the match, respectively (Figure 43.4).

Figure 43.4

Subexpressions are handy for matching portions of a RegEx.

TIP

If you need to know the number of subexpressions in a RegEx, you can use `arrayLen()` with either the `pos` or `len` array and then subtract 1 from the result (because the first values of the array is for the match as a whole). In Listing 43.4, you could output the value of `arrayLen(MatchStruct.pos)-1` to find the number of subexpressions in the email RegEx (the answer would be 3).

Working with Multiple Matches

So far, this chapter's listings have shown you how to find the first match in a given chunk of text. Often, that's all you need to do. There are times, however, when you might need to match multiple phone numbers, email addresses, or something else.

The `reFind()` and `reFindNoCase()` functions don't specifically provide any means to find multiple matches at once, but you can use the `start` argument mentioned in Table 43.3 to achieve the same result. This is a manual process that is now much easier with `reMatch()` and `reMatchAll()`. But if you want to do it the older, more manual way, listing 43.5 shows an example

Listing 43.5 `RegExFindEmail4.cfm`—Finding Multiple Matches with a `<cfloop>` Block

```
<!---
 Filename: RegExFindEmail4.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates basic use of REFind()
--->

<html>
<head><title>Using a Regular Expression</title></head>
<body>

<!--- The text to search --->
<cfset text = "My email address is nate@nateweiss.com. Write to me anytime. "
 & "You can also use nate@nateweiss.co.uk or Weiss_Nate@nateweiss.com.">

<!--- Start at the beginning of the text --->
<cfset startPos = 1>

<!--- Continue looping indefinitely (until a <CFBREAK> is encountered) --->
<cfloop condition="True">

 <!--- Attempt to find a match --->
 <cfset matchStruct =
 reFind("([\w._]+)@([\w_]+(\.[\w_]+)+)", text, startPos, True)>

 <!--- Break out of the loop if no match was found --->
 <cfif matchStruct.pos[1] eq 0>
  <cfbreak>

 <!--- Otherwise, display the match --->
 <cfelse>
   <!--- Advance the StartPos so the next iteration finds the next match --->
   <cfset startPos = matchStruct.pos[1] + matchStruct.len[1]>
```

Listing 43.5 (continued)

```
      <!--- The first elements of the arrays represent the overall match --->
      <cfset foundString = mid(text, matchStruct.pos[1], matchStruct.len[1])>
      <!--- The subsequent elements represent each of the subexpressions --->
      <cfset userNamePart = mid(text, matchStruct.pos[2], matchStruct.len[2])>
      <cfset domainPart = mid(text, matchStruct.pos[3], matchStruct.len[3])>
      <cfset suffixPart = mid(text, matchStruct.pos[4], matchStruct.len[4])>

      <cfoutput>
      <p>A match was found at position #matchStruct.pos[1]#.<br>
      The actual email address is: <b>#foundString#</B><BR>
      The username part of the address is: #userNamePart#<br>
      The domain part of the address is: #domainPart#<br>
      The suffix part of the address is: #suffixPart#<br>
      </cfoutput>
    </cfif>

  </cfloop>

  </body>
  </html>
```

The key difference between this listing and the preceding one is the addition of the `startPos` variable and the `<cfloop>` tags that now surround most of the code. (The loop uses a `condition="True"` attribute that causes the block to loop forever unless a `<cfbreak>` tag is encountered).

At the beginning, `startPos` is set to 1. Then, within the loop, `startPos` is fed to the `reFind()` function, meaning that the first iteration of the loop will find matches starting from the beginning of the text. If no match is found, `<cfbreak>` is used to break out of the loop. Otherwise, the `pos[1]` and `len[1]` values are combined to set `startPos` to the character position immediately following the match.

So, if the first match is found at position 50 and is 15 characters long, the next iteration of the loop will use a `startPos` of 65, thereby finding the next match (if any) in the `text`. The process will repeat until no match is found after `startPos`, at which point the `<cfbreak>` kicks in to end the loop. The result is a simple page that finds and displays multiple email addresses (Figure 43.5).

CAUTION

> Be careful when you use `<cfloop>` tags that use `condition="True"` in this manner. If your code doesn't include a `<cfbreak>` that is guaranteed to execute at some point, your loop will go on forever, occupying more and more of ColdFusion's time and resources. You would probably need to restart the server as a result.

As stated above, it is even easier now in ColdFusion 8 to find multiple matches using `reMatch` and `reMatchAll`. While `reFind` only returns the matching position of a RegEx, or the position and length of all the matches, the `reMatch` function can simply return all the matches at once. Listing 43.6 shows an example.

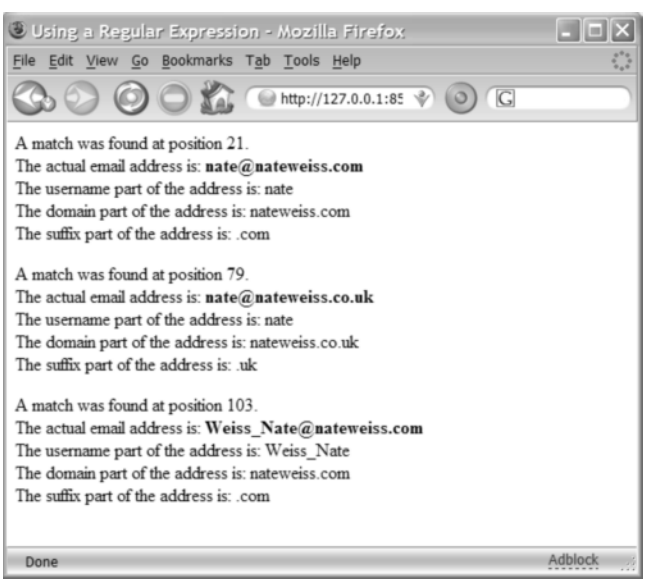


Figure 43.5

Using simple loops, you can easily find multiple matches.

Listing 43.6 `RegExFindEmail5.cfm`—Finding Multiple Matches with a `reMatch()` Call

```
<!--- The Text to Search --->
<cfsavecontent variable="text">
Here is text with names and email address.
Ray's email is ray@camdenfamily.com,
Todd's email address is todd@cfsilence.com.
Lastly, Luke Skywalker's email address is luke@newalliance.gov.
</cfsavecontent>

<!--- Find all matches. --->
<cfset matches = reMatch("([\w._]+)@([\w_]+(\.[\w_]+)+)", text)>

<cfoutput>
<p>
There were #arrayLen(matches)# matches.
</p>

<p>
Matches:<br />
</cfoutput>

<cfloop index="match" array="#matches#">
  <cfoutput>#match#<br /></cfoutput>
</cfloop>
```

NOTE

The template begins by creating a variable from a block of text using the `cfsavecontent` tag. This is a handy way to quickly create a large block of text as a variable. Next the `reMatch` function is called using the email RegEx against the text variable. The size of the resultant array is then displayed along with a list of all the matches. Note the use of the new array attribute to `cfloop` to make it easier to loop over an array. This method of grabbing a set of matches is much easier than listing 43.5.

Replacing Text using `reReplace()`

As you learned from Table 43.1, ColdFusion provides `reReplace()` and `reReplaceNoCase()` functions in addition to the functions you've seen so far.

The `reReplace()` and `reReplaceNoCase()` functions each take three required arguments and one optional argument, as follows:

```
reReplace(string, regex, substring [, scope ])
```

The meaning of each argument is explained in Table 43.3.

Table 43.3 `reReplace()` Function Syntax

ARGUMENT	DESCRIPTION
`string`	Required. The string in which you want to find matches.
`regex`	Required. The regular expression criteria you want to use to find matches.
`substring`	Required. The string that you want each match to be replaced with. You can use backreferences in the string to include pieces of the original match in the replacement.
`scope`	Optional. The default is `ONE`, which means that only the first match is replaced. You can also set this argument to `ALL`, which will cause all matches to be replaced.

The function returns the altered version of the string (the original string is not modified). Think of it as being like the `replace()` function on steroids, since the text you're looking for can be expressed using RegEx wildcards instead of a literal substring.

NOTE

The syntax for both `reReplace()` and `reReplaceNoCase()` is the same. Anywhere you see one, you could use the other. Just use the function that's appropriate for the task, depending on how you want the replacement operation to behave in regard to capitalization. Again, though, do not forget that you can actually do case-insensitive regular-expression matching, so you need not ever use `reReplaceNoCase()`.

Using `reReplace` to Filter Posted Content

The next few examples will implement an editable home page for the fictitious Orange Whip Studios company. The basic idea is for the application to maintain a text message in the `APPLICATION` scope; this message appears on the home page. An edit link allows the user to type a new message in a simple form (Figure 43.6). When the form is submitted, the new message is displayed on the home page from that point forward (Figure 43.7). Listing 43.7 shows the simple logic for this example.

Figure 43.6
Users can edit the home page message with this simple form.

Figure 43.7
Regular expressions can be used to filter what gets displayed on the home page.

Listing 43.7 `EditableHomePage1.cfm`—Removing Text Based on a Regular Expression

```
<!---
 Filename: EditableHomePage1.cfm
 Author: Nate Weiss (NMW)
 Purpose: Example of altering text with regular expressions
--->

<!--- Enable application variables --->
<cfapplication name="OrangeWhipIntranet">

<!--- Declare the HomePage variables and give them initial values --->
<cfparam name="APPLICATION.homePage" default="#structNew()#">
<cfparam name="APPLICATION.homePage.messageAsPosted" type="string" default="">
<CFPARAM NAME="APPLICATION.homePage.messageToDisplay" type="string" default="">

<!--- If the user is submitting an edited message --->
<cfif isDefined("FORM.messageText")>

 <!--- First of all, remove all tags from the posted message --->
 <cfset messageWithoutTags = reReplace(FORM.messageText,
 "<[^>]*>", <!--- (matches tags) --->
 "", <!--- (replace with empty string) --->
 "ALL")>

 <!--- Save the "before" version of the new message --->
 <cfset APPLICATION.homePage.messageAsPosted = messageWithoutTags>

 <!---
 (other code will be added here in following examples)
```

Listing 43.7 (continued)

```
 --->

 <!--- Save the "after" version of the new message --->
 <cfset APPLICATION.homePage.messageToDisplay = messageWithoutTags>
</cfif>

<!--- This include file takes care of dispaying the actual page --->
<!--- (including the message) or the form for editing the message --->
<cfinclude template="EditableHomePageDisplay.cfm">
```

At the top of this listing, three application variables called `homepage`, `homePage.messageAsPosted`, and `HomePage.messageToDisplay` are established. If the user is currently posting a new message, the `<cfif>` block executes. This block is responsible for saving the edited message. Inside the `<cfif>` block, the `reReplace()` function is used to find all HTML (or XML, CFML, or any other type of tag) and replace the tags with an empty string. In other words, all tags are removed from the user's message in order to prevent users from entering HTML that might look bad or generally mess things up.

NOTE

Once again, you have to take it on faith that the `<[^>]*>` regular expression used in this example is an appropriate one to use for removing tags from a chunk of text. For details, see the section "Crafting Your Own Regular Expressions" in this chapter.

Once the tags have been removed, the resulting text is saved to the `homePage.messageAsPosted` and `homePage.messageToDisplay` variables, which will be displayed by the next listing. For now, the two variables will always hold the same value, but you will see a few different versions of this listing that save slightly different values in each.

Finally, a `<cfinclude>` tag is used to include the `EditableHomePageDisplay.cfm` template, shown in Listing 43.8. This code is responsible for displaying the message on the home page (as shown in Figure 43.6) or displaying the edit form (as shown in Figure 43.8) if the user clicks the edit link.

Listing 43.8 `EditableHomePageDisplay.cfm`—Form and Display Portion of Editable Home Page

```
<!---
 Filename: EditableHomePageDisplay.cfm
 Author: Nate Weiss (NMW)
 Please Note Included by the EditableHomePage.cfm examples
--->


<html>
<head><title>Orange Whip Studios Home Page</title></head>
<body>
<cfoutput>
 <!--- Orange Whip Studios logo and page title --->
 <img src="../images/logo_c.gif" width="101" height="101" alt="" align="absmiddle">
 <b>Orange Whip Studio Home Page</b><br clear="all">

 <!--- Assuming that the user is not trying to edit the page --->
 <cfif not isDefined("URL.edit")>
```

Listing 43.8 (continued)

```
      <!--- Display the home page message --->
      <p>#paragraphFormat(APPLICATION.homePage.messageToDisplay)#

      <!--- Provide a link to edit the message --->
      <p>[<a href="#CGI.script_name#?edit=Yes">edit message</a>]</p>

    <!--- If the user wants to edit the page --->
    <cfelse>

      <!--- Simple form to edit the home page message --->
      <form action="#CGI.script_name#" method="post">

      <!--- Text area for typing the new message --->
      <textarea
      name="messageText"
      cols="60"
      rows="10">#htmlEditFormat(APPLICATION.homePage.messageAsPosted)#</textarea><br>

      <!--- Submit button to save the message --->
      <input
      type="submit"
      value="Save Text">
      </form>

    </cfif>
   </cfoutput>

   </body>
   </html>
```

There is nothing particularly interesting about this listing. It's a simple file that either displays the home page or an edit form, as appropriate. Note that the `homePage.messageToDisplay` is what is normally displayed on the home page, whereas `homePage.messageAsPosted` is what appears in the edit form. Right now, these two values are always the same, but subsequent versions of Listing 43.7 will change that.

Clearly, you aren't limited to only removing the tags; you can replace them with any string you want. If you wanted the user to get a visual cue about the removal of any tags from the message, you could change the third argument of the `reReplace()` function so that the tags are replaced with a message such as `[tags removed]`. And in the next section, you'll learn how to use the RegEx backreference wildcard so that the actual match can be incorporated into the replacement string dynamically.

NOTE

Of course, in a real application you wouldn't allow just anyone to edit the message on the home page. At a minimum, you would require a username and password to make sure that only the proper people had access to the edit form.

Altering Text with Backreferences

Listing 43.7 showed you how to use `reReplace()` to replace any matches for a regular expression with a replacement string (in that example, the replacement was an empty string). Using a simple

replacement string is fine when you want to remove matches from a chunk of text, or to replace all matches with the same replacement string.

But what if you want the replacements to be more flexible, so that the replaced text is based somehow on the actual match? The `reReplace()` function supports *backreferences*, which allow you to do just that. A backreference is a special RegEx wildcard that can be used in the replacement string to represent the actual value of a subexpression. Backreferences are commonly used to alter or reformat the substrings matched by a regular expression.

In ColdFusion, you include backreferences in your replacement strings using `\1`, `\2`, `\3`, and so on, where the number after the backslash indicates the number of a subexpression. If your replacement string contains a `\1`, the actual value matched by the first subexpression (that is, the first parenthesized part of the RegEx) will appear in place of the `\1`. If the replacement includes `\2`, the result will have the value of the second subexpression in place of the `\2`, and so on.

TIP

Think of backreferences as a special kind of variable. For each actual match, these special variables are filled with the values of each subexpression that contributed to the match. The replacement is then made using the values of the special variables. The process is repeated for each match.

The next example listing is a new version of the earlier code (Listing 43.7) for tweaking the home page message submitted by users. This version uses backreferences to make two additional changes to the message posted by the user:

- "Malformed" phone numbers are rearranged so that the area code appears in parentheses, in the form `(999)999-9999`. If the user enters a phone number as `800/555-1212` or `800 555 1212`, it will be rearranged to read `(800)555-1212`.
- Any email addresses in the text will be surrounded by "mailto" hyperlinks that activate the user's email client when clicked. If `bfoxile@orangewhipstudios.com` is found in the text, it will be changed to an `<a>` link that includes an `href="mailto:bfoxile@orangewhipstudios.com"` attribute.

The user can type a message that contains phone numbers and email addresses (Figure 43.8); the home page will display a version of the message that has been altered in a reasonably intelligent and consistent fashion (Figure 43.9). Listing 43.9 shows the code for this new version of the home page example.

Figure 43.8

Regular expressions are used to scan for phone numbers and email addresses.

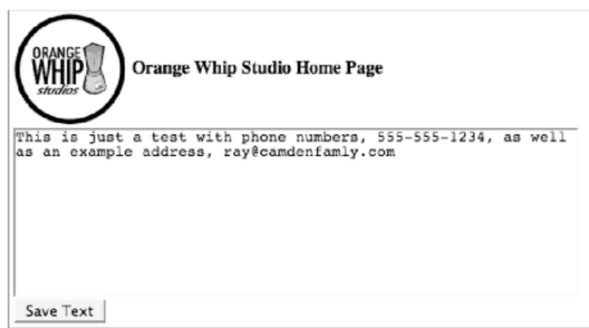
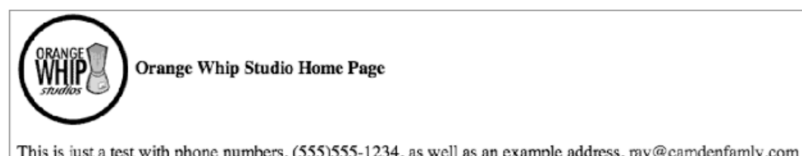


Figure 43.9

The phone numbers and email addresses are reformatted using RegEx backreferences.



Listing 43.9 `EditableHomePage2.cfm`—Using Backreferences to Make Intelligent Alterations

```
<!---
 Filename: EditableHomePage2.cfm
 Author: Nate Weiss (NMW)
 Purpose: Example of altering text with regular expressions
--->

<!--- Enable application variables --->
<cfapplication name="OrangeWhipIntranet">

<!--- Declare the HomePage variables and give them initial values --->
<cfparam name="APPLICATION.HomePage" default="#structNew()#">
<cfparam name="APPLICATION.homePage.messageAsPosted" type="string" default="">
<cfparam name="APPLICATION.homePage.messageToDisplay" type="string" default="">

<!--- If the user is submitting an edited message --->
<cfif isDefined("FORM.messageText")>

 <!--- First of all, remove all tags from the posted message --->
 <cfset FORM.messageText = reReplace(FORM.messageText,
 "<[^>]*>", <!--- (matches tags) --->
 "", <!--- (replace with empty string) --->
 "ALL")>

 <!--- Save the "before" version of the new message --->
 <cfset APPLICATION.homePage.messageAsPosted = FORM.messageText>

 <!--- Format any lazily-typed phone numbers in (999)999-999 format --->
 <cfset FORM.MessageText = reReplaceNoCase(FORM.messageText,
 "([0-9]{3})[-/ ]([0-9]{3})[- ]([0-9]{4})", <!--- (matches phone) --->
 "(\1)\2-\3", <!--- (phone format) --->
 "ALL")>

 <!--- Surround all email addresses with "mailto" links --->
 <cfset FORM.messageText = reReplaceNoCase(FORM.messageText,
 "(([\w._]+)@([\w_]+(\.[\w_]+)+))", <!--- (matches email addresses) --->
 "<a href=mailto:\1>\1</a>", <!--- (email address in link) --->
 "ALL")>

 <!--- Save the "after" version of the new message --->
 <cfset APPLICATION.homePage.messageToDisplay = FORM.messageText>
</cfif>

<!--- This include file takes care of dispaying the actual page --->
<!--- (including the message) or the form for editing the message --->
<cfinclude template="EditableHomePageDisplay.cfm">
```

Much of this listing is unchanged from the version in Listing 43.7. The difference is the addition of the second and third uses of `reReplace()` (the first `reReplace()` was in the previous version).

The second `reReplace()` is the one that reformats the phone numbers. This function contains three parenthesized subexpressions (which correspond to the area code, exchange, and last four digits of the phone number, respectively). Therefore, the `\1` in the replacement string will contain the area code when an actual match is encountered, the `\2` will contain the exchange portion of the phone number, and so on.

The final `reReplace()` does something similar except for email addresses. This replacement is interested in working only with the match as a whole, so an additional set of parentheses have been added around the entire regular expression, so that the entire RegEx is considered a subexpression. Therefore, the entire match will appear in place of the `\1` in the replacement string when this code executes. This is different from the behavior of `reFind` and `reFindNoCase` where `returnSubExpressions` is true. These functions will return the entire match automatically. An alternative is to omit the extra set of parentheses and refer to each part of the email address separately in the replacement string, like so:

```
<!--- Surround all email addresses with "mailto" links --->
<cfset FORM.messageText = reReplaceNoCase( FORM.messageText,
"([\w._]+)@([\w_]+(\.[\w_]+)+)", <!--- (matches email addresses) --->
"<a href=mailto:\1\@2\3>\1\@2\3</a>", <!--- (email address in link) --->
"ALL")>
```

NOTE

In Perl, you use `$1`, `$2`, and so on, rather than `\1` and `\2`, because the `$` is special to Perl.

NOTE

You can also use backreferences in the regular expression itself, often to match repeating patterns. For details, see the "Metacharacters 303: Backreferences Redux" section near the end of this chapter.

Altering Text Using a Loop

Sometimes you might want to make changes that are too complex to be made with a `reReplace()`, even using backreferences. In such a situation, you can use `reFind()` in its `returnSubEexpressions` form to loop over the matches (Listing 43.5), altering the original chunk of text as you go.

Listing 43.10 shows another distillation of the editable home-page logic. This code is similar to the last version (Listing 43.9), except that it now performs the replacement in a more manual fashion.

Listing 43.10 `editableHomePage3.cfm`—Making Changes Based on `reFind()` Results

```
<!---
 Filename: EditableHomePage3.cfm
 Author: Nate Weiss (NMW)
 Purpose: Example of altering text with regular expressions
--->

<!--- Enable application variables --->
<cfapplication name="OrangeWhipIntranet">
```

Listing 43.10 (continued)

```
<!--- Declare the HomePage variables and give them initial values --->
<cfparam name="APPLICATION.homePage" default="#structNew()#">
<cfparam name="APPLICATION.homePage.messageAsPosted" type="string" default="">
<cfparam name="APPLICATION.homePage.messageToDisplay" type="string" default="">

<!--- If the user is submitting an edited message --->
<cfif isDefined("FORM.messageText")>

 <!--- First of all, remove all tags from the posted message --->
 <cfset FORM.messageText = reReplace(FORM.messageText,
 "<[^>]*>", <!--- (matches tags) --->
 "", <!--- (replace with empty string) --->
 "ALL")>

 <!--- Save the "before" version of the new message --->
 <cfset APPLICATION.homePage.messageAsPosted = FORM.messageText>

 <!--- Find all email addresses --->
 <cfset matches = reMatchNoCase("([\w._]+)@([\w_]+(\.[\w_]+)+)",
   FORM.messageText)>

 <!--- loop through results --->
 <cfloop index="match" array="#matches#">

     <!--- Try to find email address in the database --->
     <cfquery name="emailQuery" datasource="ows">
     SELECT FirstName, LastName
     FROM Contacts
     WHERE EMail = '#match#'
     </cfquery>

     <!--- If the email address was found in the database --->
     <cfif emailQuery.recordCount eq 1>
       <cfset linkText = '<a href="mailto:#match#">'
       & "#emailQuery.FirstName# #emailQuery.LastName#</a>">

     <!--- If it was not found --->
     <cfelse>

       <cfset linkText =
       '<a href="mailto:#match#">#match#</a>'>

     </cfif>

  <!--- Replace the email --->
     <cfset FORM.messageText = replaceNoCase(form.messageText,
  match, linkText)>

 </cfloop>

 <!--- Save the "after" version of the new message --->
 <cfset APPLICATION.homePage.messageToDisplay = FORM.messageText>

</cfif>
```

Listing 43.10 (continued)

```
<!--- This include file takes care of dispaying the actual page --->
<!--- (including the message) or the form for editing the message --->
<cfinclude template="EditableHomePageDisplay.cfm">
.
```

Within the loop, this version of the code checks each email address to see if it's in the Contacts table of the OWS example database. If so, the portion of the mailto link between the <a> tags will show the person's first and last names, rather than just the email address (Figure 43.10). If you want to test this feature, be sure to use ben@forta.com in your text.

Figure 43.10

Ben's email address is in the database, but Raymond's isn't.

This is just a test with phone numbers, 555-555-1234, as well as two email addresses: Ben Forta and ray@camdenfamily.com

[edit message]

Some Convenient RegEx UDFs

The listings for this chapter include a file called `RegExFunctions.cfm`, which creates several user-defined functions (UDFs) that might come in handy when you're working with regular expressions. To use the library, simply `<cfinclude>` it in your own templates. Table 43.4 lists the functions included in this simple UDF library.

Table 43.4 Functions in UDF Library `RegExFunctions.cfm`

FUNCTION	DESCRIPTION
`reFindString()`	Performs a regular expression match and returns the matched string. Returns an empty string if no match is found. This is a shortcut for using the `pos[1]` and `len[1]` values as shown in Listing 43.3.
`adjustNewlinesToLinefeeds()`	Replaces any CRLF or CR sequences in a chunk of text with LF characters. This is handy when using multiline mode with `(?m)`. Accepts just one argument, `str`, as shown in the section "Understanding Multiline Mode" in this chapter.

Using `reFindString()`

The `reFindString()` UDF function takes two required arguments and two optional arguments:

```
reFindString(regex, string [, start] [, casesensitive])
```

The required `regex` and `string` arguments are the regular expression to use, and the chunk of text to search, respectively. The optional `start` argument is the text position at which to start the search (the default is 1). The optional `casesensitive` argument is a Boolean value that indicates whether the search should be case-sensitive (the default is `False` for no case-sensitivity).

The function returns the matched string. If no match is found, it returns an empty string. Listing 43.11 is a simple example that shows how the function might be used. It is similar to Listing 43.3, except that it uses the UDF function instead of `reFind()`.

Listing 43.11 `RegExFindEmail2a.cfm`—Using the `reFindString()` Function from the UDF Library

```
<!---
 Filename: RegExFindEmail2a.cfm
 Author: Nate Weiss (NMW)
 Purpose: Demonstrates use of RegExFunctions.cfm library
--->

<html>
<head><title>Using a Regular Expression</title></head>
<body>

<!--- Include UDF library of regular expression functions --->
<!--- This allows us to use the REFindString() function --->
<cfinclude template="RegExFunctions.cfm">

<!--- The text to search --->
<cfset text = "My email address is nate@nateweiss.com. Write to me anytime.">

<!--- Attempt to find a match --->
<cfset matchedString = reFindString("([\w._]+)\@([\w_]+(\.[\w_]+)+)", text)>

<!--- Display the result --->
<cfif matchedString neq "">
 <cfoutput><p>A match was found: #matchedString#</p></cfoutput>
<cfelse>
 <p>No matches were found.</p>
</cfif>

</body>
</html>
```

Listing 43.12 provides the code for the `RegExFunctions.cfm` UDF library.

Listing 43.12 `RegExFunctions.cfm`—A UDF Function Library for Working with Regular Expressions

```
<!---
 Filename: RegExFunctions.cfm
 Author: Nate Weiss (NMW)
 Purpose: Implements a UDF library for working with regular expression
--->


<!--- REFindString() function --->
<cffunction name="reFindString" returntype="string" output="false">
 <!--- Function arguments --->
```

Listing 43.12 (continued)

```
<cfargument name="regEx" type="string" required="yes">
<cfargument name="string" type="string" required="yes">
<cfargument name="start" type="numeric" required="no" default="1">
<cfargument name="caseSensitive" type="boolean" required="no" default="no">

<!--- The value to return (start off with an empty string) --->
<cfset var result = "">
<cfset var foundstruct = "">

<!--- Perform the regular expression operation --->
<cfif ARGUMENTS.caseSensitive>
  <cfset foundStruct = reFind(regEx, string, start, true)>
<cfelse>
 <cfset foundStruct = reFindNoCase(regEx, string, start, true)>
</cfif>

<!--- If a match was found, use the found string as the result --->
<cfif foundStruct.pos[1] gt 0>
<cfset result = mid(string, foundStruct.pos[1], foundStruct.len[1])>
</cfif>

<!--- Return the result --->
<cfreturn result>
</cffunction>


<!--- AdjustNewlinesToLinefeeds() function --->
<cffunction name="adjustNewlinesToLinefeeds" returntype="string" output="false">
 <!--- argument: string --->
 <cfargument name="string" type="string" required="yes">

 <!--- Replace all CRLF sequences with just LF --->
 <cfset var result = reReplace(string, chr(13)&chr(10), chr(10), "ALL")>

 <!--- Replace any remaining CR characters with LF --->
 <cfset result = reReplace(string, chr(13), chr(10), "ALL")>

 <!--- Return the result --->
 <cfreturn result>
</cffunction>
```

NOTE

For information about the `<cffunction>`, `<cfargument>`, and `<cfreturn>` code used here, see Chapter 27, “Creating Advanced ColdFusion Components,” in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 2: Application Development*.

Building a RegEx Testing Page

Sometimes it’s a lot easier to craft a regular expression if you have an interactive environment to play with. The example listings for this chapter include a convenient regular expression “tester” page for creating or troubleshooting your regular expressions. You can also use this tester page to work through some of the RegEx syntax examples in the section “Crafting Your Own Regular Expressions.”

To use the page, follow these steps:

1. Visit the `RegExTester.cfm` page with your browser. The Regular Expression Tester page is illustrated in Figure 43.11.

Figure 43.11

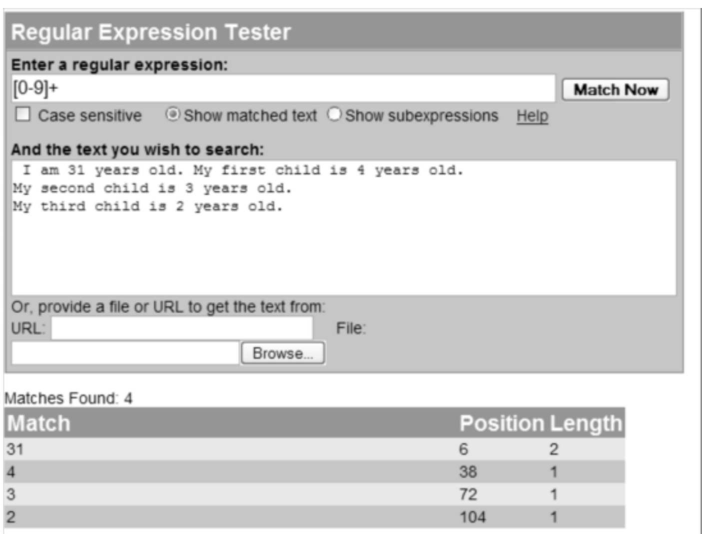
The Regular Expression Tester is handy for crafting your own Regular Expressions.

Regular Expression Tester
Enter a regular expression:
Match Now
Case sensitive
Show matched text
Show subexpressions
Help
And the text you wish to search:
Or, provide a file or URL to get the text from:
URL:
File:
Browse...

2. Enter your regular expression. If you want case to be considered, check the Case Sensitive option.
3. Enter the text you want to search or get the text from a Web page on your server or elsewhere on the Internet, and provide the URL (including the `http://` part) in the field provided. To get the text from a file on your computer, use the Browse button to select the file to upload.
4. Click the Match Now button to display the matches (Figure 43.12).

Figure 43.12

Each of your RegEx's matches is displayed in a scrolling table.



Match	Position	Length
31	6	2
4	38	1
3	72	1
2	104	1

TIP

If you need to jog your memory on a particular metacharacter, you can click the Help link to bring up the RegEx portion of the ColdFusion documentation.

TIP

If your regular expression contains subexpressions, choose the Show Subexpressions option to display the actual values matched by each subexpression.

Listing 43.13 provides the code for the Regular Expression Tester. You are invited to adapt the page to suit your needs.

Listing 43.13 `RegExTester.cfm`

```
<!---
 Filename: RegExTester.cfm
 Author: Nate Weiss (NMW)
 Purpose: A page for crafting, testing, and debugging regular expressions
--->

<!--- Form parameters --->
<cfparam name="FORM.regEx" type="string" default="">
<cfparam name="FORM.searchText" type="string" default="">
<cfparam name="FORM.searchTextFile" type="string" default="">
<cfparam name="FORM.searchTextURL" type="string" default="">
<cfparam name="FORM.ShowSubExpr" type="boolean" default="no">
<cfparam name="FORM.CaseSensitive" type="boolean" default="No">

<!--- Location of the RexEx pages within the ColdFusion documentation --->
<cfset regExDocURL =
 "http://livedocs.macromedia.com/coldfusion/6/" &
 "Developing_ColdFusion_MX_Applications_with_CFML/regexp.htm">

<!--- If the user is uploading a file --->
<cfif FORM.searchTextFile neq "">

 <!--- Obtain a temporary file to store the uploaded text in --->
 <cfset tempFileName = getTempFile(getTempDirectory(), "rgx")>

 <!--- Accept the file upload --->
 <cffile action="upload" filefield="FORM.searchTextFile"
 destination="#tempFileName#" nameconflict="Overwrite">

 <!--- Read the contents of the file into the FORM.SearchText variable --->
 <cffile action="read" file="#tempFileName#" variable="FORM.SearchText">

 <!--- Delete the temporary file --->
 <cffile action="delete" file="#tempFileName#">

<!--- If the user is providing a URL to get the search text from --->
<cfelseif left(FORM.searchTextURL, 4) eq "http">
 <!--- Fetch the text over HTTP --->
 <cfhttp method="get" url="#FORM.searchTextURL#">

 <!--- If we appear to have connected succesfully to the URL --->
 <cfif cfhttp.fileContent neq "Connection Failure">
```

Listing 43.13 (continued)

```
   <!--- Use the fetched text from here on --->
   <cfset FORM.searchText = cfhttp.fileContent>
  </cfif>
 </cfif>

 <html>
 <head><title>Regular Expression Test Page</title></head>
 <body>

 <!--- Some styles to make the page look nice --->
 <style>
  body {font-family:arial;font-size:12px}
  input {font-size:11px}
  textarea {font-size:11px}
  th {background:#8888FF;color:white;text-align:left}
  td {background:#CCCC99;color:black;font-size:12px}
  td.RowA {background:#CCCC99}
  td.RowB {background:#EEEEBB}
 </style>

 <cfoutput>
  <!--- Create a self-submitting form --->
  <FORM ACTION="#CGI.script_name#" METHOD="post" ENCTYPE="multipart/form-data">

  <table border="1" cellPadding="4" cellSpacing="0" width="550">
  <tr>
  <th class="Dialog">Regular Expression Tester</th>
  </tr>
  <tr>
  <td class="Dialog">

  <!--- Text input for the regular expression itself --->
  <b>Enter a regular expression:</b><br>
  <input name="regEx" type="text" size="65" style="font-size:13px"
  value="#htmlEditFormat(FORM.regEx)#">

  <!--- Submit button to start the search --->
  <input type="submit" value="Match Now" style="font-weight:bold"><br>

  <!--- Checkbox to control case-sensitivity --->
  <input type="checkbox" name="casesensitive" id="casesensitive"
  <cfif FORM.caseSensitive>checked</cfif>
  value="yes">
  <label for="casesensitive">Case sensitive</label>

    
  <!--- Radio buttons to display matches vs. subexpressions --->
  <input type="radio" name="showSubExpr" id="showSubExprno"
  <cfif not FORM.showSubExpr>checked</cfif>
  value="no"><label for="showSubExprNo">Show matched text</label>

  <input type="radio" name="showSubExpr"id="showSubExpryes"
  <cfif FORM.showSubExpr>checked</cfif>
 value="yes"><label for="showSubExprYes">Show subexpressions</label>
```

Listing 43.13 (continued)

```
      
    <a href="#regExDocURL#" target="RegExDocs">Help</a>

    <!--- Textarea where user can type the text to search --->
    <p><b>And the text you wish to search:</b><br>
    <textarea name="searchText" wrap="off" cols="70" rows="6">
    #htmlEditFormat(FORM.searchText)#
    </textarea><br>

    <!--- Text input for providing a URL --->
    Or, provide a file or URL to get the text from:<br>
    URL:
    <input name="searchTextURL" type="text" size="35">

    <!--- File input for uploading a text file to search --->
       
    File:
    <input name="searchTextFile" type="file" size="30">

    </td>
    </tr>
    </table>

    </form>
   </cfoutput>

   <!--- When the form is submitted... --->
   <cfif FORM.regEx neq "">

    <!--- Include UDF library of regular expression functions --->
    <!--- This allows us to use the REFindMatches() function --->
    <cfinclude template="RegExFunctions.cfm">

    <!--- If the RegEx is using "multiline mode", adjust the text --->
    <!--- so that only linefeed characters are used to separate lines --->
    <cfif left(FORM.regEx, 4) eq "(?m)">
      <cfset FORM.searchText = adjustNewlinesToLinefeeds(FORM.searchText)>
    </cfif>

    <cftry>
    <!--- Perform the regular expression search --->
    <!--- Results are returned as a query object with three columns: --->
    <!--- "Pos" - the position of the match --->
    <!--- "Len" - the length of the match --->
    <!--- "Found" - the actual text of the match --->
    <cfset matchQuery = reFindMatches(FORM.regEx, FORM.searchText, FORM.caseSensitive)>

    <!--- If any errors occur --->
    <cfcatch type="Expression">
    <!--- If it has to do with regular expression, show friendly message --->
    <cfif CFCATCH.message contains "malformed regular expression">
    <p><b>Sorry, there is a problem with your regular expression.</b><br>
    <cfoutput>#CFCATCH.Detail#<br></cfoutput>
    <cfabort>

    <cfelse>
      <cfrethrow>
```

Listing 43.13 (continued)

```
</cfif>
</cfcatch>
</cftry>

<cfoutput>
<!--- Display the number of matches found --->
Matches Found: #matchQuery.RecordCount#<br>

<!--- If there is at least one match... --->
<cfif matchQuery.recordCount gt 0>

  <!--- Display the matches in a simple HTML table --->
  <div style="height:200px;overflow-y:auto;width:550">
  <table border="0" cellPadding="2" cellSpacing="0" width="550">

  <tr>
  <cfif FORM.showSubExpr>
    <cfloop from="1" to="#matchQuery.SubExprCount#" index="i">
      <th>Sub-Expr #i#</th>
    </cfloop>
  <cfelse>
    <th width="*">Match</th>
    <th width="50">Position</th>
    <th width="50">Length</th></tr>
  </cfif>
  </tr>

  <cfloop query="matchQuery">
    <!--- Alternating row colors --->
    <cfset class = iif(currentRow mod 2 eq 0, "'RowA'", "'RowB'")>
    <tr>
    <cfif FORM.showSubExpr>
      <cfloop from="1" to="#matchQuery.SubExprCount#" index="i">
      <td class="#class#"
      width="#round(100/matchQuery.SubExprCount)#%">
      #htmlEditFormat(matchQuery["SubExpr#i#"][currentRow])#
      </td>
      </cfloop>
    <cfelse>
      <td class="#class#" width="450">
      #htmlEditFormat(matchQuery.Found)#
      </td>
      <td class="#class#" width="50">#matchQuery.pos#</td>
      <td class="#class#" width="50">#matchQuery.len#</td>
    </cfif>
    </tr>
  </cfloop>
  </table>
  </div>

</cfif>
</cfoutput>
</cfif>

</body>
</html>
```

NOTE

Listing 43.13 makes use of the `reFindMatches()` and `adjustNewlinesToLinefeeds()` functions discussed in the previous section, "Some Convenient RegEx UDFs." As such, it requires the `RegExFunctions.cfm` UDF library file to be present.

Crafting Your Own Regular Expressions

Up to this point, this chapter has introduced you to the `reFind(), reMatch` and `reReplace()` functions (and their case-insensitive counterparts). Along the way, you learned about a number of RegEx concepts, such as subexpressions and backreferences. You've also seen some decent examples of actual RegEx criteria syntax (that is, the various wildcards you can use in regular expressions)—but you haven't been formally introduced to what each of the wildcards does.

The remainder of this chapter will focus on the regular expressions themselves.

Understanding Literals and Metacharacters

Every regular expression you write includes two types of characters: *literals* and *metacharacters*.

Literals, or *literal characters*, are normal text characters that represent themselves literally. In other words, literals are all the characters in a RegEx that aren't wildcards of one form or another. In the email RegEx that's been used several times in this chapter (see Listing 43.1), the only literal character is the `@` sign. If your search involves the word *dog*, your RegEx will likely contain the literal `d`, `o`, and `g` characters.

Metacharacters are the various special characters (what I've been calling *wildcards* up to this point) that have special meaning to the regular expression engine. You've already seen a few of the most common metacharacters, such as the `[`, `]`, `{`, `}`, and `+` characters. You'll learn about all the rest in the pages to come.

NOTE

Up to this point, I've been using the term wildcard as an approximate synonym for metacharacter. Wildcard is less technical and perhaps a bit less precise, but it rolls off the tongue a lot more easily and is more intuitively understood. I imagine you've understood what I've meant by wildcard all along, whereas metacharacter might have slipped us up a bit. I'll continue to use wildcard during the less formal parts of the remaining discussion.

Including Metacharacters Literally

Sometimes, you need to include one of the metacharacters as a literal. To do so, you escape the metacharacter by preceding it with a backslash. You saw this demonstrated in Listing 43.2, where the sequences `\(` and `\)` were used to denote literal parentheses characters (that is, parentheses that should actually be searched for, rather than having their usual special meaning of indicating a subexpression).

NOTE

If you need to search for a literal backslash, escape the backslash with another backslash. Just use two backslashes together, as in `\\`.

NOTE

Remember that the backslash serves another purpose, too: indicating a backreference. For details, see the earlier section "Altering Text with Backreferences."

Introducing the Cast of Metacharacters

The RegEx implementation in ColdFusion supports a lot of metacharacters, which can be broken into the conceptual groups shown in Table 43.5.

Table 43.5 Metacharacter Types

TYPE	DESCRIPTION
Character classes	Character classes define a set of characters that will match. They are defined with square brackets: `[aeiou]` matches any single vowel; `[0-9]` matches any single number, and `[^0-9]` matches any single character except numbers. There are also special shortcuts for often-used sets of characters, such as `\w` or for any letter or number, or `\s` for any whitespace character. Finally, there's the dot character (`.`), which matches any character at all.
Quantifiers	These metacharacters allow you to specify how many times a certain item can appear to still be considered a match. Quantifiers include `?` for optional matches, `+` for one or more matches, and `*` for any number of matches (including none). There are also the *interval quantifiers*: `{num}` for `num` number of matches; `{num,max}` for `num` to `max` number of matches; and `{num,}` for `num` or more matches.
Alternation	You can establish OR conditions in your regular expressions with the `|` character. Parentheses constrain how far the `|` reaches, so `(you|we)` matches `you` or `we`.
String anchors	String anchors let you specify that a match must occur at a particular location in a chunk of text. Anchors include `^` for matches at the beginning of the text (or line) and `$` for matches at the end. There are also the `\A` and `\Z` anchors, which are similar, except do not work in multiline mode.
Escape sequences	Escape sequences are mostly for matching certain unprintable characters; for example, `\t` to match tabs or `\n` to match newlines.
Modifiers	Modifiers allow you to turn on different types of RegEx behavior for use in special cases. Modifiers include `(?m)` for line-by-line matching and `(?=)` for lookahead matching.

The next few sections present a kind of crash course in metacharacters. I've titled these sections Metacharacters 101, Metacharacters 102, and so on. By the end of this little course, you'll have a pretty good understanding of regular expression syntax. Aren't you glad you didn't actually have a course like this in school?

Metacharacters 101: Character Classes

Of all of the metacharacters available in regular expressions, character classes are probably the most important. Character classes are a way of specifying a set of characters, any one of which can be

considered a match. You can specify your own classes or use any number of predefined classes supported by RegEx.

Specifying Character Classes with `[ ]`

You can specify any set of characters as a character class with the square bracket characters `[` and `]`. The class `[aeiouAEIOU]` will match any vowel; `[12345]` will match a 1, 2, 3, 4, or 5 character. For instance, perhaps your last name is Andersen and people often misspell it as Anderson or forget to capitalize the first letter. You could find any of the various spellings using `[Aa]nders[eo]n` as the regular expression.

The hyphen character has special meaning when it is between a set of square brackets: It indicates a range of acceptable characters. For instance, `[1-5]` is easier to type than `[12345]` and will still match a 1, 2, 3, 4, or 5 character. Very common character classes are `[A-Za-z]` for matching any letter, and `[0-9]` for matching any single number character. If your company uses an ID number composed of two letters followed by a dash and then three numbers, you could use this as the regular expression:

```
[A-Z][A-Z]-[0-9][0-9][0-9]
```

As you'll learn in Metacharacters 102, you could use quantifiers as an easier way of specifying the part consisting of three numbers at the end.

Negating a Character Class with `^`

If the square bracket contents for a character class start with a caret character, the character class is negated, meaning that the class will match any character that isn't in the class. For example, `[^A-Za-z0-9]` matches anything other than a number or letter, and `[^aeiouAEIOU]` matches anything other than a vowel.

NOTE

> Keep in mind that there are lots of other characters other than letters and numbers, including unprintable characters such as tabs and newlines. So, while you may think at first glance that `[^aeiouAEIOU]` would simply match all consonants, that's not all it will match. It will also match unprintable characters, and all other characters, too, including punctuation characters (commas, periods, and the like).

Common Character Classes

Because certain character classes are called for frequently (such as `[A-Za-z]` for matching any letter, and `[0-9]` for matching any digit), ColdFusion supports a number of shortcuts for the most commonly needed character classes. Different regular expression tools support slightly different ways of specifying these shortcuts, but most adhere to the shortcuts supported by Perl or by POSIX. ColdFusion's RegEx implementation supports both. The Perl shortcuts, in particular, are really easy to type.

Table 43.6 shows common character classes you might need to use in your regular expressions, with the Perl-style and POSIX-style shortcuts for each. The Normal column shows how to write the character class using the normal square bracket syntax. The Perl Shortcut and POSIX Shortcut

columns show the shortcuts for each class; for some of the classes, there is a POSIX shortcut but no corresponding Perl shortcut, in which case the Perl Shortcut column is left blank. A few shortcuts shown at the bottom of the table would be virtually impossible to type using the manual `[ ]` syntax, so the Normal column is left blank.

Table 43.6 Common Character Classes and Their Shortcuts

NORMAL	PERL	POSIXSHORTCUT	MATCHESSHORTCUT
`[A-Z]`		`[[:upper:]]`	Any uppercase letter.
`[a-z]`		`[[:lower:]]`	Any lowercase letter.
`[A-Za-z]`		`[[:alpha:]]`	Any letter, regardless of case.
`[0-9]`	`\d`	`[[:digit:]]`	Any number character (digit).
`[^0-9]`	`\D`	`[^[:digit:]]`	Any character other than a number.
`[0-9A-Za-z]`	`\w`	`[[:alnum:]]`	Any letter or number character.
`[^0-9A-Za-z]`	`\W`	`[^[:alnum:]]`	Any character other than a number or letter.
`[ \t]`		`[[:blank:]]`	A space or a tab.
`[ \t\n\r\f]`	`\s`	`[[:space:]]`	Any whitespace character, which means any spaces, tabs, or any of the end-of-line indicators (newlines, form feeds, and carriage returns).
`[^ \t\n\r\f]`	`\S`	`[[:graph:]]`	Any nonwhitespace character.
	`.` (dot)		Any character at all. It's important to understand that in ColdFusion, the dot character always matches newlines, which is not always the case with Perl.

NOTE

As noted in this table, ColdFusion's dot metacharacter always matches any character, including newlines. In other words, the behavior is consistent with Perl behavior when Perl's `/s` switch is in effect.

In the preceding section, we discussed a regular expression for matching an ID number that comprised two letters, a dash, and three numbers. The RegEx looked like this:

```
[A-Z][A-Z]-[0-9][0-9][0-9]
```

You can use Perl-style shortcuts to make the RegEx easier to type and look at, like this:

```
[A-Z][A-Z]-\d\d\d
```

Or, you can use POSIX-style shortcuts, like so:

```
[[:upper:]][[:upper:]]-[[:digit:]][[:digit:]][[:digit:]]
```

Feel free to mix and match the two types of shortcuts, like so:

```
[[:upper:]][[:upper:]]-\n\n
```

NOTE

The POSIX shortcuts can be negated with the ^ character, as shown in the POSIX Column for the `[^0-9]` class in Table 43.6.

NOTE

You might be wondering why you would use `[[:upper:]]` instead of `[A-Z]`, because it doesn't seem to be much of a shortcut at all (there's actually more to type). The main benefit is that the POSIX shortcuts attempt to understand uppercase and lowercase letters for each language, whereas something like `[A-Z]` will work only for English and other roman-style character sets.

Metacharacters 102: Quantifiers

As you learned in Table 43.5, *quantifiers* allow you to specify how many times certain parts of a RegEx can match for the overall regular expression to be considered a match. You will learn about the many quantifiers in detail as we work through the Metacharacters section.

Regardless of which quantifier you're using, you always place it right after the item that you want to affect. That item might be a single character, a character class, or the set of parentheses that sets off a subexpression. If character classes are the foundation of what regular expressions are about, quantifiers give the technology its muscle; without them, it would be hard to solve anything but simple problems with RegEx.

Table 43.7 lists the quantifier metacharacters available for your use.

Table 43.7 RegEx Quantifiers

QUANTIFIER	DESCRIPTION
`?`	Means that the preceding item is optional. In more technical terms, `?` matches the preceding item zero or one times. The preceding item might be a single character, a character class, or a subexpression.
`+`	Means that the preceding item appears at least once; that is, `+` matches the preceding item one or more times. Again, the preceding item might be a single character, a character class, or a subexpression.
`*`	Means that the preceding item is optional, but also may appear any number of times. Conceptually, it's like combining `?` and `+`. That is, it matches the preceding item zero or more times.
`{num}`	Matches the preceding item exactly `num` times; for instance, either `[0-9]{3}` or `\d{3}` will match three numbers (digits).
`{num,max}`	Matches the preceding item between `num` and `max` times. So, if you were looking for all words between 5 and 10 letters in length, you could use `[A-Za-z]{5,10}` or `[[:alpha:]]{5,10}`.
`{num,}`	Matches the preceding item at least `num` times, without a maximum, so `[A-Z]{10,}` could be used to find long words (longer than 10 letters). If you think about it, the `+` character (above in this table) could be considered a shortcut for `{1,}`.

Using Quantifiers

Let's look at a few examples of using character classes and quantifiers. Say you need to create a regular expression that will match a U.S. ZIP code. Let's start off with the simple five-digit version of a ZIP code. Using the character class skills you learned in Metacharacters 101, you know you could use this:

```
[0-9][0-9][0-9][0-9][0-9]
```

or this:

```
\d\d\d\d\d
```

You can use the `{num}` quantifier from Table 13.8 to avoid having to type a separate class for each digit, like so:

```
[0-9]{5}
```

or like so:

```
\d{5}
```

Now let's say you want to match the nine-digit version of a ZIP code. Just add another class and quantifier sequence, like so:

```
\d{5}-\d{4}
```

NOTE

For those of you who aren't from the U.S., sorry to use such a culturally myopic example. It's just a natural one to start off with. Anyway, U.S. ZIP codes are just the postal code used in a mailing address. ZIP codes come in two forms. For a long time, they were simply five-digit numbers. Later, the postal service introduced a nine-digit version, in the form 99999-9999. Both forms are used in practice today..

Making Certain Portions Be Optional with `?`

Okay, what if you wanted to accept either five- or nine-digit ZIP codes? You can use the `?` quantifier to say that the second portion of the code is optional, as in the following (Figure 43.13):

```
\d{5}(-\d{5})?
```

Note that the `?` quantifier respects parentheses, so in this example everything within the parentheses is modified by the `?`.

Including One or More Matches with +

Another cool quantifier is the `+` metacharacter. Because it matches one or more times, `+` is essential for matching substrings that will vary in length. That turns out to describe the majority of regular expression problems, so you'll be using `+` a lot.

The following matches any number of digits:

```
[0-9]+
```

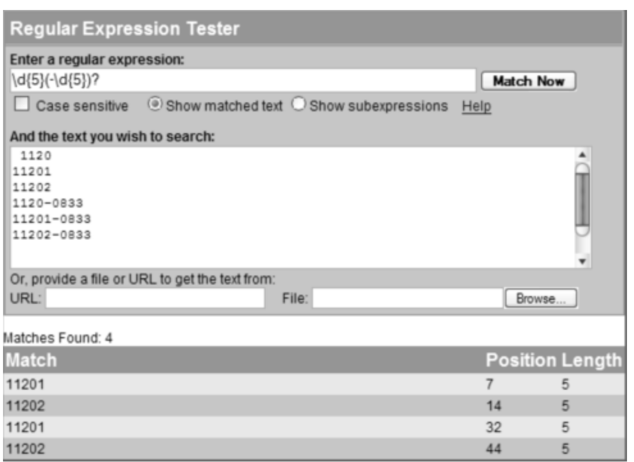


Figure 43.13

The `?` operator handles items that don't necessarily need to be present.

Like `?` and all the other quantifiers, the `+` character respects parentheses. When it follows a parenthesized group, `+` matches the entire group one or more times. You can also nest these sets of parentheses within one another, an approach that forms the basis of the email address RegEx you have seen throughout this chapter:

```
[\w._]+@[\w_]+(\.[\w_]+)+
```

That looks complex at first, but it's not so bad if you concentrate on each portion separately. The first portion is in charge of matching the username part of the email address (the part before the `@` sign). I came up with `[\w._]+` for this part, which matches any number of letters, numbers, dots, or underscores. After the `@` sign, the next portion is `[\w_]+`, which is almost the same except that it doesn't match dots. Next comes a parenthesized group. Inside the parentheses, the expression reads `\.[\w_]+`, which means a dot, then any number of letters, numbers, or underscores. The fact that there's a `+` after the parentheses means that this pattern (a dot, then other stuff) can be repeated any number of times.

In plain English, then, the expression reads "any number of normal characters, then an `@` sign, then any number of groups, where the groups each have dots at the beginning," which is a fair description of a validly formed email address.

NOTE

Some of the examples in this chapter add a few additional sets of parentheses to this regular expression so that it contains subexpressions for each part of the email address. Those parentheses don't have anything to do with the + sign, and don't affect which addresses actually match. They just make it possible to capture each portion of the match separately.

Matching Any Number of Matches with *

The `*` metacharacter is similar to `+` in that it will match one, two, or any other number of whatever preceded it. The difference is that it will also match zero times: It matches even if the preceding item isn't present at all. I like to think of this quantifier as meaning "any amount of the preceding, but let it be optional."

For instance, it could be used to find `<b>` (boldface) tags in a chunk of HTML:

```
<b>.*</b>
```

In plain English, this means to match a `<b>`, then any amount of anything, then `</b>`. This seems sensible enough. If you try it against this text:

```
The <b>Bear</b> walked alone
```

you will find that the `<b>Bear</b>` part is what matches, which is what you would expect. However, if you try it against this text:

```
The <b>Bear</b> and the <b>Fox</b> walked hand in hand.
```

it will match the `<b>Bear</b> and the <b>Fox</b>` part of the text. That is, the RegEx engine finds the first `<b>`, then matches everything up to the last `</b>`. What's going on? Although it might seem counterintuitive at first, it's important to understand that the `.*` part *really does* mean "any number of any characters." There's nothing in the `</b>.*</b>` expression that says that the `.*` part isn't supposed to match the characters in the `</b>` part. It's an important concept that is crucial to understand when crafting regular expressions.

By default, regular expressions are "greedy," which means that the processor is always willing to return the least rigorous interpretation of your RegEx as possible. Or, to put it another way, the engine will always assume that you want the longest possible match. The ColdFusion documentation refers to this as *maximal matching*, but most regular expression references call it *greedy matching*.

One way to fix the boldface-text example is to replace the `.*` with `[^<]*`, like so:

```
<b>[^<]*</b>
```

See the difference? In plain English, this now means "match `<b>`, then match any number of anything that isn't a <, then match `</b>`."

When used against the previous text sample, this version of the RegEx will correctly match `<b>Bear</b>` and `<b>Fox</b>`, making it a pretty good solution to the problem. However, it will fail if the text contains any < characters between the `<b>` and `</b>`, like this:

```
The <b><i>Bear</i></b> and the <b>Fox</b> walked hand in hand.
```

Using this text, the `<b>[^<]*</b>` expression will only match `<b>Fox</b>`. Bummer. All is not lost, though. You can tell the RegEx engine not to use greedy matching, which brings us to our next topic.

Using Minimal Matching (Non-Greedy) Quantifiers

As you have seen, the fact that regular expressions will match the longest possible substring by default, maximal matching (greedy matching) can sometimes be a problem. In such situations, you can use slightly different quantifiers to tell the RegEx engine to match the shortest possible substring instead. The ColdFusion documentation refers to this as *minimal matching* (as opposed to maximal matching), but most RegEx texts call it *non-greedy matching*.

There is a non-greedy version of each of the quantifiers shown in Table 13.8. To indicate that you want to use the non-greedy version, follow the quantifier with a `?` character, as shown in Table 43.8.

Using your newfound knowledge of non-greedy quantifiers, the boldfaced-text problem becomes easy to solve:

```
<b>(.*?)</b>
```

Table 43.8 Minimal Matching (Non-Greedy) Quantifiers

QUANTIFIER	DESCRIPTION
`??`	Non-greedy version of `?`, which means that the preceding item is optional. The difference in the non-greedy version is that the RegEx engine will first try to match based on the item's absence. In other words, the item will only be included in the match if it is not possible to get a match without the item.
`+?`	Non-greedy version of `+`, which means that the preceding item will match at least once, but as few times as possible.
`*?`	Non-greedy version of `*`, which means that the preceding item can appear any number of times (including none at all), but the shortest possible string will always be found.
`{num,max}?`	Non-greedy version of `{num,max}`, which means that the preceding item will match between `num` and `max` times, but as few times as actually possible.
`{num,}?`	Non-greedy version of `{num,}`, which means that the preceding item will match at least `num` times, but as few times as actually possible.

If you wanted to ensure that there was at least one character between the `<b>` and `</b>` tags, you could use the non-greedy version of `+` instead of `*`, like so:

```
<b>(.+?)</b>
```

This expression will match all bold text, but not empty `<b></b>` tags.

NOTE

Non-greedy matching is sometimes called lazy matching, meaning that the RegEx engine is "lazily" trying to match as little text as possible.

Metacharacters 201: Alternation

Sometimes you might need to find matches that contain one string or pattern, or another string or pattern. That is, sometimes you need the conceptual equivalent of what would be called an "or" in normal programming languages, or the `OR` part of a SQL query.

To perform "or" matches with regular expressions, use the `|` character (usually called the *pipe* character). Each pipe represents the idea of "or." Just as in normal programming, the `|` character's effect can be constrained with parentheses, so `Number (1|2)` is different from `Number 1|2`. The first would match the string `Number 1` or `Number 2`, whereas the second would match `Number 1` or just the number `2`.

The following RegEx would match the phrase My Red Fox, My Brown Fox, or My Beige Fox. It would also match My 1 Fox, My 2 Foxes, My 3 Foxes, or any other number of foxes:

```
My ((Red|Brown|Beige|1) Fox|[0-9]+Foxes)\b
```

Metacharacters 202: Word Boundaries

Often, you will need to write regular expressions that are aware of word boundaries. ColdFusion supports the Perl-style `\b` and `\B` boundary sequences, as described in Table 43.9.

Table 43.9 Perl-Style Boundary Sequences

SEQUENCE	MEANING
`\b`	Matches what can generally be described in plain English as a *word boundary*. Technically, a boundary is defined as the transition between an alphanumeric character and a nonalphanumeric character.
`\B`	The opposite of `\b`, matching any character that is not a word boundary. Generally less useful than \b in most scenarios.

The `\b` boundary sequence is particularly handy for making sure that your regular expression matches only whole words. For instance, the regular expression `\b[Cc]at\b` would match cat or Cat, but not Cats, Catsup, or Scat.

Metacharacters 203: String Anchors

String anchors are conceptually similar to boundary sequences (see the preceding section), because they are another way of making sure that your regular expression doesn't find undesired "partial matches." Whereas boundaries are about making sure the match "bumps up" against the beginning or end of a word, string anchors are about making sure the match "bumps up" against the beginning or end of the entire chunk of text being searched.

The RegEx string anchors are listed in Table 43.10.

Table 43.10 String Anchors

ANCHOR	DESCRIPTION
`^`	Matches the beginning the chunk of text being searched. Or, in multiline mode, matches the beginning of a line (multiline mode is discussed next).
`$`	Matches the end of the text being searched. Or, in multiline mode, matches the end of a line.
`\A`	Always matches the beginning of the chunk of text being searched, regardless of whether multiline mode is being used.
`\Z`	Always matches the end of the text being searched, regardless of multiline mode.

For instance, perhaps you have a form field called ZipFieldPlus4, which you want to validate to make sure it contains a properly formatted U.S. postal ZIP code (the nine-digit ”+4“ variety). If you didn't know about string anchors, you might decide to use \d{5}-\d{4} as the regular expression, like so:

```
<cfif reFind("\d{5}-\d{4}", FORM.zipCodePlus4)>
 Okay
<cfelse>
 Not Valid
</cfif>
```

This regular expression seems to do the job. It displays "Okay" if the user enters something like 01201-9809, and "Not Valid" if the user enters 01201-98 or just 01201.

However, it will also display "Okay" if the user types Foo 01201-9809 or 01201-9809Bar, because there is nothing about the regular expression that says the ZIP code must be the *only* thing the user enters. The solution is to anchor the regular expression to the beginning and end of the string using ^ and $, like so:

```
<cfif reFind("^\d{5}-\d{4}$", FORM.zipCodePlus4)>
 Okay
<cfelse>
 Not Valid
</cfif>
```

Alternatively, you could use the \A and \Z sequences, like so:

```
<cfif reFind("\A\d{5}-\d{4}\Z", FORM.zipCodePlus4)>
 Okay
<cfelse>
 Not Valid
</cfif>
```

These two snippets will perform the same way, because ^ is synonymous with \A (and $ is synonymous with \Z) unless the regular expression uses multiline mode.

Understanding Multiline Mode

If you start your regular expression with the special sequence (?m), the regular expression is processed in what the ColdFusion and Perl engines call *multiline mode*. Multiline mode means that the ^ and $ characters match the beginning and end of a line within the chunk of text being searched, rather than the beginning and end of the entire chunk of text (Figure 43.14).

Let's say you were going to search the following chunk of text:

```
1 frog a leaping
2 foxes jumping
100 programs crashing
5 golden rings
```

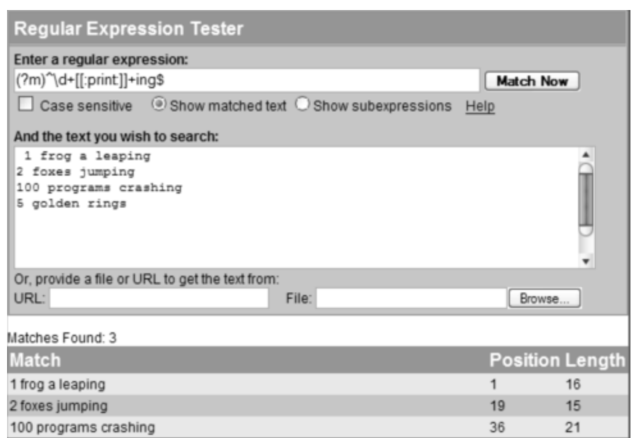


Figure 43.14

Multiline mode anchors matches to lines in the text being searched.

The following regular expression would get only the first line; because multiline mode is not in effect, `^` will match only the very beginning of the text:

```
^\d+[[:print:]]+
```

This one matches all four lines; because multimode is on, `^` matches the beginning of a line:

```
(?m)^\d+[[:print:]]+
```

This next one matches the first three lines (because they all end with `ing`), but not the last line (Figure 13.14):

```
(?m)^\d+[[:print:]]+ing$
```

All this said, it is very important to understand what the definition of a line is for the purposes of multiline mode processing. When you use `(?m)` with ColdFusion, each linefeed character (that's ASCII character 10) is considered to start a new line; this is the Unix method of indicating new lines. Carriage return characters (ASCII code 13) are not considered the start of new lines, which means that

- Multimode processing won't work correctly with chunks of text that originate on Macintosh computers, because the text might contain only carriage return characters and no linefeeds.
- Chunks of text that originate on Windows/MS-DOS machines probably contain CRLF sequences (a carriage return followed by a linefeed), to separate the lines. As far as RegEx's multimode processing is concerned, a carriage return character sits at the very end of every line, which means that the `$` will not work properly because it matches only linefeeds, not carriage returns.
- Chunks of text that originate on Unix machines will work fine (but if the chunks of text are coming from the public, it's unlikely that they are using Unix browsers).

Therefore, if you are going to use multiline mode, I recommend that you use ColdFusion's normal `replace()` method to massage the chunk of text that you're going to search. First, replace each CRLF with a linefeed (that should take care of the Windows text), and then replace any remaining carriage returns with linefeeds (to deal with the Mac text). Assuming that the chunk of text you will be searching is in a string variable called `str`, the following two lines will do the job:

```
<cfset str = reReplace(str, Chr(13)&Chr(10), Chr(10), "ALL")>
<cfset str = reReplace(str, Chr(13), Chr(10), "ALL")>
```

Another option would be to use the `adjustNewlinesToLinefeeds()` function included in the `RegEx-Functions.cfm` UDF library (Table 43.4), like so:

```
<cfset str = adjustNewlinesToLinefeeds(str)>
```

Metacharacters 301: Match Modifiers

Perl 5 introduced a number of special modifiers that begin with the sequence `(?`, as listed in Table 43.11. Most of these modifiers are discussed elsewhere in this chapter, as indicated.

Table 43.11 Match Modifiers Supported in ColdFusion

MODIFIER	DESCRIPTION
`(?x)`	Allows you to write the rest of the expression with indentation, whitespace, and comments. A nice alternative to writing a very complex expression all on one long line (see example after this table).
`(?m)`	Tells the engine to use multiline mode for purposes of matching `^` and `$` (discussed in the preceding section, "Understanding Multiline Mode").
`(?i)`	Tells the engine to perform case-insensitive matches, regardless of whether you are using `reFind()` or `reFindNoCase()`—or, for that matter, `reReplace()` versus `reReplaceNoCase()`.
`?:`	Used at the beginning of a set of parentheses, tells the engine not to consider the value as a subexpression. That is, `(?:)` means that the parentheses will not add an item to the `len` and `pos` arrays (see Listing 43.4). The parentheses still behave normally in all other respects (for instance, a quantifier after a set of the parentheses still applies to everything within the set).
`?=`	Used at the beginning of a set of parentheses, tells the engine to match whatever is inside the parentheses using positive lookahead. This means you want to make sure that the pattern exists but that you don't need it to be part of the actual match.
`?!`	Used at the beginning of a set of parentheses, tells the engine to match whatever is inside the parentheses using negative lookahead, which means that you want to make sure that the pattern does not exist.

NOTE

The ColdFusion documentation implies that you can use only `(?x)` or `(?m)` or `(?i)` at the very beginning of a regular expression. Actually, you can use them anywhere in the expression, but they always affect the whole expression, ignoring parentheses. There is no way to say that you only want part of the expression to be affected by `(?i)`, for instance. This is consistent with Perl's behavior. Just the same, I recommend putting these match modifiers at the beginning of the expression, because that's the documented usage.

As an example of using the `(?x)` modifier, consider the simple phone number RegEx that has been used elsewhere in this chapter. When used in a `reFind()`, it can look a bit unwieldy and somewhat inscrutable:

```
<cfset match = reFind("(\([0-9]{3}\))([0-9]{3}-[0-9]{4})", text, 1, true)>
```

Using `(?x)`, you can spread the regular expression over as many lines as you want, using whatever indention you want. You can also use the # sign to add comments, like this:

```
<cfset match = reFind("(?x)

 ( ## (begin capturing area code with subexpression)
 \([0-9]{3}\) ## Area Code portion, surrounded by literal parentheses
 ) ## (end capturing of area code)

 ( ## (begin capturing actual phone number)
 [0-9]{3} ## "Exchange" portion of phone number,
 - ## then a hyphen,
 [0-9]{4} ## then the last four digits of phone number
 ) ## (end capturing of phone number)

", text, 1, True)>
```

Anything from a `##` to the end of the line is considered to be a comment.

NOTE

Actually, the RegEx comment indicator is a single `#`, not `##`, but because `#` has special meaning to ColdFusion, you need to use two pound signs together in order to get the `#` character into the RegEx string. This is the case anytime you need to embed `#` within a quoted string in CFML.

TIP

If you need to match a space character while using `(?x)`, escape the space character by typing a `\` followed by a space. That tells the processor to consider the space as an actual part of the match criteria, rather than part of the indention and other decorative whitespace.

Metacharacters 302: Lookahead Matching

As noted in Table 43.11, you can use the positive lookahead modifier at the beginning of any parenthesized set of items. Positive lookahead means that you want to test that a pattern exists, but without it actually being considered part of the match. For instance, consider the following regular expression:

```
\bBelinda (?=Foxile)
```

This expression will match `Belinda` in a chunk of text, but only if it is followed by `Foxile`. `Belinda` followed by `Carlisle` will not match.

Negative lookahead, conversely, means that you want to test that a pattern does not exist. Conceptually, it's kind of like being able to say "this but not that." The following expression will match any `Belinda`, as long as it's not `Belinda Carlisle`:

```
\bBelinda (?!Carlisle)
```

Here's another example of using lookahead. Say you are using a simple regular expression such as the following to match telephone numbers in the form (999)999-9999:

```
(\([0-9]{3}\))([0-9]{3}-[0-9]{4})
```

The following variation adds negative lookahead to match only the phone numbers that are not in the 212 area code (see Figure 43.15):

```
(\((?!212)[0-9]{3}\))([0-9]{3}-[0-9]{4})
```

This last variation adds negative lookahead together with backreferences in the regular expression to match only the phone numbers that are not in the 212 area code, but where the phrase `(new listing)` appears after the number:

```
(\((?!212)[0-9]{3}\))([0-9]{3}-[0-9]{4})\s+(?=\(new listing\))
```

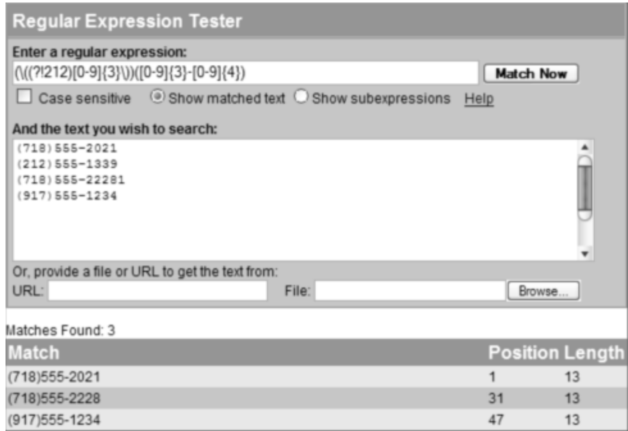


Figure 43.15

Lookahead matching allows for "this but not that" matches.

NOTE

ColdFusion does not support lookbehind processing (Perl's `(?<=)` and `(?<!)` sequences).

Metacharacters 303: Backreferences Redux

Earlier in this chapter, you learned about using backreferences such as `\1` and `\2` in the replacement string when using `REReplace()`, which allowed you to perform replacements that were far more intelligent than with static replacement strings. You can also use backreferences within the regular expression itself: Each backreference is like a variable that holds the value of the corresponding subexpression.

For instance, let's look at our telephone number RegEx again. Here's the normal version of the expression:

```
(\([0-9]{3}\))([0-9]{3}-[0-9]{4})
```

The following variation matches only those phone numbers where the last four digits are the same:

```
(\([0-9]{3}\))([0-9]{3}-(\d)\3\3\3)
```

This variation adds negative lookahead (discussed in the preceding section) to match only phone numbers in which the last four digits are not the same:

```
(\([0-9]{3}\))([0-9]{3}-(?!(\d)\3\3\3))
```

Metacharacters 304: Escape Sequences

ColdFusion supports the use of normal Perl escape sequences in regular expressions, as shown in Table 43.12. Previously, you needed to add these special characters to your RegEx string using the `Chr()` function. You can still do so, but these escape sequences are more standard and easier to type and read.

Table 43.12 RegEx Escape Sequences

ESCAPE SEQUENCE	DESCRIPTION
`\n`	Newline.
`\t`	Tab.
`\f`	Form feed.
`\r`	Carriage return.
`\x00`	Allows you to specify any character, using a two-digit hexadecimal number. For instance, the ASCII code for an exclamation point is 33 using normal (decimal) numbers; this is 21 in hexadecimal, so you could use `\x21` to specify an exclamation point in a RegEx. (Clearly, there would be more point to this if it were a character that's not on your keyboard, but you get the idea.)
`\000`	Allows you to specify any character, using a three-digit octal character. The octal version of 33 is 41, so you could also use `\041` to specify an exclamation point.

It's worth nothing that these escape sequences can be used in character classes, so `[\x00-xC8]` would match any of the first 200 characters in the character set (`C8` is hexadecimal for what we humans call 200).

Concluding Remarks

This chapter has introduced you to ColdFusion's regular expression support and armed you with some helpful examples. In the latter half of the chapter, you also learned a lot about RegEx syntax (the various wildcards or metacharacters that you can use to find matches).

Solving really tough problems with regular expressions is something of an art form, however, and their terse, concise nature doesn't make them particularly easy to understand by example. In short, the learning curve can be somewhat brutal, and there's no way all things RegEx can be discussed in just one chapter.

CHAPTER 44

ColdFusion Scripting

IN THIS CHAPTER

What Is <cfscript>? E161
Implementing <cfscript> E166
Common Problems and Solutions E175
User-Defined Functions in <cfscript> E176
Exception Handling in <cfscript> E178

What Is <cfscript>?

ColdFusion scripting allows you to write portions of your templates with script-based syntax, which is often more concise and straightforward than ColdFusion's traditional tag-based syntax. While you can't do everything with `<cfscript>` that you currently do with tag-based syntax, as you will later see, you may find that writing substantial portions of your code using scripting syntax is more natural for you.

Scripting syntax is very similar to JavaScript syntax with a couple of exceptions that we'll discuss soon. Listing 44.1 is an example of a tag-based index loop and its `<cfscript>` counterpart. Try running Listing 44.1 by alternately commenting-out the tag-based version and the `<cfscript>` version, and you'll see that each version does the same thing.

Listing 44.1 `ScriptingExample.cfm`—An Example of ColdFusion Scripting

```
<!--- Author: Charlie Arehart -- carehart.org --->
<!--- Find cookies other than CFID/CFTOKEN --->
<cfparam name="test" default="1">

<!--- Tag-based structure loop --->
<cfloop collection="#cookie#" item="cname">
   <cfif cname neq "cfid" and cname neq "cftoken">
      <cfoutput>
         #cname# <br>
      </cfoutput>
   </cfif>
</cfloop>

<p>

<cfscript>
// Script-based structure loop
```

Listing 44.1 (continued)

```
for (cname in cookie) {
   if (cname != "cfid" && cname != "cftoken"){
      writeoutput(cname & "<br>");
   }
}
</cfscript>
```

For many, the scripted loop is more readable and intuitive than the tag-based loop. It uses familiar constructs from other script-based languages (such as `//` for comments, `!=` for "not equal," `&&` for "and," semicolons to end each statement, bracketed `loop` and `if` blocks, and not a single pound-sign needed). While this sort of syntax may be foreign to long-time ColdFusion developers, it's found in many other languages and so makes some developers more comfortable.

Even so, scripting may also appeal to traditional CFML developers, at least in some circumstances. Often, a block of code will simply look cleaner if replaced with a script-based equivalent (a good example is a long list of `<cfset>` tags.) Similarly, ColdFusion 8's new implicit array and structure creation tags, though not unique to CFSCRIPT, do add to the improved experience of creating cleaner, more concise CFML code. (Implicit array and structure creation is discussed in Chapter 8, "The Basics of CFML," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 1: Getting Started*.)

As you'll learn while you read this chapter, ColdFusion scripting has various differences (some advantages and disadvantages) compared to tag-based CFML.

What You Can and Cannot Do with `<cfscript>`

`<cfscript>` is a useful alternative to tag-based CFML, but you can't just replace all your tag-based CFML with scripting because there are things you can do with tags that you can't do with scripting. For example, scripting can only contain variable assignments, flow-control logic, exception handling, and function calls. You can't directly query a database, read a file from disk, or perform any other task that requires the use of CFML tags.

The following section explains the differences between `<cfscript>` and tag-based syntax. In each section, the `<cfscript>` example is shown first, followed by the tag-based version.

Assignment

```
myVariable = "Value";

<cfset myVariable = "Value">
```

Output

```
Writeoutput(somevar);

<cfoutput>#somevar#</cfoutput>
```

If-Elseif-Else

```
if(condition1) {
  [logic];
} else if(condition2) {
  [logic];
} else {
  [logic];
}

<cfif condition1>
  [logic]
<cfelseif condition2>
  [logic]
<cfelse>
  [logic]
</cfif>
```

Switch-Case

```
switch(expression) {
  case "value1":
    [logic];
    break;
  case "value2":
    [logic];
    break;
  default:
    [logic];
}

<cfswitch expression="">
  <cfcase value="value1">
    [logic]
  </cfcase>
  <cfcase value="value2">
    [logic]
  </cfcase>
  <cfdefaultcase>
    [logic]
  </cfdefaultcase>
</cfswitch>
```

For Loops

```
for(i=1; i <= 10; i++) {
  [logic];
}

<cfloop index="i" from="1" to="10">
  [logic]
</cfloop>
```

While Loops

```
while(expression) {
  [logic];
}

<cfloop condition="#expression#">
  [logic];
</cfloop>
```

Do-While Loops

```
do {
  [logic];
} while(expression);

(No tag-based equivalent)
```

For-In Loops

```
for(key in structure) {
  [logic];
}

<cfloop collection="#structure#" item="key">
  [logic]
</cfloop>
```

Array Loops

```
for(i=1; i <= ArrayLen(array); i++) {
  [logic]
}

<cfloop array="#array#" index="currentArrayElement">
  [logic]
</cfloop>
```

List Loops

```
for(i=1; i <= ListLen(list); i++) {
  [logic]
}

<cfloop list="#listOfValues#" index="currentListItem">
  [logic]
</cfloop>
```

Query Loops

```
(no script-based equivalent)

<cfloop query="queryName">
  [logic]
</cfloop>
```

Break

```
break;

<cfbreak>
```

Continue

```
continue;

(No tag-based equivalent)
```

Object Processing

```
cfobject(…);

<cfobject …>
```

Exception Handling

```
try {
  [logic];
}
catch(Expression exception) {
  [logic];
}
catch(Any exception) {
  [logic];
}

<cftry>
  [logic]
  <cfcatch type="Expression">
    [logic];
  </cfcatch>
  <cfcatch type="Any">
    [logic];
  </cfcatch>
</cftry>
```

Throwing a Custom Exception

```
(no script-based equivalent)

<cfthrow
  type="BusinessRule.Customer"
  errorCode="66020"
  message="Non-wholesale customer"
  detail="Only registered wholesale customers may place bulk orders.">
```

Rethrowing a Caught Exception

```
(no script-based equivalent)

<cfrethrow>
```

Functions

```
function myFunction(arg1, arg2) {
  var value = 0;
  [logic];
  return value;
}

<cffunction name="myFunction" returntype="numeric">
  <cfargument name="arg1" type="numeric">
  <cfargument name="arg2" type="numeric">
  <cfset var value = 0>
  [logic]
  <cfreturn value>
</cffunction>
```

Besides the few examples above which indicate no script-based equivalent, there are some other differences between CFML tags and Script-based alternatives, such as with switch-case and try-catch processing, which will be explored later in the sections, "Flow Control Using `<cfscript>`"and "Exception Handling in `<cfscript>`," respectively,

Differences Between `<cfscript>` and JavaScript

You'll notice that <cfscript> looks a lot like JavaScript and other familiar scripting languages, but there are some differences compared to those as well. First, a difference compared to JavaScript is that ColdFusion is absolutely fanatical about terminating all statements with a semicolon. Another major difference is that ColdFusion scripting is executed entirely on the server side, as opposed to JavaScript, which is executed entirely in the user's browser. There is no Document Object Model (DOM) for ColdFusion scripting to access or manipulate, because ColdFusion has no sense of a "document."

Why Use `<cfscript>`?

Sure, ColdFusion scripting does offer a few things that tag-based CFML lacks, such as `do-while` loops and the ability to skip loop iterations using `continue`. On the other hand, there are just as many if not more things offered by tag-based CFML that ColdFusion scripting lacks. So why use `<cfscript>` for anything other than its few unique capabilities?

- `<cfscript>` integrates a familiar, JavaScript-like syntax into ColdFusion.
- `<cfscript>` provides access to all ColdFusion variables and objects, using a more-intuitive and easier-to-read format.
- `<cfscript>` supports all ColdFusion functions and introduces a few features and functions not available to its tag-based counterpart.

Implementing `<cfscript>`

To use ColdFusion scripting in your templates, place a `<cfscript></cfscript>` tag pair in a template, and then write ColdFusion scripting statements in the body of the tag.

You can intersperse `<cfscript>` blocks with blocks of tag-based CFML throughout a ColdFusion template, but you should strive to combine your ColdFusion scripting into as few `<cfscript>` blocks as is feasible. ColdFusion Server executes both ColdFusion scripting statements and tag-based CFML in the order they are placed on the page, so if scripting statements create variables or other objects that will be used by tag-based CFML, then you must place that script before those tags.

The only exception to this rule is with user-defined functions, which you can call before they appear in the template. However, it is considered a best practice to define functions above the statements that call them. This is less confusing for you and other developers because the code reads in a natural order.

Creating and Using Variables

To assign a value to a variable in a script, just provide the variable name, the equal sign (=), and then the actual value you want to assign to the new variable. If you want, the value can come from an expression made up of any valid combination of ColdFusion operators and functions.

In Listing 44.2, I'm showing both the scripting and tag-based versions of the same variable assignments so you can compare them. Try commenting and uncommenting each section and perhaps varying the values being assigned.

Listing 44.2 `CreatingVariables.cfm`—Creating Variables Using ColdFusion Scripting

```
<!--- Author: Adam Phillip Churvis -- ProductivityEnhancement.com --->
<!--- Creating variables --->

<!--- A simple variable --->
<cfset welcomeMessage = "Welcome to ColdFusion scripting!">

<!--- An array --->
<cfset gameScores = ["93","87","96"]>


<!--- A structure --->
<cfset player = {name="John Doe", age="27", gender="Male"}>

<cfscript>
  // A simple variable
  welcomeMessage = "Welcome to ColdFusion scripting!";

  // An array
  gameScores = ["93","87","96"];
  // A structure
    player = {name="John Doe", age="27", gender="Male"};
</cfscript>

<p><cfoutput>#welcomeMessage#</cfoutput></p>
<p><cfdump var="#gameScores#" label="gameScores"></p>
<p><cfdump var="#player#" label="player"></p>
```

Basically, to convert a tag-based variable declaration into a script-based declaration, you just use the guts of the `<cfset>` tag and terminate it with a semicolon.

Commenting Code

Code comments in ColdFusion scripting differ from those in tag-based code, as shown in Listing 44.3.

Listing 44.3 `CommentingCode.cfm`—Comparing Code Comments in Script vs. Tags

```
<!--- Author: Adam Phillip Churvis -- ProductivityEnhancement.com --->
<!--- Types of comments --->

<!--- Tag-based single-line comment before code block --->
<cfset example = "before block">

<cfset example = "inline">  <!--- Tag-based single-line comment after code--->

<!--- Tag-based
multi-line comment --->
<cfset example = "multi-line">

<cfscript>
  // Script-based single-line comment before code block
  example = "before block";

  example = "inline";  // Script-based single-line comment after code

  /* Script-based
  multi-line comment */
  example = "multi-line";
</cfscript>
```

NOTE

Don't feel embarrassed when your script breaks because you accidentally put a tag-based comment inside your `<cfscript>` code. We've all done it more times than we care to admit.

Calling ColdFusion Functions

Calling a ColdFusion function in script is just like assigning a static value to a variable—you're just assigning the result of a function call instead of a static value. Take a look at Listing 44.4.

Listing 44.4 `ColdFusionFunctions.cfm`—Calling ColdFusion Functions from ColdFusion Script

```
<!--- Author: Adam Phillip Churvis -- ProductivityEnhancement.com --->
<!--- Calling ColdFusion functions within cfscript --->

<cfscript>
  todaysDate = Now();
  dateDisplay = "The date is #DateFormat(todaysDate, 'dddd, mmmm d, yyyy')#
    and the time is " & TimeFormat(todaysDate, 'h:mm tt');
</cfscript>

<cfoutput>#dateDisplay#</cfoutput>
```

Flow Control Using `<cfscript>`

As already mentioned, there are some important differences between how CFML and `<cfscritp>` perform similar operations. Now let's see how those differences manifest in actual code.

`if-elseif-else` Constructs

`If-else` constructs are used to control flow based on a single test, and `if-elseif-else` constructs are used to control flow based on multiple tests. In Listing 44.5, the first test is based on the value of `colorCode`, and the second test is based on the value of `temperature`. If both of those tests are false, then control flows to the `else` clause.

Listing 44.5 `IfElseifElse.cfm`—Deciding Flow Based on Multiple Tests

```
<!--- Author: Adam Phillip Churvis -- ProductivityEnhancement.com --->
<!--- If-Elseif-Else --->
<!--- Call this page using various URL values for colorCode and emergency --->
<cfparam name="url.term" default="red">
<!--- Tag-based --->
<cfif URL.colorCode EQ "Red">
  <cfset result = "Emergency">
<cfelseif URL.temperature GTE "100">
  <cfset result = "Hot">
<cfelse>
  <cfset result = "Normal">
</cfif>

<cfscript>
  // Script-based
  if(URL.colorCode == "Red") {
    result = "Emergency";
  } else if(URL.temperature >= "100") {
    result = "Hot";
  } else {
    result = "Normal";
  }
</cfscript>

<cfoutput>#result#</cfoutput>
```

Try using these URLs to see how Listing 44.5 reacts:

```
<!--- Displays "Emergency" --->
http://localhost:8500/chapter44/ifelseifelse.cfm?colorcode=red

<!--- Displays "Normal" --->
http://localhost:8500/chapter44/ifelseifelse.cfm?colorcode=blue&temperature=72

<!--- Displays "Hot" --->
http://localhost:8500/chapter44/ifelseifelse.cfm?colorcode=green&temperature=115
```

Also try commenting and uncommenting the script-based and syntax-based sections of code to see that each section works exactly the same.

Whenever you have more than one test that controls the flow of logic, you'll use an `if-elseif-else` construct. Don't confuse multiple *tests* with multiple *results* from a single test, though. That sort of flow control is best handled by a `switch-case` construct, which is discussed next.

`switch-case` Constructs

If you have a single test that can have an enumerated set of results (in other words, a specific set of predefined result values), and these results need to control the flow of logic, then you'll use a `switch-case` statement, as in Listing 44.6.

Listing 44.6 `SwitchCase.cfm`—Deciding Flow Based on a Single Test with Enumerated Results

```
<!--- Author: Adam Phillip Churvis -- ProductivityEnhancement.com --->
<!--- Switch-Case --->

<!--- Tag-based --->
<cfset status = "Moderate">
<cfswitch expression="#status#">
  <cfcase value="Low">
    <cfset score = 25>
  </cfcase>
  <cfcase value="Subnormal,Moderate,Elevated">
    <cfset score = 50>
  </cfcase>
  <cfcase value="High">
    <cfset score = 75>
  </cfcase>
  <cfdefaultcase>
    <cfset score = 0>
  </cfdefaultcase>
</cfswitch>

<cfscript>
  // Script-based
  status = "Moderate";
  switch(status) {
    case "Low":
      score = 25;
      break;

    case "Subnormal":
    case "Moderate":
    case "Elevated":
      score = 50;
      break;

    case "High":
      score = 75;
      break;

    default:
      score = 0;
  }
</cfscript>

<cfoutput><p>#score#</p></cfoutput>
```

Notice that there are more `case` statements in the script version than in the tag-based version. One of the useful features of the `<cfcase>` tag is that you can specify a comma-separated list of values to satisfy a case. Unfortunately, this isn't possible with the script version because it follows the JavaScript model of *fall-through* and *breaks*.

Fall-through is a feature that causes code to continue executing until a `break` statement is reached. For example, the second, third, and fourth `case` statements in Listing 44.6 are all grouped together. So if the value of `status` is `Subnormal`, flow of control will begin at that `case` statement and then continue executing lines of code until a `break` statement is reached. This means the next line of code to execute would be `case "Moderate":`, which wouldn't do anything; then `case "Elevated":`, which also wouldn't do anything; and then `score = 50;`, which would assign `50` to the `score` variable. Finally, the `break;` line would be processed, and flow of control would pass to the end of the `switch` construct and then continue to the `<cfoutput>` line.

So as you start writing `switch-case` constructs in ColdFusion scripting, make sure to include a `break;` statement at the end of each `case` (except the `default` case, which is at the end of the `switch` construct and so doesn't need a `break`).

Loops

Of course, no scripting language would be complete without ways to place a chunk of code into a loop of some kind. ColdFusion scripting provides for four types of loops: `for`, `while`, `do-while`, and `for-in`. Unfortunately, ColdFusion scripting doesn't provide for query loops or list loops.

`for` Loops

`for` loops execute a block of script a predetermined number of times and use a numerical index as a starting point. This index is incremented or decremented with each iteration of the loop, and looping continues until the index reaches the ending value. Listing 44.7 demonstrates a typical `for` loop.

Listing 44.7 `ForLoops.cfm`—Looping a Predetermined Number of Times

```
<!--- Author: Adam Phillip Churvis -- ProductivityEnhancement.com --->
<!--- Edited by: Charlie Arehart, carehart.org --->
<!--- For loop (used when you can determine the number of loops beforehand) --->

<!--- Tag-based --->
<cfset colorCodes = ["Red,Orange,Yellow"]>
<cfloop index="i" from="1" to="#ArrayLen(colorCodes)#">
  <cfoutput><p>#colorCodes[i]#</p></cfoutput>
</cfloop>

<cfscript>
  // Script-based
  colorCodes = ["Red,Orange,Yellow"];
  for(i=1; i LTE ArrayLen(colorCodes); i=i+1) {
    WriteOutput("<p>" & colorCodes[i] & "</p>");
  }
</cfscript>
```

`for` loops can be used to loop over the items in a list; the number of iterations is simply the result of the function `ListLen()`.

`while` Loops

`while` loops execute a block of script as long as a condition is `true`. When the condition is no longer `true`, the loop exits. Listing 44.8 shows how a `while` loop could be used.

Listing 44.8 `WhileLoops.cfm`—Looping While a Condition Is True

```
<!--- Author: Charlie Arehart -- carehart.org --->
<!--- While loop (used when the number of loops depends on a condition) --->

<!--- Tag-based --->
<cfset rate=5>
<cfset init = 1>
<cfset total = init>
<cfset years = 0>
<cfloop condition="total lt 2">
   <cfset total = total + (total*rate/100)>
   <cfset years++>
</cfloop>
<cfoutput>At a rate of %#rate# interest, it will take #years# years for $#init# to
at least double, to #dollarformat(total)#</cfoutput><p>

<cfscript>
  // Script-based
  rate=5;
  init = 1;
  total = init;
  years = 0;
  while(total lt 2) {
    total = total + (total*rate/100);
    years++;
  }
  writeoutput("At a rate of %#rate# interest, it will take #years# years for $#init#
to at least double, to #dollarformat(total)#");
</cfscript>
```

`while` loops are most often used when you don't know how many times a loop should iterate—or whether the loop should iterate at all—but you can formulate an expression that, while `true`, permits the loop to iterate. If you attempt to enter a `while` loop while the expression evaluates to `false`, the loop will never iterate at all.

`do-while` Loops

Unlike `while` loops, `do-while` loops always iterate at least once. This occurs because the `while` condition's expression is evaluated at the end of each iteration rather than before, and if the condition defined in the `while` clause is `false`, the loop exits. Otherwise, the loop continues while the condition is `true`. In Listing 44.9, we create a typical `do-while` loop.

Listing 44.9 `DoWhileLoops.cfm`—Looping At Least Once and While a Condition Is True

```
<!--- Author: Charlie Arehart -- carehart.org --->
<!--- Do-While loop (a conditional loop that will iterate at least once) --->

<!--- Tag-based --->
<!--- (There is no tag-based equivalent to a do-while loop) --->

<cfscript>
  // Script-based
  // Calculates factorial of counter (eg, for 4, caclulate 4 * 3 * 2 * 1)
  counter=4;
  factorial=1;
  do {
    factorial = factorial * counter ;
    counter--;
  } while(counter > 0);
  writeoutput (factorial);
</cfscript>
```

`do-while` loops are most often used in scenarios where you are certain to perform an operation on a nonempty value or set of elements, and each iteration of the loop may affect the number remaining in the set. Notice that the above code would have executed even if the starting counter value was 0, in which case factorial would be set (correctly) to 0.

`for-in` Loops

`for-in` loops iterate through a block of script once for every item in a structure and populate a local variable with the name of the key of the current item. Once every item in the structure has been looped through, the loop exits. Listing 44.10 demonstrates how a `for-in` loop iterates over a ColdFusion structure.

Listing 44.10 `ForInLoops.cfm`—Looping Over the Keys in a Structure

```
<!--- Author: Adam Phillip Churvis -- ProductivityEnhancement.com --->
<!--- Edited by: Charlie Arehart, carehart.org --->
<!--- For-In loop (used to loop over a structure or "collection") --->

<!--- Tag-based --->
<cfset player = {name="John Doe", age="27", gender="Male"}>
<cfloop collection="#player#" item="keyName">
  <cfoutput><p>#keyName#: #player[keyName]#</p></cfoutput>
</cfloop>

<cfscript>
  // Script-based
  player = {name="John Doe", age="27", gender="Male"};
  for(keyName in player) {
    WriteOutput("<p>#keyName#: #player[keyName]#</p>");
  }
</cfscript>
```

`for-in` loops are useful for testing or summarizing the values of keys in a structure. This comes in handy when you consider that ColdFusion variable scopes are all structures.

Using `continue` and `break`

Sometimes you may want to abort a particular iteration of a loop without aborting the loop itself—in other words, you want to skip the remaining code in the current iteration and `continue` at the beginning of the next iteration of the loop. Other times, your logic may encounter a condition that requires you to `break` out of the loop completely. ColdFusion scripting easily handles both of these situations (whereas tag-based code can directly handle only breaking out of a loop).

Listing 44.11 loops over an array of quantities in a shopping cart. If a quantity is less than 100, there is no wholesale discount applied, so the line of code that outputs the wholesale discount message is skipped by the `continue` statement. Likewise, if someone attempts to order more than 10,000 of any given item, the shopping cart is considered invalid and the flow of logic breaks out of the loop entirely. Notice also that there is no tag-based equivalent to `continue`.

Listing 44.11 `ContinueAndBreak.cfm`—Skipping Iterations and Breaking Out of a Loop

```
<!--- Author: Adam Phillip Churvis -- ProductivityEnhancement.com --->
<!--- Break and Continue --->

<!--- Tag-based --->
<!--- (There is no tag-based equivalent to Continue, so we'll kludge one) --->
<cfset quantityOrdered = [16,400,25000,47]>
<cfloop index="i" from="1" to="#ArrayLen(quantityOrdered)#">
  <cfset continue = FALSE>
  <cfoutput><p>Quantity ordered: #quantityOrdered[i]#</p></cfoutput>
  <cfif quantityOrdered[i] LTE 100>
    <cfset continue = TRUE>
  </cfif>

  <cfif NOT continue>
    <cfif quantityOrdered[i] GT 10000>
      <p>You cannot request more than 10000 items per order.</p>
      <p>Order terminated.</p>
      <cfbreak>
    </cfif>

    <p>(wholesale discount will be applied to previous item)</p>
  </cfif>
</cfloop>

<cfscript>
  // Script-based
  quantityOrdered = [16,400,25000,47];
  for(i = 1; i <= ArrayLen(quantityOrdered); i++) {
    WriteOutput("<p>Quantity ordered: #quantityOrdered[i]#</p>");
    if(quantityOrdered[i] <= 100) {
      continue;
    }

    if(quantityOrdered[i] > 10000) {
      WriteOutput("<p>You cannot request more than 10000 items per order.</p>");
      WriteOutput("<p>Order terminated.</p>");
      break;
    }
```

Listing 44.11 (continued)

```
      WriteOutput("<p>(wholesale discount will be applied to previous item)</p>");
    }
  </cfscript>
```

Notice the rather clumsy way we had to handle "continuing" in the tag-based version? Good enough reason to use scripting rather than tags for such a loop.

Common Problems and Solutions

Even with the removal of many incompatibilities in script-based operators (as compared to JavaScript and other languages), there are some challenges as ones switches between using tag- and script-based coding. To resolve errors in your scripts, follow these simple guidelines:

- Make sure each statement ends with a semicolon (;).
- For every opening `(` and `{`, make sure there is a corresponding closing `)` and `}`.
- Check for closing tag > symbols when converting tag-based code to script. It's easy to forget to remove the > symbol at the end of each tag, which you need to replace with a semicolon (;).
- Make sure your statements don't contain any ColdFusion tags, because `<cfscript>` doesn't allow these.

A Note About the `{}` Symbols

As you've seen in the script examples so far, `<cfscript>` uses `{}` to create blocks of script code that the `if` or `else` part of an `if-else` statement should execute. Actually, the use of `{}` is optional if either the `if` or `else` keyword is handling only one line of code. If you leave the brackets out, however, things can get confusing should you need to add additional lines to your code later. For that reason, including these symbols in your script code is a good habit, even when they aren't explicitly needed.

The following code is an example of when you need to use `{}`. By itself, this code throws an error:

```
<cfscript>
  a = 1;
  if(a == 1)
   b = 1;
   b = 2;
  else
   b = 3;
</cfscript>

<cfoutput>#b#</cfoutput>
```

By definition, this code should set the value of 2 to the variable b, but it doesn't do that because the second expression, b = 2, isn't evaluated. To make this code behave correctly, we must insert {} around clauses that contain two or more expressions:

```
<cfscript>
  a = 1;
  if(a == 1) {
   b = 1;
   b = 2;
  } else {
   b = 3;
  }
</cfscript>

<cfoutput>#b#</cfoutput>
```

Now the code works as it should and returns 2 as the value held in the variable b.

A Note About Quotes

As in JavaScript, strings in <cfscript> containing single or double quotes (' or ") are a source of confusion and error. Any string surrounded with double quotes cannot contain unescaped double quotes within the string. In other words, if this string contains a quoted word or phrase, we have to escape each instance of double quotes with another instance of double quotes. To see a string that causes an error, follow the next example:

```
<cfscript>
  MyVar = "Take it to the "next" level...";
</cfscript>
```

Because the quotes delimit a string, having quotes of the same type within the string causes the interpreter to end prematurely. You can work around this issue by alternating between double and single quotes within a string. If double quotes surround your string, use single quotes within it:

```
<cfscript>
  MyVar = "Take it to the 'next' level...";
</cfscript>
```

If you need to maintain double quotes within your string, simply insert a second set to escape them:

```
<cfscript>
  MyVar = "Take it to the ""next"" level...";
</cfscript>
```

TIP

Unlike JavaScript, <cfscript> does not allow you to escape quotes with the backslash character (\).

User-Defined Functions in <cfscript>

To create a custom function using <cfscript>, you declare your function just as you would in JavaScript, using the function keyword. Like JavaScript, UDFs can accept and return values and can be embedded within other functions. And because UDFs support recursion, you can call each

function from within the function itself. (User-defined functions can be created using `<cffunction>`, as well.)

Listing 44.12 shows an example of a `<cfscript>`-based UDF named `TitleCase()` that takes a string as its one argument and returns the string converted to title case (with the first letter of each word capitalized, and the rest in lowercase). `TitleCase()` can be called like any other function, as shown in the listing.

Listing 44.12 `UserDefinedFunction.cfm`—Creating and Calling a UDF

```
<!--- Author: Adam Phillip Churvis -- ProductivityEnhancement.com --->
<!--- Script-based user defined function (UDF) --->

<cfscript>
  function TitleCase(rawSentence) {
    var titleCaseSentence = "";
    var numberOfListElements = ListLen(rawSentence, ", .;-:");
    var currentElement = "";
    var currentWord = "";
    for(i=1; i <= numberOfListElements; i++) {
      currentElement = ListGetAt(rawSentence, i, ", .;-:");
      currentWord = UCase(Left(currentElement, 1));
        if(Len(currentElement) > 1) {
          currentWord = currentWord &
            LCase(Right(currentElement, Len(currentElement) - 1));
        }
      titleCaseSentence = titleCaseSentence & " " & currentWord;
    }
    titleCaseSentence = Trim(titleCaseSentence) & ".";
    return titleCaseSentence;
  }
</cfscript>

<cfoutput>
  #TitleCase("i AM the very moDEL of a MODERN major general.")#
</cfoutput>
```

Where to Define UDFs

You can define UDFs in two places: in the same ColdFusion template where the function is called, or in a separate `.cfm` file that is included in the same ColdFusion template where the function is called. You can also assign them into shared variable scopes to permit reuse across multiple requests, or place them into a CFC (an instance of which itself could be assigned to a shared variable scope). Which technique should you use? It depends on how and where the function will be used.

If the user-defined function is used only in a single ColdFusion template, you can define the function directly in that template, typically at the top of the page. But since UDFs are built for reuse, you'll most likely have occasion to call the same UDFs from multiple templates. In that case, you could define your UDF in an external `.cfm` file and then `<cfinclude>` that file in the ColdFusion templates that call the UDF.

But what if your application uses many different UDFs? To accommodate this, you could categorize your UDFs by the type of function they perform and then group them together in `.cfm` files that are named for each type of function. For example, if you have three functions that manipulate strings and seven functions that handle mathematical calculations, you might create two files named `StringFunctions.cfm` and `MathFunctions.cfm` and place the function definitions in these two files. Then, whenever a ColdFusion template needs to call a math function, you would simply `<cfinclude>` the `MathFunctions.cfm` in that template and then call the function wherever it was needed.

That approach was our only option prior to ColdFusion MX 6, but with the advent of CFCs, it may be better to place such highly reused user-defined functions in a CFC instead. In fact, you may be surprised to learn that you can place a script-based UDF into a CFC and call it just like any other CFC method. Clearly though, `<cffunction>`-based UDFs offer a great deal more power, with definable arguments, return types, ability to make them remote, etc. See Chapter 24, "Building User-Defined Functions," in *ColdFusion 8 Web Application Construction Kit, Volume 2: Application Development*," for more details. While you may want to convert such script-based UDFs to use `<cffunction>` instead when inside a CFC, note that you could leave the bulk of the UDF within `<cfscript>` inside the `<cffunction>`, removing only the opening/closing function declaration block.

Finally, just as CFC instances can be placed into shared variable scopes for reuse across multiple requests, note as well that user-defined functions (defined outside a CFC, whether with `<cfscript>` or `<cffunction>`) can also be placed into shared variable scopes. This is beyond the scope of this chapter to detail, but just know that you can assign a function to a variable just like any other assignment and can then refer to the function as being within that scope just as you would a scope-based variable.

There are a few things you should keep in mind about user-defined functions:

- The UDF name cannot match a built-in function name and cannot start with `cf`.
- The number of arguments passed in the function call must be greater than or equal to the number of arguments declared in the script-based function definition.
- The `var` keyword initializes variables local to the function. All variables created inside a user-defined function should be declared as local function variables using the `var` keyword. You must declare all local function variables immediately after the function is declared and before any logic is executed. Arguments are already local to the function that defines them, so you don't need to separately declare them in the body of the function.

No discussion of user-defined functions would be complete without referring you to `cflib.org`, a Web site that contains hundreds of useful UDFs and libraries of UDFs available for free download and use. Do yourself a favor and head over there now for a quick tour of what they offer.

Exception Handling in `<cfscript>`

CFML Error handling in general is covered in Chapter 51, "Error Handling," online. If you're already familiar with implementing structured exception handling in your CFML code, then you're

also familiar with using `<cftry>-<cfcatch>` constructs, and the various types of errors that can be handled by specifying type attributes in your `<cfctach>` blocks.

Fortunately, ColdFusion scripting provides an equivalent with `try-catch`. Listing 44.13 demonstrates a comparison of tag-based and script-based try-catch handling. To run Listing 44.13, note that there it tries to open a file named `test.txt` which has been provided in the sample source code. Try changing filename to `test2.txt` (which does not exist) and notice the types of exceptions that are thrown, caught, and handled.

Listing 44.13 `ExceptionHandling.cfm`—Handling Exceptions in ColdFusion Script

```
<!--- Author: Charlie Arehart -- carehart.org --->
<!--- Exception handling --->

<cfset filename="test.txt">
<!--- Tag-based --->
<cftry>
  <cfset result = FileOpen(expandpath(filename))>
  <p>It worked!</p>

  <cfcatch type="Expression">
    <p>An Expression exception was thrown.</p>
   <cfoutput>#cfcatch.message#</cfoutput>
  </cfcatch>
  <cfcatch type="Security">
    <p>A Security exception was thrown.</p>
   <cfoutput>#cfcatch.message#</cfoutput>
  </cfcatch>
</cftry>

<cfscript>
  // Script-based
  try {
    result = FileOpen(expandpath(filename));
    WriteOutput("<p>It worked!</p>");
  }
  catch(Expression exceptionVariable) {
    WriteOutput("<p>An Expression exception was thrown.</p>");
    WriteOutput("<p>#exceptionVariable.message#</p>");
  }
  catch(Security exceptionVariable) {
    WriteOutput("<p>A Security exception was thrown.</p>");
    WriteOutput("<p>#exceptionVariable.message#</p>");
  }
</cfscript>
```

There are a couple of differences between `try-catch` in `<cfscript>` and the tag-based equivalents. First, note that the tag-based `<cfcatch>` is placed within a `<cftry>`, where a script-based `catch` is placed after a `try`. Also, note that with `<cfcatch>`, there's an available, pre-defined catch scope holding details of the error, where with the `catch` statement, you instead name a variable to hold those details.

Perhaps a more significant difference is that you can't execute tags that you might normally for error handling, such as `<cfmail>`, `<cflog>`, `<cfquery>`, etc. At least in tag-based try-catch processing, you could `<cfrethrow>` a caught exception or `<cfthrow>` a custom exception of your own design, to be handled by a higher-level error handler. Unfortunately, there is no way to rethrow an exception once it is caught in the scripting version of a `try-catch` construct; nor can you throw custom exceptions in ColdFusion scripting. You could, however, call a `<cffunction>-based` user-defined function, which then could perform normal tag-based processing to send email, create a log, etc.

Script-based processing in CFML is just another option in your CFML toolbelt. There may be times when its simpler syntax (fewer tags) will appeal to you as you write code. More often, it will appeal to those coming to CFML from other languages. As so often with ColdFusion, the choice is yours.

CHAPTER 45 Working with XML

IN THIS CHAPTER

XML Document Structure E181
Reading and Creating XML Documents E184
Using Special Characters in XML E191
XML Namespaces E193
Validating XML E196
More XML Resources E204

XML files (eXtensible Markup Language) have become a common way to share and exchange data between systems and applications, as well as between servers and browsers. In contrast to more traditional data exchange formats, like CSV and other flat-file formats, XML files describe and give structure to content by embedding it within tags.

In ColdFusion, we may be interested in reading or writing XML files. They can take the form of flat text files, content generated in a web page or web service, data shared via an RSS feed (see Chapter 69, "Working with Feeds," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 3: Advanced Application Development*), and so on. Various CF8 features also work with XML, including `<CFPDF>`, `<CFREPORT>`, as well as older tags like `<CFWDDX>`.

This chapter gives you a practical description of how XML is used in ColdFusion 8. We will cover just the basics of reading, writing, and returning XML, XML schemas, namespaces, and other XML concepts; an exhaustive discussion of the subject and all uses of XML in CFML would require a book of its own.

XML Document Structure

Listing 45.1 is an XML document that describes a company's employees.

Listing 45.1 `Company.xml`—An XML Document

```
<?xml version="1.0" ?>
<company name="ABC MegaCorp, Inc." location="NY">
  <comments>A very big company that does many different things.</comments>
  <employee ssn="123-45-6789">
    <first-name>Ed</first-name>
    <last-name>Johnson</last-name>
    <department>Human Resources</department>
    <children>
```

Listing 45.1 (continued)

```
        <child name="Sean" />
        <child name="Polly" />
      </children>
    </employee>
    <employee ssn="541-29-8376">
      <first-name>Maria</first-name>
      <last-name>Smith</last-name>
      <department>Accounting</department>
      <children>
        <child name="Sandra" />
      </children>
    </employee>
    <employee ssn="568-73-1924">
      <first-name>Eric</first-name>
      <last-name>Masters</last-name>
      <department>Accounting</department>
    </employee>
  </company>
```

Let's break down the composition of this document.

The first line is called an *XML Declaration*. This tells an XML parser that the ensuing content is intended to be properly formed XML that it conforms to version 1.0 of the XML specification. Though this declaration is not required in all XML documents, it's a good habit to always put it at the top of the document.

The second line is the opening tag for this document's *root element*. Every XML document must have one and only one element at the root of its hierarchy, and in this case the root element is `company`.

The `company` element contains a `comments` element that describes the company. Between the opening and closing tags of the `company` element, there are `employee` elements for each of the company's employees. The `employee` element contains an `ssn` attribute as well as `first-name`, `last-name`, and `department` elements, as well as an optional `children` element to describe the employee's children.

While an XML document may seem familiar compared to HTML documents, the structure of an XML document is more formal. XML is case-sensitive (so `<a>` and `<A>` are two different tags), whereas HTML is not (so `<a>` and `<A>` are equivalent). XML also requires that all attribute values be surrounded with double quotes. In HTML, we could write `location=CA`, but XML requires that we write `location="CA"` instead.

Probably the biggest problem people have with XML concerns the requirement for closing tags. In an XML document, every tag must be closed. Even if the tag has no content (such as the `child` elements in Listing 45.1), you must close the tag; if you don't, the XML parser that processes the document will throw an error. In the case of the `<child>` tags in Listing 45.1, we used a special shorthand closing syntax: `<child name="Sandra" />` is equivalent to `<child name="Sandra"></child>`.

One last note—the following characters are illegal in attribute values and element content and must be escaped using their equivalent entity escape codes:

- Ampersand (&): `&`
- Greater-than sign (>): `>`

- Less-than sign (<): `<`
- Double quote ("): `"`

Fortunately, CFML includes an `XMLFormat` function to help with this challenge. That and entity references will be discussed in more detail later in the chapter.

Following the example and guidelines outlined in this section ensures that an XML document is *well formed*, meaning that it follows the standard rules of XML document structure. Any XML parser should be able to read a well-formed document.

Elements and Their Attributes

In listing 45.1, notice that the description of data about each employee was provided as a nested XML element, such as `<first-name>`. You may have wondered why such information wasn't offered as attributes of the `<employee>` element, as with the `ssn` attribute. There is flexibility in XML design about when to use elements to store data and when to use attributes. For instance, this portion of Listing 45.1:

```
<employee ssn="568-73-1924">
  <first-name>Eric</first-name>
  <last-name>Masters</last-name>
  <department>Accounting</department>
</employee>
```

could also have been represented like this:

```
<employee ssn="568-73-1924" first-name="Eric" last-name="Masters"
department="Accounting" />
```

Some people like using elements; others prefer attributes. Different groups claim that one method is inherently superior to the other, but this is not really true. The decision as to whether to use an attribute or a child element in any given circumstance is one that should be made according to the developer's opinion of what will work for that situation.

Most important, the decision for choosing the XML format rests with whoever creates the XML file (or designates how XML files are to be created). If you're reading someone else's XML or creating XML to be read by someone else's process, you have no choice but to process the XML according to that defined format. If you're creating an XML file to let others read it and there's no defined format, you're then free to decide how to format the XML.

In making your decision, here a few rules to keep in mind:

- All attributes of a single element must have unique names.
- Attributes cannot contain embedded tags. If something has a substructure of its own, it must be represented as an element.
- Child elements are often more difficult to handle in code than are attributes. This is because accessing an element's content means using a property, whereas accessing an attribute can be done directly.

Naming Conventions

There seem to be as many "standard" XML naming conventions as there are XML developers. Some people would name my `first-name` element from Listing 45.1 `FirstName`; others would name it `first_name`; yet others would name it `fn`. All of these are valid names, but `first-name` may be the simplest and easiest for most of us to understand.

There is no one standard naming convention. Rather, there is the naming convention created by whoever defined the XML file format you're reading from or writing to, or if you're creating the XML file format, use what is most comfortable for you and the other developers on your team. The conventions presented here are merely a guideline.

- Make all element and attribute names lowercase, because XML is case-sensitive. On most platforms, if you have a `first-name` element and you look for an element named `First-Name`, the application will not find the element you're looking for. Because different people have different capitalization rules, it's best to eliminate the issue entirely and use lowercase for all names.
- Use hyphens to separate multiple words in an element or attribute name. Some teams use underscores, but I avoid them because it's often difficult to see an underscore when code is underlined or outlined in an IDE. Hyphens are always easy to see.
- Don't abbreviate element names unless you absolutely must. XML is known for being a very verbose format. Because of this, developers often abbreviate the names of elements, using `fn`, for instance, instead of the more verbose `first-name`. However, `fn` does not describe what the element does, whereas `first-name` describes it perfectly. Remember that XML was invented to make data understandable by both machines and humans.

Reading and Creating XML Documents

In this section, we'll show three ways to work with XML documents, whether reading them from a file or URL using `XMLParse()`, or creating them within CFML using either `<CFXML>` or `XMLNew()`.

If you want to make it easy to read and/or manipulate the XML you may obtain from a file or web page, you can convert it into a ColdFusion-specific *XML object*, using the `XMLParse()` function. The XML as found in a file or web page is just a string of text. By *parsing* it, ColdFusion lets you then access the information in the XML file much like a collection of structures and arrays.

Reading an XML File Using `XmlParse()`

To parse an XML document stored somewhere on disk, use `XmlParse()`, as demonstrated in Listing 45.2. As of ColdFusion 7, you can name a file or even a web page URL from which to obtain the XML content to be parsed. Let's look at each of these.

Listing 45.2 `ParseXML.cfm`—Parsing an XML Document on Disk

```
<cfset xmlObject = XmlParse("company.xml","yes")>

<cfdump var="#xmlObject#">
```

Figure 45.1 shows the output of `xmlObject` in the browser. You could have replaced the filename with a URL, to retrieve an XML string from a web page. An example is `http://www.adobe.com/crossdomain.xml`. The `"Yes"` is the value of the `caseSensitive` argument to `XmlParse()`, which makes the created object case-sensitive.

That was pretty painless, wasn't it?

By parsing the file's contents with the `XmlParse()` function, we converted the XML document (a string of XML) into a variable called `xmlObject`, which is an object that ColdFusion recognizes as a set of employees, rather than just as a string of text.

CAUTION

The two terms XML document and XML object are not interchangeable, although some people use them as such. In this book, XML document refers to a string of XML-encoded text, and XML object refers to a complex variable that ColdFusion can manipulate directly.

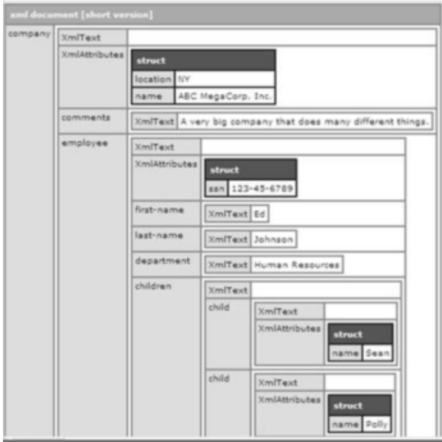


Figure 45.1
A ColdFusion XML object as displayed by `<cfdump>`

Creating XML Documents Using CFXML

Sometimes, rather than reading an XML document, you will want to create one from within CFML. A good example of doing this is to querying a database and converting its results into XML. That XML could be written it to the browser or returned as a CFC method result, as will be shown later in this section, or it can simply be saves to a file, as shown in the next example:

Listing 45.3 `XMLFromADatabase.cfm`—Creating an XML Document from Database Content

```
<cfquery name="employeesQuery" datasource="Chapter45_DSN">
SELECT SSN, FirstName, LastName, Department
FROM Employee
</cfquery>

<cfxml variable="xmlObject" casesensitive="yes">
<company>
  <cfoutput query="employeesQuery">
  <employee ssn="#XmlFormat(SSN)#">
```

Listing 45.3 (continued)

```
        <first-name>#XmlFormat(FirstName)#</first-name>
        <last-name>#XmlFormat(LastName)#</last-name>
        <department>#XmlFormat(Department)#</department>
      </employee>
      </cfoutput>
    </company>
    </cfxml>

    <cffile action="WRITE"
        file="#ExpandPath('EmployeeXML.xml')#"
        output="#ToString(xmlObject)#">
```

NOTE

To set up `Chapter45_DSN`, see the `Readme.txt` inside the Chapter45 folder in the downloadable code for this book.

Listing 45.3 queries the database to find employee information, then creates an XML object and writes the XML object to disk. Using `ToString()` converts an XML object back into an XML document (text string), and the XML document is then written to disk.

Running Listing 45.3 produces a file named `EmployeeXML.xml`, as shown in Listing 45.4.

Listing 45.4 `EmployeeXML.xml`—XML Generated From A Database

```
<?xml version="1.0" encoding="UTF-8"?>
<company>

  <employee ssn="123-45-6789">
    <first-name>Ed</first-name>
    <last-name>Johnson</last-name>
    <department>Human Resources</department>
  </employee>

  <employee ssn="541-29-8376">
    <first-name>Maria</first-name>
    <last-name>Smith</last-name>
    <department>Accounting</department>
  </employee>

  <employee ssn="568-73-1924">
    <first-name>Eric</first-name>
    <last-name>Masters</last-name>
    <department>Accounting</department>
  </employee>

</company>
```

`<cfxml>` creates the same kind of XML object that is created by `XmlParse()`, with the difference being that `XmlParse()` creates an XML object based on an XML document.

TIP

You may have noticed the `caseSensitive` attribute in the `<cfxml>` call earlier. By default, ColdFusion is not case-sensitive in its XML handling. However, because other platforms may be case-sensitive, you should make sure your XML documents are case-sensitive as well.

Notice the use of `XmlFormat()`. Anytime you place data into an XML document, you should always use `XmlFormat()`, which replaces special or high-ASCII characters with their entity reference equivalents (see the section "Entity References" later in this chapter for more information).

You may ask why we should bother to use `<CFXML>`, since you end up outputting the XML as a string, and it's true that you could just as well have created the XML and written it to a file without using `<CFXML>`. The advantage is that `<CFXML>`will ensure that the XML is well-formed. If it's not, you'll receive an error during processing of the `<CFXML>` tag.

You could also have changed the code to write the XML to a browser (to be retrieved by anyone requesting the page in a browser), by replacing the `<CFFILE>` in listing 45.3 with `<CFOUTPUT>`, or more particularly:

```
<cfsetting showdebugoutput="No">

<cfcontent type="text/xml">

<cfoutput>#ToString(xmlObject)#</cfoutput>
```

Note the use of `<CFCONTENT>` to tell the browser to expect a plain text xml file (rather than the default of a text/html file), and the use of `<cfsetting>` to turn off CF's debugging (which is also HTML). An example is provided in the online files as `XMLFromADatabaseToBrowser.cfm`.

You could also create and return XML this way within a CFC method or user-defined function using `<CFFUNCTION>`, by specifying its `ReturnType="xml"` option and returning the XML. This could either be used to return data to another CFML template invoking the CFC, or an application (perhaps on a remote server) calling the CFC as a web service.

If you return the raw `xmlObject`, it can only be read by another ColdFusion server calling the web service (such as in returning a query result set). If you use the `ToString()` as above, it will return just XML as text. See the example provided on disk as `XMLFromADatabaseInCFC.cfc` and `invoke_xml_method.cfm`.

Creating XML Documents Using `XmlNew()`

While `<cfxml>` is a way to quickly create an XML document within CFML, there is another way—one that is often more flexible because you can take very granular control over the content of the document. Listing 45.5 shows the `XmlNew()` and `XmlElemNew()` functions in action.

Listing 45.5 `XmlNew.cfm`—Using `XmlNew()` and `XmlElemNew()` to Dynamically Create an XML Object

```
<cfscript>
  xmlObject = XmlNew("Yes");
  xmlObject["company"] = XmlElemNew(xmlObject, "company");

  employeeNode1 = XmlElemNew(xmlObject, "employee");
  employeeNode1.xmlAttributes["ssn"] = "123-45-6789";
  employeeNode1["first-name"] = XmlElemNew(xmlObject, "first-name");
  employeeNode1["first-name"].xmlText = "Ed";
  employeeNode1["last-name"] = XmlElemNew(xmlObject, "last-name");
  employeeNode1["last-name"].xmlText = "Johnson";
```

Listing 45.5 (continued)

```
  employeeNode1["department"] = XmlElemNew(xmlObject, "department");
  employeeNode1["department"].xmlText = "Human Resources";

  employeeNode2 = XmlElemNew(xmlObject, "employee");
  employeeNode2.xmlAttributes["ssn"] = "541-29-8376";
  employeeNode2["first-name"] = XmlElemNew(xmlObject, "first-name");
  employeeNode2["first-name"].xmlText = "Maria";
  employeeNode2["last-name"] = XmlElemNew(xmlObject, "last-name");
  employeeNode2["last-name"].xmlText = "Smith";
  employeeNode2["department"] = XmlElemNew(xmlObject, "department");
  employeeNode2["department"].xmlText = "Accounting";

  employeeNode3 = XmlElemNew(xmlObject, "employee");
  employeeNode3.xmlAttributes["ssn"] = "568-73-1924";
  employeeNode3["first-name"] = XmlElemNew(xmlObject, "first-name");
  employeeNode3["first-name"].xmlText = "Eric";
  employeeNode3["last-name"] = XmlElemNew(xmlObject, "last-name");
  employeeNode3["last-name"].xmlText = "Masters";
  employeeNode3["department"] = XmlElemNew(xmlObject, "department");
  employeeNode3["department"].xmlText = "Accounting";

  ArrayAppend(xmlObject["company"].xmlChildren, employeeNode1);
  ArrayAppend(xmlObject["company"].xmlChildren, employeeNode2);
  ArrayAppend(xmlObject["company"].xmlChildren, employeeNode3);
</cfscript>

<cfdump var="#xmlObject#">
```

The XML object created by Listing 45.5 is exactly the same as the one created by the `<CFXML>` tag in Listing 45.3.

When dynamically creating XML documents as we've done in this example, we first use `XmlNew()` to create a new XML object (the `"Yes"` is the value of the `caseSensitive` argument to `XmlNew()`, which makes the XML object case-sensitive):

```
xmlObject = XmlNew("Yes");
```

Then we create elements inside this object using `XmlElemNew()`:

```
xmlObject["company"] = XmlElemNew(xmlObject, "company");
employeeNode1 = XmlElemNew(xmlObject, "employee");
employeeNode1["first-name"] = XmlElemNew(xmlObject, "first-name");
```

We then use `ArrayAppend()` to place the employee nodes into the company's child node array:

```
ArrayAppend(xmlObject["company"].xmlChildren, employeeNode1);
```

This method of assembling nodes and then using array operations to connect all the nodes together is a common one. Its chief benefit is making code more readable and modular. Otherwise, we'd be using code like this to set the first name:

```
xmlObject["company"]["employee"][1]["first-name"] = XmlElemNew(xmlObject,
"first-name");
xmlObject["company"]["employee"][1]["first-name"].xmlText = "Ed";
```

While the example above does not use `XMLFormat()`, we could and should, especially if we were creating the XML elements from variables rather than strings. The issue is that if the data being used to create the XML elements or content included any special characters, it could cause an error. This issue is discussed further in the section, "Using Special Characters in XML".

Accessing XML Elements and Attributes

Whether we created an XML object using `XMLParse()`, `<CFXML>`, or `XMLNew()`, we now have an XML object in memory. If we wanted to explore or manipulate it, how would we access its elements and their properties? ColdFusion makes it easy for us by exposing the elements and attributes as if they were arrays and structures. For example, to access the second employee element, I'd use syntax like this:

```
xmlObject["company"]["employee"][2]
```

Notice the bracket notation, just as if I were accessing a set of nested structures with an array at the end. What if I wanted to access the second employee's first name? I'd extend the syntax like this:

```
xmlObject["company"]["employee"][2]["first-name"].xmlText
```

Notice the `xmlText` property at the end of the reference. If I had referred instead to just the `first-name` element, that expression would have returned an XML element object reference rather than a string. The `xmlText` contains the text stored in the element object itself.

What about attributes? Let's say I wanted to access the SSN of the third employee. I'd use similar syntax, like this:

```
xmlObject["company"]["employee"][3].xmlAttributes["ssn"]
```

Because attributes are always strings, attributes do not require that you put `xmlText` at the end of the reference.

TIP

> When referencing element and attribute names, I always use bracket notation (`["company"]`) rather than dot notation (`.company`). I do this for two reasons. First, XML is case-sensitive, and dot notation often has problems with mixed-case documents. Second, you cannot use dot notation with element names that contain hyphens, so for consistency's sake, I use bracket notation everywhere.

The form of XML notation we've been using so far, in which the element names are used like the names of arrays, is known as *short form notation*:

```
xmlObject["company"]["employee"][2]
```

This is different from *long form notation*, which would reference the second employee node as follows:

```
xmlObject.xmlRoot.xmlChildren[2]
```

Instead of using `company`, we use `xmlRoot` (because it is the document's *root node*), and instead of using `employee[2]`, we use `xmlChildren[2]` (because it is the second child of `xmlRoot`). There is no array reference after `xmlRoot` because, like all XML documents, this one has only one root element.

The advantage to using long form notation is that it doesn't matter what the name of each individual node is, because nodes are referenced using the `xmlChildren` array of the parent element. The disadvantage is that now you have to do more work to find out what node is being referenced.

The two different notation methods exist for an important reason. Long form notation is like looking at text-based driving directions from a site like MapQuest: The instructions are very detailed but they don't give any perspective on where you are at any given point. In contrast, short form notation is like looking at a map: You can easily see where you are at any given time, but it's not always easy to find out how to get there.

You're not limited to using only one method at a time. You can combine both long form and short form, as I'm doing here:

```
xmlObject["company"].xmlChildren[2]["first-name"].xmlText
```

Instead of directly referencing the `employee` array, I'm using `xmlChildren`.

So far, the only element properties we've seen are `xmlText`, `xmlAttributes`, and `xmlChildren`. These properties are present for every node in a document, and they are, for the most part, the only properties you'll use in most of your code.

Table 45.1 shows a listing of the properties of every element within an XML object.

Table 45.1 XMLNode Properties

PROPERTY NAME	TYPE	DESCRIPTION
`xmlName`	String	The name of the element. If the element has a namespace prefix, `xmlName` contains the fully qualified name. (See the later section "XML Namespaces" for more information.)
`xmlNsPrefix`	String	The element's namespace prefix. If the element does not have a namespace, `xmlNsPrefix` is blank.
`xmlNsURI`	String	The element's namespace URI. If the element does not have a namespace, `xmlNsURI` is blank.
`xmlText`	String	The text between the element's opening and closing tags (not counting text inside of any child elements).
`xmlCData`	String	Any `CDATA` content between the element's opening and closing tags (not counting `CDATA` sections inside of any child elements). See the section on "`CDATA` Sections" later in this chapter for more details.
`xmlComment`	String	A single value containing the text of all the comments inside the element (not counting comments inside of any child elements).
`xmlAttributes`	Structure	A structure containing a key for each of the element's attributes.
`xmlChildren`	Array of XML elements	An array containing an entry for each of the element's child elements.

Table 45.1 (continued)

PROPERTY NAME	TYPE	DESCRIPTION
`xmlParent`	XML element	A reference to the element's parent element.
`xmlNodes`	Array of XML nodes	An array containing an entry for each child *node* of the element (not including attribute nodes). This is very rarely used.

Using Array and Structure Functions

Table 45.1 shows that `xmlChildren` is an array, and `xmlAttributes` is a structure. This means we can use ColdFusion's native structure and array functions to modify XML objects. If we wanted to delete the third employee node from the document, for instance, we'd use `ArrayDeleteAt()`:

```
ArrayDeleteAt(xmlObject["company"].xmlChildren, 3)
```

Similarly, to find out how many attributes a particular element has, we'd use `StructCount()`:

```
StructCount(xmlObject["company"]["employee"][1].xmlAttributes)
```

The *ColdFusion Developer's Guide*, which is part of the ColdFusion 8 documentation set, gives a listing of which array and structure functions can be used with XML objects and when.

Using Special Characters in XML

The previous examples of creating XML have been simple enough, but there are some special considerations that you should be aware of when creating XML. We showed the use of `XMLFormat()` earlier in Listing 45.3. Let's look at the reason why.

One of the frustrations developers encounter is trying to use special characters inside an otherwise well-formed XML document. For example, let's say we want to embed some text with an ampersand inside of a document:

```
<company name="Baker & Associates" location="CA">
  ...
</company>
```

XML parsers would not be able to process that document because the ampersand is a special character. This section shows you two different ways to deal with this problem: entity references and `CDATA` sections.

Entity References

If you've ever used a sequence such as ` ` or `&` within HTML or another markup language, then you've used *entity references*. Entity references are a way to encode special characters so that

they can be used within an XML document without affecting parsers' abilities to read them. Here's an example:

```
<company name="Baker &amp; Associates" location="CA">
  ...
</company>
```

An entity reference begins with an ampersand and ends with a semicolon. Following are some of the entity references you may have seen in HTML:

```
  (nonbreaking space)
&amp; (ampersand)
&iexcl; (inverted exclamation mark)
&Uuml; (capital U with umlaut)
```

Whenever a browser sees one of these entity references, the browser knows to display the corresponding character instead.

There are approximately 250 references defined in HTML. In XML, however, there are only five:

```
&amp; (ampersand)
&lt; (less-than sign)
&gt; (greater-than sign)
&quot; (double-quote)
&apos; (single-quote)
```

Any special characters outside of this set (specifically, high-ASCII characters above ASCII 127) must be encoded using a *character reference*, which looks like one of these two examples:

```
 
 
```

Both of these examples represent a nonbreaking space, ASCII 160. The `&#` introduces a decimal number, and `&#x` introduces a hexadecimal (base 16) number.

Fortunately, you may not need to remember all these rules, if you remember instead to use `XmlFormat()` whenever you put data into an XML document, as shown in Listing 45.3. The `XmlFormat()` automatically handles special characters by replacing them with their appropriate entity and character references before putting them into the document.

CDATA Sections

`XmlFormat()` is well suited for special characters that may occur in isolated areas of your XML document, but what if you have a section of your document that represents HTML markup with lots of special characters, or tags that don't parse correctly because they're not well formed?

In situations like these, `XmlFormat()` can go from being a blessing to being a curse—as a result of all the escaping that's done to the stored text. A good alternative is to use a CDATA section as shown in Listing 45.6.

Listing 45.6 `Namespaces.xml`—A Portion of Listing 45.1, Using `CDATA` Rather Than Regular Text

```
<company name="ABC MegaCorp, Inc." location="NY">
  <comments>
    <![CDATA[
      <P>A very large company with 4 divisions:
      <UL>
        <LI>Financial Services
        <LI>Baby Food
        <LI>Large Vehicles
        <LI>Fashion Consultation
      </UL>
    ]]>
  </comments>
  <employee ssn="123-45-6789">
    ...
  </employee>
  <employee ssn="541-29-8376">
    ...
  </employee>
</company>
```

The content inside the `CDATA` section is unparsed, so it can contain any characters—even the special characters that are normally unusable in XML content. The only restriction is that you cannot use the sequence `]]>` except to end the `CDATA` block.

To use `CDATA` within ColdFusion, use a node's `xmlCData` property rather than its `xmlText` property. The content you place into `xmlCData` is automatically placed into a `CDATA` section within the document, whereas content inside of `xmlText` is encoded and not put inside of a `CDATA` section:

```
<cfset myXmlNode.xmlCData = "<P>This is content I do not want to escape">
```

`CDATA` sections are also ideal for long passages of text with many special characters, because escaped characters can take up anywhere from three to eight times as much space as their unescaped counterparts. The disadvantage to using a `CDATA` section is slightly more complex code.

From this point forward we will not use any more `CDATA` sections in order to keep the focus on new topics.

XML Namespaces

Whether reading or creating XML, inexperienced programmers often overlook XML's ability to partition a single document into multiple sections or to represent multiple concepts in a single document by using *namespaces*. These collections of tag and attribute names are bound together using prefixes, and kept separate from tags and attributes in other namespaces.

In Listing 45.7, the single company document in Listing 45.1 is extended by adding starting salary information and putting directory information in its own namespace.

Listing 45.7 `ExtendedCompany.xml`—The `<company>` Entry with Starting Salary Information Added

```
<company
  xmlns:directory="http://www.mycompany.com/directory"
  xmlns:salary="http://www.mycompany.com/salary">
  <directory:employee ssn="123-45-6789">
    <directory:first-name>Ed</directory:first-name>
    <directory:last-name>Johnson</directory:last-name>
    <directory:department>Human Resources</directory:department>
  </directory:employee>
  <directory:employee ssn="541-29-8376">
    <directory:first-name>Maria</directory:first-name>
    <directory:last-name>Smith</directory:last-name>
    <directory:department>Accounting</directory:department>
  </directory:employee>
  <directory:employee ssn="568-73-1924">
    <directory:first-name>Eric</directory:first-name>
    <directory:last-name>Masters</directory:last-name>
    <directory:department>Accounting</directory:department>
  </directory:employee>
  <salary:starting-salaries>
    <salary:low>32000</salary:low>
    <salary:avg>56000</salary:avg>
    <salary:high>78000</salary:high>
  </salary:starting-salaries>
</company>
```

Namespaces are also important to XML schemas, as you will see later. In essence, namespaces separate parts of an XML document that serve distinct functionality.

In Listing 45.7, we declare two namespaces: one for elements concerning a company's directory entry, and another for the company's starting salary information. As a result, while I'm storing only one document, that document can store two different kinds of information together, with two completely separate schemas.

Another benefit of namespaces has to do with portability. Listing 45.8 shows another XML document with the same information as in Listing 45.7, but using different namespace prefixes.

Listing 45.8 `ExtendedCompany2.xml`—Another XML Document with Different Namespace Prefixes

```
<company
  xmlns:dir="http://www.mycompany.com/directory"
  xmlns:sal="http://www.mycompany.com/salary">
  <dir:employee ssn="123-45-6789">
    <dir:first-name>Ed</dir:first-name>
    <dir:last-name>Johnson</dir:last-name>
    <dir:department>Human Resources</dir:department>
  </dir:employee>
  <dir:employee ssn="541-29-8376">
    <dir:first-name>Maria</dir:first-name>
    <dir:last-name>Smith</dir:last-name>
    <dir:department>Accounting</dir:department>
  </dir:employee>
  <dir:employee ssn="568-73-1924">
    <dir:first-name>Eric</dir:first-name>
    <dir:last-name>Masters</dir:last-name>
```

Listing 45.8 (continued)

```
    <dir:department>Accounting</dir:department>
  </dir:employee>
  <sal:starting-salaries>
    <sal:low>32000</sal:low>
    <sal:avg>56000</sal:avg>
    <sal:high>78000</sal:high>
  </sal:starting-salaries>
</company>
```

Listings 45.7 and 45.8 are the exact same document, even though the namespace prefixes are different. What makes the namespaces unique is not the prefix before the name, but rather the URI (Uniform Resource Identifier) to which you bind the prefix. (In Listings 45.7 and 45.8, the URIs were `http://www.mycompany.com/directory` and `http://www.mycompany.com/salary`.) When two different documents bind two distinct prefixes to the same URI, then those two prefixes represent the same namespace.

NOTE

The difference between a URI and a URL is very small. A URL (Uniform Resource Locator) is always a Web address. A URI is a more general term for any unique identifier used to identify something. A URL is a type of URI.

The URIs used in the `xmlns` attributes in Listings 45.7 and 45.8 do not have to point to anything in particular. Parsers will not retrieve anything from the URI specified at that address. The namespace URI is merely a unique identifier for that namespace that keeps one namespace separate from another. That said, it's usually a good idea to use the namespace URI to point to one of two things:

- *Your company's home page.* In cases where multiple companies use XML to transfer and store information in a single document, the namespace URI can be a way to self-document the relationship of nodes to companies.
- *A Web address at your company that gives information about the schema.* This is a way of linking your XML file directly to its documentation. Although parsers will not use the document at this address, someone who reads the file and wants to know what's going on will be able to visit the URL and find out. (The W3C does this.)

The Default Namespace

Most developers are surprised to learn that even if no namespaces are declared in their document, they are still using a namespace. Even Listing 45.1 had a namespace. The only difference between the namespace in Listing 45.1 and the ones in Listing 45.7 are that the namespaces in Listing 45.7 are explicitly declared.

A namespace without a prefix is called a *default namespace*. Any tags that are not part of an explicitly declared namespace are part of this default namespace. To bind the default namespace to a URI, use the following syntax:

```
<element xmlns="http://www.mycompany.com" />
```

When to Use Namespaces

Namespaces are not always useful, but here are some situations when you *should* consider using a namespace:

- *When you are writing a solution to communicate between companies (or even divisions within a company).* Separating namespaces lets each company or division define its own functionality and own rules for interaction.
- *When you are using XML to control the behavior of a modular application.* Separating functionality into discrete namespaces lets you change parts of the application without affecting others, especially when a schema change is involved. Usually you will want to use one namespace per module.

This list is not exhaustive; it is only meant to give you an idea of some of the situations in which namespaces are useful.

Because the rest of the chapter focuses on schemas, we will be making heavy use of namespaces. In the next chapter, you will also get a real-world idea of how namespaces make XSLT possible.

Validating XML

Any XML parser will tell you whether your XML document is well formed (that is, whether all tags are closed and nothing is improperly nested). But just because a document is well formed does not mean it adheres to the structure your business logic expects.

To know that your XML document is not only well formed, but also valid in accordance with your business rules, you must *validate* the document. Validating an XML document ensures that tags are nested in their proper hierarchy (so that something that's supposed to be a child is not actually a parent); that there are no extraneous attributes; and that all required elements and attributes are present.

There are two major methods for validating XML: the older DTD (Document Type Definition) standard, and the newer and more flexible XML Schema standard. This chapter does not go into much depth on DTDs; XML Schema is a more capable standard that is widely accepted.

NOTE

Some other validation standards have come and gone. Chief among them are XDR (XML Data-Reduced), which was Microsoft's attempt at a proprietary schema language, and Relax NG, yet another attempt at defining an XML syntax for creating schemas. However, XML Schema will most likely be the XML validation standard for some time, thanks to its flexibility, wide-ranging support, and status as a W3C recommendation.

DTDs

DTDs (Document Type Definitions) are a holdover from the days of SGML. The DTD describes the elements and attributes available in a document, how they can be nested, and which ones are required and which are optional. But its lack of support for namespaces and inheritance, as well as its somewhat confusing syntax, make the DTD too limited for most programmers' needs.

With this in mind, the remainder of this chapter is devoted to examining and understanding the XML Schema standard, which does everything DTDs can do and more.

XML Schemas

An *XML schema* is the definition of the required structure of an XML document. By defining a schema, we can then use it to validate that a given XML document is not only well-formed but follows the format defined in the schema. For demonstration purposes, Listing 45.9 shows a simplified version of the original single company XML shown in Listing 45.1.

Listing 45.9 `SimpleCompany.xml`—A Simpler XML Document

```
<?xml version="1.0" ?>
<company>
  <employee ssn="123-45-6789">
    <first-name>Ed</first-name>
    <last-name>Johnson</last-name>
    <department>Human Resources</department>
  </employee>
  <employee ssn="541-29-8376">
    <first-name>Maria</first-name>
    <last-name>Smith</last-name>
    <department>Accounting</department>
  </employee>
  <employee ssn="568-73-1924">
    <first-name>Eric</first-name>
    <last-name>Masters</last-name>
    <department>Accounting</department>
  </employee>
</company>
```

Listing 45.10 shows a schema definition for that document. The remainder of this section will explain its components, and the next section will show how to use that schema file to validate a given XML document.

Listing 45.10 `CompanyDirectory.xsd`—An XML Schema Document

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="company" type="CompanyType" />

  <xsd:complexType name="CompanyType">
    <xsd:sequence>
      <xsd:element name="employee" type="EmployeeType" minOccurs="0"
maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="EmployeeType">
    <xsd:sequence>
      <xsd:element name="first-name" type="xsd:string" />
      <xsd:element name="last-name" type="xsd:string" />
      <xsd:element name="department" type="xsd:string" />
    </xsd:sequence>
    <xsd:attribute name="ssn" type="SSNType" />
```

Listing 45.10 (continued)

```
  </xsd:complexType>

  <xsd:simpleType name="SSNType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-\d{2}-\d{4}" />
      <xsd:length value="11" />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

This schema defines a top-level `company` element and its lower-level `employee` elements. Let's examine its parts.

Walkthrough of a Basic Schema

The first line is the schema header. Notice that XML Schema's tags are in a separate namespace, which by convention is `xsd`. The separate namespace is always necessary to prevent the collision of XML Schema tags and your own tags.

An XML schema always has `xsd:schema` as its root element. (Although the prefix may be different, the element name must always be `schema`.)

The first thing inside the schema is typically the global elements defined for the schema. (They do not have to be first, but it's usually the most logical place for them.) In this schema, there is only one global element:

```
<xsd:element name="company" type="CompanyType" />
```

A *global element* is defined as a property of the schema, rather than as a property of a type. Usually the only global element will be the root element for the document (in this case, `company`).

Elements in an XML schema are defined by using the `xsd:element` tag. In Listing 45.10, the following line of code declares the XML document's root `company` element:

```
<xsd:element name="company" type="CompanyType" />
```

When you declare an element in an XML schema, you must tell the validator the name of the element and the type of the element's content. Here, the element's name is `company`, and its type is `CompanyType`, which is a custom type defined later in the schema.

Next in the schema is a custom type definition, introduced by the element `xsd:complexType`. A *complex type* is any element type that contains element or attribute definitions of its own. In this definition, the complex type contains only a single element, represented by an `xsd:element` nested inside of an `xsd:sequence`.

The `xsd:sequence` denotes that a set of elements must occur in the specified order. The `xsd:sequence` is required here even though there is only one element; this is a quirk of the XML Schema language.

Now let's take a closer look at the `xsd:element` tag used to define the employee element:

```
<xsd:element name="employee" type="EmployeeType" minOccurs="0" maxOccurs="unbounded"
/>
```

The company element did not define minOccurs and maxOccurs attributes because it was the root element and as such had to have one and only one occurrence. However, it's not necessary for any employee elements at all to be defined, and there is no upper limit on the number of employee elements that can be nested within a company.

Table 45.2 shows the possible values for minOccurs and maxOccurs and their effects.

Table 45.2 minOccurs and maxOccurs Values

PROPERTY	VALUE	EFFECT
minOccurs	0	This element does not have to be present.
minOccurs	1...*n*	This element must be present at least this many times.
maxOccurs	0	This element cannot be present.
maxOccurs	1...*n*	This element can be present, at most, this many times.
maxOccurs	unbounded	This element can be present any number of times.

minOccurs and maxOccurs both default to 1, meaning that if neither is specified, the element is required and may only be present once.

The type of the employee element is another custom type, defined by the next xsd:complexType in the schema. EmployeeType contains three elements and one attribute:

```
<xsd:complexType name="EmployeeType">
  <xsd:sequence>
    <xsd:element name="first-name" type="xsd:string" />
    <xsd:element name="last-name" type="xsd:string" />
    <xsd:element name="department" type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="ssn" type="SSNType" />
</xsd:complexType>
```

The three xsd:element tags are inside of an xsd:sequence tag. This means they must always occur in the same order that they appear in the schema. If we wanted it such that elements could occur in any order, we would instead use xsd:choice:

```
<xsd:choice minOccurs="3" maxOccurs="3">
  <xsd:element name="first-name" type="xsd:string" />
  <xsd:element name="last-name" type="xsd:string" />
  <xsd:element name="department" type="xsd:string" />
</xsd:choice>
```

This means that the developer can choose three of these elements in any order and in any combination. Be careful using the xsd:choice syntax, because the developer could use three first-name elements if desired. Using xsd:sequence is typically your best bet.

Also notice that the type attributes of the elements do not refer to a custom type. The first-name, last-name, and department elements all have a type of xsd:string, which is a built-in XML Schema type.

A full list of XML Schema types is beyond the scope of this book, but Table 45.3 lists the most common ones. There are many more built-in XML schema types, which are described in the XML Schema Primer at the W3C website. See "More XML Resources" at the end of this chapter for more information.

Table 45.3 Built-in XML Schema Types

TYPE NAME	DESCRIPTION
`xsd:string`	A string of characters with no special formatting.
`xsd:integer`	A generic integer value.
`xsd:positiveInteger`	A generic integer value greater than zero.
`xsd:negativeInteger`	A generic integer value less than zero.
`xsd:nonPositiveInteger`	A generic integer value that is zero or less than zero.
`xsd:nonNegativeInteger`	A generic integer value that is zero or greater than zero.
`xsd:long`	A 64-bit integer value.
`xsd:unsignedLong`	An unsigned 64-bit integer value.
`xsd:int`	A 32-bit integer value.
`xsd:unsignedInt`	An unsigned 32-bit integer value.
`xsd:decimal`	An exact decimal number value.
`xsd:float`	A 32-bit floating-point single-precision value.
`xsd:double`	A 64-bit floating-point double-precision value.
`xsd:dateTime`	A date/time value in the format 2005-01-18T13:20:00.000-05:00 (representing January 18, 2005, at 1:20 P.M., 5 hours behind UTC).
`xsd:boolean`	True, false, 1, or 0 (no other value will be accepted as valid, according to the schema).

After the three string elements in `EmployeeType`, there is an `xsd:attribute` tag describing the `ssn` attribute. Notice that the `xsd:attribute` is outside the `xsd:sequence` tag (because attributes are never ordered), and that its type attribute refers to `SSNType`, which is defined by an `xsd:simpleType`:

```
<xsd:simpleType name="SSNType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-\d{2}-\d{4}" />
    <xsd:length value="11" />
  </xsd:restriction>
</xsd:simpleType>
```

In this case, `SSNType` is a special kind of `xsd:string` that restricts its value to a valid SSN using a regular expression. (See Chapter 43, "Using Regular Expressions," online, for the full discussion of regular expression syntax.)

NOTE

`xsd:complexType` defines an element type possibly containing other elements and attributes, whereas `xsd:simpleType` defines a simple string type based on an existing type. Element values can be either simple or complex types because elements can contain other elements, but attribute values must always be simple types because simple types can always be expressed as string values.

The `xsd:pattern` and `xsd:length` tags are *facets* of the simple type's restriction. (*Facet* is an XML Schema term meaning "a kind of restriction.") Other facets that can be used inside an `xsd:restriction` tag are summarized in Table 45.4.

Table 45.4 Facets Used Inside an `xsd:restriction` Element

XSD ELEMENT	DESCRIPTION	EXAMPLE
`xsd:length`	Specifies that a string value must be a specific length.	`<xsd:length value="8" />`
`xsd:minLength`	Specifies that a string value must be at least a certain length.	`<xsd:minLength value="3" />`
`xsd:maxLength`	Specifies that a string value must be at most a certain length.	`<xsd:maxLength value="7" />`
`xsd:pattern`	Specifies that a value must adhere to a given regular expression pattern.	`<xsd:pattern value="[0-9]{5}(-[0-9]{4})?" />`
`xsd:enumeration`	Specifies that a value must be in a given set of values. Note that there is one `xsd:enumeration` tag for every value in the allowed set.	`<xsd:enumeration value="0" />` `<xsd:enumeration value="1" />`
`xsd:whiteSpace`	Specifies how extraneous whitespace is handled before other facets are validated. Can be `preserve` (to preserve all whitespace); `replace` (to replace all tabs and carriage returns with spaces); or `collapse` (to replace all tabs and carriage returns with spaces, replace all whitespace with a single space, and then trim the front and back).	`<xsd:whiteSpace value="collapse" />`
`xsd:minInclusive`	Specifies that a value must be greater than or equal to a given value.	`<xsd:minInclusive value="1" />`
`xsd:maxInclusive`	Specifies that a value must be less than or equal to a given value.	`<xsd:maxInclusive value="20" />`
`xsd:minExclusive`	Specifies that a value must be greater than a given value.	`<xsd:minExclusive value="0" />`
`xsd:maxExclusive`	Specifies that a value must be less than a given value.	`<xsd:maxExclusive value="21" />`
`xsd:totalDigits`	Specifies that a numeric value can have, at most, this many digits.	`<xsd:totalDigits value="12" />`
`xsd:fractionDigits`	Specifies that a numeric value can have, at most, this many digits after the decimal place.	`<xsd:fractionDigits value="2" />`

Note that you can combine multiple facets to have granular control over your data. For example, if you wanted to ensure that a value was between three and eight characters long, you could use the following `xsd:simpleType` declaration:

```
<xsd:simpleType name="SKUType">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="3" />
    <xsd:maxLength value="8" />
  </xsd:restriction>
</xsd:simpleType>
```

As you can see, even complicated schemas consist of simple building blocks. For more information on the XML Schema language and how to build schemas, see "More XML Resources" at the end of this chapter.

Validating XML in ColdFusion

Validating XML in ColdFusion is a simple process, assuming you've already written the appropriate schema or DTD. Both `XmlParse()` and the new function `XmlValidate()` provide the capability to target a schema and validate a document's contents.

In the case of `XMLParse()`, as of ColdFusion 7 it offers a new optional 3rd argument to name a validator (an XSD file). Going back to listing 45.2, the first line could be changed to use the validator in listing 45.10:

```
<cfset xmlObject = XmlParse('Company.xml', true,"CompanyDirectory.xsd")>
```

This would attempt to validate the XML in `company.xml` against the XSD file in Listing 45.10, but it would fail because the XSD is defined for the simpler XML file defined in Listing 45.9.

More important, though, using validation in `XMLParse()` has a drawback in that if and when there is an error, the tag fails and your ColdFusion page gets an error, unless you have handled it using `CFTRY/CFCATCH`. The advantage of `XMLValidate()` is discussed in the next section.

Validating by Using `XmlValidate()`

Using `XMLValidate()` offers greater control over error handling and also provides additional information in the result structure returned from the function. `XmlValidate()` takes two arguments, as shown in Listing 45.11.

Listing 45.11 `ValidateXML.cfm`—Using `XmlValidate()` to Validate a Document

```
<cfset xmlObject = XmlParse('Company.xml', true)>

<cfset errorStruct = XmlValidate(xmlObject, "CompanyDirectory.xsd")>

<cfdump var="#errorStruct#">
```

The first argument is a reference to the XML document that needs validating, and the second argument is a reference to the schema document against which the XML document must be validated. Note that each argument can take a number of inputs:

- Both arguments can take a string representation of the XML markup.
- Both arguments can take a filename.
- Both arguments can take a URL.
- The first argument (the XML document) can also take a parsed XML object.

`XmlValidate()` returns a structure describing whether validation succeeded, as well as any errors that may have occurred during validation. Most often you will use `errorStruct.status`, which is `YES` or `NO`, depending on whether validation was successful, to determine whether or not to continue processing the document. (The other keys in the structure are usually only helpful during debugging.)

Note that by passing in the schema object to `XmlValidate()`, you can dynamically control the schema against which a given document is validated, as shown in the next section.

Embedding Schema Information within a Document

You will most often pass a schema to the `XmlValidate()` function in order to control how a document is validated, but it is sometimes necessary to embed a reference to the schema within the document itself. This is done by using syntax like this:

```
<company xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="CompanyDirectory.xsd">
  <employee ssn="123-45-6789">
    <first-name>Ed</first-name>
    <last-name>Johnson</last-name>
    <department>Human Resources</department>
  </employee>
  <employee ssn="541-29-8376">
    <first-name>Maria</first-name>
    <last-name>Smith</last-name>
    <department>Accounting</department>
  </employee>
  <employee ssn="568-73-1924">
    <first-name>Eric</first-name>
    <last-name>Masters</last-name>
    <department>Accounting</department>
  </employee>
</company>
```

This syntax example points the parser to the schema at Listing 45.10.

Note that this method is far from foolproof, because when the calling program validates this document using the provided schema, there is no guarantee that the schema is actually the correct one. The person who created the XML file could very well point the URL anywhere and still have a valid document at that location.

To make use of a schema location provided within the source document, you would call `XmlValidate()` without a second argument, like this:

```
XmlValidate(xmlObject);
```

This tells ColdFusion to use the embedded schema location.

You can also use this form of embedded Schema with `XMLParse()`, by leaving off the 3rd argument (don't name a validator). But note that the URL offered within the XML file's `xsi:noNamespaceSchemaLocation` (as shown above) must be a complete URL. It cannot be a filename only, as with `XmlValidate()`.

More XML Resources

This chapter has touched on many of the most important topics in the world of XML. But there is a wealth of additional information available elsewhere. Here is a list of Web sites that should prove valuable to you in learning more about XML.

- `http://www.w3c.org` is the Web site for the World Wide Web Consortium and has information about XML standards. The articles here tend to be very dry and wordy, but they are full of excellent information.
- `http://www.zvon.org` is an indispensable site for learning about the world of XML and its various satellite technologies. There is a substantial amount of information here, and the site has excellent tutorials for people just starting out.
- `http://msdn2.microsoft.com/en-us/library/ms256177.aspx` is the MSDN main reference page for XML standards. It has an excellent XML Schema reference, as well as information on namespaces and DTDs. (MSDN is the website of the Microsoft Developer Network.)

CHAPTER 46

Manipulating XML with XSLT and XPath

IN THIS CHAPTER

Understanding XPath E205

Transforming XML into Content by Using XSLT E210

More XPath and XSLT Resources E230

As you saw in the previous chapter, XML is a way to structure data hierarchically using elements and attributes, much like a relational database structures data using tables and columns. Ultimately, XML is just a different way of structuring and storing data.

This also means that you will eventually want to start pulling data out of the XML structure and formatting it for use within your application. This could mean searching an XML document to find a node or set of nodes (using XPath), or it could mean transforming the XML into another format using a style sheet (using XSLT). Both of these technologies are discussed within this chapter.

Understanding XPath

XPath is a simple query language for XML that mimics standard directory access syntax. For example, if I had a company directory and wanted to access the third company node, I could use syntax like:

```
/companies/company[3]
```

XPath syntax has the advantage of being both more understandable and more portable than a platform-specific notation, such as the ColdFusion variable syntax described last chapter. In addition, because XPath was created for searching documents, it tends to be more flexible than a platform-specific method.

XPath is to XML as URLs are to the Internet. Where a URL is a patternistic way to search for resources online, XPath is a language used to search for nodes within an XML document.

Example: A CD Collection

Listing 46.1 is an XML document describing a few of the CDs in my collection. This CD collection will be the input XML document for all of the examples in this chapter.

Listing 46.1 `CDCollection.xml`—The XML Document Used for This Chapter

```
<?xml version="1.0" ?>
<cdcollection>
  <artist id="1" name="Air">
    <genre>Electronic</genre>
    <cd id="1" name="10,000 Hz Legend" rating="3">
      <recommend cd="2" />
    </cd>
    <cd id="2" name="Talkie Walkie" rating="4">
      <recommend cd="1" />
      <recommend cd="6" />
    </cd>
  </artist>
  <artist id="2" name="Kylie Minogue">
    <genre>Dance</genre>
    <cd id="3" name="Fever" rating="3">
      <recommend cd="4" />
    </cd>
    <cd id="4" name="Body Language" rating="4">
      <recommend cd="5" />
    </cd>
    <recommend artist="3" />
  </artist>
  <artist id="3" name="Dannii Minogue">
    <genre>Dance</genre>
    <genre>Electronic</genre>
    <cd id="5" name="Neon Nights" rating="5">
      <recommend cd="4" />
    </cd>
    <cd id="6" name="You Won't Forget About Me EP" rating="5">
      <recommend cd="5" />
    </cd>
    <recommend artist="2" />
  </artist>
  <artist id="4" name="Brooklyn Funk Essentials">
    <genre>Funk</genre>
    <genre>Dance</genre>
    <genre>Spoken Word</genre>
    <cd id="7" name="Cool &amp; Steady &amp; Easy" rating="5">
      <recommend cd="4" />
      <recommend cd="5" />
    </cd>
    <recommend artist="1" />
    <recommend artist="5" />
  </artist>
  <artist id="5" name="Felix Da Housecat">
    <genre>Electronica</genre>
    <genre>Dance</genre>
    <genre>Retro</genre>
    <cd id="8" name="A Bugged Out Mix" rating="3">
      <recommend cd="4" />
      <recommend cd="6" />
      <recommend cd="9" />
    </cd>
    <cd id="9" name="Kittenz and Thee Glitz" rating="5" />
    <cd id="10" name="Devin Dazzle &amp; The Neon Fever" rating="5">
```

Listing 46.1 (continued)

```
          <recommend cd="9" />
        </cd>
        <recommend artist="2" />
        <recommend artist="4" />
      </artist>
    </cdcollection>
```

The collection is broken down by artist, each of whom can have one or more genres, one or more CDs, and one or more recommendations for other artists. Each CD can have one or more recommendations for other CDs. By the end of the chapter we will have turned this XML structure into an HTML listing that displays all this information in a user-friendly format.

XPath Syntax

The syntax of an XPath search expression is based on the same syntax as file paths in UNIX or Windows, with the addition of features that allow the developer to restrict the returned node set based on some criteria. For example, to find all `cd` nodes with a `rating` higher than 3, I would use syntax like:

```
/cdcollection/artist/cd[@rating > 3]
```

In essence, I am using typical directory hierarchy syntax to *select* nodes, then using typical array/structure syntax to *restrict* nodes. When I run the XPath search against my document, I will get back an array containing all the nodes that match the expression.

Selections Using `/` and `//`

Much like you *select* columns from a table in an SQL statement, you *select* nodes from an XML document in an XPath expression. For instance, in this syntax:

```
/cdcollection
```

We're selecting the `cdcollection` element directly underneath the root node of the document.

If we wanted to select children of `cdcollection`, we'd extend the previous selection with another one:

```
/cdcollection/artist
```

And so forth until we've drilled down to the level of the XML hierarchy for which we're looking.

It is possible to shorten certain XPath expressions. For instance, this expression:

```
/cdcollection/artist/cd
```

could also be written as:

```
//cd
```

The two expressions will return the same results; however, they do not mean the same thing. The first expression specifies that we want to retrieve the `cd` nodes at that *exact* position in the hierarchy, whereas the second specifies that we want to retrieve *all* `cd` nodes anywhere in the hierarchy. In effect, `//` means "search the entire document."

While `//` is certainly more convenient in some cases, its use can present some problems. First, because it is not doing any kind of restriction by document structure, the XPath engine must visit every node in the document to find all the possible matches. This makes searches using `//` much slower in most cases.

Second, consider this expression:

```
/cdcollection/artist/recommend
```

Shortening that expression to:

```
//recommend
```

Returns not only those `recommend` nodes underneath an `artist` node, but also the `recommend` nodes found under any `cd` nodes as well. In order to return the same data as the original expression, we'd have to use syntax like:

```
//artist/recommend
```

But again, using the `//` instead of giving a fully qualified path will make the search take longer, especially on large documents.

Restrictions Using `[]`

Now, let's say that we have an XPath expression like:

```
/cdcollection/artist/cd
```

That would select all the `cd` nodes at that position in the hierarchy (regardless of which artist element where they were contained). But what if I only wanted to select the `cd` nodes underneath the first `artist` node? I would use a *restriction* like:

```
/cdcollection/artist[1]/cd
```

much as I would if I were accessing an array element in ColdFusion. Let's walk through the expression as it stands right now.

1. XPath selects the `cdcollection` node underneath the root.
2. XPath then selects all of the `artist` nodes underneath the node that's been found so far.
3. Then XPath applies the restriction `[1]` to the currently selected set, meaning to take only the first `artist` node it finds.
4. Finally, XPath selects all the `cd` child nodes of the currently selected `artist` node.

There are other restriction formats. For instance, if I wanted to find only those CD nodes with a rating higher than 3, I would use syntax like:

```
/cdcollection/artist/cd[@rating > 3]
```

The `@` symbol is XPath shorthand for "attribute," so whenever you are referencing an attribute name in XPath, just remember to prefix it with `@`.

Restrictions can contain further selections, as in this expression that retrieves all artist nodes containing at least one CD with a rating higher than 3:

```
/cdcollection/artist[cd/@rating > 3]
```

Notice that the selection inside of the square brackets did not start with `/` or `//`. This means that the selection starts from the *context node*, meaning the node immediately outside the square brackets.

What if you were searching for artists in a particular genre? (Remember that genre is an element rather than an attribute like rating.) Restrictions by element value look similar to restrictions by attribute value:

```
/cdcollection/artist[genre = 'Electronica']
```

That expression would return all `artist` elements containing at least one `genre` element with a value of "Electronica."

It is also possible to combine expressions using Boolean operators. If I wanted to find all artist nodes in the Electronica genre who also have at least one CD with a rating higher than 3, I could use syntax like:

```
/cdcollection/artist[genre = 'Electronica' and cd/@rating > 3]
```

If I wanted to find artists who were either in the Electronica genre *or* had a CD rated higher than 3, I could use an or search like:

```
/cdcollection/artist[genre = 'Electronica' or cd/@rating > 3]
```

There is also a negation operator; if I wanted to find artists in the Electronica genre but did *not* have any CDs rated higher than 3, I could use syntax like:

```
/cdcollection/artist[genre = 'Electronica' and not(cd/@rating > 3)]
```

Using `XmlSearch()` to Retrieve an Array of Nodes

Given the CD collection shown in Listing 46.1, how would I find all of the artist recommendation nodes? I could use ColdFusion to loop through the artist nodes, and then find each artist's recommendation nodes; or I could use `XmlSearch()` to run an XPath search, as in Listing 46.2.

Listing 46.2 `FindArtistRecommendations.cfm`—Using XPath to Search the CD Collection

```
<cffile action="READ"
  file="#ExpandPath('CDCollection.xml')#"
  variable="xmlDocument">

<cfset xmlObject = XmlParse(xmlDocument)>
<cfset results = XmlSearch(xmlObject, "/cdcollection/artist/recommend")>

<cfdump var="#results#">
```

The ColdFusion function `XmlSearch()` takes the XPath string (in this case `/cdcollection/artist/recommend`) and returns an array of XML elements, as shown in Figure 46.1.

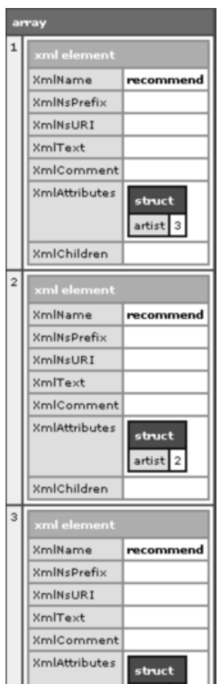


Figure 46.1
The result of an XPath search using `XmlSearch()`.

Each element in the array represents one of the elements found by the XPath search.

Your knowledge of XPath can also be used with the new ColdFusion 8 tag, `CFSPRYDATASET`, which through its optional `XPATH` attribute allows you to extract data when processing XML data. For more information, see the Adobe ColdFusion CFML Reference manual.

There's much more you can learn about XPath, as is discussed in the Resources section at the end of this chapter.

Transforming XML into Content by Using XSLT

CFML is an excellent tool for transforming content from a relational database into HTML, XML, or almost any form of content markup due to its native support for looping over a query set. However, this ease of looping and data handling does not extend to XML. ColdFusion can certainly loop over data in an XML object, and can search for data using `XmlSearch()`, but there is a much better solution for transforming XML into other forms of content.

XSLT (Extensible Stylesheet Language for Transformations) was created by the W3C in 1998 as a way to easily define a transformation from an XML document to some other content format. Using XSLT, I could transform a document like Listing 46.1 into the HTML table shown in Figure 46.2 using a simple stylesheet.

Figure 46.2
HTML table.

Artist/CD Name	Rating (for CDs)	Recommendations
Air Electronic		
10,000 Hz Legend	Like It	• Talkie Walkie
Talkie Walkie	Love It	• 10,000 Hz Legend • You Won't Forget About Me EP
Kylie Minogue Dance		• Dannii Minogue
Fever	Like It	• Body Language
Body Language	Love It	• Neon Nights
Dannii Minogue Dance, Electronic		• Kylie Minogue
Neon Nights	Favorite!	• Body Language
You Won't Forget About Me EP	Favorite!	• Neon Nights
Brooklyn Funk Essentials New Artist! Funk, Dance, Spoken Word		• Air • Felix Da Housecat
Cool & Steady & Easy	Favorite!	• Body Language • Neon Nights
Felix Da Housecat Electronica, Dance, Retro		• Kylie Minogue • Brooklyn Funk Essentials
A Bugged Out Mix	Like It	• Body Language • You Won't Forget About Me EP • Kittenz and Thee Glitz
Kittenz and Thee Glitz	Favorite!	
Devin Dazzle & The Neon Fever	Favorite!	• Kittenz and Thee Glitz

We will build the stylesheet that creates this table throughout the chapter. If you want to take a look at the stylesheet that created Figure 46.2, look at Listing 46.11 at the end of the chapter.

Creating a Basic Transformation

Before we can build a complex listing of artists, CDs, ratings, and recommendations, let's start out with a simple nested bulleted list of artists and CDs. The list will display each artist's name in bold, followed by a bulleted list of the artist's genres and a separate list of the artist's CDs. Listing 46.3 shows an XSL stylesheet that gives us this information.

Listing 46.3 `BasicTransformation.xsl`—An XSLT Stylesheet to Create a Simple Listing of Artists and CDs

```
<xsl:transform
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Listing 46.3 (continued)

```
    <xsl:output omit-xml-declaration="yes" />

    <xsl:template match="/cdcollection">
      <ul>
        <xsl:apply-templates />
      </ul>
    </xsl:template>

    <xsl:template match="/cdcollection/artist">
      <li>
        <b><xsl:value-of select="@name" /></b>
        <br />

        Genre:
        <ul>
          <xsl:apply-templates select="genre" />
        </ul>

        CDs:
        <ul>
          <xsl:apply-templates select="cd" />
        </ul>
      </li>
    </xsl:template>

    <xsl:template match="/cdcollection/artist/genre">
      <li><xsl:value-of select="." /></li>
    </xsl:template>

    <xsl:template match="/cdcollection/artist/cd">
      <li><xsl:value-of select="@name" /></li>
    </xsl:template>

  </xsl:transform>
```

Running that transformation (see "Performing the Transformation by Using `XmlTransform()`" later in the chapter) produces the HTML in Listing 46.4.

Listing 46.4 `ArtistsAndCDs.htm`—A Simple Listing of Artists and CDs

```
<ul>
  <li>
    <b>Air</b><br/>
    Genre:
    <ul>
      <li>Electronic</li>
    </ul>
    CDs:
    <ul>
      <li>10,000 Hz Legend</li>
      <li>Talkie Walkie</li>
    </ul>
  </li>
  <li>
    <b>Kylie Minogue</b><br/>
    Genre:
```

Listing 46.4 (continued)

```
        <ul>
          <li>Dance</li>
        </ul>
        CDs:
        <ul>
          <li>Fever</li>
          <li>Body Language</li>
        </ul>
      </li>
      <li>
        <b>Dannii Minogue</b><br/>
        Genre:
        <ul>
          <li>Dance</li>
          <li>Electronic</li>
        </ul>
        CDs:
        <ul>
          <li>Neon Nights</li>
          <li>You Won't Forget About Me EP</li>
        </ul>
      </li>
      <li>
        <b>Brooklyn Funk Essentials</b><br/>
        Genre:
        <ul>
          <li>Funk</li>
          <li>Dance</li>
          <li>Spoken Word</li>
        </ul>
        CDs:
        <ul>
          <li>Cool &amp; Steady &amp; Easy</li>
        </ul>
      </li>
      <li>
        <b>Felix Da Housecat</b><br/>
        Genre:
        <ul>
          <li>Electronica</li>
          <li>Dance</li>
          <li>Retro</li>
        </ul>
        CDs:
        <ul>
          <li>A Bugged Out Mix</li>
          <li>Kittenz and Thee Glitz</li>
          <li>Devin Dazzle &amp; The Neon Fever</li>
        </ul>
      </li>
    </ul>
```

Don't be overwhelmed by the stylesheet. It may seem completely foreign, so let's break it down bit by bit.

<xsl:transform>

XSL stylesheets are well-formed XML documents that use a specific set of tags to tell the XSL processor how to run the transformation. All transformation stylesheets use `<xsl:transform>` as their root element. (You may sometimes see `<xsl:stylesheet>` used instead of `<xsl:transform>`; they are interchangeable in most cases.)

The `version` attribute of `<xsl:transform>` specifies the version of the XSL standard that will be used to process the stylesheet, and the `xmlns:xsl` attribute identifies the namespace that contains the XSL tags. By and large, the transformations you create will always use the same values for these attributes as you see in Listing 46.3.

<xsl:output>

The first tag inside `<xsl:transform>` in Listing 46.3 is an `<xsl:output>` tag, which in this case tells the XSL processor not to output the XML declaration in the result HTML. If this `<xsl:output>` were not present, the resulting HTML file would have as its first line:

```
<?xml version="1.0" encoding="UTF-8"?>
```

which is not valid in an HTML document. There are nine other attributes of `<xsl:output>` that let you specify other behaviors of the output engine; these are summarized in Table 46.1.

By far the most commonly used attribute of `<xsl:output>` is `omit-xml-declaration`.

Table 46.1 Other Attributes of `<xsl:output>`

ATTRIBUTE NAME	POSSIBLE VALUES	DESCRIPTION
`method`	`xml`, `html`, or `text`	Tells the XSL processor what rules are used when creating output content. `method="html"` allows tags to be unclosed and `method="text"` does not validate the output at all.
`version`	Any number	Specifies the version number of the output method; usually only useful with `method="html"`
`indent`	`yes` or `no`	Specifies whether the XSL processor may add whitespace when generating the output content.
`encoding`		Tells the XSL processor what character set to use when outputting content.
`media-type`		Specifies the MIME type of the output content.
`doctype-system`	a valid doctype URI	Tells the XSL processor to put out a `<!DOCTYPE>` element with the specified system URI.
`doctype-public`	a valid doctype URI	Tells the XSL processor to put out a `<!DOCTYPE>` element with the specified public URI.
`standalone`	`yes` or `no`	Tells the XSL processor whether to make the resulting output content a standalone XML document. The XSL processor will put the contents of all elements of this name into a CDATA section in the output document.

<xsl:template> and <xsl:apply-templates>

After specifying the output rules for the stylesheet by using <xsl:output>, we can now define the templates used to create the output content by using <xsl:template>.

There are four templates in this stylesheet, corresponding to the four elements we want to process. This is not to say that there will always be a one-to-one correspondence between templates and elements; in fact, there are usually only a few templates in a stylesheet compared to the number of elements present. In this first example, however, we want to keep things simple.

The first <xsl:template> is:

```
<xsl:template match="/cdcollection">
  <ul>
    <xsl:apply-templates />
  </ul>
</xsl:template>
```

The first line of the template is the opening <xsl:template> tag, containing a single `match` attribute. `match` tells the XSL processor which nodes in the source document this template processes; in this case, this template handles `cdcollection` nodes located immediately under the document root.

Inside the template is a <ul> element with an <xsl:apply-templates> tag in between its opening and closing tag. <xsl:apply-templates> tells the XSL processor to start looping through the children of the current `cdcollection` node and to start applying XSL templates to each one.

One of the hardest things for beginning XSL programmers to grasp is exactly how an XSL processor uses the stylesheet. The conception that many have is that the stylesheet tells the processor in which order to process nodes in the XML document. However, the reverse is true. The XSL processor loops over nodes in the XML document, and for each one, attempts to find a template in the stylesheet that matches the current node. Once it finds the template, it calls the template almost as if the template were a function. It's more of an event-driven model than a procedural one.

<xsl:value-of>

So once the XSL processor sees the <xsl:apply-templates> in the `/cdcollection` template, the processor loops over all the children of the `cdcollection` node. When it sees that the child node is an `artist` node, it looks for a template that matches and finds this one:

```
<xsl:template match="/cdcollection/artist">
  <li>
    <b><xsl:value-of select="@name" /></b>
    <br />

    Genre:
    <ul>
      <xsl:apply-templates select="genre" />
    </ul>

    CDs:
    <ul>
      <xsl:apply-templates select="cd" />
```

```
    </ul>
  </li>
</xsl:template>
```

For each artist node, the XSL processor outputs an opening `<li>` tag, then puts out the artist's name in bold by using `<xsl:value-of>`, which is another XSL tag. Its select attribute takes an XPath expression and outputs the returned value, which in this case is the name of the current artist.

`<xsl:apply-templates>` Revisited

After putting out the name of the artist by using `<xsl:value-of>`, the artist template then generates the lists of genres and CDs for the artist. Unlike the previous use of `<xsl:apply-templates>`, we are no longer indiscriminately looping over all child nodes. Rather, we are using the select attribute of `<xsl:apply-templates>` to selectively loop over only certain children. (`select` works the same way for both `<xsl:value-of>` and `<xsl:apply-templates>`; in both cases select specifies an XPath expression that works from the current node.)

That covers elements within three of the four templates shown in Listing 46-3. The only thing remaining to be explained is the `<xsl:value-of>` in the `genre` template:

```
<xsl:value-of select="." />
```

The period is an XPath expression meaning "current node." Getting the value of the current node returns the content between the opening and closing tags; as such, the `<xsl:value-of>` means to output whatever the current genre is.

Performing the Transformation by Using `XmlTransform()`

An XSL stylesheet by itself does nothing. In order to transform the input XML according to the stylesheet, you must use ColdFusion to run the transformation as shown in Listing 46.5.

Listing 46.5 `TransformArtistAndCDList.cfm`—Transforming XML content using XSL

```
<cffile action="READ"
  file="#ExpandPath('CDCollection.xml')#"
  variable="xmlDocument">

<cfset transformedContent = XmlTransform(xmlDocument, "BasicTransformation.xsl")>

<cffile action="WRITE"
  file="#ExpandPath('ArtistsAndCDs.htm')#"
  output="#transformedContent#">

Finished transforming content!
```

The `XmlTransform()` function applies the specified stylesheet to the passed-in XML document and returns the generated output as a string. In Listing 46.5 we are writing it to a file, but you could just as easily output it to the screen, return it from a CFC method call, or use it in some other way.

The XSL file argument can be either a relative or absolute file path. It can also be an XSL stylesheet stored in a string or a URL to a file, where valid protocol identifiers include `http`, `https`, `ftp`, and `file`.

Ignoring Nodes in the Hierarchy

So far we've been able to completely ignore the recommend nodes in the XML document by simply never including them in the select attributes that retrieve sets of nodes. But what if we wanted to skip elements in the document hierarchy (for instance, what if we wanted to show a list of CDs without showing artists at all)? Listing 46.6 shows a stylesheet that does just that.

Listing 46.6 `CDList.xsl`—A Stylesheet that Bypasses Artist Information

```
<xsl:transform
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output omit-xml-declaration="yes" />

  <xsl:template match="/cdcollection">
    <ul>
      <xsl:apply-templates select="//cd" />
    </ul>
  </xsl:template>

  <xsl:template match="/cdcollection/artist/cd">
    <li><xsl:value-of select="@name" /></li>
  </xsl:template>

</xsl:transform>
```

All we have to do is create a template for the root node, then use XPath to find all `cd` nodes in the file and output each one using the separately defined template.

NOTE

Why did we create a match for the root node of the file? One of the lesser-known caveats of XSL development is that if you don't include a root node, the XSL processor will output the value of every element it comes across until it finds the first match. This means that you run the risk of having errant text pop up if you don't always include a match for the root node.

Creating a More Complex Transformation

So now that we've created a few simple transformations, let's tackle the complex listing seen at the beginning of the chapter.

`<xsl:if>`

Although XSL is not a programming language in the traditional sense, it does have some of the standard flow control constructs present in other languages. The first of these is `<xsl:if>`, as demonstrated in Listing 46.7.

Listing 46.7 `IfInATransformation.xsl`—Using `<xsl:if>`

```
<xsl:transform
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Listing 46.7 (continued)

```
   <xsl:output omit-xml-declaration="yes" />

   <xsl:template match="/cdcollection">
     <style>
       td {
         vertical-align: top;
       }
     </style>

     <table cellspacing="0">
       <tr>
         <th>Artist/CD Name</th>
       </tr>
       <xsl:apply-templates />
     </table>
   </xsl:template>

   <xsl:template match="/cdcollection/artist">
     <tr>
       <td>
         <b><xsl:value-of select="@name" /></b>
         <xsl:if test="count(cd) = 1">
           <br /><span style="color: red;">New Artist!</span>
         </xsl:if>
       </td>
     </tr>
     <xsl:apply-templates select="cd" />
     <tr>
       <td><hr /></td>
     </tr>
   </xsl:template>

   <xsl:template match="/cdcollection/artist/cd">
     <tr>
       <td><xsl:value-of select="@name" /></td>
     </tr>
   </xsl:template>

</xsl:transform>
```

In Listing 46.7, we use `<xsl:if>` to output a block of text that says "New Artist" if the artist only has one CD in the collection. `count()` is an XPath function that counts the number of nodes that match the given expression.

It is worth noting that XSL does not have an else construct like `CFELSE` or the `else` keyword in other languages. One alternative is to use syntax like:

```
<xsl:if test="count(cd) = 1">
  ... code goes here ...
</xsl:if>
<xsl:if test="not(count(cd) = 1)">
  ... other code goes here ...
</xsl:if>
```

The other alternative is to use `<xsl:choose>` and `<xsl:otherwise>` as described in the next section.

<xsl:choose>, <xsl:when>, and <xsl:otherwise>

The second XSL flow control construct is almost like the equivalent of CFSWITCH. In Listing 46.8, we use <xsl:choose>, <xsl:when>, and <xsl:otherwise> to convert the numeric rating into a user-friendly text string.

Listing 46.8 ChooseInATransformation.xsl—Using <xsl:choose>

```
<xsl:transform
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output omit-xml-declaration="yes" />

  <xsl:template match="/cdcollection">
    <style>
      td {
        vertical-align: top;
      }
    </style>

    <table cellspacing="0">
      <tr>
        <th>Artist/CD Name</th>
        <th>Rating (for CDs)</th>
      </tr>
      <xsl:apply-templates />
    </table>
  </xsl:template>

  <xsl:template match="/cdcollection/artist">
    <tr>
      <td colspan="2">
        <b><xsl:value-of select="@name" /></b>
        <xsl:if test="count(cd) = 1">
          <br /><span style="color: red;">New Artist!</span>
        </xsl:if>
      </td>
    </tr>
    <xsl:apply-templates select="/cdcollection/artist/cd" />
    <tr>
      <td colspan="2"><hr /></td>
    </tr>
  </xsl:template>

  <xsl:template match="cd">
    <tr>
      <td><xsl:value-of select="@name" /></td>
      <td>
        <xsl:choose>
          <xsl:when test="@rating = 1">It's OK</xsl:when>
          <xsl:when test="@rating = 2">Decent</xsl:when>
          <xsl:when test="@rating = 3">Like It</xsl:when>
          <xsl:when test="@rating = 4">Love It</xsl:when>
          <xsl:when test="@rating = 5">Favorite!</xsl:when>
          <xsl:otherwise>Unknown</xsl:otherwise>
```

Listing 46.8 (continued)

```
      </xsl:choose>
    </td>
  </tr>
</xsl:template>

</xsl:transform>
```

The major difference between `<xsl:choose>` and switch constructs in other languages is that XSL evaluates each condition separately, like a set of elseifs, making `<xsl:choose>` more flexible (but also slower) than switch constructs other languages.

`<xsl:for-each>` and `<xsl:text>`

The last of the flow control constructs in XSL is the loop construct, `<xsl:for-each>`. Where `<xsl:apply-templates>` tells the XSL processor to loop over child nodes and find a matching template for each one, `<xsl:for-each>` gives a specific action to be taken for each node.

In Listing 46.9, I am using `<xsl:for-each>` to put out the list of genres in which each artist performs.

Listing 46.9 `LoopInATransformation.xsl`—Using `<xsl:for-each>`

```
<xsl:transform
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output omit-xml-declaration="yes" />

  <xsl:template match="/cdcollection">
    <style>
      td {
        vertical-align: top;
      }
    </style>

    <table cellspacing="0">
      <tr>
        <th>Artist/CD Name</th>
        <th>Rating (for CDs)</th>
      </tr>
      <xsl:apply-templates />
    </table>
  </xsl:template>

  <xsl:template match="/cdcollection/artist">
    <tr>
      <td colspan="2">
        <b><xsl:value-of select="@name" /></b><br />
        <xsl:if test="count(cd) = 1">
          <span style="color: red;">New Artist!</span><br />
        </xsl:if>
        <xsl:for-each select="genre">
```

Listing 46.9 (continued)

```
            <xsl:value-of select="." />
            <xsl:if test="position() &lt; last()">
              <xsl:text>, </xsl:text>
            </xsl:if>
          </xsl:for-each>
        </td>
      </tr>
      <xsl:apply-templates select="cd" />
      <tr>
        <td colspan="2"><hr /></td>
      </tr>
    </xsl:template>

    <xsl:template match="/cdcollection/artist/cd">
      <tr>
        <td><xsl:value-of select="@name" /></td>
        <td>
          <xsl:choose>
            <xsl:when test="@rating = 1">It's OK</xsl:when>
            <xsl:when test="@rating = 2">Decent</xsl:when>
            <xsl:when test="@rating = 3">Like It</xsl:when>
            <xsl:when test="@rating = 4">Love It</xsl:when>
            <xsl:when test="@rating = 5">Favorite!</xsl:when>
            <xsl:otherwise>Unknown</xsl:otherwise>
          </xsl:choose>
        </td>
      </tr>
    </xsl:template>

  </xsl:transform>
```

There are several new concepts in the `<xsl:for-each>` loop above, so let's break it down line-by-line.

The first line is the `<xsl:for-each>` loop, and the `select` attribute works exactly like the `select` attribute of `<xsl:apply-templates>`.

You're familiar with `<xsl:value-of>` by now, but there's something peculiar about the `<xsl:if>` test. What we're doing is putting out a comma only if the current node is not the last genre in the selected set. `position()` and `last()` are XPath functions that return the position of the current node and last node in the current set, respectively.

However, the `<` seems out of place. In any other language, we would just say:

```
position() < last()
```

But remember that an XSL stylesheet is just another XML document. Meaning that it must be well-formed, so special characters must be escaped. As such, we must replace the `<` with `<`.

Finally, let's turn our attention to the `<xsl:text>` element. Unfortunately, XSL can be very sloppy with whitespace, but there are times when we want to include literal whitespace or other text within a stylesheet, and embedding text within an `<xsl:text>` does just that.

`<xsl:element>` and `<xsl:attribute>`

Now that we've covered flow control in XSL, it's time for one of the most difficult concepts for many programmers to master. I want to surround the names of artists and CDs in my list with a link to another ColdFusion page, such that in the output stream I get markup like

```
<a href="ArtistDetail.cfm?id=1">10,000 Hz Legend</a>
```

Most developers' first instinct would be something like the following:

```
<xsl:template match="/cdcollection/artist">
  <a href="ArtistDetail.cfm?id=<xsl:value-of select="@id" />"><xsl:value-of
select="@name" /></a>
</xsl:template>
```

This is much like what would be done in ColdFusion, but it is not valid in XSL because the markup is not valid XML. It would seem that we are out of luck, but XSL provides the `<xsl:element>` and `<xsl:attribute>` tags to do just what we're trying to do, as shown in Listing 46.10.

Listing 46.10 `DynamicElements.xsl`—Using `<xsl:element>` to Create Dynamic HTML Elements

```
<xsl:transform
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:output omit-xml-declaration="yes" />

 <xsl:template match="/cdcollection">
   <style>
     td {
       vertical-align: top;
     }
   </style>

   <table cellspacing="0">
     <tr>
       <th>Artist/CD Name</th>
       <th>Rating (for CDs)</th>
     </tr>
     <xsl:apply-templates />
   </table>
 </xsl:template>

 <xsl:template match="/cdcollection/artist">
   <tr>
     <td colspan="2">
       <b>
         <xsl:element name="a">
           <xsl:attribute name="href">
             <xsl:text>ArtistDetail.cfm?id=</xsl:text>
             <xsl:value-of select="@id" />
           </xsl:attribute>
           <xsl:value-of select="@name" />
         </xsl:element>
       </b><br />
       <xsl:if test="count(cd) = 1">
         <span style="color: red;">New Artist!</span><br />
```

Listing 46.10 (continued)

```
          </xsl:if>
          <xsl:for-each select="genre">
            <xsl:value-of select="." />
            <xsl:if test="position() &lt; last()">
              <xsl:text>, </xsl:text>
            </xsl:if>
          </xsl:for-each>
        </td>
      </tr>
      <xsl:apply-templates select="cd" />
      <tr>
        <td colspan="2"><hr /></td>
      </tr>
    </xsl:template>
    <xsl:template match="/cdcollection/artist/cd">
      <tr>
        <td>
          <xsl:element name="a">
            <xsl:attribute name="href">
              <xsl:text>CDDetail.cfm?id=</xsl:text>
              <xsl:value-of select="@id" />
            </xsl:attribute>
            <xsl:value-of select="@name" />
          </xsl:element>
        </td>
        <td>
          <xsl:choose>
            <xsl:when test="@rating = 1">It's OK</xsl:when>
            <xsl:when test="@rating = 2">Decent</xsl:when>
            <xsl:when test="@rating = 3">Like It</xsl:when>
            <xsl:when test="@rating = 4">Love It</xsl:when>
            <xsl:when test="@rating = 5">Favorite!</xsl:when>
            <xsl:otherwise>Unknown</xsl:otherwise>
          </xsl:choose>
        </td>
      </tr>
    </xsl:template>

  </xsl:transform>
```

Because the two sections containing `<xsl:element>` are so similar, let's just dissect the first one:

```
<xsl:element name="a">
  <xsl:attribute name="href">
    <xsl:text>ArtistDetail.cfm?id=</xsl:text>
    <xsl:value-of select="@id" />
  </xsl:attribute>
  <xsl:value-of select="@name" />
</xsl:element>
```

Here I am creating an `<a>` element with a single `href` attribute. I can break the attribute value across multiple lines because `<xsl:text>` ensures that the only whitespace in the attribute is what's inside of the `<xsl:text>` block.

Using Named Templates

Now we finally come to displaying artist and CD recommendations. Basically, we want to display the name of the recommended artists and/or CDs next to each artist and/or CD that has `recommend` nodes in the source document. This means that we need a way to look up an artist's or CD's name given its ID. In most languages, you'd create a function to do this work for you, but remember that XSL for the most part does not have the concept of a "function."

However, XSL does have templates. Until now, all of our templates have used the `match` attribute to specify that they be called automatically for certain nodes in the input XML document:

```
<xsl:template match="/cdcollection/artist/cd">
...
</xsl:template>
```

There is also, however, a name attribute used to specify that a given template is programmatically called from within another template:

```
<xsl:template name="MyTemplateName">
...
</xsl:template>
```

This second form of `<xsl:template>` can be called using the `<xsl:call-template>` tag:

```
<xsl:call-template name="MyTemplateName" />
```

The real beauty of named templates, however, is the ability to pass parameters. To define a parameter in the named template, you use the `<xsl:param>` tag:

```
<xsl:template name="MyTemplateName">
  <xsl:param name="MyParamName" />
  ...
</xsl:template>
```

And to pass the parameter in the template call, you use the `<xsl:with-param>` tag:

```
<xsl:call-template name="MyTemplateName">
  <xsl:with-param name="MyParamName" select="XPathExpression" />
</xsl:call-template>
```

So coming back to the main Artist/CD Listing transformation, Listing 46.11 shows the complete transformation with all the new features present.

Listing 46.11 `CompleteTransformation.xsl`—A Complete Listing of Artists, CDs, and Recommendations

```
<xsl:transform
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output omit-xml-declaration="yes" />

  <xsl:template match="/cdcollection">
    <style>
      td {
        vertical-align: top;
```

Listing 46.11 (continued)

```
    }
  </style>

  <table cellspacing="0">
    <tr>
      <th>Artist/CD Name</th>
      <th>Rating (for CDs)</th>
      <th>Recommendations</th>
    </tr>
    <xsl:apply-templates />
  </table>
</xsl:template>

<xsl:template match="/cdcollection/artist">
  <tr>
    <td colspan="2">
      <b>
        <xsl:element name="a">
          <xsl:attribute name="href">
            <xsl:text>ArtistDetail.cfm?id=</xsl:text>
            <xsl:value-of select="@id" />
          </xsl:attribute>
          <xsl:value-of select="@name" />
        </xsl:element>
      </b><br />
      <xsl:if test="count(cd) = 1">
        <span style="color: red;">New Artist!</span><br />
      </xsl:if>
      <xsl:for-each select="genre">
        <xsl:value-of select="." />
        <xsl:if test="position() &lt; last()">
          <xsl:text>, </xsl:text>
        </xsl:if>
      </xsl:for-each>
    </td>
    <td>
      <xsl:if test="count(recommend)">
        <ul>
          <xsl:for-each select="recommend">
            <li>
              <xsl:call-template name="FindArtistName">
                <xsl:with-param name="id" select="@artist" />
              </xsl:call-template>
            </li>
          </xsl:for-each>
        </ul>
      </xsl:if>
    </td>
  </tr>
  <xsl:apply-templates select="cd" />
  <tr>
    <td colspan="3"><hr /></td>
  </tr>
</xsl:template>
```

Listing 46.11 (continued)

```
<xsl:template match="/cdcollection/artist/cd">
  <tr>
    <td>
      <xsl:element name="a">
        <xsl:attribute name="href">
          <xsl:text>CDDetail.cfm?id=</xsl:text>
          <xsl:value-of select="@id" />
        </xsl:attribute>
        <xsl:value-of select="@name" />
      </xsl:element>
    </td>
    <td>
      <xsl:choose>
        <xsl:when test="@rating = 1">It's OK</xsl:when>
        <xsl:when test="@rating = 2">Decent</xsl:when>
        <xsl:when test="@rating = 3">Like It</xsl:when>
        <xsl:when test="@rating = 4">Love It</xsl:when>
        <xsl:when test="@rating = 5">Favorite!</xsl:when>
        <xsl:otherwise>Unknown</xsl:otherwise>
      </xsl:choose>
    </td>
    <td>
      <xsl:if test="count(recommend)">
        <ul>
          <xsl:for-each select="recommend">
            <li>
              <xsl:call-template name="FindCDName">
                <xsl:with-param name="id" select="@cd" />
              </xsl:call-template>
            </li>
          </xsl:for-each>
        </ul>
      </xsl:if>
    </td>
  </tr>
</xsl:template>

<xsl:template name="FindArtistName">
  <xsl:param name="id" />

  <xsl:value-of select="//artist[@id = $id]/@name" />
</xsl:template>

<xsl:template name="FindCDName">
  <xsl:param name="id" />

  <xsl:value-of select="//cd[@id = $id]/@name" />
</xsl:template>

</xsl:transform>
```

In the main templates for `cd` and `artist`, I loop over the `recommend` nodes (if they exist) using `<xsl:for-each>`, and for each one, I call the `FindArtistName` or `FindCDName` template, passing in the `artist` or `cd` attribute. Notice that inside the named templates, I reference the parameter value by using `$id`. `$` is an XPath prefix that means “parameter” or “variable.”

Listing 46.12 shows the HTML markup generated by `CompleteTransformation.xsl`. To generate this file, run `TransformComplexList.cfm`, which is included in the sample code for this chapter. Note that I have cleaned up the markup to make it easier to read; this in no way affects the displayed content.

Listing 46.12 `CompleteArtistsAndCDs.htm`—The Output of the Final Transformation

```
<style>
  td {
    vertical-align: top;
  }
</style>

<table cellspacing="0">
<tr>
  <th>Artist/CD Name</th>
<th>Rating (for CDs)</th>
  <th>Recommendations</th>
</tr>
<tr>
  <td colspan="2">
    <b><a href="ArtistDetail.cfm?id=1">Air</a></b><br/>
    Electronic
  </td>
  <td/>
</tr>
<tr>
  <td><a href="CDDetail.cfm?id=1">10,000 Hz Legend</a></td>
  <td>Like It</td>
  <td>
    <ul>
      <li>Talkie Walkie</li>
    </ul>
  </td>
</tr>
<tr>
  <td><a href="CDDetail.cfm?id=2">Talkie Walkie</a></td>
  <td>Love It</td>
  <td>
    <ul>
      <li>10,000 Hz Legend</li>
      <li>You Won't Forget About Me EP</li>
    </ul>
  </td>
</tr>
<tr><td colspan="3"><hr/></td></tr>
<tr>
  <td colspan="2">
    <b><a href="ArtistDetail.cfm?id=2">Kylie Minogue</a></b><br/>
    Dance
  </td>
  <td>
    <ul>
      <li>Dannii Minogue</li>
    </ul>
  </td>
</tr>
```

Listing 46.12 (continued)

```
<tr>
  <td><a href="CDDetail.cfm?id=3">Fever</a></td>
  <td>Like It</td>
  <td>
    <ul>
      <li>Body Language</li>
    </ul>
  </td>
</tr>
<tr>
  <td><a href="CDDetail.cfm?id=4">Body Language</a></td>
  <td>Love It</td>
  <td>
    <ul>
      <li>Neon Nights</li>
    </ul>
  </td>
</tr>
<tr><td colspan="3"><hr/></td></tr>
<tr>
  <td colspan="2">
    <b><a href="ArtistDetail.cfm?id=3">Dannii Minogue</a></b><br/>
    Dance, Electronic
  </td>
  <td>
    <ul>
      <li>Kylie Minogue</li>
    </ul>
  </td>
</tr>
<tr>
  <td><a href="CDDetail.cfm?id=5">Neon Nights</a></td>
  <td>Favorite!</td>
  <td>
    <ul>
      <li>Body Language</li>
    </ul>
  </td>
</tr>
<tr>
  <td><a href="CDDetail.cfm?id=6">You Won't Forget About Me EP</a></td>
  <td>Favorite!</td>
  <td>
    <ul>
      <li>Neon Nights</li>
    </ul>
  </td>
</tr>
<tr><td colspan="3"><hr/></td></tr>
<tr>
  <td colspan="2">
    <b><a href="ArtistDetail.cfm?id=4">Brooklyn Funk Essentials</a></b><br/>
    <span style="color: red;">New Artist!</span><br/>
    Funk, Dance, Spoken Word
  </td>
  <td>
```

Listing 46.12 (continued)

```
        <ul>
          <li>Air</li>
          <li>Felix Da Housecat</li>
        </ul>
      </td>
    </tr>
    <tr>
      <td><a href="CDDetail.cfm?id=7">Cool &amp; Steady &amp; Easy</a></td>
      <td>Favorite!</td>
      <td>
        <ul>
          <li>Body Language</li>
          <li>Neon Nights</li>
        </ul>
      </td>
    </tr>
    <tr><td colspan="3"><hr/></td></tr>
    <tr>
      <td colspan="2">
        <b><a href="ArtistDetail.cfm?id=5">Felix Da Housecat</a></b><br/>
        Electronica, Dance, Retro
      </td>
      <td>
        <ul>
          <li>Kylie Minogue</li>
          <li>Brooklyn Funk Essentials</li>
        </ul>
      </td>
    </tr>
    <tr>
      <td><a href="CDDetail.cfm?id=8">A Bugged Out Mix</a></td>
      <td>Like It</td>
      <td>
        <ul>
          <li>Body Language</li>
          <li>You Won't Forget About Me EP</li>
          <li>Kittenz and Thee Glitz</li>
        </ul>
      </td>
    </tr>
    <tr>
      <td><a href="CDDetail.cfm?id=9">Kittenz and Thee Glitz</a></td>
      <td>Favorite!</td>
      <td/>
    </tr>
    <tr>
      <td><a href="CDDetail.cfm?id=10">Devin Dazzle &amp; The Neon Fever</a></td>
      <td>Favorite!</td>
      <td>
        <ul>
          <li>Kittenz and Thee Glitz</li>
        </ul>
      </td>
    </tr>
    <tr><td colspan="3"><hr/></td></tr>
    </table>
```

Take some time to compare Listings 46.11 and 46.12 to see how the XSL stylesheet generated the HTML markup. Also compare these listings to Figure 46.2 to see how it all fits together.

NOTE

It's also worth noting that as of ColdFusion 7, it's possible to pass XSLT parameter values on the `XmlTransform()` function itself, using a new optional third argument. With this, one can name a structure created to hold keys whose names are mapped to the XSLT parameters in the XSL file. For more information, see the Adobe ColdFusion CFML Reference manual.

Concluding the discussion of XSLT and ColdFusion, note that ColdFusion 7 also added the feature called XML forms, which are forms that generate XForms-compliant XML. These are normally formatted using an XSLT skin. You can use XML forms with skins that ColdFusion provides or use the knowledge you've learned in this chapter to explore customizing them.

More XPath and XSLT Resources

This chapter only scratched the surface of the many things you can do with XSLT and XPath. For more information, check out these web sites.

- `http://www.w3c.org/TR/xpath` is the W3C's working specification for XPath. This document is not for the faint of heart, but it is the definitive standard on XPath syntax and operation.
- `http://www.zvon.org/xxl/XPathTutorial/General/examples.html` is the XPath Tutorial at `www.zvon.org`. It's very much like an immersion course in XPath, so it's easy to feel lost at first, but there is some excellent information here.
- `http://www.w3.org/TR/xslt` is the W3C's working specification for XSLT. This document is a very scientific document and is not recommended for beginners. However, for those looking for a deep understanding of how XSLT works, this is just the place to go.
- `http://www.zvon.org/xxl/XSLTutorial/Output/index.html` is the XSLT tutorial at `www.zvon.org`. This is an excellent starting point for people who want an easy introduction to the world of practical XSLT.
- `http://www.zvon.org/xxl/XSLTreference/Output/index.html` is the XSLT/XPath reference at `www.zvon.org`. This is an indispensable tool for anyone developing XSLT stylesheets.

CHAPTER 47

Using WDDX

IN THIS CHAPTER

Introducing WDDX E231
Using WDDX with ColdFusion E234
Anatomy of a WDDX Packet E241
Using WDDX Packets to Store Information in Files E245
Other Places to Store WDDX Packets E258
Exchanging WDDX Packets Among Web Pages E260
Binary Content in WDDX Packets E268

When you're developing Web applications with ColdFusion, you often deal with complex chunks of data, such as structures, recordsets returned from queries, and arrays. Often, you need to save that data to disk, store it in a database, or move it from one place to another. Sometimes you even need to pass these chunks of data between environments. For instance, you might want to move information from ColdFusion to JavaScript or from a .NET application to PHP or Java or ColdFusion.

Yet the easiest way to store and share information is just to use ordinary ASCII text. If you had some way to turn your recordsets, arrays, and structures into blocks of ordinary ASCII text and back again, it would be easy to store that text in files or databases, exchange it via HTTP, pass it around in Web pages or email messages, and more.

This chapter introduces the *Web Dynamic Data Exchange (WDDX)* format, a simple XML vocabulary that makes it painless to convert any type of complex data structure to text and back again.

Introducing WDDX

The WDDX format was created in 1998 by Allaire's legendary Sim Simeonov as a simplified way to use XML for exchanging data among Internet applications. At that time, many of the XML tools commonly available today were just beginning to emerge, and high-level XML technologies such as SOAP, SAX, and XSLT for representing and massaging data were not yet in the mainstream.

The idea was to come up with a simple way of thinking about XML that allowed ColdFusion users to start reaping its benefits right away, without having to learn about XML parsers, DTDs, document object models, entities, namespaces, and so on. Of course, all of those concepts are important and very useful, but Sim realized that plenty of uses for XML could be facilitated without forcing people to get that deep into the theory and vocabulary of it all. He set out to create a system that would sit on top of XML, hiding all the complexities of parsing, creating, and populating XML documents.

The result was WDDX and the `<cfwddx>` tag, which first appeared in ColdFusion 4.0. In true ColdFusion style, `<cfwddx>` gave developers the ability to convert data to and from XML in a single step. All productivity, zero theory.

In a nutshell, WDDX's mission is to take any kind of data—a single number or a complex structure of arrays within other arrays—and instantly turn it into a chunk of simple XML. That chunk of XML can then be passed from place to place with reckless abandon. When it's time to actually use the data again, it can be read from the WDDX format back to the way it was before the whole process began.

Perhaps the coolest thing about exchanging data with WDDX is that it takes care of preserving data types for you. So, if part of the data started as a date variable on the way into the WDDX format, it ends up as a date variable when it comes back out. This holds true even if the data gets passed between two different programming environments.

For instance, consider a ColdFusion array that holds a number, a date, and an ordinary string. This array can be converted to WDDX and provided to a JavaScript routine on a Web browser. The JavaScript routine can refer to the array just like any other JavaScript-style array. In addition, the date stored in the array is a true JavaScript `Date` object, and the number is a true JavaScript `Number` object. The array is successfully "passed" from CFML to an entirely different type of language (JavaScript). The same goes for other types of applications, such as Perl, Active Server Pages, and Java.

Some WDDX Terminology

I'll start off by introducing some WDDX concepts and terminology.

WDDX Packet

A *WDDX packet* is any chunk of data stored in the WDDX format. As you will soon learn, a WDDX packet looks similar to any other XML document, as well as to HTML or CFML, for that matter. Because it's tag based, a WDDX packet is simple, is easy to read, and practically describes itself. Each piece of information is surrounded by special opening and closing tags that allow complex data types such as structures and arrays to be stored in the packet. That's pretty hard to accomplish with other text-based formats—comma-separated or space-delimited text, for example. Depending on the nature of the data, it can even be a bit tricky when using a relational database system.

Serializing

Serializing is the process of converting a piece of data into a WDDX packet. To serialize data, you need to use a language or an environment that has access to a function or procedure that knows about the WDDX format and how to serialize data properly. For instance, in a ColdFusion template, the serialization process is performed by the `<cfwddx>` tag. In other languages and environments, the functions or methods you use to serialize a particular chunk of data are different, but the resulting WDDX packet should be the same and can be understood by any program that supports WDDX.

Consider this example: Serializing the string `Hello, World!` creates a WDDX packet that includes the string itself, surrounded by a pair of `<string>` tags, like this:

```
<string>Hello, World!</string>
```

The strategy of surrounding each value with tags that explain its type enables the WDDX packet to hold descriptions of your data right along with the data itself. That's one of the coolest things about the WDDX format: The packet can describe itself to whatever application needs to read it.

Deserializing

Deserializing is the opposite of serializing. It's the process of pulling the actual data out of a WDDX packet. For instance, if an application needs to "unpack" the `Hello, World` snippet shown just above, it first looks at the tags to learn what type of data is in the packet. Because WDDX says that this packet contains string data, the application knows it should store the value sitting between the tags as a string. For strongly typed development environments such as Java, C++, and Delphi, the data type is often very important.

Tools and Languages Supported by WDDX

WDDX support is available for a number of languages and development tools. Of course, it's supported by CFML, which means that Adobe has built WDDX into ColdFusion as the easy-to-use `<cfwddx>` tag. You'll find the `<cfwddx>` tag in many of the code listings in this chapter.

There is also a COM object available that brings WDDX functionality to any COM-enabled development tool or application. This means you can use WDDX to share information among ColdFusion, Active Server Pages (ASP), Visual Basic, and applications built with COM-enabled development tools such as Visual Basic, Delphi, Visual C++, and so on. This COM object is capable of performing all the WDDX-related tasks that ColdFusion templates can do natively.

There is a complete Java implementation of WDDX, a Perl 5 package, and support for WDDX is built into PHP 4 natively. Adobe also provides WDDX support for JavaScript, which enables you to easily make complex, server-side variables visible to your Web pages.

NOTE

WDDX can also be used to communicate between ColdFusion and other technologies (like Java and COM). To facilitate WDDX communication you will need to add WDDX support to those applications. This is accomplished using the WDDX SDK available from `http://www.openwddx.org/`.

WDDX or XML?

It's important to understand that WDDX is simply a type of XML document, designed to provide some of the benefits of XML without your having to first define XML schemas and data definitions. As such, WDDX is not so much an alternative to XML as it is a simple way to leverage XML.

When an XML type has been defined and agreed upon, applications should definitely use that type instead of the more generic WDDX. But if no XML type has been defined, WDDX can be an extremely and immediately useful tool.

NOTE

WDDX is used extensively by ColdFusion itself. In fact, ColdFusion's XML configuration files (stored in the `lib` directory under the ColdFusion root) are all in WDDX format.

Using WDDX with ColdFusion

Now that you have an idea what WDDX is all about, you can start learning how to use it in your ColdFusion applications. This section introduces you to the `<cfwddx>` tag and gets you thinking about different ways you can use WDDX in your own pages. You will find that WDDX is a very flexible technology, appropriate for solving many types of problems, from simple to complex, lofty to mundane.

NOTE

WDDX is not just for ColdFusion developers. The same basic techniques explained in this book can be used within Java, Perl, and more. And data that has been converted to WDDX can almost always be effortlessly shared between any of these applications with no loss of integrity.

Introducing the `<cfwddx>` Tag

Each language or environment that supports WDDX has some way to serialize and deserialize WDDX packets. In ColdFusion, it's the `<cfwddx>` tag. You use `<cfwddx>` to serialize data from native CFML variables to the WDDX packet format. You also use it to deserialize the data from the WDDX packet back into native ColdFusion variables.

First, I'll show you how to use the `<cfwddx>` tag to do some simple serialization and deserialization of WDDX packets. Then we'll take a closer look at what the actual packets look like. The first thing for you to understand is the syntax supported by the `<cfwddx>` tag, as outlined in Table 47.1.

Table 47.1 `<cfwddx>` Tag Syntax

ATTRIBUTE	DESCRIPTION
`actio`	Required. Specifies whether to convert to or from the WDDX format. Use `action="cfml2wddx"` to serialize a ColdFusion variable to a WDDX packet. Use `action="wddx2cfml"` to deserialize a WDDX packet back into a native ColdFusion variable.
`input`	Required. The value to be converted. If you are using `action="cfml2wddx"`, here is where you provide the value you want to serialize. If you are using `action="wddx2cfml"`, provide the WDDX packet you want to deserialize.
`OUTPUT`	The name of a variable to hold the result of the conversion. If you are using `action="cfml2wddx"`, the serialized WDDX packet will be stored in the variable you specify here. If you are using `action="wddx2cfml"`, the data from the WDDX packet will be deserialized and stored in this variable.
`usetimezoneinfo`	Optional. Relevant only when deserializing data with `action="wddx2cfml"`. If `Yes` (the default) and the WDDX packet contains dates that contain time zone information, ColdFusion will convert the dates to the server's time zone during the deserialization process. If `No`, all time zone information in the packet is ignored.
`validate`	Optional. Relevant only when deserializing data with `action="wddx2cfml"`. If `No` (the default), it is assumed that the packet provided to the `input` attribute is known to be a valid WDDX packet. If `Yes`, the packet is checked for validity first, which adds a small amount of overhead. In general, the `IsWDDX()` function is a better way to make sure a packet is valid; for details, see the section "Validating Packets with `IsWDDX()`" later in this chapter.

NOTE

`<cfwddx>` supports two other `action` values, as well (`cfml2js` and `wddx2js`), and one more attribute (`toplevelvariable`). These items are all specific to using ColdFusion with JavaScript and are discussed in Chapter 48, "Using JavaScript and ColdFusion Together." online.

Creating Your First WDDX Packet

Listing 47.1 shows how to use `<cfwddx>` to serialize a simple string value into a WDDX packet. This listing then displays the packet and also saves the packet to the server's drive as a file called `StringPacket.txt` (see Figure 47.1).

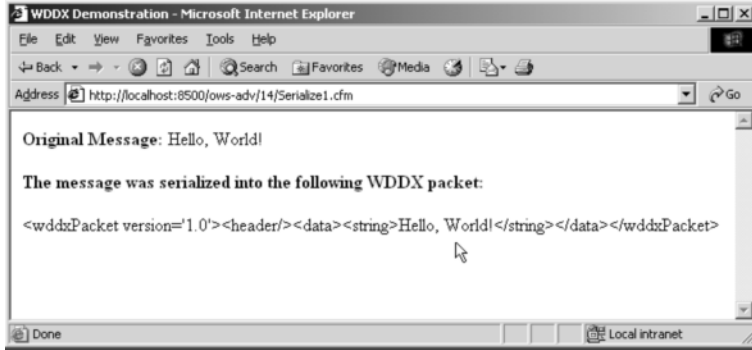


Figure 47.1

Simple strings get placed between `<string>` tags in the WDDX format.

Listing 47.1 `Serialize1.cfm`—Converting a Simple String to WDDX

```
<!---
Name:        Serialize1.cfm
Author:      Nate Weiss and Ben Forta
Description: Serialize data into a WDDX packet
Created:     02/01/05
--->

<html>

<head>
 <title>WDDX Demonstration</title>
</head>

<body>

<!--- set the #message# variable to a simple string value --->
<cfset Message="Hello, World!">

<!--- Serialize the #Message# variable into a WDDX Packet --->
<cfwddx action="CFML2WDDX"
        input="#Message#"
        output="MyWDDXPacket">

<!--- Output WDDX packet so we can see what it looks like --->
<!--- (HTMLEditFormat function lets us see tags properly) --->
```

Listing 47.1 (continued)

```
<cfoutput>
 <p><strong>Original Message:</strong> #Message#</p>
 <p><strong>The message was serialized into
           the following WDDX packet:</strong></p>

 #HTMLEditFormat(MyWDDXPacket)#
</cfoutput>

<!--- Save the WDDX packet to a file on the server's drive --->
<cffile action="WRITE"
        file="#ExpandPath('StringPacket.txt')#"
        output="#MyWDDXPacket#">

</body>
</html>
```

NOTE

Because the `MyWDDXPacket` variable contains tags that look like HTML tags, most Web browsers will not display the packet's contents unless each `<and>` character is converted to a `<` or `>` symbol. ColdFusion's `HTMLEditFormat()` function escapes these types of special characters automatically, which is the reason that function is used in Listing 47.1. You can leave out this function if you want, but in that case you must use the browser's View Source command to actually see the packet's contents. Alternatively, you could use the `HTMLCodeFormat()` function, which would cause the browser to display the packet's contents using a fixed-width ("code") font, always on one long line.

Deserializing Your First WDDX Packet

Listing 47.2 shows how to deserialize a WDDX packet. As you can see, the process is very similar to the serialization process; you just use `action="wddx2cfml"` instead of `action="cfml2wddx"` in the `<cfwddx>` tag, and supply the text of the WDDX packet as the tag's `input` attribute.

Whatever is stored in the WDDX packet will become available in the variable you specify in the `output` attribute. In this case, the contents of the packet is the "Hello, World" message from Listing 47.1. So, after the `<cfwddx>` tag, the `Message` variable contains that string and can be displayed in a `<cfoutput>` block just like any other string variable (see Figure 47.2).

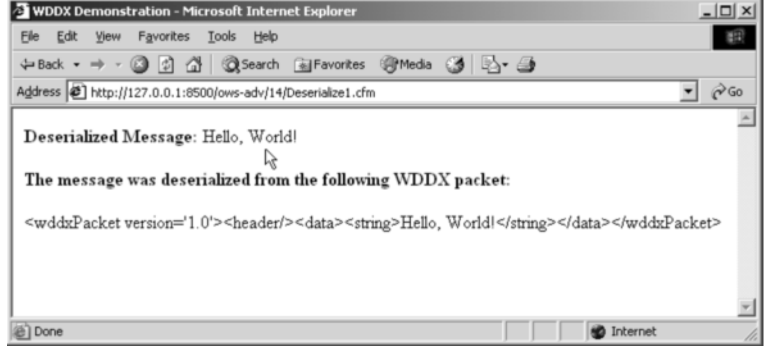


Figure 47.2

You can easily deserialize any WDDX packet with the `<CFWDDX>` tag.

Listing 47.2 `Deserialize1.cfm`—Deserializing the Packet Created with Listing 47.1

```
<!---
Name:        Deserialize1.cfm
Author:      Nate Weiss and Ben Forta
Description: Deserialize data from a WDDX packet
Created:     02/01/05
--->

<html>

<head>
 <title>WDDX Demonstration</title>
</head>

<body>

<!--- Read the WDDX packet from the file on the server's drive --->
<cffile action="READ"
        file="#ExpandPath('StringPacket.txt')#"
        variable="MyWDDXPacket">

<!--- Deserialize the WDDX packet back into native #Message# variable --->
<cfwddx action="WDDX2CFML"
        input="#MyWDDXPacket#"
        output="Message">

<cfoutput>
 <!--- Display the message we retrieved from the WDDX packet --->
 <p><strong>Deserialized Message:</strong> #Message#</p>
 <p><strong>The message was deserialized from
            the following WDDX packet:</strong></p>

 <!--- Output WDDX packet so we can see what it looks like --->
 <!--- (HTMLEditFormat function lets us see tags properly) --->
 #HTMLEditFormat(MyWDDXPacket)#
</cfoutput>

</body>
</html>
```

Serializing and Deserializing Complex Data

Listings 47.1 and 47.2 showed you how to serialize and deserialize a simple string value. Although those listings are a useful demonstration of the `<cfwddx>` tag, the actual result is not that interesting. Those listings simply stored a string value in a file; you could have achieved that result by saving the string to a simple text file with the `<cffile>` tag alone.

Things get a lot more interesting when you use `<cfwddx>` to serialize and deserialize complex data types such as arrays, query recordsets, and structures. In fact, just about any CFML variable can be serialized (and then deserialized) with WDDX, and the `<cfwddx>` tag syntax remains exactly the same.

Listing 47.3 creates a structure called `MyStruct`, fills it with various types of data (including a nested array and a nested query recordset), and serializes it with the `<cfwddx>` tag (see Figure 47.3). The packet is stored in a text file called `StructPacket.txt`.

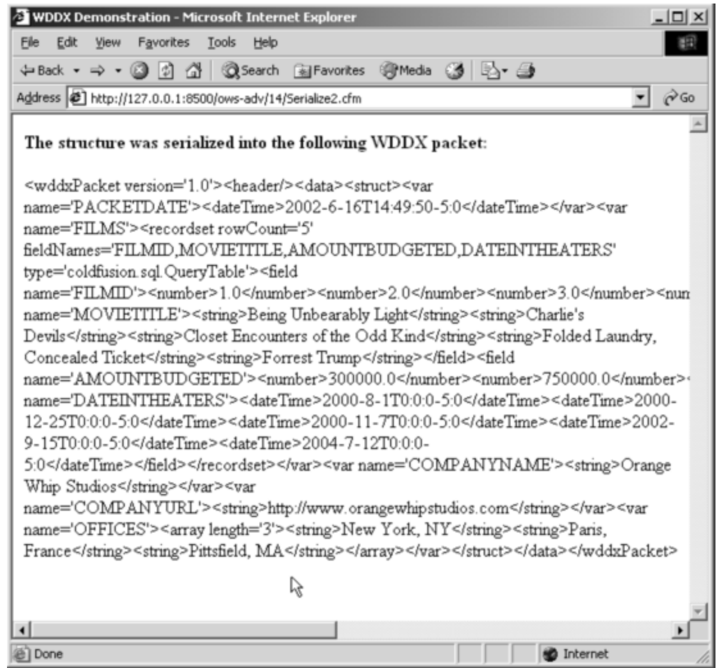


Figure 47.3

Complex values such as structures, recordsets, and arrays can be serialized with `<cfwddx>`.

Listing 47.4 uses `<cfwddx>` to deserialize the packet and then display some of the information that it contained (see Figure 47.4). The output proves that the deserialized `MyStruct` variable holds exactly the same information as it did before the serialization/deserialization process. Even if the `MyStruct` structure contained nested structures that in turn contained other structures, or arrays that contained recordsets, you could still serialize it using the same approach.

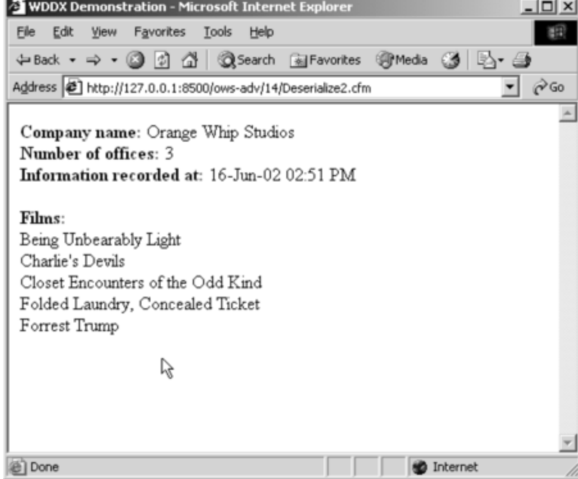


Figure 47.4

After deserialization, the data from a WDDX packet can be used just like any other data.

Listing 47.3 `Serialize2.cfm`—Serializing a Structure that Contains an Array and Recordset

```
<!---
Name:        Serialize2.cfm
Author:      Nate Weiss and Ben Forta
Description: Serialize data into a WDDX packet
Created:     02/01/05
--->

<html>

<head>
 <title>WDDX Demonstration</title>
</head>

<body>

<!--- Run a simple database query to include in the WDDX packet --->
<!--- Limit the query to just 5 rows to keep things simple --->
<cfquery name="filmsquery" datasource="ows" maxrows="5">
 SELECT FilmID, MovieTitle, AmountBudgeted, DateInTheaters
 FROM Films
 ORDER BY MovieTitle
</cfquery>

<!--- Create a structure --->
<cfset MyStruct=StructNew()>
<!--- Add a few simple string values --->
<cfset MyStruct.CompanyName="Orange Whip Studios">
<cfset MyStruct.CompanyURL="http://www.orangewhipstudios.com">
<!--- Add the current date and time --->
<cfset MyStruct.PacketDate=Now()>
<!--- Add the contents of the FilmsQuery query --->
<cfset MyStruct.Films=FilmsQuery>
<!--- Add a simple array --->
<cfset MyStruct.Offices=ArrayNew(1)>
<cfset MyStruct.Offices[1]="New York, NY">
<cfset MyStruct.Offices[2]="Paris, France">
<cfset MyStruct.Offices[3]="Pittsfield, MA">

<!--- Serialize the #MyStruct# structure into a WDDX Packet --->
<cfwddx action="CFML2WDDX"
        input="#MyStruct#"
        output="MyWDDXPacket">

<!--- Output WDDX packet so we can see what it looks like --->
<!--- (HTMLEditFormat function lets us see tags properly) --->
<cfoutput>
 <p><strong>The structure was serialized into the
            following WDDX packet:</strong></p>

 #HTMLEditFormat(MyWDDXPacket)#
</cfoutput>

<!--- Save the WDDX packet to a file on the server's drive --->
<cffile action="WRITE"
        file="#ExpandPath('StructPacket.txt')#"
```

Listing 47.3 (continued)

```
        output="#MyWDDXPacket#">

</body>
</html>
```

Listing 47.4 `Deserialize2.cfm`—Deserializing a Multifaceted Data Structure

```
<!---
Name:        Deserialize2.cfm
Author:      Nate Weiss and Ben Forta
Description: Deserialize data from a WDDX packet
Created:     02/01/05
--->

<html>

<head>
 <title>WDDX Demonstration</title>
</head>

<body>

<!--- Read the WDDX packet from the file on the server's drive --->
<cffile action="READ"
        file="#ExpandPath('StructPacket.txt')#"
        variable="MyWDDXPacket">

<!--- Deserialize the WDDX packet back into native #MyStruct# variable --->
<cfwddx action="WDDX2CFML"
        input="#MyWDDXPacket#"
        output="MyStruct">

<!--- Output various information from the packet, to prove that --->
<!--- the structure contains all of the original information --->
<cfoutput>
 <!--- MyStruct.CompanyName should be a string value --->
 <strong>Company name:</strong>
 #MyStruct.CompanyName#<br>

 <!--- MyStruct.Offices should be an array --->
 <strong>Number of offices:</strong>
 #ArrayLen(MyStruct.Offices)#<br>

 <!--- MyStruct.PacketDate should be a date/time object --->
 <strong>Information recorded at:</strong>
 #DateFormat(MyStruct.PacketDate)# #TimeFormat(MyStruct.PacketDate)#<br>

 <!--- MyStruct.Films should be a query recordset --->
 <p><strong>Films:</strong><br>
 <cfloop query="MyStruct.Films">
  #MovieTitle#<br>
 </cfloop>
</cfoutput>

</body>
</html>
```

It's worth emphasizing that `<cfwddx>` doesn't care what a variable contains when you serialize it. You feed it whatever data you want converted to a packet, and it obliges. Compare this to traditional XML approaches, with which you would normally have to decide on what tag and attribute names you wanted to use, perhaps creating a DTD along the way, and then populate an XML document using DOM-like syntax. Don't get me wrong here. I'm not trying to suggest that WDDX is better than other types of XML. But WDDX is unquestionably easier to use for quick-and-dirty tasks where all you want to do is convert data to XML in a quick and reliable fashion.

Validating Packets with `IsWDDX()`

Sometimes you'll want to deserialize packets that may be coming from some other application or location. If you're unsure whether a WDDX packet is valid, you can use the `IsWDDX()` function to validate it before attempting to deserialize it with the `<cfwddx>` tag.

The `IsWDDX()` function accepts a single argument, which is the string value that you suspect to be a WDDX packet. The function returns `true` or `false`, depending on whether the packet is indeed valid. If the result is false, the packet cannot be deserialized with `<cfwddx>`.

For instance, you could add the following `<cfif>` block to Listing 47.4, after the `<cffile>` tag but before the `<cfwddx>` tag:

```
<!--- Make sure the packet is valid before deserializing it --->
<cfif not IsWDDX(MyWDDXPacket)>
 Sorry, the StructPacket.txt file does not contain a valid WDDX packet.
 <cfabort>
</cfif>
```

Anatomy of a WDDX Packet

Now that you have an idea about how the `<cfwddx>` tag works, let's take a closer look at the WDDX packets themselves. You've already seen WDDX packets as displayed in a browser (Figures 47.1 and 47.3), but packets aren't really meant to be displayed on Web pages. It makes more sense to look at them as if they were data files, kind of like a database or delimited text file.

NOTE

> Really, the principal idea behind WDDX is that the XML for each WDDX packet is created and parsed automatically, so you don't actually ever need to know or understand the anatomy of the packets themselves. That said, I thought you might be curious about the various elements (tags) in the packets. Feel free to skip this section if you want!

Listing 47.5 shows the `StringPacket.txt` file that was created by Listing 47.1. I have added some carriage returns and indention to make the packet easier on the eyes. The whitespace I added doesn't make the packet any less valid, but it makes it easier for us humans to understand.

Listing 47.5 `StringPacketFormattted.txt`—The Simple WDDX Packet Created by Listing 47.1

```
<wddxPacket version='1.0'>
 <header/>
 <data>
```

Listing 47.5 (continued)

```
 <string>Hello, World!</string>
 </data>
</wddxPacket>
```

The entire packet is enclosed between a pair of `<wddxPacket>` tags. As of this writing, the version attribute will always be 1.0. If the WDDX specification changes in the future, the version number will be updated accordingly. This way, before an application attempts to deserialize a packet, it can check the version number to ensure that it knows how to read all the tags in the packet before it starts.

NOTE

For history buffs out there, the first version of WDDX was version 0.9 and was introduced in ColdFusion 4.0. A few minor additions to WDDX were made in the months thereafter, resulting in version 1.0. The main change from 0.9 to 1.0 was the introduction of the `<binary>` element, which allows WDDX packets to contain raw bits of binary data such as images. `<cfwddx>` has been producing version 1.0 packets since ColdFusion 4.5 and continues to do so in the current version.

Within the `<wddxPacket>` block, a `<header>` tag always appears next. The `<header>` tag serves no purpose in WDDX at this time but might come to hold significant information in a future version of WDDX. After the `<header>` tag, a pair of `<data>` tags appear, and all the tags that contain the actual serialized information are placed between them.

In Listing 47.5, a single pair of `<string>` tags is placed between the `<data>` tags. If the data in the packet were a date value instead of a string, a pair of `<dateTime>` tags would appear there instead.

For a glimpse inside a more interesting WDDX packet, take a look at Listing 47.6, which shows the `StructPacket.txt` file generated by my second serialization example (Listing 47.3). The same `<wddxPacket>`, `<header>`, and `<data>` elements are present and will always be present in any valid packet. Not surprisingly, this packet contains quite a few additional elements nested within its `<data>` block.

Again, I've added indention to make the packet more readable on the printed page, but the indention doesn't affect the validity of the packet.

Listing 47.6 `StructPacketFormattted.txt`—The Complex Packet Created by Listing 47.3

```
<wddxPacket version='1.0'>
 <header/>
 <data>
 <struct>
 <var name='PACKETDATE'>
 <dateTime>2002-6-16T14:51:5-5:0</dateTime>
 </var>
 <var name='FILMS'>
 <recordset
 fieldNames='FILMID,MOVIETITLE,AMOUNTBUDGETED,DATEINTHEATERS'
 rowCount='5'>

 <field name='FILMID'>
 <number>1.0</number>
 <number>2.0</number>
```

Listing 47.6 (continued)

```
<number>3.0</number>
<number>18.0</number>
<number>21.0</number>
</field>
<field name='MOVIETITLE'>
<string>Being Unbearably Light</string>
<string>Charlie's Devils</string>
<string>Closet Encounters of the Odd Kind</string>
<string>Folded Laundry, Concealed Ticket</string>
<string>Forrest Trump</string>
</field>
<field name='AMOUNTBUDGETED'>
<number>300000.0</number>
<number>750000.0</number>
<number>350000.0</number>
<number>7000000.0</number>
<number>1.35E8</number>
</field>
<field name='DATEINTHEATERS'>
<dateTime>2000-8-1T0:0:0-5:0</dateTime>
<dateTime>2000-12-25T0:0:0-5:0</dateTime>
<dateTime>2000-11-7T0:0:0-5:0</dateTime>
<dateTime>2002-9-15T0:0:0-5:0</dateTime>
<dateTime>2004-7-12T0:0:0-5:0</dateTime>
</field>

</recordset>
</var>
<var name='COMPANYNAME'>
<string>Orange Whip Studios</string>
</var>
<var name='COMPANYURL'>
<string>http://www.orangewhipstudios.com</string>
</var>
<var name='OFFICES'>
<array length='3'>
<string>New York, NY</string>
<string>Paris, France</string>
<string>Pittsfield, MA</string>
</array>
</var>
</struct>
</data>
</wddxPacket>
```

Tables 47.2 and 47.3 provide a brief explanation of the various XML elements and attributes found in WDDX packets. Basically, the idea is to define the basic types of information that can be stored in packets (see Table 47.2), and then allow these types to be arranged as complex structures that mimic arrays, structures (or associative arrays), or recordsets (see Table 47.3). The result is a system that allows just about any type of information typically tracked by applications, regardless of the programming language used, to be represented cleanly and clearly.

Table 47.2 Basic Data Elements Found in WDDX Packets

ELEMENT	DESCRIPTION
`<string>`	Surrounds any string value in the packet. Within the `<string>`, any extended or nonprintable characters can be represented by a `<char code=''>` element, where the `code` attribute is the UTF-8 number for the character, expressed as a two-digit hex value, such as `0C` for a form feed.
`<number>`	Surrounds any numeric value. Note that WDDX does not get into issues regarding the range or precision of numbers. That is, there is no special consideration given to whether a number is an integer, a floating-point number, a `single`, a `double`, or the like.
`<dateTime>`	Surrounds any date/time value. In WDDX, dates always include a time portion as well, and may optionally include time zone information. Dates must be formatted according to the ISO8601 standard, as in `2006-12-25T09:05:32-5:0` to represent 9:05 a.m. (Eastern Standard Time) on December 25, 2006.
`<boolean>`	Represents a Boolean (true/false) value. The `<boolean>` element will always contain a `value='true'` or `value='false'` attribute accordingly.
`<null>`	Represents a null value, such as a `NULL` value retrieved from a database table.
`<binary>`	Represents a binary value, such as the contents of an image or other nontextual information. At this time, `<binary>` elements will always contain an `encoding='base64'` attribute. Between the `<binary>` tags, the actual binary data will appear, having first been converted to the Base 64 format.

NOTE

In general, you never have to actually type the tags in Table 47.2 or Table 47.3. They are generated automatically for you by the `<cfwddx>` tag (or whatever WDDX serializer you are using).

Table 47.3 Container Elements Found in WDDX Packets

ELEMENT	DESCRIPTION
`<wddxPacket>`	Required. Surrounds the entire WDDX packet. At this time, always contains a `version='1.0'` attribute.
`<header>`	Required. Reserved for future use.
`<data>`	Required. Surrounds the actual data in the packet.
`<array>`	Represents an array. The `<array>` element always contains a `length` attribute that indicates how many items are in the array. Then the actual items in the array are included between the opening and closing `<array>` tags, with each item contained within its own `<string>`, `<number>`, `<dateTime>`, or whatever other element is appropriate.
`<recordset>`	Represents a recordset, such as a query object returned by `<cfquery>`. Within the `<recordset>` element, a `<field>` element is used to represent each column in the recordset.

Table 47.3 (CONTINUED)

ELEMENT	DESCRIPTION
<field>	Used only as a child of the <recordset> element. Represents a single column within the recordset. Within the <field> element, each row of data is represented by a <string>, <number>, or whatever element is appropriate.
<struct>	Represents a CFML structure (or whatever the corresponding data type is called in other programming languages). Within the <struct> element, a <var> element is used to represent each individual value in the structure.
<var>	Used only as a child of the <struct> element. Represents a single name/value pair within the structure. The name of the value is provided with the name attribute; the actual value appears between the opening and closing <struct> tags, surrounded by a <string>, <number>, or whatever element is appropriate.

NOTE

The WDDX DTD says it is legal for the <header> element to contain a single comment attribute that could hold some kind of human-readable description of what the packet contains. However, <cfwddx> provides no direct way to insert or read such a comment.

Using WDDX Packets to Store Information in Files

In the listings you've seen so far in this chapter (especially Listing 47.3), you have learned how easy it is to convert any variable or data structure to XML with the <cfwddx> tag. Because WDDX packets are so easy to create, and contain just about any type of information, and because the packet itself is just simple text (as is any XML document), ColdFusion developers often use WDDX as a way to store complex information in places where it's usually only possible to store text.

The next sections discuss storing WDDX packets in text files, client variables, and string columns in database tables. These are just examples; you can apply the basic idea in other ways as well. Anytime you want to store any kind of information in a place that normally can store only text, consider using WDDX to get the data into a simple text format. It's fast, simple, proven, and lightweight. Best of all, it's supported not only by ColdFusion but by ASP, .NET, Java, PHP, Perl, and all the other environments listed in the earlier section "Tools and Languages Supported by WDDX."

About Storing Packets in Files

You have already seen how you can use <cffile> and <cfwddx> to create WDDX packets, store them in files, and deserialize the packets back into native ColdFusion variables. There are many situations in which you might want to save information in such files.

For instance, you might want to build an application whose behavior or appearance can be tweaked with various settings. Let's say you are building an intranet for the fictitious Orange Whip Studios

company, and you want certain aspects of the application to be flexible. One setting will be for the background color of the application's home page, another setting will be for the name of the company, and so on. This way, if the desired color or the name of the company changes next month, you simply change the setting. Conceptually, this is equivalent to the Options or Preferences dialog box found in many Windows or Mac applications.

Building a Simple WDDX Function Library

The serialization and deserialization examples you've seen so far in this chapter use the `<cfwddx>` and `<cffile>` tags to read or store WDDX packets on the server's drive. As you've seen, it's really easy. We can make it even easier by creating a few simple user-defined functions.

The UDF function library called `WDDXFunctions.cfm` (included with the listings for this chapter) contains a few simple functions for reading and writing WDDX packets on the server's drive. These functions are just shorthand for using the `<cfwddx>`, `<cfhttp>`, and `<cfwddx>` tags. The library also contains similar functions for reading and writing packets in the `CLIENT` scope, and for reading packets from other Web servers using HTTP.

Table 47.4 shows the functions provided by this simple library (you'll see the code to create the functions in a moment).

Table 47.4 Functions in the `WDDXFunctions.cfm` UDF Library

FUNCTION	DESCRIPTION
`WDDXFileWrite(file, value)`	Stores the `value` as a WDDX packet at the location on the server's drive indicated by `file`. The value can be any structure, recordset, or other serializable value.
`WDDXFileRead(file)`	Reads the WDDX packet at the location on the server's drive indicated by `file`, deserializes the packet, and returns the deserialized data.
`WDDXHttpGet(url)`	Similar to `WDDXFileRead()`, except that the WDDX packet is read from another Web server using HTTP.
`WDDXClientWrite(name, value)`	Like `WDDXFileWrite()`, except that the WDDX packet is stored as a client variable with the name specified by the `name` argument.
`WDDXClientRead(name)`	Like `WDDXFileRead()`, except that the WDDX packet is read from the `CLIENT` scope. If the variable does not exist or does not contain a valid packet, the function returns an empty string.

Listing 47.7 shows the code used to create the functions listed in Table 47.4. As you can see, the code in each of the individual `<cffunction>` blocks is fairly simple, and very similar to the code used earlier in Listings 47.1 and 47.2. Wrapping them into functions makes them even easier to use.

Listing 47.7 `WDDXFunctions.cfm`—A UDF Library for Reading and Writing WDDX Packets

```
<!---
Name:        WDDXFunctions.cfm
Author:      Nate Weiss and Ben Forta
Description: A general-purpose UDF library to make using WDDX easier
Created:     02/01/05
--->


<!---
Function to write any value to the
server's drive as a WDDX packet.
--->
<cffunction name="WDDXFileWrite"
            returntype="void">
  <!--- Required arguments --->
  <cfargument name="File"
              type="string"
              required="Yes">
  <cfargument name="Value"
              type="any"
              required="Yes">

  <!--- This variable is for this function's use only --->
  <cfset var WddxPacket="">

  <!--- Convert the value to a WDDX packet --->
  <cfwddx action="CFML2WDDX"
          input="#ARGUMENTS.Value#"
          output="WddxPacket">

  <!--- Save the WDDX packet to the server's drive --->
  <cffile action="WRITE"
          file="#ARGUMENTS.File#"
          output="#WddxPacket#">

</cffunction>


<!---
Function to read a value from a WDDX packet on
the server's drive. Returns the value in the packet,
after deserialization.
--->
<cffunction name="WDDXFileRead"
            returntype="any">
  <!--- Required argument --->
  <cfargument name="File"
              type="string"
              required="Yes">

  <!--- These variables are for this function's use only --->
  <cfset var Result="">
  <cfset var WddxPacket="">

  <!--- Read the WDDX packet from the server's drive --->
```

Listing 47.7 (continued)

```
  <cffile action="READ"
          file="#ARGUMENTS.File#"
          variable="WddxPacket">

  <!--- Deserialize the value in the WDDX packet --->
  <cfwddx action="WDDX2CFML"
          input="#WddxPacket#"
          output="Result">

  <!--- Return the result --->
  <cfreturn Result>
</cffunction>


<!---
Function to read a value from a WDDX packet
on a Web server. Returns the value in the
packet, after deserialization.
--->
<cffunction name="WDDXHttpGet"
            returntype="any">
  <!--- Required argument --->
  <cfargument name="URL"
              type="string"
              required="Yes">

  <!--- The Result variable is for this function's use only --->
  <cfset var Result="">

  <!--- Fetch the WDDX packet over the wire --->
  <cfhttp method="GET"
          url="#ARGUMENTS.URL#">

  <!--- Deserialize the value in the WDDX packet --->
  <cfwddx action="WDDX2CFML"
          input="#CFHTTP.FileContent#"
          output="Result">

  <!--- Return the result --->
  <cfreturn Result>
</cffunction>



<!---
Function to write any value to a client variable
as a WDDX packet.
--->
<cffunction name="WDDXClientWrite"
            returntype="void">
  <!--- Required arguments --->
  <cfargument name="Name"
              type="string"
              required="Yes">
  <cfargument name="Value"
```

Listing 47.7 (continued)

```
              type="any"
              required="Yes">

  <!--- This variable is for this function's use only --->
  <cfset var WddxPacket="">

  <!--- Convert the value to a WDDX packet --->
  <cfwddx action="CFML2WDDX"
          input="#ARGUMENTS.Value#"
          output="WddxPacket">

  <!--- Save the packet as a CLIENT variable --->
  <cfset CLIENT[ARGUMENTS.Name]=WddxPacket>
</cffunction>


<!---
Function to retrieve a value stored with
WDDXClientWrite().
--->
<cffunction name="WDDXClientRead"
            returntype="any">
  <!--- Required argument --->
  <cfargument name="Name"
              type="string"
              required="Yes">

  <!--- These variables are for this function's use only --->
  <cfset var Result="">
  <cfset var WddxPacket="">

  <!--- If the client variable exists and is valid --->
  <cfif IsDefined("CLIENT.#ARGUMENTS.Name#")>
    <cfif IsWddx(CLIENT[ARGUMENTS.Name])>
      <!--- Deserialize the value in the WDDX packet --->
      <cfwddx action="WDDX2CFML"
              input="#CLIENT[ARGUMENTS.Name]#"
              output="Result">

    </cfif>
  </cfif>

  <!--- Return the result --->
  <cfreturn Result>
</cffunction>
```

Storing Application Settings as a WDDX Packet

The new UDF library can be put to work right away. As mentioned, the example for this section will be an application that has a few settings for controlling things like the background color, company name, and so on.

Listing 47.8 is a simple `Application.cfm` file, which you can modify to suit your needs. The idea is to check and see if the application is being accessed for the first time (that is, since the ColdFusion

server has been restarted). If so, the application's settings are read in from a file called `AppSettings.xml` and stored as an application variable called `APPLICATION.AppSettings`. After that's done, any of the information in the packet can be referred to as `APPLICATION.AppSettings.AppTitle`, `APPLICATION.AppSettings.HTML.PageColor`, and so on.

Listing 47.8 `Application.cfm`—Reading Application Settings from a WDDX Packet on Disk

```
<!---
Name:        Application.cfm
Author:      Nate Weiss and Ben Forta
Description: Executes on every request
Created:     02/01/05
--->

<!--- Define the application --->
<cfapplication name="OrangeWhipIntranet"
               clientmanagement="Yes">

<!--- Include the WDDXFunctions UDF library --->
<cfinclude template="WDDXFunctions.cfm">

<!--- If the application has not been initialized yet, or if the --->
<!--- user is currently trying to change the application's settings... --->
<cfif (NOT IsDefined("APPLICATION.Initialized"))
  OR IsDefined("FORM.IsSavingSettings")>

  <!--- Initialize the application --->
  <cftry>
    <!--- Location of AppSettings.xml file --->
    <cfset SettingsFile = GetDirectoryFromPath(GetCurrentTemplatePath())
      & "/AppSettings.xml">

    <!--- Attempt to initialize application.
          If this fails for any reason, --->
    <!--- the <cfcatch> block will display the Settings form page. --->
    <cfset APPLICATION.AppSettings = WddxFileRead(SettingsFile)>

    <!--- Remember that the application has been initialized, so that --->
    <!--- this whole section will be skipped until server is restarted --->
    <cfset APPLICATION.Initialized = True>

    <!--- Display the Settings form page if any exceptions are thrown --->
    <cfcatch type="Any">
      <cfinclude template="AppSettingsForm.cfm">
      <cfabort>
    </cfcatch>
  </cftry>
</cfif>
```

First, the WDDX function library from Listing 47.7 is included using the `<cfinclude>` tag. Then a simple `<cfif>` test is used to check to see if the application's settings have already been read from the server's drive. If they have not, a variable called `SettingsFile` is created that holds the location of the `AppSettings.xml` file (the `GetDirectoryFromPath()` and `GetCurrentTemplatePath()` functions

are used to indicate that the file should be stored in the same folder as the `Application.cfm` file itself).

Next, the `WddxFileRead()` function from Listing 47.7 is used to read the WDDX packet stored in the `AppSettings.xml` file, deserialize it, and save the data from the packet in the `APPLICATION.AppSettings` variable. Finally, the `APPLICATION.Initialized` variable is set to `True` to indicate that the application has been initialized. This step will cause the entire `<cfif>` block in this listing to be skipped for all subsequent page executions (until the server is restarted), which means that this code adds virtually no overhead to the application as a whole.

If, for some reason, there is a problem reading and deserializing the application settings (perhaps the `AppSettings.xml` file does not exist, or someone has edited it in such a way that it is no longer valid), the `<cfcatch>` block will execute. The `<cfcatch>` block includes a file called `AppSettingsForm.cfm`, which displays a form for creating the application's settings, and then halts further execution. In other words, if the settings file is missing or invalid, the application will force the user to provide new application settings before any pages can be accessed.

NOTE

> You could ship or deploy your ColdFusion application with the `AppSettings.xml` file deliberately missing. The first time the application is used, it will demand that the first user (presumably the person installing or deploying the application) provide the correct settings.

Listing 47.9 shows the code to create the HTML form for editing the application's settings (see Figure 47.5).

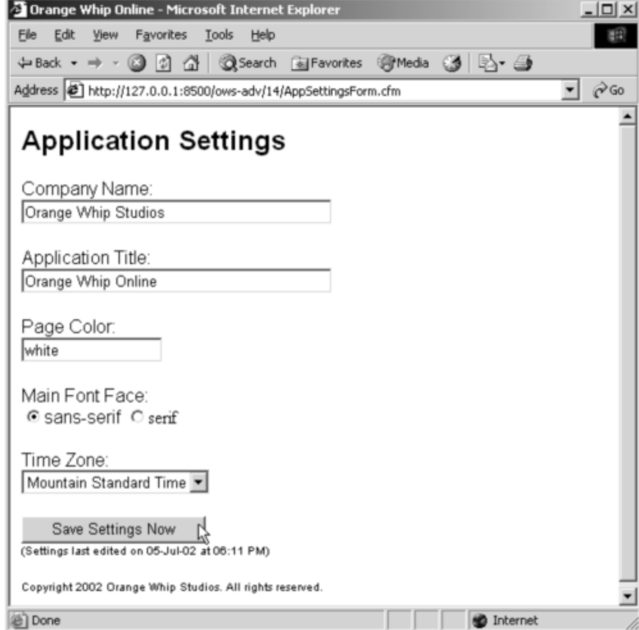


Figure 47.5

The application's settings can be edited with this HTML form.

Listing 47.9 `AppSettingsForm.cfm`—Reading Application Settings from a WDDX Packet on Disk

```
<!---
Name:        AppSettingsForm.cfm
Author:      Nate Weiss and Ben Forta
Description: Provides a form for editing this application's settings
Created:     02/01/05
--->

<!--- Location of AppSettings.xml file --->
<cfset ThisFolder=GetDirectoryFromPath(GetCurrentTemplatePath())>
<cfset SettingsFile=ThisFolder & "AppSettings.xml">

<!--- Read time zone recordset from WDDX packet on the server's drive --->
<cfset TimeZones=WDDXFileRead(ThisFolder & "TimeZoneRecordsetPacket.xml")>

<!--- If the form is being submitted --->
<cfif IsDefined("FORM.IsSavingSettings")>
  <!--- Make new structure called Settings, which contains data from form --->
  <cfset Settings.CompanyName=FORM.CompanyName>
  <cfset Settings.AppTitle=FORM.AppTitle>
  <cfset Settings.HTML.PageColor=FORM.PageColor>
  <cfset Settings.HTML.FontFace=FORM.FontFace>

  <!--- Use in-memory query to get information about selected time zone --->
  <cfquery dbtype="query" name="SelectedTimeZone">
    SELECT * FROM TimeZones
    WHERE Code='#FORM.TimeZoneCode#'
  </cfquery>

  <!--- Add information about the selected time zone --->
  <cfset Settings.TimeZone.Code=SelectedTimeZone.Code>
  <cfset Settings.TimeZone.Offset=SelectedTimeZone.Offset>
  <cfset Settings.TimeZone.Description=SelectedTimeZone.Description>

  <!--- Remember when these edits were made --->
  <cfset Settings.SettingsLastEdited=Now()>

  <!--- Save the settings as a WDDX packet on the server's drive --->
  <cfset WddxFileWrite(SettingsFile, Settings)>

  <!--- Clear the application's Initialized flag --->
  <!--- This will cause settings to be re-read on the next page request --->
  <cfset StructDelete(APPLICATION, "Initialized")>

  <!--- Reload whatever page was requested --->
  <cflocation url="#CGI.SCRIPT_NAME#?#CGI.QUERY_STRING#">
</cfif>



<!--- Read the settings from the WDDX Packet on the server's drive --->
<cfset AppSettings=WDDXFileRead(SettingsFile)>

<!--- The application settings should include the following --->
<!--- These default values will be used if the settings file is missing --->
<cfparam name="AppSettings.CompanyName" type="string" default="">
```

Listing 47.9 (continued)

```
<cfparam name="AppSettings.AppTitle" type="string" default="">
<cfparam name="AppSettings.HTML.PageColor" type="string" default="white">
<cfparam name="AppSettings.HTML.FontFace" type="string" default="sans-serif">
<cfparam name="AppSettings.TimeZone.Code" type="string" default="EST">


<html>
<head>
 <title>Application Settings</title>
</head>

<body>
<h2>Application Settings</h2>


<!--- Simple form to gather application settings --->
<cfform action="#CGI.SCRIPT_NAME#"
        method="POST">

  <!--- Hidden field for detecting when the form is being submitted --->
  <cfinput type="Hidden"
           name="IsSavingSettings"
           value="Yes">

  <!--- Text field for company name --->
  <p>Company Name:<br>
  <cfinput name="CompanyName"
           value="#AppSettings.CompanyName#"
           size="40"
           required="Yes"
           message="Please do not leave the company name blank.">

  <!--- Text field for application title --->
  <p>Application Title:<br>
  <cfinput name="AppTitle"
           value="#AppSettings.AppTitle#"
           size="40"
           required="Yes"
           message="Please do not leave the application title blank.">

  <!--- Text field for page color --->
  <p>Page Color:<br>
  <CFinput name="PageColor"
           value="#AppSettings.HTML.PageColor#"
           size="15"
           required="Yes"
           message="Please do not leave the page color blank.">

  <!--- Radio buttons for font face --->
  <p>Main Font Face:<br>
  <cfif AppSettings.HTML.FontFace EQ "sans-serif">
   <cfset checked="yes">
  <cfelse>
   <cfset checked="no">
  </cfif>
```

Listing 47.9 (continued)

```
      <cfinput type="Radio"
               name="FontFace"
               value="sans-serif"
               checked="#checked#">
      <font face="sans-serif">sans-serif</font>
      <cfif AppSettings.HTML.FontFace EQ "serif">
       <cfset checked="yes">
      <cfelse>
       <cfset checked="no">
      </cfif>
      <cfinput type="Radio"
               name="FontFace"
               value="serif"
               checked="#checked#">
      <font face="serif">serif</font>

      <p>Time Zone:<br>
      <cfselect name="TimeZoneCode"
                selected="#AppSettings.TimeZone.Code#"
                query="TimeZones"
                value="Code"
                display="Description"/>

      <!--- Submit button to save settings --->
      <p>
      <cfinput type="Submit"
               name="submit"
               value="Save Settings Now"><br>

      <!--- Display when the settings were last edited, if available --->
      <cfif IsDefined("AppSettings.SettingsLastEdited")>
        <cfoutput>
          <font size="1">
            (Settings last edited on #DateFormat(AppSettings.SettingsLastEdited)#
            at #TimeFormat(AppSettings.SettingsLastEdited)#)
          </font>
        </cfoutput>
      </cfif>

    </cfform>

    </body>
    </html>
```

First, the `WDDXFunction.cfm` library is included, and the location of the `AppSettings.xml` file is determined (similar to Listing 47.8). Next, information about time zones is read in from a different WDDX packet and stored in the `TimeZoneRecordsetPacket.xml` file (skip ahead to Listing 47.11 if you want to have a look at this file). The packet contains a recordset, which means that after this line executes, the `TimeZones` variable can be used like the results of a `<CFQUERY>` tag.

The bulk of the work is done in the large `<cfif>` block that follows, which executes when the user submits the form (refer to Figure 47.5). Inside the `<cfif>`, a new structure called `Settings` is created and filled with the information being submitted by the user, by referring to various `FORM` variables.

Note that three pieces of information are stored with respect to the chosen time zone (a three-letter code, the numeric offset from Greenwich Mean Time, and a description). Only the code for the time zone is submitted by the form, so an in-memory query is used to get the corresponding offset and description for the selected time zone. It's pretty neat that the `TimeZones` recordset can be queried directly like this, even though the recordset came from a WDDX packet on the server's drive rather than from a database.

Once the `Settings` structure has been filled with the appropriate information, the `WddxFileWrite()` function from Listing 47.7 is used to save the structure to disk as a WDDX packet in the `AppSettings.xml` file. Then the `StructKeyDelete()` function is used to remove the `Initialized` flag (if it exists) from the `APPLICATION` scope. This will cause the application to no longer consider itself initialized, which in turn means that the settings will be read in afresh from disk the next time one of the application's pages is visited.

Listing 47.10 shows the `AppSettings.xml` file created when the form shown earlier in Figure 47.5 is submitted. Again, I have added some indention and whitespace to make the packet easier for us humans to read, but this doesn't affect the validity of the packet.

Listing 47.10 `AppSettings.xml`—WDDX Packet Containing Settings for the Application

```
<wddxPacket version='1.0'>
 <header/>
 <data>
 <struct>

 <var name='HTML'>
 <struct>
 <var name='PAGECOLOR'><string>white</string></var>
 <var name='FONTFACE'><string>sans-serif</string></var>
 </struct>
 </var>

 <var name='APPTITLE'>
 <string>Orange Whip Online</string>
 </var>

 <var name='TIMEZONE'>
 <struct>
 <var name='OFFSET'><number>-7.0</number></var>
 <var name='CODE'><string>MST</string></var>
 <var name='DESCRIPTION'><string>Mountain Standard Time</string></var>
 </struct>
 </var>

 <var name='COMPANYNAME'>
 <string>Orange Whip Studios</string>
 </var>

 <var name='SETTINGSLASTEDITED'>
 <dateTime>2002-7-5T18:11:26-5:0</dateTime>
 </var>

 </struct>
 </data>
</wddxPacket>
```

NOTE

Some developers prefer to use a file name extension of `wddx` (instead of `xml`) for WDDX packets, to emphasize that the XML in the file uses the WDDX vocabulary. Others prefer the `xml` extension to emphasize that WDDX is actually XML under the hood. The truth is, the file name extension doesn't matter much; use whatever extension makes sense to you.

Listing 47.11 shows the WDDX packet that contains the time zone information used by Listing 47.9. I created this packet by hand, but you could easily put together a ColdFusion page that creates the packet programmatically with the `<cfwddx>` tag.

Listing 47.11 `TimeZoneRecordsetPacket.xml`—WDDX Packet with a Recordset About U.S. Time Zones

```
<wddxPacket version='1.0'>
 <header/>
 <data>

 <recordset
 rowCount='4'
 fieldNames='CODE,OFFSET,DESCRIPTION'>

 <field name='CODE'>
 <string>EST</string>
 <string>CST</string>
 <string>MST</string>
 <string>PST</string>
 </field>

 <field name='OFFSET'>
 <number>-5.0</number>
 <number>-6.0</number>
 <number>-7.0</number>
 <number>-8.0</number>
 </field>

 <field name='DESCRIPTION'>
 <string>Eastern Standard Time</string>
 <string>Central Standard Time</string>
 <string>Mountain Standard Time</string>
 <string>Pacific Standard Time</string>
 </field>

 </recordset>

 </data>
</wddxPacket>
```

Note how much easier it is to ship this packet file with your application, rather than worrying about creating a database table, a corresponding data source, and so on. As you saw in Listing 47.9, you can use ColdFusion's in-memory querying feature (also known as Query of Queries) with this information, which means that the time zone data can still be queried, sorted, and joined against other tables. Since this data has only a minor role in the application, and because it is used in an essentially read-only fashion (it's unlikely that Listing 47.11 will need to be edited often, if at all), it makes a lot of sense to just store it in a WDDX packet. This is especially true if you're building a simple application that doesn't need a full-blown database in the first place.

NOTE

In practice, you would have information about all time zones in this packet, not just for the United States. I'm just trying to keep the example listing short.

Naturally, now that the application's settings have been established, you can access the settings by referring to the `APPLICATION.AppSettings` structure within any of the application's pages. As a quick example, Listing 47.12 shows how some of the settings could be displayed in the application's home page (Figure 47.6).

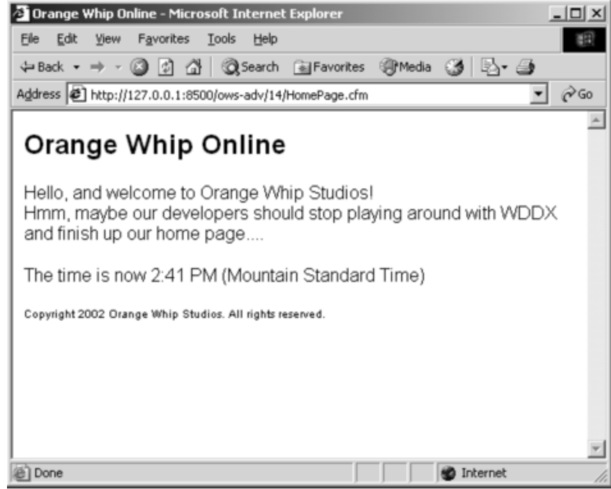


Figure 47.6

Once in place, the application's settings are straightforward.

Listing 47.12 `HomePage.cfm`—WDDX Packet Containing a Recordset

```
<!---
Name:        HomePage.cfm
Author:      Nate Weiss and Ben Forta
Description: Use AppSettings.xml settings
Created:     02/01/05
--->

<!--- Begin HTML page, incorporating some of the application settings --->
<!--- (you could <CFINCLUDE> a seperate Header.cfm page here instead) --->
<cfoutput>
  <!doctype html public "-//w3c//dtd html 3.2 final//en">
  <html>
  <head>
  <title>#APPLICATION.AppSettings.AppTitle#</title>
  </head>
  <body bgcolor="#application.appsettings.html.pagecolor#">
  <font face="#application.appsettings.html.fontface#">
</cfoutput>


<!--- Normal page content could go here --->
<cfoutput>
```

Listing 47.12 (continued)

```
    <h2>#APPLICATION.AppSettings.AppTitle#</h2>
    Hello, and welcome to #APPLICATION.AppSettings.CompanyName#!<br>
    Hmm, maybe our developers should stop playing around with WDDX
    and finish up our home page....<br>

    <p>The time is now #TimeFormat(Now(), "h:mm tt")#
    (#APPLICATION.AppSettings.TimeZone.Description#)<br>
  </cfoutput>


  <!--- Footer area at bottom of page --->
  <!--- (you could <cfinclude> a seperate Footer.cfm page here instead) --->
  <cfoutput>
    <font size="1">
    <p>Copyright #Year(Now())# #APPLICATION.AppSettings.CompanyName#.
    All rights reserved.<br>
    </font>
  </cfoutput>
```

What Have We Learned?

This simple example has shown how easy it is to store any sort of ad-hoc settings or other data in files on the server's drive, using WDDX as the storage format. You could do the same thing using a database, `.ini` files, or your own XML vocabulary. But WDDX makes it particularly easy.

There are other advantages, too. With the WDDX approach, new information can be added to the files at any time without having to worry about re-declaring the format or structure of the files. If you were using a database to store these settings, you might need to change the structure of your tables depending on the type of information you wanted to store. You'd have to make similar structural changes if you were using your own XML vocabulary. And `.ini` files, though simple, aren't particularly good at storing complex information like recordsets, structures, or arrays.

This isn't to say that WDDX is always the best solution to a given problem, or that you should abandon databases or other XML vocabularies. Databases have their own sets of advantages and disadvantages, as do custom XML schemas and vocabularies. But WDDX is a very useful tool that can make short work of many everyday tasks.

Other Places to Store WDDX Packets

So far, the examples in this chapter use text files as a place to store WDDX packets. But the great thing about WDDX packets (okay, one of the great things) is that they can be stored anywhere ordinary text can be stored.

Storing Packets as Client Variables

One useful place to store WDDX packets is in ColdFusion's built-in `CLIENT` scope.. As you probably already know, once you store a value in the `CLIENT` scope, that value will remain associated with the client (basically, the browser machine). Client variables are nifty because they are stored on the

server side (by default), yet they follow each of your users around as they use your pages over time. In many respects, client variables are superior to session variables, because they survive between server restarts and can be shared by multiple servers in a cluster.

One of the limitations of the `CLIENT` scope, however, is that it can be used to store only what ColdFusion considers to be simple values (strings, dates, numbers, and Booleans)—because the underlying storage mechanism is only capable of storing simple strings. ColdFusion won't let you store a structure, array, or query object in the client scope because it can't be expressed as a string.

Of course, you can just convert the structure, array, or recordset to a WDDX packet and then store the packet in the `CLIENT` scope. Because the packet is made up of ordinary text, ColdFusion won't mind storing it as a client variable. When you want to use the value, you can just read the WDDX packet back from the client variable and deserialize it.

For instance, suppose you have a structure called `UserSettings` that holds information specific to each user. You can easily store the structure as a client variable using this code:

```
<cfwddx action="CFML2WDDX"
        input="#UserSettings#"
        output="CLIENT.MySettings">
```

Later, you could read the values back from the `CLIENT` scope using code similar to the following. If the `CLIENT.MySettings` variable doesn't exist or doesn't contain a valid packet, a new and empty structure is created with `StructNew()`.

```
<cfif IsDefined("CLIENT.MySettings") and IsWDDX(Client.MySettings)>
 <cfwddx action="WDDX2CFML"
         input="#CLIENT.MySettings#"
         output="UserSettings">
<cfelse>
 <cfset UserSettings=StructNew()>
</cfif>
```

Alternatively, you could use the `WDDXClientRead()` and `WDDXClientWrite()` functions from Listing 47.7 (refer to Table 47.4). Using the functions, you would save the `UserSettings` structure like this:

```
<cfset WDDXClientWrite("MySettings", UserSettings)>
```

The structure could later be retrieved like this:

```
<cfset UserSettings=WDDXClientRead("MySettings")>
```

NOTE

If you are using the Registry to store your client variables, it's possible that a very large WDDX packet would be too large to store as a client variable.

Storing Packets in Databases

I've explained the advantages to storing user-specific information as WDDX packets in client variables. Depending on how you have configured ColdFusion, your client variables are probably being

stored in a database, which means that the special database tables added by ColdFusion to your database are being used to store the packets.

You can also store WDDX packets in your own database tables. Just create a text, memo, or varchar type of column in the appropriate table. Serialize whatever data you want to store using `<cfwddx>`, and store the resulting WDDX packet using ordinary SQL `INSERT` or `UPDATE` syntax. To retrieve the data, just get the packet from the database using a `SELECT` query, and deserialize the packet.

All that said, many databases and database drivers were not necessarily designed for selecting and updating large amounts of text on a high-volume basis, so this type of solution may not scale particularly well. A related idea would be to save the WDDX packet as a separate file on the server's drive, using the database primary key as the file name.

TIP

You could easily create convenience functions (similar to the `WDDXFileRead()` and `WDDXFileWrite()` functions from Listing 47.7) that include the `<cfquery>` code needed to move the packets in and out of your database.

Exchanging WDDX Packets Among Web Pages

If you've done any work with ColdFusion's `<cfhttp>` tag, you know that you can use it to fetch Web pages from any Web server on the Internet. Essentially, the `<cfhttp>` tag pretends to be a Web browser, supplying any parameters that would normally be supplied by form input, cookies, or CGI variables. ColdFusion developers already use this tag to have their applications automatically visit other Web pages programmatically.

For instance, a ColdFusion application might need to know the current temperature. By using `<cfhttp>` to fetch a page that includes the current temperature—perhaps the "current conditions" page of the local airport's Web site—the application can obtain a document that has the necessary information in it. Then, using ColdFusion's string manipulation functions or some regular expressions, the application can parse through the page's source code and extract the few characters that represent the temperature.

➜ See Chapter 67, "Using Server-Side HTTP and FTP," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 3: Advanced Application Development*, for a complete discussion of using the `<cfhttp>` tag to fetch Web pages from other Web servers on the Internet, or from servers on your intranet or extranet.

The Concept of a Back-End Web Page

Okay, now imagine taking things a step farther. What if the airport's Web site has a special page that isn't meant to be looked at, but rather is meant only to supply information to other systems? That is, instead of including pictures, links, table and font tags, explanatory text, and so on, what if all the page contains is a WDDX packet with the temperature? Maybe the packet includes other information as well, such as the barometric pressure, runway conditions, and so on.

In that case, any ColdFusion application could use the `<cfhttp>` tag to pick up this packet and then use `<cfwddx>` to extract all the information from the packet into local variables. Just two lines of

CFML code later, the application has the information it needs. You can imagine that other airports around the world might set up the same type of back-end Web pages to report the current conditions. The airports might even use these pages to get information about each other's current conditions to be able to tell customers what the weather is like at their destinations.

Suddenly, the airport's Web site is no longer just supplying information to people who happen to visit the Web site and click the "current conditions" page. It's now part of an ambitious information and automation network. No expensive communications channels were set up, and no complicated integration work was done. By using the infrastructures already in place—namely, the airport's Web server and Internet connection—the airport can transform itself into a source of raw data for any application that knows how to fetch a Web page and deserialize a WDDX packet.

Back-End WDDX Pages vs. Web Services

If you're familiar with SOAP, XML-RPC, or the general concept of Web Services, you will recognize that my description of a back-end page is a very similar concept. Making back-end Web pages with WDDX as discussed in this section is a sort of roll-your-own approach to putting together pages that behave like formalized Web Services. You might prefer to just go ahead and adopt the official Web Services frameworks by using CFCs to create services, and `<cfinvoke>` to use services, as discussed in Chapter 68, "Creating and Consuming Web Services," in Vol. 3, *Advanced Application Development*.

That said, here are some reasons why you might want to use a roll-your-own approach using WDDX rather than formalized Web services:

- Perhaps you need to integrate legacy applications or systems that don't support Web services but that do support COM or Java. Since WDDX support is provided for COM and Java, you're all set.
- Perhaps you like the idea of being able to very easily understand every aspect of what's going on. WDDX is simple and intuitive.
- Perhaps you don't think the various Web Services frameworks are mature and proven enough for your particular needs. WDDX doesn't tie you to .NET, J2EE, or anything else.

Indeed, homegrown WDDX-enabled back-end Web pages (I often call them "robot" pages) make up many of the examples for the remainder of this chapter. I use them as examples because they are clear, and because they illustrate how easy it is to get different applications working together. As you read on, just keep in mind that any of the WDDX-related code and techniques used for the back-end page scenario (where packets are exchanged over the Web via HTTP) are just as relevant when you're using WDDX to exchange or save packets via other delivery or storage mechanisms, such as files, client variables, databases, or even email messages.

Creating a Back-End Web Page

Take a look at the `FilmsRobot1.cfm` template shown in Listing 47.13. This page selects information about films from the database and outputs the query results as a WDDX packet. It supports a few

URL parameters to control which films are selected, and how much information about each film is included in the packet.

Listing 47.13 `FilmsRobot1.cfm`—A Back-End Web Page That Exposes Film Data as WDDX Packets

```
<!---
Name:        FilmsRobot1.cfm
Author:      Nate Weiss and Ben Forta
Description: Creates a back-end web page that
             supplies data about films
Created:     02/01/05
--->

<!--- URL Parameters to control what film data the page responds with --->
<cfparam name="URL.FilmID"
         type="numeric"
         default="0">
<cfparam name="URL.Details"
         type="boolean"
         default="No">
<cfparam name="URL.Keywords"
         type="string"
         default="">

<!--- Execute a database query to select film information from database --->
<cfquery name="FilmsQuery"
         datasource="ows">

  SELECT
    <!--- If all information about film(s) is desired --->
    <cfif URL.Details>
      *
    <!--- Otherwise, return the film's ID and title --->
    <cfelse>
      FilmID, MovieTitle
    </cfif>
  FROM Films
  <!--- If a specific film ID was specified --->
  <cfif URL.FilmID GT 0>
    WHERE FilmID = #URL.FilmID#
  <!--- If keywords were provided to search with --->
  <cfelseif URL.Keywords NEQ "">
    WHERE MovieTitle LIKE '%#URL.Keywords#%'
       OR Summary LIKE '%#URL.Keywords#%'
  </cfif>
  ORDER BY MovieTitle
</cfquery>


<!--- Convert the query recordset to a WDDX packet --->
<cfwddx action="CFML2WDDX"
        input="#FilmsQuery#">
```

If you visited this page normally with your browser, you'd probably see all the film titles and ID numbers smushed together on the page, because the browser doesn't know how to render the WDDX packet visually. On the other hand, if you view the source, you will see that the page is indeed responding with a packet fill of film data.

NOTE

I like to refer to this type of page as a "robot" because that term emphasizes the metaphor for a kind of automated process that is always waiting for requests and responding to them. Of course, any Web page can be considered robotic in nature, but something about the fact that the content is WDDX rather than HTML (and thus not designed to be read by humans) makes "robot" appropriate. If you prefer, think of this type of page as a "service." Just don't get this confused with official Web services as discussed in Chapter 68.

Listing 47.14 shows the packet returned by Listing 47.13 when visited normally (that is, without providing any URL parameters). For clarity, I have abbreviated the listing and added indention.

Listing 47.14 Response from `FilmsRobot1.cfm` When Visited with No URL Parameters

```
<wddxPacket version='1.0'>
<header/>
<data>

 <recordset
 rowCount='23'
 fieldNames='FILMID,MOVIETITLE'
 type='coldfusion.sql.QueryTable'>

 <field name='FILMID'>
 <number>1.0</number>
 <number>2.0</number>
 <number>3.0</number>
 <number>18.0</number>

 ...and so on...
 </field>

 <field name='MOVIETITLE'>
 <string>Being Unbearably Light</string>
 <string>Charlie's Devils</string>
 <string>Closet Encounters of the Odd Kind</string>
 <string>Folded Laundry, Concealed Ticket</string>

 ...and so on...
 </field>

 </recordset>

</data>
</wddxPacket>
```

Now if you visit the page, this time supplying `FilmID=3` and `Details=Yes` parameters in the URL, the robot will respond with the packet shown in Listing 47.15. Again, I have abbreviated the packet slightly to make it appear more clearly on the printed page.

Listing 47.15 Response from `FilmsRobot1.cfm` When Details for a Particular Film Are Requested

```
<wddxPacket version='1.0'>
 <header/>
 <data>

 <recordset
 rowCount='1'
 fieldNames='FILMID,MOVIETITLE,PITCHTEXT,AMOUNTBUDGETED,RATINGID,...'
 type='coldfusion.sql.QueryTable'>

 <field name='FILMID'>
 <number>3.0</number>
 </field>
 <field name='MOVIETITLE'>
 <string>Closet Encounters of the Odd Kind</string>
 </field>
 <field name='PITCHTEXT'>
 <string>Some things should remain in the closet</string>
 </field>
 <field name='AMOUNTBUDGETED'>
 <number>350000.0</number>
 </field>
 <field name='RATINGID'><number>5.0</number>
 </field>
 <field name='SUMMARY'>
 <string>One man finds out more than he ever wanted to know...</string>
 </field>
 <field name='IMAGENAME'>
 <string>f3.gif</string>
 </field>
 <field name='DATEINTHEATERS'>
 <dateTime>2000-11-7T0:0:0-5:0</dateTime>
 </field>
 </recordset>

 </data>
</wddxPacket>
```

Putting the Back-End Web Page to Use

Now that the back-end film robot page has been constructed, it's time to try incorporating the robot's responses into ordinary ColdFusion pages. Take a look at the simple code in Listing 47. It fetches the WDDX packet from the `FilmsRobot1.cfm` template (shown in Listing 47.14), and then deserializes the packet with `<cfwddx>`. Because the packet contains a recordset, the resulting `FilmsQuery` variable can be used just like the recordset returned by an ordinary `<cfquery>` tag. In this case, the query is used to display a list of movie titles (see Figure 47.7).

NOTE

The `RobotURL` value used in this listing assumes that you are saving the listings for this chapter in a folder called `16` within a folder called `ows-adv`, which is within your server's document root. It's also assumed that you are running ColdFusion in stand-alone mode (thus, the `:8500` part of the URL). You may need to adjust the URL slightly depending on where you are storing the listings.

Figure 47.7

Film data is fetched over the Internet from the robot page, then displayed to the user.

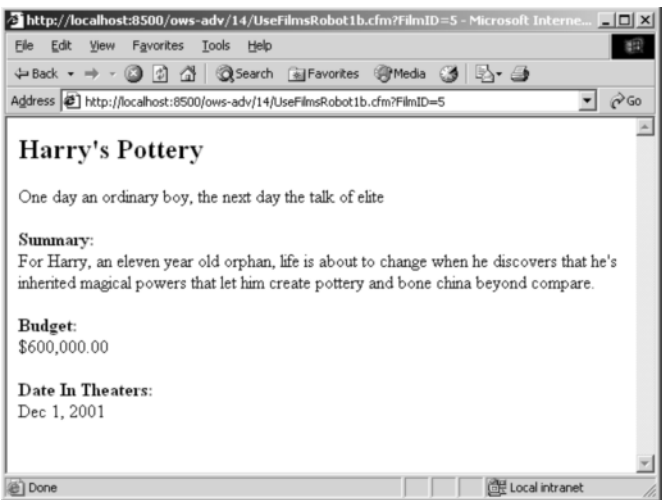
Listing 47.16 `UseFilmsRobot1a.cfm`—Connecting to a Back-End Robot Page

```
<!---
Name:        UseFilmsRobot1a.cfm
Author:      Nate Weiss and Ben Forta
Description: Fetches a WDDX packet via
             HTTP and uses the query
             contained within
Created:     02/01/05
--->

<!--- Location of the robot page --->
<!--- The URL could be anywhere in world, not just on this server --->
<cfset RobotURL="http://localhost:8500/ows_adv/16/FilmsRobot1.cfm">

<!--- Contact the robot page and retrieve the WDDX packet it returns --->
<cfhttp method="Get"
        url="#RobotURL#">

<!--- Deserialize the packet, which we know holds a query recordset --->
<cfwddx action="WDDX2CFML"
        input="#CFHTTP.FileContent#"
        output="FilmsQuery">

<!---
We can now use the query object normally,
just as if it came directly from a <cfquery> tag
--->
<h2>Live data retrieved from robot page</h2>
<cfoutput query="FilmsQuery">
  <a href="UseFilmsRobot1b.cfm?FilmID=#FilmID#">#MovieTitle#</a><br>
</cfoutput>
```

When the user clicks on any of the links produced by this listing (see Figure 47.7), they are brought to the `UseFilmsRobot1b.cfm` page, which displays the details about the film (see Figure 47.8). Listing 47.17 shows the code needed to put together the detail page.

Figure 47.8

The films robot is contacted again to get detailed information about individual films.



Listing 47.17 `UseFilmsRobot1b.cfm`—Connecting to a Back-End Robot Page

```
<!---
Name:        UseFilmsRobot1b.cfm
Author:      Nate Weiss and Ben Forta
Description: Fetches a WDDX packet via
             HTTP and uses the query
             contained within
Created:     02/01/05
--->

<!--- We need a FilmID --->
<cfparam name="URL.FilmID"
         type="numeric">

<!--- Location of the robot page --->
<!--- The URL could be anywhere in world, not just on this server --->
<cfset RobotURL="http://localhost:8500/ows_adv/16/FilmsRobot1.cfm">

<!--- Add parameters so the robot knows to return detailed information --->
<cfset RobotURL=RobotURL & "?FilmID=#URL.FilmID#&Details=Yes">

<!--- Contact the robot page and retrieve the WDDX packet it returns --->
<cfhttp method="Get"
        url="#RobotURL#">

<!--- Deserialize the packet, which we know holds a query recordset --->
<cfwddx action="WDDX2CFML"
        input="#CFHTTP.FileContent#"
        output="FilmQuery">

<!---
We can now use the query object normally,
just as if it came directly from a <cfquery> tag
--->
```

Listing 47.17 (continued)

```
<cfoutput query="FilmQuery">
  <h2>#MovieTitle#</h2>
  #PitchText#

  <p><strong>Summary:</strong><br>
  #Summary#<br>

  <p><strong>Budget:</strong><br>
  #LSCurrencyFormat(AmountBudgeted)#<br>

  <p><strong>Date In Theaters:</strong><br>
  #LSDateFormat(DateInTheaters)#<br>
</cfoutput>
```

As you can see, this listing is quite similar to the one that came before it (Listing 47.16). The only real difference is that this listing passes `FilmID` and `Details=Yes` parameters to the robot page, which causes the robot to respond with a WDDX packet similar to Listing 47.15 instead of Listing 47.14. Once the packet has been fetched and deserialized, the data from the packet can once again be used just like the results of a normal `<cfquery>` tag.

Understanding the Possibilities

You've seen how WDDX can be used to quickly and easily allow one ColdFusion page to grab data from another ColdFusion page. The code is simple and elegant, and easy to put together and understand. But what, exactly, is the benefit of doing things this way?

For most ColdFusion pages, there probably isn't any benefit. `<cfhttp>` and `<cfwddx>` are fast, lightweight processes, but they do introduce a small amount of additional processing time and general overhead. In terms of raw performance, just querying the database directly with a `<cfquery>` and using it all in the same page is clearly more straightforward and efficient.

But suppose, for whatever reason, you have one ColdFusion server that can connect to the database, and another ColdFusion server that cannot (perhaps it's outside the firewall, or overseas, or is owned by a different company). In that case, this kind of WDDX setup makes a whole lot of sense. Just put a robot page (like Listing 47.13) on the database-enabled server, and then call the robot (using code like Listing 47.16 and Listing 47.17) using pages on the second server. It's a remarkably simple and easy way to integrate the two environments.

Additionally, neither of the two servers need be running ColdFusion in order to supply or use the WDDX packets. Any of the other technologies discussed in the second portion of this chapter (such as ASP, Java, or Perl) could be used to create the robot page or the pages that use the robot. Mix and match to your heart's content.

Like the airport scenario mentioned in the section "The Concept of a Back-End Web Page," you can create robotlike pages that make statistics or other real-time information publicly available as WDDX packets. Of course, you can also put a password or other security mechanism on the robot pages if you only want the information to be available to your company's partners.

In short, the ability to exchange WDDX packets over the Internet via HTTP is a simple and powerful means to integrate machines that are separated from one another in some way, either physically or in terms of the software they are running. As mentioned, this is really the same concept that is at the heart of the Web Services movement, and you might want to consider creating and consuming official Web services instead (as discussed in Chapter 68). But if you go the home-grown, WDDX-based route, you have the advantage of comprehension: WDDX's simplicity becomes pretty compelling in and of itself.

Binary Content in WDDX Packets

So far, the WDDX packets you've seen in this chapter have contained data that can be expressed as a string. Sure, the data might be arranged in complex data structures such as arrays, recordsets, and structures, and the data might contain dates or numbers surrounded by `<number>` or `<dateTime>` elements. But the individual pieces of data have all been easy for WDDX to express as a string between the actual `<number>`, `<dateTime>`, `<string>`, and other data elements.

WDDX also allows you to include binary data in packets. For purposes of this discussion, *binary data* means any data that doesn't have an obvious plain-text counterpart. The most obvious examples of binary data are the contents of nontextual files such as image files, database files, executables, Word or other application-specific documents, and so on. Basically, any file that shows up as "garbage" in a text editor (such as Adobe Dreamweaver or Windows Notepad) should probably be considered to be a binary file for our purposes here.

You may not be very familiar with the binary object type in CFML because it's not often needed in Web applications. Here are a few notes about binary objects in ColdFusion:

- You can use `<cffile>` with `action="ReadBinary"` to read a binary file. The contents are returned to you as a binary object variable.
- Once you have a binary object variable, you can save it to the server's drive using the usual `<cffile>` tag with `action="Write"`.
- You can check whether a particular variable holds a binary object by using the `IsBinary()` function, or `IsObject("binary")`.
- A binary object variable is essentially an array of individual bytes. There wouldn't normally be much of a reason to, but you can access the individual bytes using normal array notation, such as `MyBinary[5]` to access the fifth byte. As you might expect, `ArrayLen(MyBinary)` returns the size of the object in bytes. If `MyBinary` came from a file using `<cffile>` with `action="ReadBinary"`, then `ArrayLen(MyBinary)` should match the size of the file on disk as reported by the operating system.

If you serialize a binary object with `<cfwddx>`, or if you serialize a recordset, array, or structure that contains a binary object, the object will be represented by a pair of `<binary>` elements in the resulting WDDX packet. Between the `<binary>` elements will be bunch of characters that look like some

kind of encrypted or scrambled text. There will also be a `length` attribute that indicates the number of characters between the `<binary>` elements, something like this:

```
<binary length="6614">R0lGODlhZACDANUAAP//zP//mf//M///AP/MzP/Mm...</binary>
```

I replaced most of the characters with the `...` at the end, but you get the idea. To us humans, it looks like a bunch of random text characters.

What's going on here? Well, the binary data has been encoded (converted) into a special text format called Base 64. Base 64 encoding is most often used for email attachments: Your email client converts your attached documents and images to this Base 64 format and includes the Base 64 version in the message. That's why binary files can be sent in email messages, which otherwise know only how to deal with plain text.

You can find out more about Base 64 on the Web (start at the W3C site), but it's not really so important for you to understand the mechanics of the encoding. All you need to know is that it's possible to include binary content in your WDDX packets. (And if anyone asks, you can tell them that it's done by converting to Base 64 so that the content can be represented in XML, which is only capable of handling text characters.)

NOTE

ColdFusion also provides `ToBinary()` and `ToBase64()` functions so that you can convert between Base 64 text and individual pieces of binary data. These functions have nothing directly to do with WDDX per se; however, the same conversions are used internally by `<cfwddx>` when you serialize or deserialize packets that contain binary data.

Listing 47.18 shows a more advanced version of the Films Robot page that was created earlier in Listing 47.13. Among other things, this version supports an optional `Images` parameter that can be included in the URL. If `Images=Yes`, then the images (for those films that have them) are included in the WDDX packet that the page generates.

Listing 47.18 `FilmsRobot2.cfm`—Including Binary Content in WDDX Packets

```
<!---
Name:        FilmsRobot2.cfm
Author:      Nate Weiss and Ben Forta
Description: Creates a back-end web page that
             supplies data about films
Created:     02/01/05
--->

<!---
If a WDDX packet is supplied to this page as a FORM or URL parameter
--->
<cfif IsDefined("ParamsAsWDDX")>
  <!---
  Deserialize the packet and use its contents as parameters
  to control this page's behavior
  --->
  <cfwddx action="WDDX2CFML"
          input="#ParamsAsWDDX#"
          output="IncomingParams">
<cfelse>
```

Listing 47.18 (continued)

```
  <!---
  The values in this structure will be used to control page's behavior
  --->
  <cfset IncomingParams=StructNew()>
</cfif>

<!---
If form parameters are being submitted, use them to contol behavior
--->
<cfif StructCount(FORM) GT 1>
  <cfset StructAppend(IncomingParams, FORM, "Yes")>
  <cfset StructDelete(IncomingParams, "FIELDNAMES")>
<!--- Otherwise, just use  --->
<cfelse>
  <cfset StructAppend(IncomingParams, URL, "Yes")>
</cfif>


<!---
The incoming WDDX packet may include these parameters.
The default values are used when there is no incoming packet,
or when the incoming packet doesn't include the parameter.
--->
<cfparam name="IncomingParams.usecache" type="boolean" default="yes">
<cfparam name="IncomingParams.details" type="boolean" default="no">
<cfparam name="IncomingParams.filmid" type="numeric" default="0">
<cfparam name="IncomingParams.keywords" type="string" default="">
<cfparam name="IncomingParams.orderby" type="string" default="movietitle">
<cfparam name="IncomingParams.images" type="boolean" default="no">

<!--- If a cached query may be used --->
<cfif IncomingParams.UseCache>
  <cfset CachedWithin=CreateTimeSpan(0,0,30,0)>
<cfelse>
  <cfset CachedWithin=CreateTimeSpan(0,0,0,0)>
</cfif>

<!---
Execute a database query to select film information from database
--->
<cfquery name="FilmsQuery"
         datasource="ows"
         cachedwithin="#CachedWithin#">
  SELECT
    <!--- If all information about film(s) is desired --->
    <cfif IncomingParams.Details>
      *
    <!--- Otherwise, return the film's ID and title --->
    <cfelse>
      FilmID, MovieTitle
      <cfif IncomingParams.Images>, ImageName</cfif>
    </cfif>
  FROM Films
  <!--- If a specific film ID was specified --->
  <cfif IncomingParams.FilmID GT 0>
```

Listing 47.18 (continued)

```
      WHERE FilmID = #IncomingParams.FilmID#
    <!--- If keywords were provided to search with --->
    <cfelseif IncomingParams.Keywords NEQ "">
      WHERE MovieTitle LIKE '%#IncomingParams.Keywords#%'
    </cfif>
    <!--- Order the results appropriately --->
    ORDER BY #IncomingParams.OrderBy#
  </cfquery>

  <!--- If the requesting process wants images included in the packet... --->
  <cfif IncomingParams.Images>
    <!--- Add an ImageContent column to the query recordset --->
    <cfset QueryAddColumn(FilmsQuery, "ImageContent", ArrayNew(1))>

    <!--- For each row in the recordset --->
    <cfloop query="FilmsQuery">
      <!--- If this film has an associated image
            (according to the database) --->
      <cfif ImageName NEQ "">
        <!--- Location of the image on the server's drive --->
        <cfset ImagePath=ExpandPath("/ows/images/#ImageName#")>

        <!--- If the file actually exists on the server --->
        <cfif FileExists(ImagePath)>
          <!--- Read the contents of the file --->
          <!--- ImageBinary will be a binary object variable --->
          <cffile action="READBINARY"
                  file="#ImagePath#"
                  variable="ImageBinary">

          <!--- Store the binary object variable in
                the ImageContent column --->
          <cfset FilmsQuery.ImageContent[CurrentRow] = ImageBinary>
        </cfif>

      </cfif>
    </cfloop>
  </cfif>

  <!--- Convert the query recordset to a WDDX packet --->
  <cfwddx action="CFML2WDDX"
          input="#FilmsQuery#"
          output="WDDXPacket">

  <!---
  Return the packet to whatever system requested this page.
  Use <cfcontent> to reset the output stream, so that any whitespace,
  page headers, etc., included by Application.cfm gets discarded.
  --->
  <cfcontent type="text/xml"
             reset="Yes"><cfoutput>#WDDXPacket#</cfoutput>
```

The new portion of the listing is the `<cfif>` block in the middle. If there is an `Images=Yes` parameter in the URL, a new query column called `ImageContent` is added to the `FilmsQuery` recordset. Then, for each film that has an associated image, the `<cffile>` tag is used with `action="ReadBinary"` to read

the actual contents of the file into `ImageBinary` (a binary object variable). The value of `ImageBinary` is then placed into the `ImageContent` column of the recordset. When this block of code is finished, the query object contains binary data for each film that has an associated image. The resulting WDDX packet will thus contain `<binary>` blocks (like the one shown earlier in this section) for each image.

There are a few other differences between this version of the robot page and the original from Listing 47.13:

- The system that is submitting the request to the robot can now specify the sort order with the `OrderBy` parameter, and with the `UseCache` parameter can control whether ColdFusion can use cached query information to build the packet.
- The `<cfcontent>` tag is used to specify a content type of `text/xml` as the packet is being sent back to whatever system requested the page. Since WDDX packets are XML, the addition of the content type is appropriate. However, the `<cfwddx>` tag and the other WDDX implementations discussed in this chapter will work just fine regardless of the content type.

Listing 47.19 demonstrates use of the binary data in a WDDX packet after the packet has been deserialized. This is a revised version of the film detail page from Listing 47.17 (shown in Figure 47.8).

Listing 47.19 `UseFilmsRobot2b.cfm`—Using Binary Data Included in WDDX Packets

```
<!---
Name:        UseFilmsRobot2b.cfm
Author:      Nate Weiss and Ben Forta
Description: Fetches a WDDX packet via
             HTTP and uses the query
             contained within
Created:     02/01/05
--->


<!--- We need an ID number for the desired film --->
<cfparam name="URL.FilmID"
         type="numeric">

<!--- Location of the robot page --->
<!--- The URL could be anywhere in world, not just on this server --->
<cfset RobotURL="http://localhost:8500/ows_adv/16/FilmsRobot2.cfm">

<!--- Add parameters so the robot knows to return detailed information --->
<cfset RobotURL=RobotURL&"?FilmID=#URL.FilmID#&Details=Yes&Images=Yes">

<!--- Contact the robot page and retrieve the WDDX packet it returns --->
<cfhttp method="Get"
        url="#RobotURL#">

<!--- Deserialize the packet, which we know holds a query recordset --->
<cfwddx action="WDDX2CFML"
        input="#CFHTTP.FileContent#"
        output="FilmQuery">
```

Listing 47.19 (continued)

```
<!--- If there is binary image content for this film --->
<cfif IsBinary(FilmQuery.ImageContent)>
  <!--- Folder location for storing deserialized images --->
  <cfset DeserializedImageFolder=ExpandPath("DeserializedImages")>

  <!--- Create the folder if it doesn't already exist --->
  <cfif NOT DirectoryExists(DeserializedImageFolder)>
    <cfdirectory action="Create"
                 directory="#DeserializedImageFolder#">
  </cfif>

  <!--- The image will be saved using the name in the ImageName column --->
  <cfset ImageFilePath=DeserializedImageFolder & "/" & FilmQuery.ImageName>

  <!--- Save image to DeserializedImages folder on the server's drive --->
  <cffile action="WRITE"
          file="#ImageFilePath#"
          output="#FilmQuery.ImageContent#">
</cfif>

<!---
We can now use the query object normally,
just as if it came directly from a <cfquery> tag.
--->
<cfoutput query="FilmQuery">
  <h2>#MovieTitle#</h2>
  #PitchText#

  <p><strong>Summary:</strong><br>
  #Summary#<br>

  <p><strong>Budget:</strong><br>
  #LSCurrencyFormat(AmountBudgeted)#<br>

  <p><strong>Date In Theaters:</strong><br>
  #LSDateFormat(DateInTheaters)#<br>

  <!--- If there is an image available --->
  <cfif ImageName NEQ "">
    <p><strong>Image:</strong><br>
    <img src="DeserializedImages/#ImageName#" border="0">
  </cfif>
</cfoutput>
```

When running this code be sure to pass a `FilmID` as a URL parameter (for example, `?FilmID=2`).

This version adds `Images=Yes` to the URL it uses to contact the robot page, which means the resulting packet may contain a `<binary>` element for the selected film's image. After the packet is deserialized, the `<cfif>` block in the middle of the listing checks to see if the `ImageContent` field actually contains binary data. If so, the binary content is stored as an image on the server's drive, in a folder called `DeserializedImages` within the folder in which this listing is saved. (If the folder does not exist yet, it is created with `<cfdirectory>`.)

Finally, near the end of the code, an ordinary <img> tag is used to display the image for the film, if available. The result looks just like the image in Figure 47.8, with the addition of the film's image at the bottom of the page.

Using binary data with WDDX in this way doesn't make a whole lot of sense if this page is on the same server as the robot page. But if the robot server is on a different server, any needed images will be automatically copied to the current server as the page executes. Of course, even this scenario doesn't make much sense in the context of the Web, because you could just tell the browser to access the images on the robot server without copying from one place to another—but you get the idea. The binary support in WDDX allows you to include any kind of data in packets, including images, documents, and so on. What you do with the feature is up to you.

Listing 47.20 shows another way to produce the same results. With this version, the binary content is not saved to a permanent location on the server's drive. Instead, the content is stored at a temporary location supplied by `GetTempFile()` and then streamed to the browser with <cfcontent>. The <img> tag at the bottom of the listing has been modified to request the image directly from the <cfcontent> part of this listing, rather than as a discrete `.gif` file. In other words, ColdFusion supplies the HTML for the details page (see Figure 47.8) as well as the actual image to display on the page. All the information comes from the robot page, which could be on the other side of the globe.

Listing 47.20 `UseFilmsRobot2c.cfm`—Serving Binary Content from a WDDX Content Directly

```
<!---
Name:        UseFilmsRobot2c.cfm
Author:      Nate Weiss and Ben Forta
Description: Fetches a WDDX packet via
             HTTP and uses the query
             contained within
Created:     02/01/05
--->


<!--- We need a film ID to be supplied in the URL --->
<cfparam name="URL.FilmID"
         type="numeric">

<!---
Flag used to indicate whether browser is requesting the the film's
image to display on the detail page, or the detail page itself.
--->
<cfparam name="URL.ImageOnly"
         type="boolean"
         default="No">

<!--- Location of the robot page --->
<!--- The URL could be anywhere in world, not just on this server --->
<cfset RobotURL="http://localhost:8500/ows_adv/16/FilmsRobot2.cfm">

<!--- Add parameters so the robot knows to return detailed information --->
<cfif URL.ImageOnly>
  <cfset RobotURL=RobotURL&"?FilmID=#URL.FilmID#&Details=No&Images=Yes">
<cfelse>
```

Listing 47.20 (continued)

```
    <cfset RobotURL=RobotURL&"?FilmID=#URL.FilmID#&Details=Yes&Images=No">
  </cfif>

  <!--- Contact the robot page and retrieve the WDDX packet it returns --->
  <cfhttp method="Get"
          url="#RobotURL#">

  <!--- Deserialize the packet, which we know holds a query recordset --->
  <cfwddx action="WDDX2CFML"
          input="#CFHTTP.FileContent#"
          output="FilmQuery">

  <!---
  If the ImageOnly flag is set, send back the binary image itself
  --->
  <cfif URL.ImageOnly>
    <!---
    If there is binary image content for this film
    --->
    <cfif IsBinary(FilmQuery.ImageContent)>
      <!--- Temporary location for the image --->
      <cfset TempFile=GetTempFile(GetTempDirectory(), "img")>

      <!--- Save the image content from the WDDX packet to the temp file --->
      <cffile action="WRITE"
              file="#TempFile#"
              output="#FilmQuery.ImageContent#">

      <!---
      Stream the content to the browser.
      The temporary file will be deleted when finished.
      --->
      <cfcontent type="image/gif"
                 reset="Yes"
                 file="#TempFile#"
                 deletefile="Yes">

    <!--- If we are meant to send back image, but robot didn't provide it --->
    <cfelse>
      <!--- Log the problem --->
      <cflog text="The image for FilmID #URL.FilmID# was not recieved."
             file="FilmsRobot"
             type="Information">

    </cfif>


  <!---
  When the page is called without the ImageOnly flag,
  we should create the detail page for the film (as HTML).
  --->
  <cfelse>
    <!---
    We can now use the query object normally,
    just as if it came directly from a <cfquery> tag.
```

Listing 47.20 (continued)

```
    --->
    <cfoutput query="FilmQuery">
      <h2>#MovieTitle#</h2>
      #PitchText#

      <p><strong>Summary:</strong><br>
      #Summary#<br>

      <p><strong>Budget:</strong><br>
      #LSCurrencyFormat(AmountBudgeted)#<br>

      <p><strong>Date In Theaters:</strong><br>
      #LSDateFormat(DateInTheaters)#<br>

      <!--- If there is an image available --->
      <cfif ImageName NEQ "">
        <p><strong>Image:</strong><br>

        <!---
        The SRC for image is this page's URL with ImageOnly flag added.
        The browser will call this page again to get the image itself.
        --->
        <img src="UseFilmsRobot2c.cfm?FilmID=#URL.FilmID#&ImageOnly=Yes"
             border="0">
      </cfif>
    </cfoutput>
  </cfif>
```

CHAPTER 48

Using JavaScript and ColdFusion Together

IN THIS CHAPTER

A Crash Course in JavaScript E277
Working with Form Elements E288
Passing Variables to JavaScript E290
Passing Data to JavaScript Using <cfwddx> E311
Working with WDDX Packets in JavaScript E322
Calling CFCs from JavaScript E340
Passing Simple Variables to ColdFusion E341

As you now know, ColdFusion gives you an extremely powerful, easy to use language for creating interactive Web sites. As you also know, all ColdFusion code executes on the server; the browser isn't expected to understand CFML, and it certainly isn't expected to be able to connect to your databases and other server-side resources directly.

All of which is, like, sooooo dreamy. But depending on the application, you may sometimes need to exert some kind of programmatic control over the browser itself——its windows, status bars, form fields, and so on. This is where JavaScript comes in. Rather than executing on the server, JavaScript code executes on the client machine, within the context of the browser.

There is often a bit of disconnect between the worlds of ColdFusion and JavaScript developers. ColdFusion developers often don't know much about JavaScript, and vice-versa. As such, ColdFusion developers tend to solve problems from the server side, while JavaScript folks solve problems on the client; either approach can be more elegant or appropriate, depending on the situation. It's easy to see that knowing something about both is the best thing, so that you at least know what your options are.

The intent of this chapter is to familiarize you with JavaScript and the role it can play in ColdFusion applications. In particular, it focuses on making forms more interactive and lively, and passing or sharing variables between the two environments. You will also learn how to use WDDX with JavaScript.

A Crash Course in JavaScript

This section will introduce you as quickly as possible to JavaScript. If you're already familiar with the language, you can skip this section for now, though the tables in this section may come in handy later as a kind of quick reference.

NOTE

If you have already read Chapter 44, "ColdFusion Scripting," online, much of this section may seem a bit redundant. However, ColdFusion's scripting language is quite a bit different from JavaScript. In some ways, the two languages can be said to be very similar, particularly with respect to the statements they support, and the way the basic syntax works (curly braces, parentheses, and so on). They are completely different in other ways, such as the operators they support and the way they treat objects.

JavaScript Language Elements

While it's not possible to fully introduce you to the JavaScript language in these pages, it's important for you to at least understand the basic language elements available to you. The next few pages list each of the core JavaScript statements, and can be used as a sort of mini-reference. If you need further information about these items, they will be covered in detail in any JavaScript book or comparable online resource.

For clarity, I have broken the statements into four groups:

- Basic statements
- Statements for looping
- Statements for creating functions
- Statements for error handling

The Basic Statements

Table 48.1 lists the basic JavaScript statements. By "basic", I just mean all the statements that aren't specifically for looping, creating functions, or error handling.

Table 48.1 Basic JavaScript Statements

STATEMENT	DESCRIPTION
`if .. else`	Implements simple if / else processing. Comparable to the `<cfif>` and `<cfelse>` tags in CFML.
`switch`	Executes different blocks of code based on the current value of some variable or expression. Comparable to the `<cfswitch>` and `<cfcase>` tags in CFML.
`with`	Establishes the default object for evaluating methods and properties. Generally used to make code easier to read and type. Though handy when used correctly, `with` is not for the faint of heart because it can lead to confusing unexpected behavior. There is no direct equivalent in CFML.
`var`	Creates a variable that is local to the current context. Generally, `var` is used to create variables that are only visible within the body of a function. Comparable to the `<cfset var>` syntax allowed within CFML `<cffunction>` blocks.

Statements for Looping

Like CFML, JavaScript provides a number of different ways to loop over blocks of code. Table 48.2 lists the various types of JavaScript loops, which correspond to various flavors of the `<cfloop>` tag in CFML.

Table 48.2 JavaScript Statements for Looping

STATEMENT	DESCRIPTION
`for`	The simplest and most familiar type of loop. Creates a loop block that advances the value of a counter with each iteration; the loop continues until the value reaches a certain value. Comparable to `<cfloop>` with `from` and `to` attributes in CFML.
`for .. in`	Creates a loop block that iterates over each of the values in a given object or array. Comparable to a `<cfloop>` with `collection` and `item` attributes in CFML.
`while`	Creates a loop block that executes until a particular condition is no longer true. If the condition is already false when the loop is encountered, it is skipped altogether. Similar to a `<cfloop>` that uses a `condition` attribute.
`do .. while`	Creates a loop block that executes until a particular condition is no longer true. The loop is guaranteed to execute at least once. There is no direct counterpart in CFML, but it's similar to a `<cfloop>` that uses a `condition` attribute.
`break`	Breaks out of a loop block. Execution continues on the first line following the loop block. Comparable to `<cfbreak>` in CFML.
`continue`	Skips the remainder of a loop block for the current pass through the loop. There is no direct counterpart in CFML.

Most `for` loops look like the following. This loop executes 10 times, advancing the value of `i` for each iteration:

```
for (i = 0; i < 10; i++) {
 ...your code here...
}
```

Here's a `while` loop that does the same thing:

```
i = 0;
while (i < 10) {
 ...your code here...
 i = i + 1;
}
```

Another `while` approach:

```
i = 0;
while (true) {
 ...your code here...
 i = i + 1;
 if (i >= 10) {
 break;
```

```
 }
}
```

This `for .. in` loop is functionally equivalent:

```
var myArray = [0,1,2,3,4,5,6,7,8,9];
for (i in myArray) {
 ...your code here...
}
```

Statements for Creating Functions

Elsewhere in this book, you learn how to create your own functions for use in CFML code. You can also create functions for use in JavaScript code, using the statements listed in Table 48.3.

Table 48.3 JavaScript Statements for Creating Functions

STATEMENT	DESCRIPTION
`function`	Creates a function that can be called elsewhere by name; also establishes the function's input arguments, if any. Comparable to the `<cffunction>` and `<cfargument>` tags in CFML.
`return`	Stops function execution, returning a particular value as the function's output. Comparable to `<cfreturn>`, though `return` can be used anywhere within the body of a function, rather than only at the end.

Here's a function that returns the product of two numbers:

```
function multiplyNumbers(num1, num2) {
 return num1 * num2;
}
```

Here's another function that returns the absolute value (the positive version) of whatever number is passed to it (actually, there is a built-in `Math.abs()` function that does the same thing; I'm just trying to show as many statements working together as possible):

```
function absoluteValue(number) {
 var result;

 if (number < 0) {
 result = 0 - number;
 } else {
 result = number;
 }

 return result;
}
```

Statements for Error Handling

Table 48.4 lists the statements available for throwing and handling exceptions (errors) within your JavaScript code.

Table 48.4 JavaScript Error Handling Statements (recent browsers only)

STATEMENT	DESCRIPTION
`try .. catch .. finally`	Tries to execute a block of code, catching any exceptions. You can respond to the exceptions in the `catch` block, much like the `<cfcatch>` tag in CFML. Throws a custom exception (error). Comparable to `<cfthrow>` in CFML.
`throw`	Throws a custom exception (error). Comparable to `<cfthrow>` in CFML. This is a relatively recent addition to JavaScript and is not available in all browsers.

The following code snippet shows the basic form of the `try..catch..finally` construct. If you include this code in a web page, you will first see the "I will now try..." message, then the value of the `firstName` variable. If you comment out the `var` line at the top, you will see the "Hmm, looks like..." message, which proves that the `catch` block executes whenever a problem (such as a reference to a nonexistent variable) occurs. In either case, the message in the `finally` block will always be displayed, making the `finally` statement ideal for displaying status messages or taking final action regardless of whether problems occur.

```
<script language="JavaScript" type="text/javascript">
 try {
 // Comment out the next line to see the error catching behavior
 var firstName = "Winona";

 alert("I will now try to display the value of the firstName variable.");
 alert(firstName);

 } catch(e) {
 // If any errors occur...
 alert("Hmmm, looks like there is no firstName variable. Sorry!");

 } finally {
 // This portion executes no matter what, error or no error
 alert("Well, I hope you enjoyed our little exercise.");
 }
</SCRIPT>
```

JavaScript Operators

Like any other language, JavaScript provides a set of operators for doing things like assigning values to variables or comparing values. Table 48.5 lists most of the JavaScript operators you are likely to encounter; refer to a JavaScript guide for a complete list. If you're familiar with C or Java, these operators will be very familiar to you. If not, that's okay, too.

You'll see many of these operators sprinkled throughout the example listings and code snippets throughout this chapter.

Table 48.5 Abbreviated list of JavaScript Operators

<table>
<tr><th>STATEMENT</th><th>DESCRIPTION</th></tr>
<tr><td>=</td><td>Variable assignment, like in CFML.</td></tr>
<tr><td>==</td><td>Equality test, like EQ or IS in CFML.</td></tr>
<tr><td>!=</td><td>Inequality test, like NEQ or IS NOT in CFML.</td></tr>
<tr><td>++</td><td>Increments a value by one, like in CFML.</td></tr>
<tr><td>--</td><td>Decrements a value by one, like in CFML.</td></tr>
<tr><td>&&</td><td>Logical “and” operator, like AND in CFML. Just as in CFML, you can use parentheses to clarify what you want the operator to affect (same goes for the next two operators).</td></tr>
<tr><td>||</td><td>Logical “or” operator, like OR in CFML.</td></tr>
<tr><td>!</td><td>Logical “not” operator, like NOT in CFML.</td></tr>
<tr><td>+</td><td>Numeric addition or string concatenation. Be careful not to use & for string concatenation; that works in CFML but has a different meaning in JavaScript.</td></tr>
<tr><td>-</td><td>Numeric subtraction, as you would expect.</td></tr>
<tr><td>*</td><td>Numeric multiplication.</td></tr>
<tr><td>/</td><td>Numeric division.</td></tr>
<tr><td>%</td><td>Modulus (remainder after division), like MOD in CFML.</td></tr>
</table>

Understanding the Core Objects

JavaScript also provides a number of built-in objects, often called the *core* objects. Many of these objects represent data types, like `String` or `Date`. Table 48.6 lists some of the most commonly used core objects, many of which you will see used throughout this chapter. Consult a JavaScript reference for a complete list of the methods and properties exposed by these objects.

Table 48.6 Core JavaScript Objects (Abridged)

OBJECT	DESCRIPTION
`Array`	Array objects expose helpful methods such as `.reverse()` and `.sort()`. The length of an array is available in the `.length` property.
`Date`	Date objects provide various date-manipulation methods such as `.getMonth()` and `.toLocaleString()`.
`Math`	You don't create instances of this object; instead you use its methods directly, as in `Math.round()`, `Math.abs()`, and `Math.random()`.
`Object`	Use the Object type to create your own custom objects that hold whatever data you need. Creating a new object is very similar conceptually to creating a new structure with `StructNew()` in CFML.
`String`	Any string object has a number of helpful methods, like `.substring()`, `.toLowerCase()`, `.toUpperCase()` , and `.indexOf()`, which is kind of like CFML's `Find()` function. The length of a string is available in the `.length` property.

Understanding JavaScript's Relationship to Web Browsers

JavaScript was originally designed by Netscape as a simple, lightweight scripting language that would work well in Web browsers. As such, people often get a little bit confused, thinking that JavaScript is something that is *exclusively* used in browsers.

Not so. JavaScript is a language that can be used in many different contexts, in browsers, on servers, and in many other types of applications. Here's a partial list of the places where JavaScript (the scripting language, not the `<script>` tag) can be used:

- **Web browsers**. This remains the most obvious example, and the one that will be discussed directly in this chapter.
- **Email clients**. These days, many email clients are just extensions of each vendor's respective Web browser technology, and provide much of the same support for JavaScript.
- **Adobe Flash applications**. Beginning with version 5, Flash movies are scripted using ActionScript, which is very similar to JavaScript.
- **Active Server Pages (ASP) pages**. Many people think of ASP pages as something that you can only write with Visual Basic style syntax (VBScript), but ASP pages can also be written with JScript, which is essentially yet another form of JavaScript.
- **Windows Scripting Host (WSH) scripts**. These scripts can be used to create macros that interact directly with the Windows shell.
- **Adobe Dreamweaver**, for creating tag editor dialogs and other custom extensions. HomeSite+ (formerly ColdFusion Studio) also exposes an extension API to JavaScript.
- **ColdFusion**, to the extent that CFScript bears a striking relationship to JavaScript. It can't really be considered compatible with JavaScript, since it doesn't include support for the core objects listed in Table 48.5, but it's a pretty close cousin nonetheless. For details about CFScript, see Chapter 44.

NOTE

I'm skipping over some rather acrimonious contentious history when I say that JScript, ActionScript, and CFScript are essentially synonyms for JavaScript. JavaScript came first, developed by Netscape for version 2.0 of their browser. The language was later turned over to a standards body and evolved into the open specification known as EMCAScript, upon which JScript and ActionScript are officially based upon. Okay, even that is an oversimplification. Regardless of history, and regardless of the relevance of standards, the real-world intent of JScript and ActionScript are clearly to look, feel, and otherwise work like JavaScript.

So, while although JavaScript is most often used in Web browsers, that's not the only place where it can be used. That's an important point to understand, and which brings us to our next topic: understanding scripting object models.

Understanding Scripting Object Models

Because JavaScript is designed to be used in many different contexts (see the bulleted list in the previous section), the context-specific stuff is kept separate from the language itself. Everything relevant

to the specific context (say, a Web page or a form if the context is a browser, or the position of the cursor if the context is Dreamweaver) is exposed to JavaScript in the form of a set of scriptable objects, or *object model*.

So, within any given context, there are really two things to know and understand:

- **The JavaScript language itself**. Think of the language as consisting mainly of the items listed in Table 48.1 through Table 48.5, plus the various operators and semantic elements like curly braces and semicolons.
- **The object model specific to the context**. Assuming that the context is a Web browser, then the object model includes scriptable representations of browser and Web page concepts (like the URL for the current document, the size of the browser window, and information about the image and form fields on the page).

TIP

If it helps, think of JavaScript itself as the language used to talk to the browser, and the object model as what you talk about. The language, then, is just a way of speaking; the object model is the actual subject of your conversation.

The JavaScript language is really pretty simple. It's the object model supported by each browser that can get pretty complicated and potentially confusing. While each browser and version supports more or less the same language (JavaScript, which hasn't changed very much over the years), the object model exposed to JavaScript can differ quite a bit between browsers and versions. The differences between the object models exposed by Netscape and Microsoft browsers, for instance, were wildly different during the 4.x and 5.x versions of the respective products. Version 6.0 brought the object models much closer together in terms of scope, functionality, and overall design. Going forward, you can expect most scripts to work identically in IE, Firefox, Netscape, Mozilla, and Opera, especially if you stay away from whatever the latest-and-greatest feature of the moment happens to be.

JavaScript Objects Available in Web Pages

The scripting object model exposes a large number of objects that you can control via script within each Web page. Table 48.7 provides a list of some of the most important objects you can control with JavaScript, and some of their properties and methods.

Table 48.7 A Short List of Scriptable Objects in Web Pages

OBJECT	DESCRIPTION
`document`	The `document` object contains information about the current web page, including the `location`, `forms`, `images`, and `frames` properties listed in this table. You can refer to other elements on the page (`<div>`, `<span>`, `<table>`, and other areas) using the `getElementById()` method, assuming that you have given the element an `ID` attribute in your HTML code. From there you can, among other things, change the text displayed in an area using the element's `innerHTML` property (this is demonstrated briefly in Listing 48.12).

Table 48.7 (continued)

OBJECT	DESCRIPTION
`document.location`	The `document.location` object allows you to inspect and control the URL of the current document. Reading the `document.location.href` property returns the current URL; setting the property causes the browser to navigate to another page (as if a link was clicked). The `document.location. replace()` method also navigates the browser to another page, but where the new page takes the place of the current page in the user's page history, (which means that the user can't use the Back button to return to the current page).
`document.forms`	This property is a collection (basically an array) of `<form>` blocks on the page. `document.forms[0]` refers to the first form, `document.forms[1]` refers to the second form (if any), and so on. If there are no forms on the page, `document.forms.length` will be zero. The various text input fields, drop-down lists, checkboxes, and so on in each form can be accessed as discussed in Table 48.10.
`document.images`	This property is a collection of objects that represent each `<IMG>` element in the current document. Documents can be referred to by index, such as `document.images[0]` for the first image on the page, or by the value of their `name` attribute, as in `document.images .MyImage` or `document. images["MyImage"]`. Among other things, you can cause a different image to appear by setting one of these object's' `src` properties; this is how image rollovers work. You can see an example of changing what image is displayed in the Film Browser example later in this chapter (Listing 48.7).
`document.frames`	This property is a collection of frames, if any, within the current document. The first frame is `document.frames[0]`, and so on. `<iframe>` blocks are included as well. If there are no frames (that is, if the current document is not a `<frameset>` page and contains no `<iframe>` tags), then `document.frames.length` is zero.
`window`	The `window` object represents the current browser window (as opposed to the `document`, which sits within the window). You can open new windows (including pop-up windows) with `window.open()`, and can close the current window with `window.close()`. The window object also exposes a few utility methods, such as the `setInterval()` and `clearInterval()` methods for creating timers (demonstrated in Listing 48.12).

Understanding JavaScript Events

JavaScript supports scriptable *events*. Events generally occur when something changes, such as the loaded status of a document, or the value of a form field. You can provide JavaScript code that should be executed when an event occurs by including an attribute with the same name in your HTML. Such code is called an *event handler*.

Table 48.8 provides a short list of interesting events that are fired by various objects in the scripting object model. You'll see these events used throughout the examples in this chapter. This is by no

means a complete list; it's just a sampling of the most commonly-used events, to give you an idea of what you can do. Consult a JavaScript reference for details.

Table 48.8 A Short List of Useful Events

ATTRIBUTE	DESCRIPTION
`document.onload`	Executes when the page first appears (is loaded) in the browser window. There is also an `unload` event that executes when the user leaves the page. To specify code to execute when the `onload` event fires, add an `onload` attribute to the document's `<body>` tag.
`onclick`	Many elements (especially form fields) support `onclick` events, which fire when the user clicks and then releases the mouse button over an object. The `onclick` event is used in the Film Browser example in Listing 48.7.
`onchange`	Most form field elements support `onchange` events, which execute when the user changes the value in a form field. In general, the event fires when the user is done making changes to a field (that is, when focus shifts away from the field), not for each keystroke. The event is used in Listing 48.1 as well as the various "cascading select list" examples throughout this chapter.
`onkeyup` and `onkeydown`	Most form field elements support these events, which fire when users use the keyboard to enter or change text. This event is also used in the Mortgage Calculator example in Listing 48.1.
`onsubmit`	This event fires when the user submits a form. Whatever JavaScript or function is called from this event can return a value of `false` to prevent the form from actually being submitted; this is the basis of most JavaScript form-validation routines. You can view source on a Web page that uses `<cfform>` validation to see how this works.

Including JavaScript Code in Web Pages

You can include JavaScript code in any Web page. Just place the code between a pair of `<script>` tags; you'll see examples of this throughout this chapter. Table 48.9 explains the attributes supported by the `<script>` tag.

Table 48.9 `HTML <script>` Tag Syntax

ATTRIBUTE	DESCRIPTION
`language`	The language that the script is written in. This value is usually `JavaScript`; that's the value used in this chapter's listings. You can also specify the JavaScript version, such as `JavaScript1.1` or `JavaScript1.2`; such values cause subtle changes in behavior depending on the browser. In Microsoft browsers, you can also use `JScript` or `VBScript`. Consult a scripting reference for details. This attribute is technically deprecated in the current HTML specification, but remains the most common way of specifying a scripting language.

Table 48.9 (continued)

ATTRIBUTE	DESCRIPTION
`type`	This is now the preferred way to specify the scripting language. For JavaScript, you use `type="text/javascript"`. For maximum compatibility, provide both the `language` and `type` attributes. For modern browsers that support the HTML 4.01 standard, you can use `type` alone.
`src`	Optional. You can use the `src` attribute to include an external script file, which will be downloaded and executed by the browser. Conceptually, this is kind of like a `<cfinclude>` tag in CFML.
`defer`	You can add the optional `defer` flag if you would like to give the browser the option of not interpreting the script code until the page has finished loading. This can speed up the loading of your page, especially for users with slow connections, but can have side effects if you're not a bit careful. Consult a scripting reference for details.

Browser Specific Features and Compatibility Issues

JavaScript development can be frustrating because of the varying degrees of compatibility between the various browsers in common use today. Again, in most cases the incompatibilities can be attributed to differences in the scriptable object models exposed to JavaScript rather than in the way the language itself works, but the end result is that JavaScript coders have had to work very hard to make certain kinds of scripts work with the major browsers. In particular, the portions of the scripting object model that pertain to Dynamic HTML (DHTML) have become notorious for not behaving the same way from browser to browser, version to version, and platform to platform.

NOTE

> These days, many people use Adobe Flash to perform the same kinds of things that they would otherwise do with DHTML. For this reason, I generally stay away from DHTML-related discussions in this chapter. As you will see, there are still plenty of other reasons to use JavaScript in your Web pages.

The situation got a lot better with the introduction of the Mozilla engine (usd by Netscape 6 and more recently Firefox). Mozilla and Internet Explorer share a comparatively enormous portion of their object models, all controllable via script.

For the sake of clarity and brevity, and because there would be too much to cover in this chapter otherwise, the examples in this chapter assume that you are using a new browser (IE6 or 7, or Firefox). Most of the examples will also work with earlier browsers (beginning, in general, with Netscape 3.0 and IE 4.0). Most JavaScript references and online guides will provide information on the various compatibilities and incompatibilities you need to keep in mind when working with older browsers.

That's the End of the Crash Course

The remainder of the chapter will explain how to perform some common tasks with JavaScript, focusing on situations where JavaScript and ColdFusion need to work together in some respect.

You can use the tables that have appeared up to this point as a mini-reference guide for the various statements and objects used in the remaining sections. Wherever possible, the example listings will only use JavaScript syntax and elements that you have already learned, but in some cases you may need to refer to a full-fledged JavaScript reference to get all the details about a specific object, method, or property. A good place to find online references is `http://msdn.microsoft.com`.

Working with Form Elements

Every Web page containing forms exposes a `document.forms` collection, which is an array of forms on the page. Each `<form>` block in the page can be therefore referred to by number, where `document.forms[0]` refers to the first `<form>` block on the page, `document.forms[1]` refers to the second form, and so on. The number of forms is always available as `document.forms.length`.

If a `<form>` tag includes a `name` attribute, you can also refer to it by name as a property of the `document` object. So, if a page includes a single form that has a `name="SignupForm"` attribute, you can refer to the form as either `document.forms[0]` or `document.SignupForm`.

Each form element (text inputs, select boxes, checkboxes, and so on) within a form can be referred to by name. So, if a form named `SignupForm` contains a text input field with a `name="FirstName"` attribute, then that element can be referred to as `document.forms[0].FirstName` or `document.SignupForm.FirstName`.

Table 48.10 shows the most important properties available for controlling each type of form element via JavaScript, where "most important" means the properties you need to use to obtain or change the element's value. In this table, `element` refers to the element object, so you would replace `element` with `document.forms[0].elementname`, where `elementname` corresponds to the `NAME` attribute of the `<input>`, `<select>`, or `<textarea>` element in question.

NOTE

Always remember that JavaScript is case-sensitive, so if you have an `<input>` with a `name="FirstName"` attribute, you can replace `element` in this table with `document.forms[0].FirstName` but not `document.forms[0].firstname` or `Document.Forms[0].Firstname`.

Table 48.10 Accessing the Value of Form Fields via Script

FORM ELEMENT	DESCRIPTION
Text input fields	To get the current value of a text field, use the `element.value` property. To change the text in a text field, just set `element.value` to the desired value. This applies to elements created with `<input type="Text"> or <textarea>`.
Hidden fields	You can also use `element.value` to get or set the value of hidden fields. Of course, there will be no visual reflection of any change you make to the value, but the new value should will be available to the server if the form is submitted.

Table 48.10 (continued)

FORM ELEMENT	DESCRIPTION
Select boxes	There is no value property for `<select>` elements. Instead, you use `element.selectedIndex` to find which `<option>` is selected; `selectedIndex` is 0 if the first option is selected, 1 if the second option is selected, and so on (if no items are selected, `selectedIndex` is -1). The `element.options` collection is an array of the element's options, each of which has a `text` and `value` property, which means that `element.options [element.selectedIndex].value` corresponds to the `value` attribute of the currently selected `<option>`. Similarly, `element.options [element.selectedIndex].text` is the text between the `<option>` tags for the current selection.
Checkboxes	If the checkbox has a `name` attribute that is unique to the form, you can find out whether it is checked using the `element.checked` Boolean property, and you can access its `value` attribute using `element.value`. If there is more than one checkbox with the same name, then `element` becomes a collection (basically an array), where each element in the array represents one of the checkboxes. The number of checkboxes in the collection is available as `element.length`; `element[0].checked` reflects whether the first checkbox in the group is checked; `element[1].value` holds the second checkbox's value, and so on.
Radio buttons	For purposes of scripting, radio buttons behave just like checkboxes (above).
The form itself	You can discover or change the page the form submits to (that is, the value of the `action` attribute) using the `formelement.action` property. The `formelement.elements` collection is an array of all form elements (text inputs, checkboxes, hidden fields, and so on) contained within the form; you can iterate through this array in a `for` loop if you want to do something to each element or find out what fields are actually in the form at runtime. You can also submit the form programmatically using the `formelement.submit()` method.

So, if a page contains a single form with a `name` attribute of `SignupForm`, and the form contains a text field called `FirstName` and a drop-down list (select box) called `CCType`, then the following line of JavaScript would change the value of the `FirstName` field to "Nate":

```
document.forms[0].FirstName.value = "Nate";
```

The following `if` block would execute only if the first element of the `CCType` dropdown is currently selected, or if no selection has been made at all:

```
if (document["SignupForm"].CCType.selectedIndex < 1) {
 alert("Don't leave the credit card type blank!");
}
```

The following would get the value of the currently selected option in the drop-down list. (In an actual code file, this should be all on one line, but it's too long to print in this book):

```
document["SignupForm"].CCType.options[
 document["SignupForm"].CCType.selectedIndex].value
```

You can use JavaScript's `with` statement to make that last snippet a little easier to type and read, like so:

```
with (document["SignupForm"].CCType) {
 options[selectedIndex].value;
}
```

Note that if the `<select>` allows multiple selections via the `multiple` attribute, you must iterate through the `element.options` array, accessing each option's `selected` property to find out whether it is selected. The number of options is always available as `element.options.length`, so the following snippet would allow you to perform some action for each selected option:

```
with (document["SignupForm"].CCType) {
 for (var i = 0; i < options.length; i++) {
 if ( options[i].selected ) {

 // ...your code here...

 }
 }
}
```

You will see further examples of working with form fields in nearly every example listing in this chapter.

Passing Variables to JavaScript

ColdFusion developers sometimes get confused about how to make the values of ColdFusion variables available to JavaScript. There are a lot of reasons why you might want to do this, as you'll see throughout the examples in this chapter.

There's only one conceptual leap that you need to make, and it's a pretty small one at that. You just need to realize that JavaScript variables are generally created and set within `<script>` blocks in an HTML page. Since those `<script>` blocks can be generated dynamically with ColdFusion (just like any other portion of an HTML page), all you need to do is to output the values of your CFML variables in the correct spots, generally right after the JavaScript `var` keyword.

Passing Numbers to JavaScript

Let's take a look at a bare-bones variable passing scenario. First, take a look at the following lines of code, which creates a JavaScript variable called `userInterestRate`, available to any other JavaScript code in the same page:

```
<script language="JavaScript1.1" type="text/javascript">
 var userInterestRate = 3.95;
</script>
```

As you can see, the initial value of the variable will always be `3.95`. If you wanted to change the initial value of the variable, you would need to edit the page and change the number, right? If you

were just using static HTML pages, the answer is yes. But if you're using ColdFusion, you can just output a CFML variable in place of the `3.95`, like so:

```
<cfoutput>
 <script language="JavaScript1.1" type="text/javascript">
 var userInterestRate = #SESSION.MyInterestRate#;
 </script>
</cfoutput>
```

Or, if you prefer:

```
<script language="JavaScript1.1" type="text/javascript">
 var userInterestRate = <cfoutput>#SESSION.MyInterestRate#</cfoutput>;
</script>
```

In either case, the actual value of the ColdFusion variable called `SESSION.MyInterestRate` will be inserted into the `<script>` block as the HTML code for the page is being sent back to the browser. If the value of `SESSION.MyInterestRate` is `5.32`, say, then the browser will receive this `<script>` block:

```
<script language="JavaScript1.1" type="text/javascript">
 var userInterestRate = 5.32;
</script>
```

The browser's JavaScript interpreter will create the `userInterestRate` variable with the appropriate value. The browser doesn't know that the variable is being passed from ColdFusion; as far as the browser is concerned, there is no difference between this `<script>` block and the first snippet in this section.

One more thing, while we're on the subject. If the `#SESSION.MyInterestRate#` came from a database and the value in the database is currently `NULL`, ColdFusion will generate an empty string in place of the variable when the page is generated. That means the browser will receive this, which won't make any sense to the JavaScript interpreter and will thus cause an error message to be displayed:

```
<script language="JavaScript1.1" type="text/javascript">
 var userInterestRate = ;
</script>
```

There are two easy workarounds. The first is to use ColdFusion's `Val()` function around the variable, as in `#Val(SESSION.MyInterestRate)#`. This will cause ColdFusion to send a value of `0` to the browser when the actual value of the CFML variable is empty or null. That's a fine solution as long as the nature of your application doesn't demand that your scripts know the difference between a zero and a null. The slightly more sophisticated workaround is to send a proper JavaScript value of `null` when the CFML variable is empty, like so:

```
<cfoutput>
 <script language="JavaScript1.1" type="text/javascript">
 <cfif SESSION.MyInterestRate EQ "">
 var userInterestRate = null;
 <cfelse>
 var userInterestRate = #SESSION.MyInterestRate#;
 </cfif>
 </script>
</cfoutput>
```

Passing Strings to JavaScript

The process is only a tiny bit more complicated if you want to pass a string value instead of a number. To create a string variable in JavaScript, you use double or single quotation marks to indicate the beginning and end of the string, much like you do in a `<cfset>` tag. For instance, the next snippet, if included in a static HTML page, would create an additional `userName` variable, with the value of `Belinda Foxile`:

```
<script language="JavaScript1.1" type="text/javascript">
var userInterestRate = 3.95;
var userName = "Belinda Foxile";
</script>
```

So, if you wanted to populate the `userName` variable on the fly, you just refer to your ColdFusion variable between the quote quotation marks, like so:

```
<cfoutput>
 <script language="JavaScript1.1" type="text/javascript">
 var userInterestRate = #SESSION.MyInterestRate#;
 var userName = "#SESSION.MyUserName#";
 </script>
</cfoutput>
```

This is all well and good, unless the ColdFusion variables contain characters that will confuse the JavaScript interpreter. For instance, if the value of the `SESSION.MyUserName` variable itself contains any quotation marks, like `Belinda "Red" Foxile`, the `<script>` block that gets sent to the browser is going to look like this:

```
<script language="JavaScript1.1" type="text/javascript">
var userInterestRate = 3.95;
var userName = "Belinda "Red" Foxile";
</script>
```

Understandably, this is going to confuse the JavaScript interpreter. As far as it can tell, the string seems to end after the space that follows the `Belinda` part. It then doesn't know what to do with the part that begins with `Red`. The embedded quote characters need to be escaped properly so that JavaScript knows that they are meant to be considered part of the string, rather than indicating the end of the string. As a result, JavaScript will display an error message or do some other nasty thing when the page is loaded in a browser.

The solution is to use ColdFusion's excellent and handy `JSStringFormat()` function, which escapes the quote quotation marks for you. It also takes care of any other characters that need to be escaped in JavaScript string literals. `JSStringFormat()`, the string you want to pass to JavaScript. No matter what the string contains, the `JSStringFormat()` function It returns the properly escaped string. So, the CFML snippet to create the `userName` variable would become:

```
<cfoutput>
 <script language="JavaScript1.1">
 var userInterestRate = #SESSION.MyInterestRate#;
 var userName = "#JSStringFormat(SESSION.MyUserName)#";
 </script>
</cfoutput>
```

The browser will receive the following. E, with each quote quotation mark preceded by a backslash character (which is the method of escaping special characters in JavaScript):

```
<script language="JavaScript1.1">
var userInterestRate = 3.95;
var userName = "Belinda \"Red\" Foxile";
</script>
```

Table 48.11 shows a list of characters that `JSStringFormat()` automatically escapes for you. The result is a string that can safely be included in a page as a JavaScript literal (that is, between quote quotation marks) without problems.

Table 48.11 Characters Automatically Escaped by JSStringFormat()

CHARACTER	ESCAPED SEQUENCE
Backslash	`\\`
Carriage return	`\r`
New line	`\n`
Quotation mark (double)	`\"`
Quotation mark (single)	`\'`
Tab	`\t`

Variable Passing Example: Mortgage Calculator

Listing 48.1 shows how to create a JavaScript-powered page that performs computations based on variables passed from ColdFusion. When the page first appears, it asks the user to choose one of three mortgage programs: Low, Medium, or High interest (Figure 48.1). When the user selects a rate, the mortgage calculator appears, where the user can make computations based on the selected interest rate.

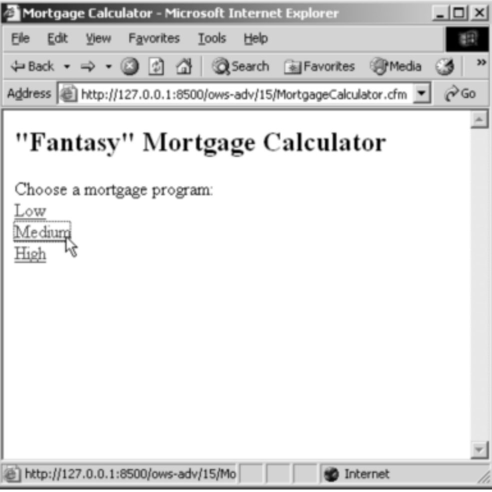


Figure 48.1
First, the user must choose a mortgage rate to pass to the calculator.

In addition to passing variables, this listing also demonstrates how to work with form fields, and how to execute a function when an element is changed or clicked.

Listing 48.1 `MortgageCalculator.cfm`—Passing Variables to JavaScript

```
<!---
Name:        MortgageCalculator.cfm
Author:      Nate Weiss and Ben Forta
Description: Demonstrates passing CFML variables
             to a JavaScript-powered page.
Created:     02/01/05
--->


<!--- User can select between several different interest types --->
<cfparam name="SESSION.MyInterestType"
         type="string"
         default="high">

<!--- If user is changing the interest type --->
<cfif IsDefined("URL.InterestType")>
  <cfset SESSION.MyInterestType=URL.InterestType>
</cfif>


<!--- Assign an actual interest rate, based on the type --->
<!--- (in a real application, this would probably come from a database) --->
<cfswitch expression="#SESSION.MyInterestType#">
  <cfcase value="High">
    <cfset SESSION.MyInterestRate=8.53>
  </cfcase>
  <cfcase value="Medium">
    <cfset SESSION.MyInterestRate=5.12>
  </cfcase>
  <cfcase value="Low">
    <cfset SESSION.MyInterestRate=2.41>
  </cfcase>
  <cfdefaultcase>
    <cfset SESSION.MyInterestType="">
  </cfdefaultcase>
</cfswitch>

<!--- Pass the interest rate and type to JavaScript --->
<cfoutput>
  <script type="text/javascript"
          language="JavaScript">
    var userInterestRate=#SESSION.MyInterestRate#;
  </script>
</cfoutput>


<!--- Use dollarFormat function defined in separate file --->
<script type="text/javascript"
        language="JavaScript"
        src="dollarFormat.js"></script>
```

Listing 48.1 (continued)

```
<script type="text/javascript"
        language="JavaScript">
  // Simple mortgage calculation function
  // ...this probably isn't how real banks do it...  :)
  function calcMortgage(amount, rate, years) {
    var result=amount;

    // Number of months
    var months=years * 12;

    // Actual mortgage calculation would go here
    for (var month=0; month <= months; month++) {
      var thisMonthsInterest=(result / months) * (rate / 12);
      result=result + thisMonthsInterest;
    }

    return result;
  }


  // Wrapper function that performs calcMortgage() based on form input
  // Also validates the form entries
  function calcMortgageBasedOnForm() {
    var lendAmount, numYears, computedResult;
    var numYears;

    // Get the amount of the mortgage from the form
    lendAmount=parseFloat(document.forms[0].LendAmount.value);

    // Get the number of years from the form
    for (var i=0; i < document.forms[0].NumYears.length; i++) {
      if (document.forms[0].NumYears[i].checked) {
        numYears=parseInt(document.forms[0].NumYears[i].value);
        break;
      }
    };

    // Validation: if the number of years and lending amount have been supplied
    if ( (numYears > 0) && (lendAmount > 0) ) {
      // Call calcMortgage() function to get the mortgage cost
      computedResult=calcMortgage(lendAmount, userInterestRate, numYears);

      // Display the computed result to the user
      document.forms[0].ComputedResult.value=dollarFormat(computedResult);

    // If the years or lending amount is blank, clear the computed result
    } else {
      document.forms[0].ComputedResult.value="";
    };
  }
</script>


<html>
<head><title>Mortgage Calculator</title></head>
```

Listing 48.1 (continued)

```
<body>
  <h2>&quot;Fantasy&quot; Mortgage Calculator</h2>

  <!--- Allow user to choose mortgage program --->
  <cfif SESSION.MyInterestType EQ "">
    Choose a mortgage program:<br>
    <!--- For each mortgage program... --->
    <cfloop list="Low,Medium,High"
            index="This">
      <!--- Provide a link to choose the program --->
      <cfoutput>
        <a href="MortgageCalculator.cfm?InterestType=#This#">#This#</a><br>
      </cfoutput>
    </cfloop>


  <cfelse>

    <!--- Use a form to hold the <input> elements --->
    <!--- The "return false" keeps the form from being submitted --->
    <form onsubmit="return false">

      <!--- Text input for lending amount --->
      <p><strong>Amount You Want to Borrow:</strong><br>
      <input type="Text"
             name="LendAmount"
             size="12"
             maxlength="10"
             onchange="calcMortgageBasedOnForm()"
             onkeyup="calcMortgageBasedOnForm()">

      <!--- Radio buttons for length of mortgage --->
      <p><strong>Mortgage Length:</strong><br>
      <input type="Radio"
             name="NumYears"
             value="7"
             onclick="calcMortgageBasedOnForm()">7 years
      <br>
      <input type="Radio"
             name="NumYears"
             value="15"
             onclick="calcMortgageBasedOnForm()">15 years
      <br>
      <input type="Radio"
             name="NumYears"
             value="30"
             onclick="calcMortgageBasedOnForm()">30 years
      <br>

      <!--- Disabled text field for displaying the computed result --->
      <p>Computed Mortgage Cost:<br>
      <input type="Text"
             name="ComputedResult"
             size="10"
             readonly
```

Listing 48.1 (continued)

```
                  style="background:lightgrey">
        </form>

        <!--- Link to choose a different rate --->
        <p>
        <a href="MortgageCalculator.cfm?InterestType=">
        Choose different rate</a>
        <br>
      </cfif>
    </body>
    </html>
```

The first portion of this listing deals with determining a numeric interest rate based upon the rate program (`Low`, `Medium`, or `High`) that the user selects. If the user chooses `Low`, the interest rate will be `0.241`; if they choose `High`, it will be `0.853`. Two `SESSION` variables, `MyInterestType` and `MyIntere-stRate`, are maintained to remember the user's selection. The variables are passed to JavaScript in the first `<script>` block.

Next, an external JavaScript file called `dollarFormat.js` is included with a `<script src>` element. This file creates a function called `dollarFormat()` which behaves like the CFML `DollarFormat()` function (you'll see the function later in Listing 48.2).

Next, a simple function called `calcMortgage()` is created, which accepts three arguments: `amount`, `rate`, and `years`. This author is a bit ashamed to say that he doesn't know how mortgages are actually calculated, but that's not really so important here (surely it's some kind of computation based on the mortgage amount, the interest rate, and the length of time the money will be borrowed). This function simply loops through each month of the mortgage period, adding the current month's interest to the `result` variable. When the loop is finished, the mortgage cost is considered to have been computed and is returned as the function's output.

With the `calcMortgage()` function in place, the next thing to do is to compute mortgages based on the user's form entries. The `calcMortgageBasedOnForm()` function takes care of this task. Conceptually, it's a wrapper around the `calcMortgage()` function. It simply gets the appropriate values from the form elements and passes them to `calcMortgage()`. Getting the `lendAmount` value is easy, since the amount is provided in a simple form field; the only twist is the addition of `parseFloat()` to convert the user's text entry into a floating-point number. Getting the `numYears` value is slightly more complicated, because it is necessary to loop through the `NumYears` array (which represents the radio buttons shown in Figure 48.2) to find which radio button the user selected.

After the `for` loop, the value of `lendAmount` will be a number, unless the form field is blank or if the user entered something that couldn't be converted to a number (in which case `lendAmount` will hold the special JavaScript value of `NaN`, which means "not a number"). The value of `numYears` will hold an integer, unless none of the radio buttons have been selected (in which case it will hold a value of `null`, since no other value has been assigned to it).

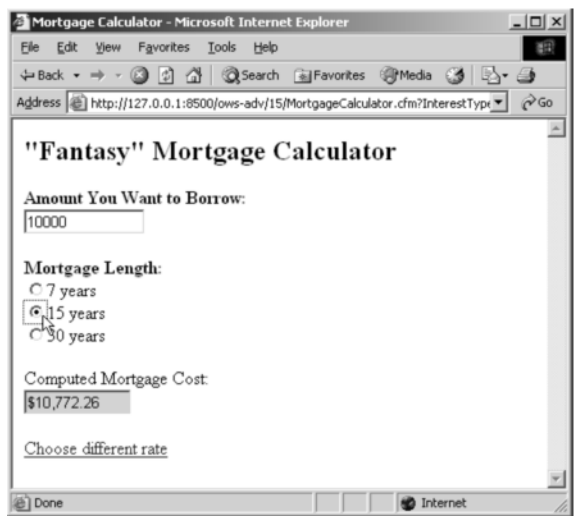


Figure 48.2
Users can make computations based on the interest rate that was passed from ColdFusion.

If both values are greater than zero, the mortgage is calculated using the `calcMortgage()` function. The computed value is placed into the visual `ComputedResult` form field (which has its `readonly` flag set and is colored gray to indicate that the user won't be able to edit the field), formatting it with `dollarFormat()` along the way. If one or both of the input values is not provided, the `ComputedResult` field is cleared.

Within the `<form>` block itself, the `onchange`, `onkeyup`, and `onclick` events of the various form controls are set to execute the `calcMortgageBasedOnForm()` function whenever the user makes a change on the form.

The result is a small JavaScript application that receives a variable from ColdFusion's `SESSION` scope (in this case, the appropriate interest rate), then allows the user to work with the variable interactively. In this case, there is only one variable being passed to JavaScript, but you could easily pass multiple values using the same basic technique. Clearly, the value of the variable could come from a database query or other server-side source.

Listing 48.2 shows the JavaScript code used to create the `dollarFormat()` function. This is just one approach; there are many other ways in which this function could be written.

NOTE

In a situation such as this (when a common utility function is needed that is not available as a built-in method), I would recommend running a few searches at sites like `www.javascript.com`, `www.webreference.com`, or `groups.google.com`. Someone else is likely to have solved the problem before.

Listing 48.2 `dollarFormat.js`—A JavaScript Function for Formatting Numbers

```
/*
  Filename: dollarFormat.js
  Author:   Nate Weiss (NMW)
  Purpose:  Mimics the CFML DollarFormat() function
```

Listing 48.2 (continued)

```
  Note     With IE, you could just use num.toLocaleString() instead
*/

// Utility function that formats a number to a "dollar format"
function dollarFormat(num) {
 var arParts, dollarPart, centPart

 // Split number based on the position of the period character, if any
 arParts = num.toString().split(".");
 // The dollar part is the part before the period
 dollarPart = arParts[0];

 // If there is a cent portion of the number, use it, otherwise use "00"
 if (arParts.length > 1) {
 centPart = (arParts[1] + "00").substr(0,2);
 } else {
 centPart = "00";
 }

 // Reset arParts to an empty array
 arParts = new Array();

 // Number of digits before first comma
 // (but zero if there will be 3 digits before first comma)
 var offset = dollarPart.length % 3;

 // If there should be some digits (other than 3) before first comma,
 // add an element to the array with that number of digits in it
 if (offset > 0) {
 arParts[0] = dollarPart.substr(0, offset);
 }

 // For all remaining groups of three digits, add additional
 // elements to the array with the next 3 digits in it
 for (i = offset; i < dollarPart.length; i = i + 3) {
 arParts[arParts.length] = dollarPart.substr(i, 3);
 };

 // Join the array back into a string, separated by commas
 dollarPart = arParts.join(",")

 // Return the various parts, concatenated together
 return "$" + dollarPart + "." + centPart;
};
```

Passing Arrays to JavaScript

So far, you have only learned how to pass simple values from ColdFusion to JavaScript. Passing multifaceted data, like arrays and structures, can be nearly as straightforward. It can also get complicated. Later in this chapter, you will learn how to use the `<cfwddx>` tag to pass any type of variable to JavaScript, no matter how complex.

Let's start simple. You can pass arrays of numbers to JavaScript using the `ArrayToList()` function (assuming the CFML array is one-dimensional), like so:

```
<cfoutput>
 <script type="text/javascript" language="JavaScript1.1">
 var userInterestRate = #SESSION.MyInterestRate#;
 var userName = "#SESSION.MyUserName#";
 var primeArray = new Array(#ArrayToList(MyPrimeNumberArray)#);
 </script>
</cfoutput>
```

When this code is received by the browser, it will look like this, assuming that the ColdFusion `MyPrimeNumberArray` array has already been populated with the first few prime numbers:

```
 <script type="text/javascript" language="JavaScript1.1">
 var userInterestRate = 3.95;
 var userName = "Belinda \"Red\" Foxile";
 var primeArray = new Array(1,3,5,7,9,11,13);
 </script>
```

NOTE

It's worth pointing out that JavaScript arrays start at 0 (instead of 1, as in ColdFusion). So, in your JavaScript code, you would refer to the first element of the array as `primeArray[0]`, the second element as `primeArray[1]`, and so on.

If you want to pass an array of strings, you can use the `ListQualify()` function. For instance:

```
<cfoutput>
 <script language="JavaScript1.1">
 var userInterestRate = #SESSION.MyInterestRate#;
 var userName = "#SESSION.MyUserName#";
 var nameArray = new Array(#ListQualify(ArrayToList(NameArray), """")#);
 </script>
</cfoutput>
```

If the strings in the array might contain quote quotation marks or the other special characters listed in Table 48.11, you could add `JSStringFormat()` around the `ArrayToList()` part (but inside the `ListQualify()` part).

But even this won't be perfect, because if the array contains any empty strings, those array elements will be missing from the JavaScript array, due to the way ColdFusion's string functions deal with consecutive delimiter characters. You can solve this problem by looping through the array, populating each of its elements manually, like so:

```
<cfoutput>
 <script language="JavaScript1.1">
 var userInterestRate = #SESSION.MyInterestRate#;
 var userName = "#SESSION.MyUserName#";
 var nameArray = new Array();

 <!--- Populate the nameArray variable --->
 <cfloop from="1" to="#ArrayLen(CFMLNameArray)#" index="i">
 nameArray[#Val(i - 1)#] = "#JSStringFormat(CFMLNameArray[i])#";
 </cfloop>
 </script>
</cfoutput>
```

The browser would receive the following (whitespace notwithstanding):

```
<script type="text/javascript" language="JavaScript1.1">
var userInterestRate = 3.95;
var userName = "Belinda \"Red\" Foxile";
var nameArray = new Array();

nameArray[0] = 1;
nameArray[1] = 3;
nameArray[2] = 5;
nameArray[3] = 7;
nameArray[4] = 9;
nameArray[5] = 11;
nameArray[6] = 13;
</script>
```

When dealing with a potentially complex object such as an array (which could contain other arrays, or structures, or record sets), it is usually a lot easier to simply use the `<cfwddx>` tag as shown below. This will automatically generate JavaScript code that is functionally equivalent to the snippet shown above:, like so:

```
<cfoutput>
 <script language="JavaScript1.1">
 var userInterestRate = #SESSION.MyInterestRate#;
 var userName = "#SESSION.MyUserName#";
 var <cfwddx action="CFML2JS"
             input="#CFMLNameArray#"
             toplevelvariable="nameArray">
 </script>
</cfoutput>
```

Contrary to the way you learned to use `<cfwddx>` in Chapter 47, "Using WDDX," online, this form of `<cfwddx>` doesn't have anything to do with WDDX packets or XML. See the "Using JavaScript with WDDX" section at the end of this chapter for details about using `<cfwddx>` in this fashion.

Passing Structures to JavaScript as Objects

The JavaScript `Object` data type corresponds fairly closely to ColdFusion's structure type. If you have a CFML structure called `MyStruct` which contains simple string data, you could use the following loop to re-create the corresponding object in JavaScript:

```
<cfoutput>
 <script language="JavaScript1.1">
 var userInterestRate = #SESSION.MyInterestRate#;
 var userName = "#SESSION.MyUserName#";
 var myObject = new Object();

 <!--- Populate the myObject variable --->
 <cfloop collection="#MyStruct#" item="ThisKey">
 myObject["#ThisKey#"] = "#JSStringFormat(MyStruct[ThisKey])#";
 </cfloop>
 </script>
</cfoutput>
```

NOTE

Again, you could do this with one step using `<cfwddx>` as discussed in the latter portion of this chapter. I'm showing how to do it the "manual" way so that you get a sense of how Objects and Structures are related conceptually.

For instance, if `MyStruct` contains three values for Belinda, Ben, and Nate, the resulting `<script>` block that gets sent to the browser might look like this:

```
<script language="JavaScript1.1">
var userInterestRate = 3.95;
var userName = "Belinda \"Red\" Foxile";
var myObject = new Object();

myObject["Belinda"] = "Teen Pop Superstar";
myObject["Ben"] = "ColdFusion Superstar";
myObject["Nate"] = "Belinda's Biggest Fan";
</script>
```

Passing Enough Data to Relate Two Select Boxes

One JavaScript-powered trick that can be effective in ColdFusion applications is the notion of "cascading" select lists. A long time ago now, this author wrote a simple CFML Custom Tag called `<CF_TwoSelectsRelated>`, which causes two drop-down lists to appear on the page, filled with data from a query. When the user chooses an option in the first select list, the second select list fills with the appropriate data, without having to reload the page.

I continue to be surprised by the number of people who use this custom tag. Many people write to me asking for one enhancement or modification or another, thinking that it would be really hard to make the changes themselves. Examining a few different approaches to connecting the select lists will be an interesting way to learn more about how to pass complex, multifaceted data from ColdFusion to JavaScript, and how the data can be used in JavaScript once it gets there. I continue to be surprised by the number of people who use this custom tag.

Examining a few different approaches to connecting the select lists will be an interesting way to learn more about how to pass complex, multifaceted data from ColdFusion to JavaScript, and how the data can be used in JavaScript once it gets there. Hopefully, the discussion will also serve to demystify what it actually takes to create interactive pages that use ColdFusion and JavaScript together.

Cascading Selects, Approach #1: Creating an Array of Films

Listing 48.3 shows one approach to solving this problem. The page contains a single form with two select lists: one for ratings, and the other for corresponding films. When the page first appears, no rating is selected and no films are displayed (Figure 48.3).

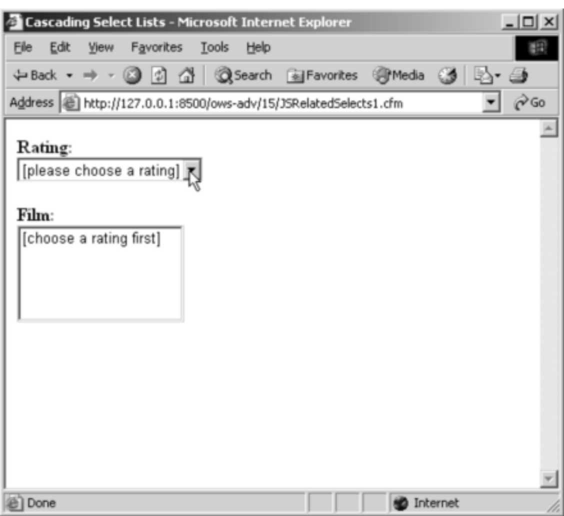


Figure 48.3

Users can choose ratings from the first select list to view matching films.

When the user selects a rating from the first list, the films that match that rating magically appear in the second list (Figure 48.4). People love this because it makes really efficient use of real estate, and because the "lookup" operation executes more or less instantly, without the need for a page refresh.

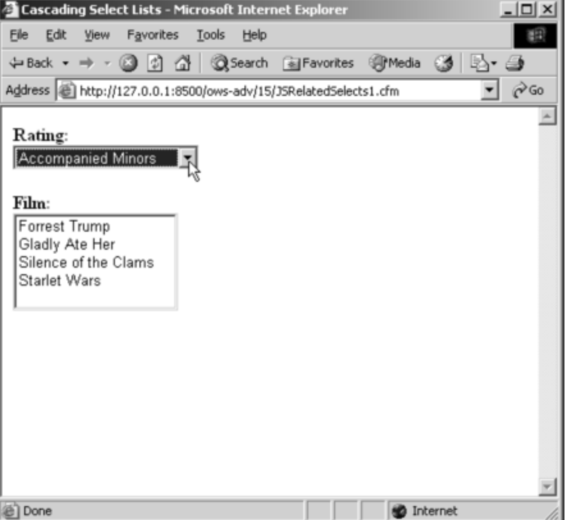


Figure 48.4

When a rating is selected, corresponding films appear in the second list.

Listing 48.3 `JSRelatedSelects1.cfm`—Using a Loop to Create an Array of Films on the Fly

```
<!---
Name:        JSRelatedSelects1.cfm
Author:      Nate Weiss and Ben Forta
Description: Demonstrates one approach to relating
             select lists via JavaScript, using
             film and rating data from the ows
             example database.
Created:     02/01/05
--->


<!--- Get film data from database --->
<cfquery name="FilmsQuery"
         datasource="ows">
  select FilmID, MovieTitle, RatingID
  FROM Films
  WHERE RatingID IS NOT NULL
  ORDER BY RatingID, MovieTitle
</cfquery>

<!--- Get rating data from database --->
<cfquery name="RatingsQuery"
         datasource="ows">
  select RatingID, Rating
  FROM FilmsRatings
  ORDER BY RatingID
</cfquery>


<html>
<head>
<title>Cascading Select Lists</title>

<!--- Custom JavaScript code --->
<script type="text/javascript" language="JavaScript">
  // Create a new array to hold film "objects"
  var arFilms=new Array;

  <!--- For each film... --->
  <cfoutput query="FilmsQuery">
    // Create a JavaScript object for this film
    var oFilm=new Object;
    oFilm.filmid=#FilmsQuery.FilmID#;
    oFilm.ratingid=#FilmsQuery.RatingID#;
    oFilm.movieTitle="#JSStringFormat(MovieTitle)#";
    // Append the object to the end of the array of films
    arFilms[arFilms.length]=oFilm;
  </cfoutput>


  // This function fills the second select box based on the first box's value
  function fillFilms() {
    // Get the currently selected rating ID from the first select box
    with (document.forms[0].RatingID) {
      var ratingid=options[selectedIndex].value;
```

Listing 48.3 (continued)

```
    }

    // Stop here if there is no selected rating
    if (ratingid == null) {
      return;
    }

    // Remove all options from the second select box
    document.FilmForm.FilmID.options.length=0;

    // For each item in the films array...
    for (var i=0; i < arFilms.length; i++) {

      // If the film's rating is the same as the currently selected rating...
      if (arFilms[i].ratingid == ratingid) {
        // Create a new visual <option> to place in the second select box
        var objOption=new Option(arFilms[i].movieTitle, arFilms[i].filmID);

        // Place the new option in the second select box
        with (document.FilmForm.FilmID) {
          options[options.length]=objOption;
        }
      }
    };
  };
</script>

</head>
<body>

<!--- Ordinary html form --->
<form action="ShowFilm.cfm"
      name="FilmForm"
      method="Post">

  <!--- First select box (displays ratings) --->
  <strong>Rating:</strong><br>
  <select name="RatingID" onchange="fillFilms()">
    <option>[please choose a rating]
    <cfoutput query="RatingsQuery">
      <option value="#RatingID#">#Rating#
    </cfoutput>
  </select>

  <!--- Second select box (displays films) --->
  <p><strong>Film:</strong><br>
  <select name="FilmID" size="5">
    <option>[choose a rating first]
  </select><br>

</form>

</body>
</html>
```

Let's start with the <form> portion of this listing, near the bottom. As you can see, this is a fairly ordinary HTML form, which uses normal <select> tags to create two select lists named RatingID and FilmID. A <cfoutput> block is used to fill the first list with options for each record in the Ratings table (from the ows example database). The second list is left empty, except for a single option that tells the user to choose a film first (see Figure 48.3). The only thing out of the ordinary is in the first <select> list: it contains an onchange attribute that tells JavaScript to execute a function called fillFilms() when the user makes a rating selection.

Now look at the <script> block at the top of the template. The first thing it does is to create a JavaScript array called arFilms. This array will be filled with JavaScript objects (which, remember, are similar to structures in CFML), where each object represents one film.

The next block uses <cfoutput> to loop over the film records in the FilmsQuery query. This is the part that creates an object for each film, appending each object to the arFilms array as it goes. This may look a bit confusing when you first see it, because JavaScript syntax and ColdFusion syntax appear to be co-mingled. The thing to keep in mind is that the ColdFusion syntax will execute on the server before the page is sent to the browser. The browser will only receive the remaining, dynamically generated JavaScript code. For instance, depending on the actual film information in the database, the browser will receive JavaScript code similar to the following (there will one such chunk for each film):

```
// Create a JavaScript object for this film
var oFilm = new Object;
oFilm.filmid = 16;
oFilm.ratingid = 1;
oFilm.movieTitle = "West End Story";
// Append the object to the end of the array of films
arFilms[arFilms.length] = oFilm;
```

So, after the <cfoutput> block has finished its work and the browser's JavaScript interpreter parses the script code, the browser's memory will contain an arFilms array that contains the ID number, title, and rating for each film. The rating of the first film is available as arFilms[0].ratingid, the title of the second film is arFilms[1].movieTitle, and so on. The ID number of the last film in the array could be accessed as arFilms[arFilms.length].filmid (or, if you prefer, arFilms[arFilms.length]["filmID"]).

The next part of the listing is the code for the fillForms() function (which executes when the user changes the selection in the first select list). First, the selected rating is obtained by getting the value attribute of the selected <option> from the <select> named ratingid. If no ratingid is currently selected, the function exits immediately.

Next, any existing options are removed from the second select list by setting the length of its options collection to 0. Then a for loop is used to iterate through the arFilms array; within the loop, the "current" film can be referred to as arFilms[i]. The if test checks to see if the current film's rating is the same as the currently selected ratingid. If so, a new Option object is created which displays the title of the film and has the film's ID number as its value. The new option is then added to the second select list by appending it to its options collection.

NOTE

Creating a new `Option` object is like creating an HTML `<option>` tag out of thin air. The first argument for `Option`'s constructor is the text that should be displayed for the option; the second argument is the value of the option (what gets sent to the server when the form is submitted). `Option` objects are only meant to be used for `<select>` elements. Consult a JavaScript reference for all the gory details.

Cascading Selects, Approach #2: Using a Custom JavaScript Object

Listing 48.4 shows a slightly different approach to the same problem. Rather than filling the `arFilms` array with instances of the generic `Object` type (which is like a CFML structure), it fills the array with instances of a custom object type called `Film`. The page behaves the same way as the previous listing (see Figure 48.3 and Figure 48.4).

Listing 48.4 `JSRelatedSelects2.cfm`—Using Instances of Custom JavaScript Objects to Hold Data

```
<!---
Name:        JSRelatedSelects2.cfm
Author:      Nate Weiss and Ben Forta
Description: Demonstrates one approach to relating
             select lists via JavaScript, using
             film and rating data from the ows
             example database.
Created:     02/01/05
--->


<!--- Get film data from database --->
<cfquery name="FilmsQuery"
         datasource="ows">
  select FilmID, MovieTitle, RatingID
  FROM Films
  WHERE RatingID IS NOT NULL
  ORDER BY RatingID, MovieTitle
</cfquery>

<!--- Get rating data from database --->
<cfquery name="RatingsQuery"
         datasource="ows">
  select RatingID, Rating
  FROM FilmsRatings
  ORDER BY RatingID
</cfquery>


<html>
<head>
<title>Cascading Select Lists</title>

<!--- Custom JavaScript code --->
<script type="text/javascript"
        language="JavaScript">
  // Create a new array to hold film "objects"
  var arFilms=new Array;
```

Listing 48.4 (continued)

```
  // Define a custom JavaScript object type to represent a single film
  function Film(filmID, ratingID, movieTitle) {
    this.filmid=filmID;
    this.ratingid=ratingID;
    this.movieTitle=movieTitle;
    this.makeOption=makeOption;
  }

  // This function becomes the makeOption() method of every Film object
  function makeOption() {
    return new Option(this.movieTitle, this.filmID);
  };


  <!--- For each film, append a new Film object to the array of films --->
  <cfoutput query="FilmsQuery">
    arFilms[arFilms.length]=
      new Film(#FilmID#, #RatingID#, "#JSStringFormat(MovieTitle)#");
  </cfoutput>


  // This function fills the second select box based on the first box's value
  function fillFilms() {
    // Get the currently selected rating ID from the first select box
    with (document.forms[0].RatingID) {
      // Stop here if there is no selected rating
      if (selectedIndex == -1) {
        return;
      }

      var ratingid=options[selectedIndex].value;
    }


    // Remove all options from the second select box
    document.FilmForm.FilmID.options.length=0;

    // For each item in the films array...
    for (var i=0; i < arFilms.length; i++) {

      // If the film's rating is the same as the currently selected rating...
      if (arFilms[i].ratingid == ratingid) {
        // Place a new option in the second select box
        with (document.FilmForm.FilmID) {
          options[options.length]=arFilms[i].makeOption();
        }
      }
    };
  };
</script>

</head>

<!--- Call the fillFilms() function when the page first appears --->
<body onload="fillFilms()">
```

Listing 48.4 (continued)

```
<!--- Ordinary html form --->
<cfform action="ShowFilm.cfm"
        name="FilmForm"
        method="Post">

  <!--- First select box (displays ratings) --->
  <strong>Rating:</strong><br>
  <cfselect name="RatingID"
            query="RatingsQuery"
            value="RatingID"
            display="Rating"
            onchange="fillFilms()"/><br>

  <!--- Second select box (displays films) --->
  <p><strong>Film:</strong><br>
  <cfselect name="FilmID"
            size="5"
            style="width:300px"/><br>

  <!--- Submit button --->
  <cfinput name="sbmt" type="Submit">
</cfform>

</body>
</html>
```

The main addition in this listing is the `Film()` function, near the top of the `<script>` block. This function isn't meant to be called normally; it's meant to be called with the `new` keyword to create instances of needed objects. Unfortunately, there isn't space here to explain everything about creating custom objects in JavaScript. Here are the basics:

- The idea is that any function can be called with the `new` keyword, which means "new instance of". Using object-oriented terminology, the function is now a *constructor* for a new object type, or *class*.
- Functions that are called in this way can use the `this` keyword to track data about each instance of the object.
- Arguments passed to the function can be stored in the `this` scope; these values will be stored separately for each instance of the object. Using object-oriented terminology, they are now the object's *properties*.
- You can also add functions to a JavaScript object, in which case the functions can be called as *methods* of the object. The body of the method's code can also refer to the `this` scope to access or manipulate that object's instance level data.

NOTE

JavaScript's custom object types are similar conceptually to ColdFusion Components (CFCs) that hold instance-level data (as discussed in Chapter 27, "Creating Advanced ColdFusion Components," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 2: Application Development*). You'll notice that both custom JavaScript objects and CFCs use the word `this` to represent the data tracked by each instance of the object being created.

So, the `function Film()` block in this listing creates a new object type called `Film` with `filmID`, `ratingID`, and `movieTitle` properties, and a single method called `makeOption()`. A new instance of the `Film` class can be created like so, if its ID number is 50 and its rating is 2:

```
var myInstance = new Film(50, 2, "Stuart Spittle");
```

Once created, its properties can be accessed as `myInstance.filmID` and `myInstance.movieTitle`. In this example, the values of these properties never change; they simply retain the values provided when the instance is created.

The `makeOption()` method returns an `Option` object (as discussed in the text after Listing 48.3). Because `this.filmID` and `this.movieTitle` are used when creating the new `Option`, calling an instance's `makeOption()` method always creates an option that contains that instance's film data. Conceptually, each instance knows how to describe itself in the form of a visual drop-down option.

The `makeOption()` method can be called like so:

```
var myNewOption = myInstance.makeOption();
```

Or, to create a new method and make it appear in a select list all at once, where element is a reference to the `<select>` element:

```
element.options[element.options.length] = myInstance.makeOption();
```

With the new `Film` object type in place, the remaining changes between Listing 48.3 and Listing 48.4 are pretty straightforward. The `<cfoutput query="FilmsQuery">` portion is now shorter, because it simply calls the `Film` object's constructor to create each object to store in the `arFilms` array. The last line of the `<script>` block, which creates the visual option for each film that matches the current rating selection, now calls the `makeOption()` method instead of creating an `Option` object on its own.

Again, the end user's experience is not any different from that of Listing 48.3. What this listing aims to demonstrate is how you can easily use JavaScript's object- oriented programming metaphors (such as `this` and `new`, and the concepts of classes and instances) even when the data is being "passed" from ColdFusion. This allows you to write JavaScript code that uses objects in much the same way that your ColdFusion code might use CFCs.

NOTE

This version of the page also uses `<cfform>` instead of `<form>` and `<cfselect>` instead of `<select>` to create the form itself. I made this change mainly to prove that the browser doesn't care which tags you use, since what it receives from ColdFusion contains the same HTML tags either way.

NOTE

This version of the page calls the `fillFilms()` function in the `<body>` tag's `onload` event, which means that the second list is filled right away when the form first appears.

Passing Data to JavaScript Using `<cfwddx>`

So far in this chapter, you have seen how to pass variables from ColdFusion to JavaScript by generating the appropriate script code on the fly, using `<cfoutput>` and other basic CFML tags. The listings have worked out just fine; there is nothing wrong with this approach. For lack of a better term, I'm going to call this the *roll-your-own* approach to passing variables to JavaScript.

ColdFusion provides a higher-level tool for passing variables to JavaScript: the `CFML2JS` action of the `<cfwddx>` tag. You were introduced to `<cfwddx>` in Chapter 47. That chapter explained how to use the tag to serialize and deserialize data in WDDX packets (a form of XML).

This section will focus on a completely different use for `<cfwddx>`, which is passing variables to JavaScript. This use of `<cfwddx>` has nothing directly to do with WDDX packets or XML. While it is consistent with WDDX's overall mission (to make it really easy to transfer data from place to place), it is not really about the WDDX format per se.

Instead of producing an XML version of a ColdFusion variable or value, this form of `<cfwddx>` produces a chunk of JavaScript code that, when executed by the browser's interpreter, re-creates the variable or value in the browser's memory. In other words, instead of converting your data into a WDDX packet, this method converts your data to a bunch of JavaScript code.

Using this Form of `<cfwddx>`

Table 48.12 shows the `<cfwddx>` syntax for sending data to JavaScript. As you can see, most of the attributes are common to the way the tag is used to produce WDDX packets (see Chapter 47).

Table 48.12 `<cfwddx>` Syntax for Passing Values to JavaScript

ATTRIBUTE	DESCRIPTION
`action`	Required. Set this attribute to `CFML2JS` to pass CFML variables to JavaScript. The other actions are for working with WDDX (XML) packets, which is a whole different topic (see Chapter 47 for details).
`input`	Required. The CFML value that you want to pass to JavaScript, surrounded by `#` signs. Can be just about any data variable, including arrays, recordsets, and structures.
`toplevelvariable`	Required. The name for the value after it is passed to JavaScript. You will use this name in your client-side script code to refer to the passed-in data. Remember that JavaScript is case-sensitive, so your script code needs to use the same exact name that you provide here.
`output`	Optional. If you provide this attribute, then a new CFML variable will be created which contains the generated JavaScript code (as a string). It is then your responsibility to output the value of this string within a `<script>` block so that it gets to the browser. If you omit this attribute, the generated JavaScript code is inserted into the current document, right where the `<cfwddx>` tag appears (it is assumed that the `<cfwddx>` tag is already positioned within a `<script>` block).

So, to transfer a ColdFusion variable called `MyValue` to JavaScript, you use code similar to the following:

```
<script language="JavaScript">
 <cfwddx action="CFML2JS"
        input="#MyValue#"
        toplevelvariable="myValueFromCF">
</script>
```

If the value of `MyValue` is a string, the code received by the browser will be similar to the following:

```
<script language="JavaScript">
 myValueFromCF = "Hello, World!";
</script>
```

If `MyValue` holds an array, the code received by the browser might be similar to this (depending on the actual data in the array, of course):

```
<script language="JavaScript">
 myValueFromCF = new Array();
 myValueFromCF[0] = "Belinda";
 myValueFromCF[1] = "Ben";
 myValueFromCF[0] = "Nate";
</script>
```

Most commonly, the `<cfwddx>` tag is placed between `<script>` tags (as shown in the first snippet). Sometimes you may prefer to store the JavaScript code in a string variable, outputting it within a `<script>` tag later in your ColdFusion page. Just use the `output` attribute to hold the generated code, like so:

```
<!--- Convert MyValue to JavaScript code --->
<cfwddx action="CFML2JS"
        input="#MyValue#"
        output="GeneratedJS"
        toplevelvariable="myValueFromCF">
<!--- Output generated JavaScript code --->
<script language="JavaScript">
 <cfoutput>#GeneratedJS#</cfoutput>
</script>
```

NOTE

If you were to output the value of `GeneratedJS` without surrounding it with `<script>` tags, the browser would just display the JavaScript code as text, rather than parsing and understanding the code.

In any case, the value you supply to `input` can be arbitrarily complex; it could be a structure that contains numerous arrays, each of which holds an arbitrary number of smaller structures. In fact, you'll see that happen in the next section.

NOTE

If the value you supply to `input` contains a query recordset object (or objects), you must use an additional `<script>` tag to include the `wddx.js` file. The file teaches JavaScript how to deal with recordsets, something it doesn't understand out of the box. For details, please see the "Working with WddxRecordset Objects" section later in this chapter.

Is This Serialization?

I find it interesting to note that both types of output (the WDDX packet and the JavaScript code) are text-only representations of the original data. As such, both operations could be said to *serialize* the data, and the JavaScript interpreter on the browser machine could be said to *deserialize* the data.

Think about the steps in a "normal" WDDX scenario, as you learned in Chapter 47:

1. Data is serialized into a WDDX packet, using `<cfwddx action="CFML2WDDX">`.
2. The packet is passed to another application, environment, or process.
3. The packet is deserialized by `<cfwddx>` or some other WDDX-aware application, effectively transferring the data to the new location.

Now think about the steps when using `<cfwddx>` in this new form:

1. Data is converted into JavaScript code, using `<cfwddx action="CFML2JS">`.
2. The JavaScript code is sent to the browser as part of a Web page.
3. The code is executed by the browser's JavaScript interpreter, effectively transferring the data to the browser's memory.

The steps are pretty similar conceptually, apart from the way the data looks while in its serialized state. The first scenario uses XML as the serialization format, and the second uses JavaScript code. Of course, the XML form has more uses, since it can be unpacked by any WDDX-aware application (or even just an XML-aware one).

Just something to think about!

Cascading Selects, Approach #3: Passing the Data Via `<cfwddx>`

Listing 48.5 demonstrates a third approach to the cascading select boxes problem. This approach assembles a CFML structure of ratings and their corresponding films (the structure is assembled on the server). Then the `<cfwddx>` tag is used to "pass" the structure to JavaScript in a single step. The JavaScript code uses this data to populate the second select list. The end-user experience is once again the same (as shown in Figure 48.4 above).

Listing 48.5 `JSRelatedSelects3.cfm`—Using `<cfwddx>` to Supply Data for the Second Select List

```
<!---
Name:        JSRelatedSelects3.cfm
Author:      Nate Weiss and Ben Forta
Description: Demonstrates one approach to relating
             select lists via JavaScript, using
             film and rating data from the ows
             example database.
Created:     02/01/05
--->
```

Listing 48.5 (continued)

```
<!--- Get film data from database --->
<cfquery name="FilmsQuery"
         datasource="ows">
  select FilmID, MovieTitle, RatingID
  FROM Films
  WHERE RatingID IS NOT NULL
  ORDER BY RatingID, MovieTitle
</cfquery>

<!--- Get rating data from database --->
<cfquery name="RatingsQuery"
         datasource="ows">
  select RatingID, Rating
  FROM FilmsRatings
  ORDER BY RatingID
</cfquery>

<!--- Create a new structure to hold the ratings --->
<cfset RatingsStruct=StructNew()>

<!--- For each rating returned by the database... --->
<cfloop query="RatingsQuery">

  <!--- We are currently working with this film rating --->
  <cfset ThisRatingid=RatingsQuery.RatingID>

  <!--- Create a new value in the structure, which is an array of films --->
  <cfset RatingsStruct[RatingID]=ArrayNew(1)>

  <!--- Use an in-memory-query to get the films with this rating --->
  <cfquery dbtype="query"
           name="MatchingFilms">
    select * FROM FilmsQuery
    WHERE Ratingid=#ThisRatingID#
  </cfquery>

  <!--- For each matching film... --->
  <cfloop query="MatchingFilms">
    <!--- Create a new structure to hold the film's ID and title --->
    <cfset FilmStruct=StructNew()>
    <cfset FilmStruct.filmid=MatchingFilms.FilmID>
    <cfset FilmStruct.movietitle=MatchingFilms.MovieTitle>

    <!--- Append the structure to the array for this rating --->
    <cfset ArrayAppend(RatingsStruct[RatingID], FilmStruct)>
  </cfloop>
</cfloop>



<html>
  <head>
<title>Cascading Select Lists</title>

<!--- Custom JavaScript code --->
```

Listing 48.5 (continued)

```
<script type="text/javascript"
        language="JavaScript">

  <!--- Output the JavaScript code needed to create a JavaScript object --->
  <!--- called objRatings that holds the same data as RatingsStruct --->
  <cfwddx action="CFML2JS"
          input="#RatingsStruct#"
          toplevelvariable="objRatings">

  // This function fills the second select box based on the first box's value
  function fillFilms() {
    // Get the currently selected rating ID from the first select box
    with (document.forms[0].RatingID) {
      // Stop here if there is no selected rating
      if (selectedIndex == -1) {
        return;
      }

      var ratingID=options[selectedIndex].value;
    }


    // Remove all options from the second select box
    document.FilmForm.FilmID.options.length=0;

    // Grab the appropriate array of films from the objRatings object
    var arFilms=objRatings[ratingID];

    // For each item in the films array...
    for (var i=0; i < arFilms.length; i++) {

      // Place a new option in the second select box
      with (document.FilmForm.FilmID) {
        options[options.length]=
          new Option(arFilms[i].movietitle, arFilms[i].filmid);
      }
    };
  };
</script>

</head>

<!--- Call the fillFilms() function when the page first appears --->
<body onload="fillFilms()">


<!--- Ordinary html form --->
<cfform action="ShowFilm.cfm"
        name="FilmForm"
        method="Post">

  <!--- First select box (displays ratings) --->
  <strong>Rating:</strong><br>
  <cfselect name="RatingID"
            query="RatingsQuery"
```

Listing 48.5 (continued)

```
              value="RatingID"
              display="Rating"
              onchange="fillFilms()"/><br>

   <!--- Second select box (displays films) --->
   <p><strong>Film:</strong><br>
   <cfselect name="FilmID"
              size="5"
              style="width:300px"/><br>

   <!--- Submit button --->
   <cfinput type="Submit"
             name="sbmt">
</cfform>

</body>
</html>
```

After the queries at the top of the page, a new CFML structure called `RatingStruct` is created. Then, within a `<cfloop>` that loops over each rating in the database, a new array is created with `ArrayNew()` and stored in `RatingsStruct` (using the rating ID as the name). Next, an in-memory-query is used to get the films for the current rating, and an inner `<cfloop>` is used to loop over each of the films. Within this inner loop, a structure is created to represent the film, holding the film's ID number and title. This film structure is then appended to the end of the array for the current rating.

NOTE

In other words, when the nested loops have finished executing, `RatingStruct` will contain an array of films for each rating. The arrays will be filled with smaller structures that contain the ID and title for each film. You could access the title of the second film with a rating of 5 using `RatingStruct[5][2].MovieTitle`.

With the `RatingsStruct` structure in place, it can be passed to JavaScript in one step using the `<cfwddx>` tag. In this listing, the `toplevelvariable` attribute establishes that the data shall be known to JavaScript as an object named `objRatings`. If you use your browser's View Source command, you can see the JavaScript code that `<cfwddx>` generates to re-create the structure on the client. Here's an excerpt:

```
objRatings = new Object();
objRatings["1"] = new Array();
objRatings["1"][0] = new Object();
objRatings["1"][0]["movietitle"] = "Charlie's Devils";
objRatings["1"][0]["filmid"] = 2;
objRatings["1"][1] = new Object();
objRatings["1"][1]["movietitle"] = "Four Bar-Mitzvahs and a Circumcision";
objRatings["1"][1]["filmid"] = 4;
...
objRatings["4"] = new Array();
objRatings["4"][0] = new Object();
objRatings["4"][0]["movietitle"] = "Ground Hog Day";
objRatings["4"][0]["filmid"] = 7;
objRatings["4"][1] = new Object();
objRatings["4"][1]["movietitle"] = "Raiders of the Lost Aardvark";
```

```
objRatings["4"][1]["filmid"] = 17;
objRatings["4"][2] = new Object();
objRatings["4"][2]["movietitle"] = "The Sixth Nonsense";
objRatings["4"][2]["filmid"] = 15;
```

As you can see, there's no magic here. This JavaScript isn't any different conceptually from the JavaScript that the previous versions of this example generated. What's cool is that `<cfwddx>` did it all for us, automatically!

The remainder of Listing 48.5 is very similar to the previous versions of this example (Listing 48.4 and Listing 48.3). Within this version's `fillFilms()` function, the JavaScript code is able to obtain an array of films for a particular rating using `objRatings[ratingID]`. This returns the JavaScript equivalent of the array that was created for the rating in the CFML portion of the listing. Because each element of the array contains an object that in turns holds `filmid` and `movietitle` properties, the appropriate JavaScript elements are created.

NOTE

Because ColdFusion is not case-sensitive (but JavaScript is), structure key names are always re-created in JavaScript using lower case. That's why the JavaScript portion of the code must refer to `arFilms[i].movietitle` instead of `arFilms[i].MovieTitle` or `arFilms[i].movieTitle`. Whenever you use `<CFWDDX>` to pass structures to JavaScript, the resulting JavaScript objects will use lower case for the property names. For this reason, I recommend that you use lowercase names when building the corresponding CFML structures (like this listing does when assigning values to `FilmStruct`). Otherwise, it can be confusing to use one spelling in the CFML portion of your code and the all-lowercase spelling in the JavaScript portion.

Working with `WddxRecordset` Objects

Most ColdFusion data types correspond to JavaScript data types rather neatly (strings become strings in JavaScript, dates become `Date` objects, arrays become `Array` objects, and so on). However, the ColdFusion notion of a recordset (or query object) has no obvious counterpart in JavaScript. For this reason, a simple, lightweight implementation of a JavaScript-based recordset object is included with ColdFusion. The object is called `WddxRecordset`. Whenever you use `<cfwddx>` to pass a query object to JavaScript, the generated JavaScript code will construct a new instance of this object.

Including the `wddx.js` JavaScript File

The `WddxRecordset` object is implemented in a file called `wddx.js`, which is automatically installed when you install ColdFusion. You must include the file with a set of `<script>` tags as discussed in this section. Otherwise, the browser will not understand what a `WddxRecordset` is, and you will likely see an error message reporting that "WddxRecordset is undefined" or something similar.

By default, the file is located in the `CFIDE/scripts` folder within your Web server's document root. This means that you can use a line like the following to properly include the `wddx.js` file:

```
<!--- Include wddx.js, located in the /CFIDE/scripts folder --->
<script language="JavaScript" src="/CFIDE/scripts/wddx.js"></script>
```

If the `wddx.js` file has been moved or deleted, or if you are using a virtual Web server instance that has a different document root, that relative URL may not be valid. If so, one option would be to configure your Web server such that the `/CFIDE/scripts` prefix maps to the folder that actually contains `wddx.js`. Or, more simply, you can just copy the `wddx.js` file to the same folder as the ColdFusion pages you need to use it in, adjusting the relative path accordingly, like so:

```
<!--- Include wddx.js, located in this folder --->
<script language="JavaScript" src="wddx.js"></script>
```

Simply to ensure that all the examples work correctly without adjustments, the example listings for this chapter use this method. Just do whatever makes sense for your situation.

NOTE

Conceptually, this line means the same thing to the browser as that a `<cfinclude>` tag means to ColdFusion. The browser will fetch the file over the Internet and execute its code inline, as if it appeared between the `<script>` tags.

Using `WddxRecordset` Methods

The `WddxRecordset` object supports a number of methods for getting data in and out of the recordset, as listed in Table 48.13. If you read Chapter 47, you may notice that the nature and scope of these methods are similar to the ones provided for the COM and Java implementations of WDDX that were discussed in that chapter. The theory in all these cases is the same: to provide a basic notion of a recordset that includes just enough methods to make the recordset useful, while keeping it easy to use, understand, and support.

NOTE

Complete reference information and further examples are provided in the WDDX Software Development Kit (SDK), which is available from http://www.openwddx.org.

Table 48.13 JavaScript `WddxRecordset` Methods

METHOD	DESCRIPTION
`.addColumn(name)`	Adds a column to the recordset. Specify the new column's name as a string.
`.addRows(num)`	Adds the specified number of rows to the recordset. Specify the number of rows as an integer.
`.isColumn(name)`	Determines whether the name you specify is a column of the recordset. Returns a Boolean (true/false) value.
`.getField(row, col)`	Returns the data in the recordset at the row and column position you specify. Specify `row` as an integer (the first row is 0, the second row is 1, and so on). Specify `col` as a string.
`.getRowCount()`	Returns the number of rows in the recordset, as an integer. Similar conceptually to the automatic `RecordCount` property for query objects in CFML.
`.setField(row, col, value)`	Places `value` into the recordset at the row and column position you specify. Specify row as an integer. Specify `col` as a string.

Table 48.13 (continued)

METHOD	DESCRIPTION
`.dump(escape)`	For debugging purposes. Conceptually, this method is the equivalent of `<cfdump>` in CFML. To dump the contents of the recordset to the screen, you could use `document.write (rs.dump())`, where `rs` is an instance of the `WddxRecordset` object. The optional `escape` argument determines whether characters that are special to HTML are escaped (in the fashion of `HTMLEditFormat()` in CFML). The default is `false`; in general, you should use `.dump(true)`.
`.wddxSerialize()`	Used internally by the `WddxSerializer` object. This method is not meant to be called on its own, but you might take a look at this portion of the `wddx.js` file to see how you can create a custom JavaScript object that serializes itself in some kind of special way. For details, consult the WDDX SDK.

NOTE

For all methods that take a `col` argument, the column name is not case sensitive (the column names are stored internally in lowercase to achieve the case-insensitivity). The only exception is if you create a new recordset from scratch on the client as discussed in the Creating Recordsets from Scratch sidebar for details, in which case you can specify that the case of the column names are preserved (details in the WDDX SDK). Otherwise, the capitalization of recordset column names is not considered to be important.

You'll see the `getRowCount()` and `getField()` methods used in the next code listing.

Cascading Selects, Approach #4: Using a `WddxRecordset` Object

As an example of how to use `WddxRecordset` in an actual Web page, let's return to the cascading select problem. Listing 48.6 shows a fourth solution to the problem, this time using `<cfwddx>` to send the `FilmsQuery` recordset to the browser as a variable called `rsFilms`. Once interpreted by the browser's script engine, the `rsFilms` object will be an instance of `WddxRecordset`, meaning that any of the methods listed in Table 48.13 can be used to retrieve (or change) the data it contains.

Listing 48.6 `JSRelatedSelects4.cfm`—Passing Recordset Data to Relate the Two Select Lists

```
<!---
Name:        JSRelatedSelects4.cfm
Author:      Nate Weiss and Ben Forta
Description: Demonstrates one approach to relating
             select lists via JavaScript, using
             film and rating data from the ows
             example database.
Created:     02/01/05
--->


<!--- Get film data from database --->
<cfquery name="FilmsQuery"
         datasource="ows">
  select FilmID, MovieTitle, RatingID
```

Listing 48.6 (continued)

```
  FROM Films
  WHERE RatingID IS NOT NULL
  ORDER BY RatingID, MovieTitle
</cfquery>

<!--- Get rating data from database --->
<cfquery name="RatingsQuery"
         datasource="ows">
  select RatingID, Rating
  FROM FilmsRatings
  ORDER BY RatingID
</cfquery>


<html>
<head>
<title>Cascading Select Lists</title>

<!--- Include the wddx.js file (in same folder as this ColdFusion page) --->
<!--- This allows us to receive and work with WddxRecordset objects --->
<script type="text/javascript"
        src="wddx.js"
        language="JavaScript"></script>

<!--- Custom JavaScript code --->
<script type="text/javascript"
        language="JavaScript">

  <!--- Output the JavaScript code needed to create a JavaScript object --->
  <!--- called objRatings that holds the same data as RatingsStruct --->
  <cfwddx action="CFML2JS"
          input="#FilmsQuery#"
          toplevelvariable="rsFilms">

  // This function fills the second select box based on the first box's value
  function fillFilms() {
    // Get the currently selected rating ID from the first select box
    with (document.forms[0].RatingID) {
      // Stop here if there is no selected rating
      if (selectedIndex == -1) {
        return;
      }

      var ratingID=options[selectedIndex].value;
    }


    // Remove all options from the second select box
    document.FilmForm.FilmID.options.length=0;

    // For each item in the films array...
    for (var row=0; row < rsFilms.getRowCount(); row++) {

      // If this is a matching film
      if ( rsFilms.getField(row, "RatingID") == ratingID ) {
```

Listing 48.6 (continued)

```
        // Place a new option in the second select box
        with (document.FilmForm.FilmID) {
          options[options.length]=new Option(
            rsFilms.getField(row, "MovieTitle"),
            rsFilms.getField(row, "FilmID"));
        }

      };
    };
  };
</script>

</head>

<!--- Call the fillFilms() function when the page first appears --->
<body onload="fillFilms()">


<!--- Ordinary html form --->
<cfform action="ShowFilm.cfm"
        name="FilmForm"
        method="Post">

  <!--- First select box (displays ratings) --->
  <strong>Rating:</strong><br>
  <cfselect name="RatingID"
            query="RatingsQuery"
            value="RatingID"
            display="Rating"
            onchange="fillFilms()"/><br>

  <!--- Second select box (displays films) --->
  <p><strong>Film:</strong><br>
  <cfselect name="FilmID"
            size="5"
            style="width:300px"/><br>

  <!--- Submit button --->
  <cfinput type="Submit"
           name="sbmt">
</cfform>

</body>
</html>
```

If you use the number of lines of code as a measure of simplicity, this is the simplest approach yet. Whether it actually feels simpler to you as a developer is a matter of perspective and personal preference.

As you can see, this listing is structurally similar to the versions that came before it. The important differences introduced in this version are as follows:

- A `<script>` block is used with `src="wddx.js"` to include support for `WddxRecordset` objects.

- `<cfwddx>` is used with `action="CFML2JS"` to generate the JavaScript code needed to create a `WddxRecordset` that contains the same data as the ColdFusion `FilmsQuery` object.
- The number of rows in the recordset is obtained using `rsFilms.getRowCount()`. This is used to create a for loop that loops over each row, where the current row is available as the integer `row`.
- The `getField()` method is used to compare the rating of each film to the user's current selection, and to retrieve each matching film's ID number and title.

Creating Recordsets from Scratch

Most of the time, you will receive a `WddxRecordset` from ColdFusion as a result of the `<cfwddx>` tag, or by deserializing a WDDX packet that contains a `<recordset>` block. That said, you may occasionally need to create a `WddxRecordset` from scratch in your JavaScript code. Just create a new recordset with `new WddxRecordset()`, then use the `addColumn()`, `addRows()`, and `setField()` methods from Table 48.12. For instance, you could create a new recordset with JavaScript code like the following:

```
rs = WddxRecordset()
rs.addColumn("firstname")
rs.addColumn("lastname")
rs.addRows(2);
rs.setField(0, "firstname", "Nate");
rs.setField(0, "lastname", "Weiss");
rs.setField(1, "firstname", "Winona");
rs.setField(1, "lastname", "Ryder");
```

Conceptually, the steps are similar to the `QueryNew()`, `QueryAddRow()`, `QueryAddColumn()`, and `QuerySetCell()` functions in CFML. You can also pass the initial column names and number of rows to the WddxRecordset() constructor; see the WDDX SDK for details.

Working with WDDX Packets in JavaScript

In Chapter 47, you learned about serializing and deserializing WDDX packets with the `<cfwddx>` tag (and with other tools like Java and Active Server Pages). This chapter has also discussed `<cfwddx>`, but only within the context of generating JavaScript code. What about serializing and deserializing WDDX packets within JavaScript?

As you might expect, support is provided for working with WDDX packets in both directions (serializing into new packets, and deserializing from existing packets). This section will explain how.

Serializing Packets with `WddxSerializer`

Earlier in this chapter, you learned about the `wddx.js` file and the `WddxRecordset` object type defined therein. That same file also defines a `WddxSerializer` object, which lets you serialize JavaScript

values and variables into WDDX packets. Conceptually, its purpose is to provide the JavaScript equivalent of `<cfwddx>`'s `CFML2WDDX` action.

To use the serializer, you follow the following basic steps:

1. Create the value or variable that you want to serialize.
2. Create a new instance of the `WddxSerializer` object.
3. Call the new instance's `serialize()` method to serialize your value. The method returns the corresponding WDDX packet.

Table 48.14 shows the methods supported by the `WddxSerializer` object. In most situations, the only one you need to use is `serialize()`.

Table 48.14 JavaScript `WddxSerializer` Methods

METHOD	DESCRIPTION
`.serialize(value)`	Serializes the `value` and returns the resulting WDDX packet (as a string). The `value` can be just about any JavaScript value, including dates, strings, numbers, `Object` instances, custom objects, arrays, and WddxRecordset objects.
Custom serialization methods	For advanced use only. `WddxSerializer` also supports `serializeVariable()`, `serializeValue()`, and `.write()` methods, which you can use to create custom objects that know how to serialize themselves in some special way. For details, consult the WDDX SDK.

To create an instance of the serializer object, use code like the following:

```
var mySer = new WddxSerializer();
```

To serialize a value, just call `serialize()` like so, assuming that `myVar` is the value that you want to serialize:

```
var wddxPacket = mySer.serialize(myVar);
```

Building a Simple Recordset-Editing Interface

Take a look at Listing 48.7. It creates a web page with a simple form on it . On the left, a list of all current films are is displayed in a multiline `<select>` list. When the user selects a film in the list, that film's title, budget, one-liner, and summary are displayed in the editable fields to the right (Figure 48.5). The user can edit the title, budget, or other information, then can press Keep These Edits to store the edits in the browser's copy of the recordset. The Commit Changes to Server button serializes the entire recordset and posts it to the server for processing.

Figure 48.5

Users can scroll through current films and make updates as needed.

NOTE

If the selected film has an image, it will be displayed as well, though this version of the browser provides no method for uploading a new image (though it would be easy enough to do with `<cffile action="Upload">`).

Listing 48.7 `JSFilmBrowser.cfm`—Serializing a Recordset Object after It has Been Edited

```
<!---
Name:        JSFilmBrowser.cfm
Author:      Nate Weiss and Ben Forta
Description: Allows the user to edit the
             records in a WddxRecordset.
             The edited records can be
             posted to the server as a WDDX packet.
Created:     02/01/05
--->

<!--- Get data about films from database --->
<cfquery name="FilmsQuery"
         datasource="ows">
  SELECT FilmID, MovieTitle, AmountBudgeted,
         PitchText, Summary, ImageName
  FROM Films
  ORDER BY MovieTitle
</cfquery>
```

Listing 48.7 (continued)

```
<!--- Workaround for bug in <cfwddx action="CFML2JS"> for NULL values --->
<!--- (see note in text) --->
<cfloop query="FilmsQuery">
  <cfif FilmsQuery.ImageName EQ "">
    <cfset FilmsQuery.ImageName="">
  </cfif>
</cfloop>


<html>
<head>
<title>Film Browser</title>

<!--- Include WddxRecordset and WddxSerializer support --->
<script type="text/javascript"
        src="wddx.js"
        language="JavaScript"></script>

<!--- Custom functions for this page --->
<script type="text/javascript"
        language="JavaScript">

   <!--- Convert query to JavaScript object named "rsFilms" --->
   <cfwddx
     action="CFML2JS"
     input="#FilmsQuery#"
     toplevelvariable="rsFilms">

  // Add a column called "wasedited" to the recordset
  // A "Yes" in this column means the row was "touched"
  rsFilms.addColumn("wasedited");

  ////////////////////////////////////////////////////
  // This function fills the select list with films
  function InitControls() {
    with (document.DataForm) {

      // Clear any current optionS from the select
      FilmID.options.length=0;

      // For each film record...
      for (var row=0; row < rsFilms.getRowCount(); row++) {

        // Create a new option object
        var NewOpt=new Option;
        NewOpt.value=rsFilms.getField(row, "FilmID");
        NewOpt.text=rsFilms.getField(row, "MovieTitle");

        // Add the new object to the select list
        FilmID.options[FilmID.options.length]=NewOpt;

      }
    }
  }
```

Listing 48.7 (continued)

```
////////////////////////////////////////////////
// This function populates other input elements
// when an option in the select box is clicked
function FillControls() {
  with (document.DataForm) {
    // Get the data row number
    var row=FilmID.selectedIndex;

    // Populate textboxes with data in that row
    AmountBudgeted.value=rsFilms.getField(row, "AmountBudgeted");
    MovieTitle.value=rsFilms.getField(row, "MovieTitle");
    PitchText.value=rsFilms.getField(row, "PitchText");
    Summary.value=rsFilms.getField(row, "Summary");

    // Get the name of the image file for this film, if any
    var imageName=rsFilms.getField(row, "ImageName");
    // Get a reference to the <img> tag on the page
    var objImage=document.images["filmImage"];

    // If there is no image for this film, make the <img> be invisible
    if (imageName == "") {
      objImage.style.visibility="hidden";
    // If there is an image, show that image in the <img> object,
    // and make sure the object is visible
    } else {
      objImage.src="images/" + imageName;
      objImage.style.visibility="visible";
    };
  }
}


////////////////////////////////////////////////
// This function "saves" data from the various
// text boxes into the wddxRecordset object
function KeepChanges() {
  with (document.DataForm) {
    // Get the data row number
    var SelectedFilm=FilmID.selectedIndex;
    var row=SelectedFilm;

    // Populate JavaScript recordset with data from form fields
    rsFilms.setField(row, "MovieTitle", MovieTitle.value);
    rsFilms.setField(row, "AmountBudgeted",
                     parseInt(AmountBudgeted.value));
    rsFilms.setField(row, "PitchText", PitchText.value);
    rsFilms.setField(row, "Summary", Summary.value);
    rsFilms.setField(row, "wasedited", "Yes");

    // Re-initialize the select list
    InitControls();

    // Re-select the film that was selected before
    FilmID.selectedIndex=SelectedFilm;
```

Listing 48.7 (continued)

```
    }
  }


  ////////////////////////////////////////////////
  // This function inserts a new row in the
  // wddxRecordset object, ready for editing
  function NewRecord() {
    with (document.DataForm) {
      // Add a new row to the recordset
      rsFilms.addRows(1);
      var NewRow=rsFilms.getRowCount()-1;

      rsFilms.setField(NewRow, "FilmID", "new");
      rsFilms.setField(NewRow, "MovieTitle", "(new)");
      rsFilms.setField(NewRow, "AmountBudgeted", "");
      rsFilms.setField(NewRow, "PitchText", "");
      rsFilms.setField(NewRow, "Summary", "");

      // Re-initialize the select list
      InitControls();

      // Re-select the film that was selected before
      FilmID.selectedIndex=NewRow;
      FillControls();
    }
  }


  ////////////////////////////////////////////////
  // This function inserts a new row in the
  // wddxRecordset object, ready for editing
  function CommitToServer() {
    with (document.DataForm) {
      // Create new WDDX Serializer object (defined in wddx.js)
      var mySer=new WddxSerializer();

      // Serialize the "rsFilms" recordset into a WDDX packet
      var FilmsAsWDDX=mySer.serialize(rsFilms);

      // Place the packet into the "WddxContent" hidden field
      WddxContent.value=FilmsAsWDDX;

      // Submit the form
      submit();
    }
  }
</script>
</head>


<!--- Run InitControls() function when page first appears --->
<body onload="InitControls();">
<h2>Film Browser</h2>
```

Listing 48.7 (continued)

```
<!--- Ordinary html form for editing the recordset --->
<cfform action="JSFilmBrowserCommit.cfm"
        method="Post"
        name="DataForm">

  <!--- CommitToServer() function gives this a value --->
  <cfinput type="Hidden"
           name="WddxContent">

  <table border cellpadding="10">
   <tr valign="TOP">
    <td>
      <!--- select populated by InitControls() function --->
      <!--- When clicked, calls FillControls() function --->
      <cfselect name="FilmID"
               size="16"
               onchange="FillControls()">
        <option>============= (loading) =================
      </cfselect>
    </td>

    <td>
      <!--- Image placeholder to display film image (when available) --->
      <!--- When the page first loads, this image will be hidden --->
      <img src=""
           name="filmImage"
           border="0"
           align="right"
           style="visibility:hidden">

      <!--- These controls get populated by FillControls() --->
      Film Title:<br>
      <cfinput name="MovieTitle"
               size="40"
               maxlength="50"><br>

      Amount Budgeted:<br>
      <cfinput name="AmountBudgeted"
               size="15"
               maxlength="50"><br>

      One-Liner:<br>
      <cfinput name="PitchText"
               size="40"
               maxlength="50"><br>

      Summary:<br>
      <cftextarea name="Summary"
                  rows="4"
                  cols="50" /><br>


      <p>
      <!--- Button to "keep" edits with KeepChanges() function --->
      <cfinput type="button"
```

Listing 48.7 (CONTINUED)

```
                  name="btn1"
                  value="Keep These Edits"
                  onclick="KeepChanges()">

         <!--- Button to cancel edits with FillControls() function --->
         <cfinput type="button"
                  name="btn2"
                  value="Cancel"
                  onclick="FillControls()">

         <!--- Button to insert new film with NewRecord() function --->
         <cfinput type="button"
                  name="btn3"
                  value="New Record"
                  onclick="NewRecord()"><br>
       </td>
      </tr>
     </table>

     <!--- Button to save to server w/ CommitChanges() function --->
     <p align="center">
      <cfinput type="button"
               name="btn4"
               value="Commit Changes To Server"
               onclick="CommitToServer()"><br>
     </p>

    </cfform>

    </body>
    </html>
```

The first few lines are familiar. First, the `wddx.js` file is included with the `src` attribute of a `<script>` tag, so that the page's JavaScript code can refer to `WddxRecordset` and `WddxSerializer` objects. For details about this line, refer to the "Including the `wddx.js` JavaScript File" section, earlier in this chapter.

NOTE

The strange-looking `<cfloop>` at the top of this listing is a workaround for a small bug in `action="CFML2JS"` in the version of ColdFusion that I was using when writing this chapter. The effect of the bug is that `NULL` values returned by database queries may not be converted into JavaScript correctly. This loop was the easiest way to fix the problem in a database-independent way. It is hoped that this bug will have been fixed in an update of some kind by the time you read this book, in which case the `<cfloop>` can be removed.

The scripting part of the template then goes on to create a user-defined function called `InitControls()`, which is responsible for populating the `<select>` box in the simple form at the bottom of the template (shown along the left side of Figure 48.5). First, `FilmID.options.length` is set to zero to clear any options that may currently be sitting in the `<select>` box. Then a `for` loop is used to fill the `<select>` with options for each film. This code is quite similar to the `fillFilms()` function in the variations of the cascading select list examples you've seen in this chapter.

The `FillControls()` function fills the four text boxes with the Title, Budget, and so on for the currently-selected film in the select list. Since the `FillControls()` function is referred to in the select list's `onchange` handler, the function will execute whenever the user chooses a different film from the list. The function itself is extremely simple—it just sets a variable called `row` that represents the currently-selected film. Then the function uses the `WddxRecordset getField()` method to retrieve each value from the `rsFilms` recordset, storing it in the `value` property of the corresponding text box.

NOTE

The `FillControls()` function also changes the image displayed in the `<img>` tag named `filmImage`. If there is an image for the current film, the image object's `src` attribute is set to display that film's image. This portion of the code also sets the object's `style.visibility` property to `hidden` or `visible` so that the image object is invisible for films that do not have an image. With modern browsers, the `style.visibility` property can be used in this way to control the visibility of nearly all elements of a page (`<div>` blocks and so on), not just images. Consult a Dynamic HTML (DHTML) reference for details.

The `KeepChanges()` function does the opposite of `FillControls()`. It reads the values from the four text boxes and uses `getField()` to place their values into the appropriate spots in the `rsFilms` recordset object. Next, it executes the `InitControls()` function to "re-draw" the items in the select list; if the movie's title was edited, the new title will now appear in the list. Lastly, it sets the list's `selectedIndex` back to the choice that was selected before the function was called. The function is assigned to the "Keep These Edits" button by referring to the function's name in the button's `onclick` handler.

The `NewRecord()` is responsible for adding a new row to the recordset when the user clicks the New Record button. All it needs to do is to call the `addRows()` function (see Table 48.13); the new row is added to the bottom of the recordset. Next, it uses the `getRowCount()` function to set a variable named `row`, which will hold the row number of the just-added row. Then it uses the `setField()` function to set each column of the new row to some initial values. Note that the `FilmID` column is set to the string `"new"`. This will indicate to the server that the record is a new record and thus should be inserted (rather than updated) to the database. Finally, the function redraws the select list with the `InitControls()` function, sets its `selectedIndex` so that the new record appears "selected" in the form, and calls the `FillControls()` function so that the data-entry inputs get filled with the new (mostly blank) values.

The `CommitToServer()` function is in charge of serializing the recordset into a new WDDX packet, then placing the packet in a hidden field and submitting the form. The serializing part requires only two lines of JavaScript code. First, a new WddxSerializer object called `mySer` is created, with the help of JavaScript's `new` keyword. This step is necessary whenever you want to serialize a value from JavaScript. Next, the `serialize()` method of the `mySer` object is used to serialize the `rsFilms` recordset into a WDDX packet, placing the packet into a JavaScript variable called `FilmssAsWDDX`. The packet (which is a string at this point, in the form of XML), is then placed in the hidden form field called `WddxContent`. Finally, the function submits the form.

The end result is that the ColdFusion template that this form submits to (`JSFilmBrowserCommit.cfm`) will be able to refer to a variable called `#Form.WddxContent#`. The variable will hold the WDDX packet that contains the edited version of the recordset.

Processing the Posted Packet on the Server

The `JSBrowserCommit.cfm` template that receives the WDDX packet from the Film Browser example (Listing 48.7) is actually quite simple. Since the packet's contents were stored in the hidden field named `WddxContent` just before the form was submitted, the packet will be available to this template in the `#Form.WddxContent#` variable. All the template needs to do is use `<cfwddx>` to deserialize the packet into a query recordset named `EditedRecordset`. Then it can use a `<cfloop>` over the query to quickly examine each data row to see if it is a new or changed record.

Listing 48.8 shows the code for the `JSFilmBrowserCommit.cfm` template.

Listing 48.8 `JSFilmBrowserCommit.cfm`—Receiving and Deserializing a Packet Created by JavaScript

```
<!---
Name:        JSFilmBrowserCommit.cfm
Author:      Nate Weiss and Ben Forta
Description: Receives an edited recordset in
             the form of a WDDX packet and
             makes changes to the corresponding
             database table accordingly.
Created:     02/01/05
--->

<!--- We are expecting to receive a form field named WDDXContent --->
<cfparam name="form.WddxContent"
         type="string">

<!--- Deserialize the WDDX packet into a native ColdFusion recordset --->
<cfwddx action="WDDX2CFML"
        input="#form.WddxContent#"
        output="EditedRecordset">

<!--- We'll increment these counters in the loop --->
<cfset InsertCount=0>
<cfset UpdateCount=0>

<!--- Loop over each of the records in the query --->
<cfloop query="EditedRecordset">

  <!--- If it's a new record (the user inserted it) --->
  <cfif EditedRecordset.FilmID EQ "new">
    <!--- Insert a new record into the database --->
    <cfquery datasource="ows">
      INSERT INTO Films (MovieTitle, AmountBudgeted, PitchText, Summary)
      valueS ('#MovieTitle#', #AmountBudgeted#, '#PitchText#', '#Summary#')
    </cfquery>
    <!--- Increment the insert counter --->
    <cfset InsertCount=InsertCount + 1>


  <!--- It's an existing record (user may have edited) --->
  <cfelseif EditedRecordset.WasEdited EQ "Yes">
    <!--- Updating the existing record --->
    <cfquery datasource="ows">
      UPDATE Films SET
```

Listing 48.8 (continued)

```
          MovieTitle='#MovieTitle#',
          AmountBudgeted=#AmountBudgeted#,
          PitchText='#PitchText#',
          Summary='#Summary#'
        WHERE Filmid=#FilmID#
      </cfquery>
      <!--- Increment the update counter --->
      <cfset UpdateCount=UpdateCount + 1>

    </cfif>
  </cfloop>


  <html>
  <head>
  <title>Committing Changes</title>
  </head>

  <body>
  <h2>Committing Changes</h2>

  <!--- Display message about what exactly happened --->
  <cfoutput>
    <p><strong>Changes Committed!</strong>
    <ul>
      <li>Records Updated: #UpdateCount#
      <li>Records Inserted: #InsertCount#
    </ul>
  </cfoutput>

  </body>
  </html>
```

If the `FilmID` column of the current record is set to the string "new", then the template knows that the record was inserted by the Film Browser's `NewRecord()` function. Therefore, it runs a simple `INSERT` query to insert the new row into the Inventory table.

If the `FilmID` column of the current record is not set to "new", the template checks to see if the `WasEdited` column has been set to "Yes". If it has, then the template knows that the record was edited by the Film Browser's `KeepChanges()` function. Therefore, it runs a simple `UPDATE` query to update the corresponding row in the Inventory table, using the `FilmID` column as the primary key.

Finally, the template displays a simple message to let the user know that the records were inserted or updated successfully. A summary is provided that shows the number of inserted records and the number of updated records (see Figure 48.6).

The WDDX SDK includes several similar examples, written in ColdFusion, ASP, and Perl. Some of the examples show how the recordset can be saved to the browser machine's hard drive using Microsoft's Scripting.FileSystemObject control (the code is IE-specific). This allows the user to save their work locally while they perform their data-entry tasks, possibly over the course of several hours or days, with or without an Internet connection.

Figure 48.6

When the edited recordset is submitted to the server, the database is updated accordingly.

Deserializing Packets with `WddxDeserializer`

As you have learned in this chapter, you can use `<cfwddx>` to send values to JavaScript in one step, without ever converting the values to XML packets. As I suggested earlier in the "Is This Serializing?" sidebar, you can think of the generated-JavaScript-code phase as the equivalent to the XML-packet-phase that you would normally expect to see in WDDX-powered applications. I know of no easier way to send complex, multifaceted data to JavaScript as a page loads.

That said, there may be times when you would like to deserialize packets within the context of a Web page, without refreshing the entire page. As a rule, the time to consider such a crazy thing is when you want the user to be able to retrieve or scroll through data in real time (like the Film Browser example you just saw), but where the amount of data or some other consideration makes it infeasible to send the entire set of data to the client at once.

Okay, see if you can guess the name of the object you use to deserialize WDDX packets in JavaScript. That's right, it's `WddxDeserializer`, and it provides a `deserialize()` method that basically does the inverse of what the `WddxSerializer` object's `serialize()` method does. Table 48.15 shows the methods supported by `WddxDeserializer`.

Table 48.15 JAVASCRIPT `WddxDeserializer` Methods

METHOD	DESCRIPTION
`.deserialize(packet)`	Deserializes the WDDX packet (supply the packet as a string). Returns the deserialized value, which could be a native JavaScript object, array, WddxRecordset, string, date, number, and so on.
`.deserializeUrl(url)`	Fetches and deserializes the WDDX packet at the given URL. You can pass parameters to the URL by adding name/value pairs to the deserializer's special `urlData` property. For details, consult the WDDX SDK.

In general, you just create a new deserializer object like this:

```
var myDes = new WddxDeserializer();
```

Then, assuming you already have a WDDX packet (that is, an XML-formatted string) in a JavaScript variable called `myPacket`, you can deserialize it like so:

```
var myObject = myDes.deserialize(myPacket);
```

The JavaScript `myObject` variable would then contain whatever data was in the packet, so it might be an Array object, a `WddxRecordset` object, or whatever custom object is appropriate.

Sounds great, right? Sure it is, but there are a few catches:

- Depending on the browsers you need to support, there isn't necessarily an easy way to fetch a WDDX packet from a Web server using JavaScript. You're fine if you need only support Internet Explorer 5 (or later) for Windows, or any other Mozilla-based browser (including Netscape 6 and Firefox). If you need to support other browsers, you may need to rely on a Java applet to fetch the text over the Internet for you. There are details about this in the WDDX SDK.
- To keep `wddx.js` as small as possible, it does not include the `WddxDeserializer` object. Instead, it is implemented in a separate file called `wddxDes.js`, which must be included by any page that wants to use the deserializer. There is also a `wddxDesIE.js` file which is specially optimized for Internet Explorer. These files are not distributed with ColdFusion. They are, however, freely available as a part of the WDDX SDK and are included in the code listings for this chapter. You'll see how to include the files in the next example listing.

Cascading Selects, Approach #5: Fetching Matching Films in Real Time

Let's take a look at a real-world example. Listing 48.9 creates another solution to the (now age-old) cascading select list problem. This one's pretty interesting. Instead of working with one large list of films that is passed to the browser when the page first loads, this version contacts the ColdFusion server each time the user selects a different rating. That is, the options to show in the second select list are retrieved in "real time" from the server.

The `<cfform>` portion of this listing is the same as the previous versions of this example. Much of the script portions have changed.

Listing 48.9 `RelatedSelectsViaWDDX.cfm`—Fetching and Deserializing Packet

```
<!---
Name:        RelatedSelectsViaWDDX.cfm
Author:      Nate Weiss and Ben Forta
Description: Demonstrates use of WDDX
             deserialization within JavaScript.
Created:     02/01/05
--->
```

Listing 48.9 (continued)

```
<!--- URL that will return WDDX packet containing recordset of film data --->
<cfset FilmComponentURL="http://127.0.0.1:8500/ows_adv/17/FilmsRobot.cfm?">

<!--- Get rating data from database --->
<cfquery name="RatingsQuery"
         datasource="ows">
  select RatingID, Rating
  FROM FilmsRatings
  ORDER BY RatingID
</cfquery>

<html>
<head>
<title>Relating Select Boxes via WDDX</title>

<!--- Pass variables to JavaScript --->
<cfoutput>
  <script type="text/javascript"
          language="JavaScript">
    var FilmComponentURL="#JSStringFormat(FilmComponentURL)#"
  </script>
</cfoutput>

<!--- Include WddxRecordset support --->
<script type="text/javascript"
        language="JavaScript"
        src="wddx.js"></script>

<!--- Include WddxDeserializer support --->
<!--- (use special file if browser is IE under Windows --->
<cfif (CGI.HTTP_USER_AGENT contains "MSIE")
  AND (CGI.HTTP_USER_AGENT contains "Win")>
  <script type="text/javascript"
          language="JavaScript"
          src="wddxDesIE.js"></script>
<cfelse>
  <script type="text/javascript"
          language="JavaScript"
          src="wddxDes.js"></script>
</cfif>

<!--- Custom JavaScript functions for this page --->
<script type="text/javascript"
        language="JavaScript">
  // showFilms() function
  // Relates two <select> boxes by fetching a WDDX recordset packet based on
  // the first box; the second box is filled with the recordset contents
  function fillFilms() {

    // Object reference for first select box
    var objSel=document.forms[0].RatingID;

    // Assuming there is a selection in the first select box
    if (objSel.selectedIndex >= 0) {
```

Listing 48.9 (continued)

```
      // Get the value of the current selection in the first select box
      var ratingID=objSel[objSel.selectedIndex].value;
      // Add the value to the URL
      var packetURL=FilmComponentURL + "&Ratingid=" + ratingID;

      // Fetch the WDDX packet from the URL
      var packet=httpGetFromURL(packetURL);

      // Deserialize the packet
      // The result is a WddxRecordset object called rsFilms
      var wddxDes=new WddxDeserializer;
      var rsFilms=wddxDes.deserialize(packet);

      // Object reference for the second select box
      objSel=document.forms[0].FilmID;

      // Remove all items from the second select box
      objSel.length=0;

      // For each row in the recordset...
      for (var i=0; i < rsFilms.getRowCount(); i++) {
        // Grab the ID and title for the current row of the recordset
        var filmID=rsFilms.getField(i, "filmid");
        var movieTitle=rsFilms.getField(i, "movietitle");

        // Add an option to the second select box
        objSel.options[objSel.options.length]=new Option(movieTitle, filmID);
      };
    }
  };



  // Utility function to fetch text from a URL
  // A wrapper around the appropriate objects exposed by Netscape 6 or IE
  function httpGetFromURL(strURL) {
    var objHTTP, result;

    // For Netscape 6+ browsers (or other browsers that support XMLHttpRequest)
    if (window.XMLHttpRequest) {
      objHTTP=new XMLHttpRequest();
      objHTTP.open("GET", strURL, false);
      objHTTP.send(null);
      result=objHTTP.responseText;

    // For IE browsers under Windows (version 5 and later)
    } else if (window.ActiveXObject) {
      objHTTP=new ActiveXObject("Microsoft.XMLHTTP");
      objHTTP.open("GET", strURL, false);
      objHTTP.send(null);
      result=objHTTP.responseText;

    } else {
      alert("Sorry, your browser can't be used for this example.");
```

Listing 48.9 (continued)

```
        }

        // Return result
        return result;
      }
    </script>

    </head>

    <body onload="fillFilms()">
    <h2>Relating Select Boxes via WDDX</h2>

    <!--- Ordinary html form --->
    <cfform action="ShowFilm.cfm"
            name="FilmForm"
            method="Post">

      <!--- First select box (displays ratings) --->
      <strong>Rating:</strong><br>
      <cfselect name="RatingID"
                query="RatingsQuery"
                value="RatingID"
                display="Rating"
                onchange="fillFilms()"/><br>

      <!--- Second select box (displays films) --->
      <p><strong>Film:</strong><br>
      <cfselect name="FilmID"
                size="5"
                style="width:300px"/><br>

      <!--- Submit button --->
      <cfinput type="Submit"
    name="sbmt">
    </cfform>

    </body>
    </html>
```

At the top of this listing, a CFML variable called `FilmComponentURL`, which contains the URL for a ColdFusion page called `FilmsRobot.cfm`. This page is a slight variation on the `FilmsRobot1.cfm` page that was created in Chapter 47 (you'll see the code for this robot page in the next listing). The `FilmComponentURL` variable is then passed to JavaScript using a simple `<script>` block.

NOTE

You may need to adjust this URL depending on how you installed ColdFusion and the location of the listings for this chapter.

Next, the usual `wddx.js` file is included, which enables the use of `WddxRecordset`. Then `wddxDes.js` or `wddxDesIE.js` is included, depending on whether the user is using IE for Windows or not. This enables the use of `WddxDeserializer`.

Within the `fillFilms()` function, the `ratingID` variable holds the currently selected rating. Another variable called `packetURL` is then constructed by adding a URL parameter called `RatingID` to the

FilmComponentURL. This URL that can be used to retrieve the appropriate WDDX packet from the server.

Next, a function called httpGetFromURL() is used to contact the robot page and retrieve the WDDX packet that it responds with. If you take a look at the body of the httpGetFromURL() function, you'll see that it executes slightly different code depending on whether the user is using IE or Netscape/Mozilla. For now, just accept that this is one relatively straightforward way to retrieve text from an arbitrary URL via JavaScript. (There are other ways, too, such as with the load() method of the respective browser's XML DOM implementations.)

In any case, the XML text of the robot's WDDX packet should be in the myPacket variable, ready for deserialization. The next two lines create an instance of WddxDeserializer called wddxDes and use it to deserialize the packet, returning what is hopefully a WddxRecordset object named rsFilms. It's now a simple matter to iterate through the rows of the recordset, filling the second select list with options on the way.

The result is a page that behaves in the same way as the previous versions, as shown back in Figure 48.4. One of the advantages to this approach is that the browser machine never needs to have the entire list of films in its memory at the same time. Another advantage is that the browser machine gets an up-to-date list every time the user chooses a different rating. If your data changes very frequently, this may be a significant benefit.

Listing 48.10 shows the code for the FilmsRobot.cfm page. This is the robot page that supplies film data to the previous listing, based on the RatingID URL parameter. Please refer to Chapter 47 for a full discussion of this type of page.

Listing 48.10 FilmsRobot.cfm—Supplying WDDX Packets to the JavaScript Page from Listing 48.9

```
<!---
Name:        FilmsRobot.cfm
Author:      Nate Weiss and Ben Forta
Description: Creates a back-end web page
             that supplies data about films
Created:     02/01/05
--->

<!--- URL Parameters to control what film data the page responds with --->
<cfparam name="URL.FilmID"
         type="numeric"
         default="0">
<cfparam name="URL.RatingID"
         type="numeric"
         default="0">
<cfparam name="URL.Details"
         type="boolean"
         default="No">
<cfparam name="URL.Keywords"
         type="string"
         default="">
```

Listing 48.10 (continued)

```
<!--- Execute a database query to select film information from database --->
<cfquery name="FilmsQuery"
         datasource="ows">
  SELECT
    <!--- If all information about film(s) is desired --->
    <cfif URL.Details>
      *
    <!--- Otherwise, return the film's ID and title --->
    <cfelse>
      FilmID, MovieTitle
    </cfif>
  FROM Films
  <!--- If a specific film ID was specified --->
  <cfif URL.FilmID GT 0>
    WHERE Filmid=#URL.FilmID#
  <cfelseif URL.RatingID GT 0>
    WHERE Ratingid=#URL.RatingID#
  <!--- If keywords were provided to search with --->
  <cfelseif URL.Keywords NEQ "">
    WHERE MovieTitle LIKE '%#URL.Keywords#%'
       OR Summary LIKE '%#URL.Keywords#%'
  </cfif>
  ORDER BY MovieTitle
</cfquery>


<!--- Convert the query recordset to a WDDX packet --->
<cfwddx action="CFML2WDDX"
        input="#FilmsQuery#">
```

Cascading Selects, Approach #6: Wrapping the WDDX Fetching in a Custom Tag

As an experiment, I created a custom tag version of the JavaScript code that powers the last version of the cascading select list example (Listing 48.9). The custom tag allows you to create any two `<select>` lists using normal HTML syntax. You then bind the two together using the `<CF_RelateTwoSelectLists>` custom tag. An example of using the tag is provided in the `UseRelateTwoSelectLists.cfm` file (included with this chapter's listings). Here's the key portion of that example:

```
<!--- Relate the two select lists in real time, using WDDX --->
<CF_RelateTwoSelectLists
 WddxRecordsetURL="#FilmComponentURL#"
 SelectObject1="document.forms[0].RatingID"
 SelectObject2="document.forms[0].FilmID"
 ValueColumn="FilmID"
 DisplayColumn="MovieTitle">
```

The custom tag itself is implemented in the `RelateTwoSelectLists.cfm` file (also included with this chapter's listings). You are invited to take a look at the listing to get yourself thinking about ways in which JavaScript functionality can be wrapped up in CFML custom tags for easy reuse.

Calling CFCs from JavaScript

As you learned in Chapter 27, any CFC method that uses `access="Remote"` can be invoked via the Adobe Flash player, as Web Services, via HTML forms, or via simple URL invocation. If you access a remote method via the URL, the value that the method returns will be automatically serialized as a WDDX packet. That means that you can call CFCs from JavaScript in much the same way that the robot page was called in Listing 48.9.

NOTE

This applies only to methods that return values via `<cfreturn>`, rather than generating HTML or some other type of ad-hoc output.

For instance, Listing 48.11 creates a simple CFC that can be used in place of the ad-hoc `FilmsRobot.cfm` page shown earlier (Listing 48.10). This component has just one method, `GetFilmsByRating()`, which accepts a rating ID and returns a recordset that contains the ID number and title of each film with that rating.

Listing 48.11 `FilmCFC.cfc`—A ColdFusion Component That Can be Accessed via JavaScript

```
<!---
Name:        FilmCFC.cfc
Author:      Nate Weiss and Ben Forta
Description: Example CFC for supplying
             recordset data to JavaScript
             via WDDX
Created:     02/01/05
--->


<cfcomponent>
  <!--- GetFilmsByRating() method --->
  <cffunction name="GetFilmsByRating"
              returntype="query"
              access="remote"
              hint="Returns films that match the specified RatingID.">

    <!--- Required argument: Rating ID --->
    <cfargument name="RatingID"
                type="numeric"
                required="Yes">

    <!--- Local variable --->
    <cfset var FilmsQuery="">

    <!--- Get film information from database --->
    <cfquery datasource="ows"
             name="FilmsQuery"
             cachedwithin="#CreateTimeSpan(0,0,10,0)#">
      SELECT FilmID, MovieTitle
      FROM Films
      WHERE Ratingid=#ARGUMENTS.RatingID#
    </cfquery>
```

Listing 48.11 (continued)

```
    <!--- Return the query --->
    <cfreturn FilmsQuery>
  </cffunction>

</cfcomponent>
```

You can invoke the `GetFilmsByRating()` method with your browser, using a URL similar to the following (again, you may need to adjust the URL a bit depending on how your ColdFusion server is configured):

```
http://127.0.0.1:8500/ows_adv/17/FilmCFC.cfc?Method=GetFilmsByRating&RatingID=1
```

The CFC should respond with approximately the same WDDX packet that this URL produces (which invokes the simple robot from Listing 48.10):

```
http://127.0.0.1:8500/ows_adv/17/FilmsRobot.cfm?RatingID=1
```

Therefore, it's a simple matter to adjust Listing 48.9 so that it interacts with the CFC instead of the robot page. Simply change this line:

```
<cfset FilmComponentURL="http://127.0.0.1:8500/ows_adv/17/FilmsRobot.cfm?">
```

to this:

```
<cfset FilmComponentURL="http://127.0.0.1:8500/ows_adv/17/
FilmCFC.cfc?Method=GetFilmsByRating">
```

Once that change is made, the Listing 48.9 page will work just as it did before, retrieving the film information from the server in real time. The only difference is that a CFC is now the supplier of data. This is cool because you can use the same CFC to supply data to JavaScript pages, Flash applications, consumers of Web Services, and your own ColdFusion pages.

Passing Simple Variables to ColdFusion

In the "Serializing Packets with WddxSerializer" section of this chapter, you learned about a rather sophisticated way to pass complex, multifaceted data from JavaScript to ColdFusion: by serializing the data into a WDDX packet and posting it to the server. If you only need to pass simple values (such as strings or numbers) to the server, you can use simpler methods.

Passing Variables as URL Parameters

The simplest method of passing variables to ColdFusion is to simply pass them in the URL.

1. Construct a URL that includes the values you want to pass as URL parameters, using the `&name=value` format you already know and love. If a value may include spaces, slashes, or any other "funny" characters (basically anything other than numbers and letters), you must use the `escape()` function to escape the characters; it does the same thing that `URLEncodedFormat()` does in CFML.
2. Tell the browser to navigate to the new URL, using the `document.location.href` property or the `document.location.replace()` function.

The following snippet would create a function that causes the browser to navigate to a fictitious ColdFusion page called `ShowFilms.cfm`, passing URL parameters called `Name` and `Rating` in the URL:

```
function loadPageBasedOnAge(name, age) {
 var rating, url;

 // Decide on a rating, based on the value
 // of the JavaScript age variable
 if (age < 13) {
 rating = "G";

 } else if (age < 18) {
 rating = "PG-13";

 } else {
 rating = "R";
 }

 // Construct a new URL, passing the rating to ColdFusion
 var url = "ShowFilms.cfm?Name=" + escape(name) + "&Rating=" + rating;

 // Navigate to the new URL
 document.location.href = url;
}
```

NOTE

This snippet is just meant to demonstrate the syntax you would use; it doesn't necessarily make any real-world sense. If you were simply collecting a name and age from the user, you would probably just use a normal HTML form to collect the information.

Passing Variables as Form Parameters

Another way to pass simple values is to use hidden form fields. To place a simple value (that is, any value that can be straightforwardly expressed as a string) into a form field, just use JavaScript code like the following, where `myValue` is the JavaScript value that you want to pass to the server:

```
document.forms[0].MyHiddenField.value = myValue;
```

If you wish, the form can then be submitted programmatically, like so:

```
document.forms[0].submit();
```

Listing 48.12 creates a page that subjects the user to a short, three-question quiz about some important dates in his or her life. The user is given only ten seconds to answer each question. The time remaining ticks off visually, at tenth-of-a-second intervals (Figure 48.7).

If the user provides a valid date in time, the amount of time that they took to answer the question is passed to the server (in milliseconds). If the user fails to provide a valid date in time, the page automatically refreshes, displaying the next question in the sequence. When all three questions have been answered (or not answered), the user gets a summary page that shows their responses and the time it took for them to answer each question (Figure 48.8).

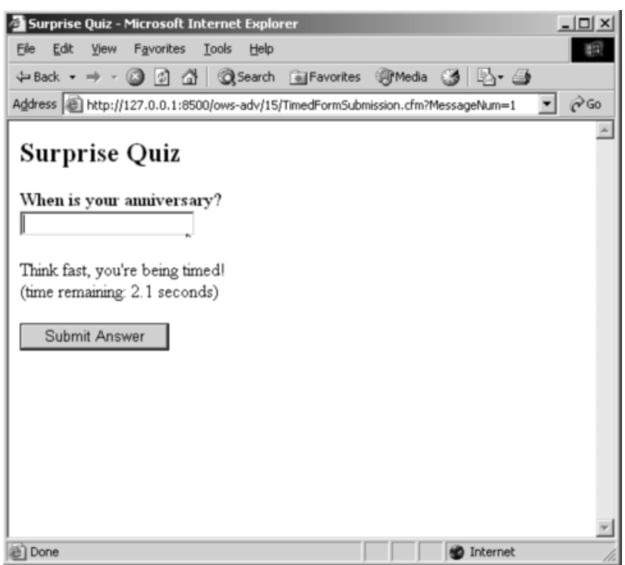


Figure 48.7

A JavaScript-based timer is used to time the user's response.

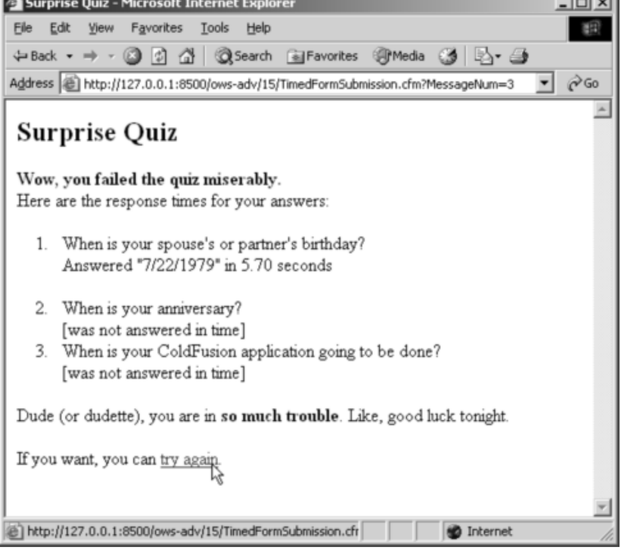


Figure 48.8

A JavaScript-based timer is used to time the user's response.

NOTE

This example is meant to introduce you to a potentially useful idea: passing values to ColdFusion that could, by their nature, only be known to JavaScript. It's not meant to serve as the foundation for a bulletproof quiz or e-learning application. That said , you could adapt this idea to produce a secure testing application that visually displays the time remaining for each section of the test.

Listing 48.12 `TimedFormSubmission.cfm`—Passing JavaScript Variables as URL Parameters

```
<!---
Name:        TimedFormSubmission.cfm
Author:      Nate Weiss and Ben Forta
Description: Demonstrates passing JavaScript
             variables to ColdFusion in the URL.
Created:     02/01/05
--->


<!--- Filename of the current ColdFusion page --->
<cfset CurrentPage=GetFileFromPath(GetBaseTemplatePath())>

<!--- Keep track of the responses on the server --->
<cfparam name="SESSION.TimedResponses"
         type="struct"
         default="#StructNew()#">

<!--- We'll ask the poor user one of these questions at random --->
<cfset Messages[1]="When is your spouse's or partner's birthday?">
<cfset Messages[2]="When is your anniversary?">
<cfset Messages[3]="When is your ColdFusion application going to be done?">

<!--- Maximum number of milliseconds allowed per answer --->
<cfset MaxMillisecondsPerPage=10000>
<!--- If the form is being submitted... --->
<cfif IsDefined("form.TimeElapsed")
  AND IsDefined("form.AnswerDate")
  AND Val(form.TimeElapsed) GT 0
  AND IsDefined("URL.MessageNum")
  AND URL.MessageNum GT 0
  AND StructCount(SESSION.TimedResponses) EQ (URL.MessageNum - 1)>

  <!--- Record the number of seconds elapsed --->
  <!--- Don't allow the user to overwrite the number by reloading the page --->
  <cfif IsDate(form.AnswerDate) OR (form.TimeElapsed GTE MaxMillisecondsPerPage)>
    <cfif StructKeyExists(SESSION.TimedResponses, URL.MessageNum) EQ False>

      <!--- Create a structure that holds user's answer and elapsed time --->
      <cfset AnswerStruct=StructNew()>
      <cfset AnswerStruct.TimeElapsed=form.TimeElapsed>
      <cfset AnswerStruct.IsAnswered=IsDate(form.AnswerDate)>
      <cfif AnswerStruct.IsAnswered>
        <cfset AnswerStruct.AnswerDate=form.AnswerDate>
      </cfif>

      <!--- Save structure in SESSION.TimedResponses by the message number --->
      <cfset SESSION.TimedResponses[URL.MessageNum]=AnswerStruct>
    </cfif>
  </cfif>
</cfif>


<html>
<head>
<title>Surprise Quiz</title>
```

Listing 48.12 (continued)

```
</head>

<h2>Surprise Quiz</h2>

<!---
If all the messages have been responded to
Display the results of the quiz --->
<cfif StructCount(SESSION.TimedResponses) GTE ArrayLen(Messages)>
  <body>

  <!--- Display message --->
  <!--- In a real application, the user would succeed sometimes... :) --->
  <p><strong>Wow, you failed the quiz miserably.</strong><br>
  Here are the response times for your answers:<br>
  <ol>

  <!--- For each of the user's answers --->
  <cfloop from="1"
          to="#ArrayLen(Messages)#"
          index="i">
    <cfoutput>
      <li>
        <!--- This is the message user was responding to --->
        #Messages[i]#<br>

        <!--- If the user provided a response --->
        <cfif SESSION.TimedResponses[i].IsAnswered>
          Answered "#SESSION.TimedResponses[i].AnswerDate#" in
          #NumberFormat(SESSION.TimedResponses[i].TimeElapsed / 1000, "9.99")#
          seconds<br><br>

        <!--- If the user wasn't able to answer the question --->
        <cfelse>
          [was not answered in time]
        </cfif>
      </li>
    </cfoutput>
  </cfloop>
  </ol>

  <!---
  Erase the SESSION.TimedResponses variable
  The quiz will start over if the page is reloaded
   --->
  <cfset StructDelete(SESSION, "TimedResponses")>

  <!--- Display message --->
  <p>Dude (or dudette), you are in <strong>so much trouble</strong>.
  Like, good luck tonight.<br>

  <!--- Link to try again --->
  <!--- Because SESSION.TimedResponses was erased, quiz will begin again --->
  <p>If you want, you can <a href="TimedFormSubmission.cfm">try again</a>.<br>
```

Listing 48.12 (continued)

```
<!---
If not all the questions have been responded to,
Provide a simple form interface for providing an answer
--->
<cfelse>
  <!--- Determine the message number to display now --->
  <cfloop from="1"
          to="#ArrayLen(Messages)#"
          index="ShowMessageNum">
    <cfif NOT StructKeyExists(SESSION.TimedResponses, ShowMessageNum)>
      <cfbreak>
    </cfif>
  </cfloop>

  <!--- Get the text for the message --->
  <cfset Message=Messages[ShowMessageNum]>

  <!--- Custom script functions --->
  <script type="text/javascript"
          language="JavaScript">
    // Because these variables is declared outside of a function block,
    // they are maintained by JavaScript at the page level
    var msMaxPerQuestion=<cfoutput>#MaxMillisecondsPerPage#</cfoutput>;
    var intervalHandle;
    var msMomentPageWasLoaded;

    // Function to increment the msTimeElapsed variable
    // and place its value in the TimeElapsed hidden field
    function fillTimeElapsed() {

      // Determine how many milliseconds have elapsed so far
      msTimeElapsed=new Date().valueOf() - msMomentPageWasLoaded;

      // Place the time elapsed (in milliseconds) in the hidden field
      document.forms[0].TimeElapsed.value=msTimeElapsed;

      // Display the timer value
      var msg=((msMaxPerQuestion - msTimeElapsed) / 1000).toFixed(1);
      document.getElementById("elSecsElapsed").innerhtml=msg;

      // If the time has elapsed
      if (msTimeElapsed >= msMaxPerQuestion) {
        // Stop the timer
        window.clearInterval(intervalHandle);
        // Submit the form
        document.forms[0].submit();
      };
    }


    // This funciton executes when the page first loads
    function initPage() {
      // Records the current time (as number of milliseconds since 1/1/1970)
      msMomentPageWasLoaded=new Date().valueOf();
```

Listing 48.12 (continued)

```
      // Start the timer
      intervalHandle=window.setInterval('fillTimeElapsed()', 100);

      // Set focus to the AnswerDate input field
      document.forms[0].AnswerDate.focus();
    };
  </script>

  <!--- Call the fillTimeElapsed() function once every 10 milliseconds --->
  <body onload="initPage()">

  <!--- Self-submitting form --->
  <cfform action="#CurrentPage#?MessageNum=#ShowMessageNum#"
          method="Post"
          onsubmit="window.clearInterval(intervalHandle)">

    <!--- Hidden field to pass secondsElapsed variable to ColdFusion --->
    <cfinput type="Hidden"
             name="TimeElapsed">

    <!--- Ordinary form field --->
    <p>
    <strong><cfoutput>#Message#</cfoutput></strong>
    <br>
    <cfinput type="Text"
             name="AnswerDate"
             required="Yes"
             validate="date"
             message="Um, fill in the date first.">
    <br>

    <p>Think fast, you're being timed!<br>
    (time remaining: <span id="elSecsElapsed"></span> seconds)<br>

    <!--- Submit button --->
    <p>
    <cfinput type="Submit"
             name="sbmt"
             value="Submit Answer">

  </cfform>

</cfif>

</body>
</html>
```

When this page first appears in the browser, the `onload` event calls the `initPage()` function. Within `initPage()`, a global variable called `msMomentPageWasLoaded` is set to the current time (according to the browser machine's clock, and expressed as the number of milliseconds since midnight on January 1, 1970). In addition, the `window.setInterval()` method is used to create a sort of internal timer which executes the `fillTimeElapsed()` function once every 100 milliseconds (that is, ten times per second).

Within `fillTimeElapsed()`, a global local variable named `msTimeElapsed` is incremented by 100 each time the function is called and is calculated by subtracting the value in `msMomentPageWasLoaded` from the current time (again, in milliseconds). The `msTimeElapsed` , thus keeping track of the approximate number of milliseconds that have passed since the page first loaded. variable thus holds the number of milliseconds that have elapsed since the `initPage()` executed, which in turn is the number of milliseconds that have passed since the page first appeared in the browser.

The number of elapsed milliseconds is stored in the hidden form field named `TimeElasped`; this value will be available to ColdFusion as a normal `FORM` variable when the form is submitted. In addition, a formatted version of the number is displayed by setting the `innerHTML` property of the `<SPAN>` element called `elSecsElapsed`. The `getElementById()` syntax used here can be used to get or set the properties of any scriptable HTML element; the exact properties available will vary from browser to browser. The `innerHTML` property used here is supported by IE 4 and above and Netscape 6 and above (or other Mozilla-based browsers).

If the number of elapsed milliseconds exceeds the maximum number of milliseconds (in this example, 10,0000, or ten seconds), the timer is cleared using `window.clearInterval()`. The form is then submitted using the form's `submit()` method.

The rest of the code is relatively straightforward ColdFusion code that uses a structure called `SESSION.TimedResponses` to remember each user's responses as they encounter the three parts of the quiz. The structure is made up of smaller sub-structures, each containing a `TimeElapsed` property, an `IsAnswered` property that indicates whether the user provided an answer in time, and an `AnswerDate` property which is the user's actual response to the question (if any).

Opening Popup Windows

For better or for worse, one of the things that JavaScript is most used for is to open popup windows. Much of the time, popup windows are an annoyance, but there are times when you may have a legitimate need to create one for your application. You can find a full discussion about popup windows in a JavaScript reference, but this section will quickly introduce you to the basics, emphasizing the fact that you can easily pass variables to ColdFusion as part of the popup-opening process.

To open a popup window, use the `window.open()` method, in the following form:

```
window.open(popupURL, popupName, popupFeatures);
```

The `popupURL` is the URL of the page to display in the popup window. The `popupName` is an optional target name for the popup window (if you provide the name of an existing window, the same window will be reused each time the method is called; if not, a new window is opened each time). The `popupFeatures` argument is an optional comma-separated list of window features, which you can use to control the size and position of the popup window. Table 48.16 lists most of the values you can supply in the `popupFeatures` string; consult a JavaScript reference for a complete listing.

Table 48.16 Window Features for the `window.popup()` Method

FEATURE	DESCRIPTION
`width`	The width of the popup window, in pixels.
`height`	The height of the popup window, in pixels.
`top`	The position of the window, in pixels, from the top of the screen.
`left`	The position of the window from the left edge of the screen.
`resizable`	Whether the window should be resizable (`yes` or `no`).
`scrollbars`	Whether scrollbars should appear in the window (`yes` or `no`). Even when `no`, the scrollbars only appear when necessary.
`status`	Whether the status bar should be displayed at the bottom of the popup window (`yes` or `no`).
`toolbar`	Whether the browser toolbar (with the next and back buttons and so on) should appear at the top of the window (`yes` or `no`).
`menubar`	Whether the usual browser menu bar should appear(`yes` or `no`).
`location`	Whether the URL location (the area where you type a new location to browse to) should appear at the top of the window (`yes` or `no`).

So, the following would open a popup window that is 300 pixels wide and 200 pixels wide, displaying content from the `ShowFilm.cfm` page:

```
window.open("ShowFilm.cfm", "filmPopup", "width=300,height=200");
```

Of course, you are free to make the values of JavaScript variables available to the page you are displaying in the popup window. For instance, if you have a JavaScript variable called `filmID`, you can easily pass it as a URL parameter to the `ShowFilm.cfm` page, like so:

```
window.open(
 "ShowFilm.cfm?FilmID=" + escape(filmID),
 "filmPopup",
 "width=300,height=200");
```

If you provide a name in the `popupName` argument, the popup window will be reused each time a new `window.open()` method executes that uses the same name. This can help avoid a situation where there are too many popup windows strewn about the user's screen. In such a case, it is often helpful to add an `onload="window.focus()"` attribute to the `<body>` tag of the window being opened, so that it moves in front of any other windows each time it is reused, like so:

```
<BODY onload="window.focus()">
```

The `JSRelatedSelects4Popup.cfm` page (included with this chapter's listings) provides a Show Film button that the user can use to display details about the selected film in a popup window, as shown in Figure 48.9. The detail page is provided by `ShowFilm.cfm` (also included with this chapter's listings), which includes the `window.focus()` line shown above so that the popup window for the film details always moves to the front as each film's details are loaded.

Figure 48.9

A pop-up window displays film details when the user clicks the Show Film button.

CHAPTER 49 Using XForms

IN THIS CHAPTER

What's Wrong with HTML Forms? E351
What Is XForms? E352
Creating XForms in CFML E353
XSL: The Extensible Stylesheet Language E367

What's Wrong with HTML Forms?

When programming for the Web, I frequently find myself doing the same things over and over: creating a form that matches my database, preloading the fields in the form from a record, and writing the JavaScript to make my particular application work. With the help of `<cfform>` and some custom tags that implement cool behavior such as date pickers and pseudo-combo boxes, I can eventually get my forms to work the way I want them to, but I find myself developing the same code again and again in each new application.

In addition, my forms need to submit to a page that know exactly what the fields on my form were, and generate new forms that contain the same data with slightly different behaviors depending on what stage of workflow the data is in. For example, creating a press release page, where the person who assigns the task of writing, the author, the editor, and the person responsible for final approval all see the same data, but can edit different parts of it, requiring a lot of conditional logic to make the form do all the different things its got to do (or redeveloping the form four times.)

The idea of XForms is to take what's been learned from years of using HTML forms, and create a new type of forms that's more generally applicable. The primary improvements of XForms, as outlined in the W3C's XForms Recommendation (see `http://www.w3.org/TR/XForms/`) are:

- **Separate Form From Data**. The data used to load a form's data can exist completely separately from the form's implementation, and is an XML document. This is similar to using CSS Style sheets and custom styles to separate the content of a document from the specifics of its implementation.
- **Strong Typing.** HTML forms submit everything as text; defining your own types allows better and more accurate form validation.
- **XML submission.** XForms can submit data to the server as an XML document, which can be stored or processed as needed.

- **Less Use of Scripting.** You can define validation rules and cross-control behaviors in XForms that don't require any scripting in the form.
- **Other Improvements,** including re-use of schemas, the ability to add your own schemas to the language, internationalization, accessibility, and support of many devices including phone systems, handheld devices, and accessible devices such as screen-readers.

What Is XForms?

So what exactly is XForms? XForms is an XML grammar in which you can define a form. It is separated into three parts: data (instance), display (control definition, and binding), and action (submission). Looking at a typical HTML form, we have a form tag that defines its action, which is where the form submits to. In XForms, submitting a form can do different things—submit to an HTML page, copy a file, or other things. This is the "submission" part of XForms. You then define the form controls, which define the display elements. In more general terms, what you are actually defining is *what types of decisions* the user can make—for example, "choose one of these options" (a radio button or single select) or "submit the form now" (an image or submit button.) In addition, using "`value=` `selected`, `checked`, and embedded textbox text defines the values that these controls initially have. Since XForms separates the data from the display, those values are defined in the "data" section of XForms and the controls and what data is displayed in them is defined in the "bind or controls section (which data is bound to which control).

Once XForms defines the information in XML, your browser still can't display it—all your browser understands is HTML. Therefore, it requires both an XForms XML document and something that translates it into HTML. HTML (or at least, the better formed version of it, XHTML) is an XML grammar, so this is accomplished using XSLT, which means Extensible Stylesheet Language Transformations. XSLT defines *transformations* that you can use to define how to translate an XML document into another XML document. XSL is the language that is used by XSLT to define these transformations, so in order to have a complete system you'll need both an XForms form definition and an XSL document that describes how to translate that forms definition into something displayed. We'll see in this chapter how ColdFusion creates XForms, and how it uses XSL to render them.

NOTE

There's no such thing as an 'XForm.' XForms is like 'elk', it describes one or all of them.

Why Use XForms?

The big idea is to explicitly divorce form contents from form presentation. Why is this a good thing? Consider the following scenarios:

- You need to reorder form fields. Instead of having to fight with `<tr>` and `<td>` tags you simply move the fields around. There is no presentation code to worry about.
- You need to copy and paste parts of one form into another. Without presentation code in the way, it really is a simple matter of copy and paste.

- You need a common look for all your forms. Instead of having presentation code in every form you have a single XSL file that all forms use.
- And when that look and feel needs to change, you now have a single file to change, and all forms will inherit that change automatically.
- It makes a lot of sense. And yet very few developers use XForms. Why is that?

Barriers to XForms Adoption

There have been three primary barriers to XForms adoption:

- Unlike HTML forms, XForms syntax is very rigid and strict, and creating the well-formed XML that XForms requires isn't trivial.
- Creating XSL is even less trivial. It isn't an easy language to learn and use.
- Worst of all, many browsers won't know what to do with XForms and XSL, and won't apply the transformation for you.

All things considered, it's easy to see why XForms adoption has been slow. Yet XForms does indeed have value, as already explained. Fortunately, ColdFusion does all the hard work for you.

Creating XForms in CFML

Creating XForms with ColdFusion is very simple because ColdFusion both contains the tags you use to create the form in XML, and the engine that renders the XML as an HTML form, complete with JavaScript, that can be used in any Web browser.

Let's look at a form and see how ColdFusion renders it in XML. The first thing to know about XForms is that they exist entirely as XML documents, and therefore you can't use standard HTML inside an XForms specification.

Or put differently, when we first looked at forms (in Chapter 12, "ColdFusion Forms," in *Adobe ColdFusion 8 Web Application Construction Kit*, *Volume 1: Getting Started*), the forms we created contained controls (the form fields themselves) as well as presentation information (where the label should be displayed, HTML tables, and so on). An XForms form only contains a list of controls and what their attributes are. Attributes include things like:

- Field name
- Field type
- Field label
- Level of nesting (if using fields within groups)
- Type of validation required

There is no presentation at all in the XForms, just lots of information that could be used by whoever (or whatever) needs to provide the presentation.

Listing 49.1 shows a form used to update an actor, using in `<cfform>`. It's been validates the actor's age, and the actor's first and last names are required.

Listing 49.1 `actorForm.cfm`—Actor Update Form, Using `<cfform>`

```
<!---
  actorForm.cfm
  Ken Fricklas (kenf@fricklas.com)
  Modified: 11/1/2007
  Listing 49.1
--->
<!DOCTYPE HTML PUBLIC " -//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd" >
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" >
<title>Sample Actor Form</title>
</head>

<body>
<cfparam name="URL.ActorID" default="1" >
<!--- note: update code left out for brevity --->
<cfquery name="myActor" datasource="ows" >
  select * from Actors where ActorID = #URL.ActorID#
</cfquery>
<cfform action="#cgi.script_name#" method="post" >
<h1>Edit Actor</h1>
<input type="hidden" name="ActorID" value="#myActor.ActorID#" >
<table border="1" >
  <tr>
    <td>First Name</td>
    <td><cfinput name="NameFirst" type="text" value="#myActor.NameFirst#"
required="yes" ></td>
  </tr>
  <tr>
    <td><EM>Real</EM> First Name</td>
    <td><cfinput name="NameFirstReal" type="text" value="#myActor.NameFirstReal#"
required="no" ></td>
  </tr>
  <tr>
    <td>Last Name</td>
    <td><cfinput name="NameLast" type="text" value="#myActor.NameLast#"
required="yes" ></td>
  </tr>
  <tr>
  <tr>
    <td>Age</td>
    <td><cfinput name="Age" type="text" value="#myActor.Age#" required="yes"
validate="integer" range="1,110" message="Age must be a number between 1 and 110."
></td>
  </tr>
  <tr>
    <td>A total babe?</td>
```

Listing 49.1 (continued)

```
        <td>
          yes
          <cfinput name="isTotalBabe" type="radio" value="1"
    checked="#myActor.IsTotalBabe#" >
              
          no
          <cfinput name="isTotalBabe" type="radio" value="0" checked="#NOT
    myActor.IsTotalBabe#" >
        </td>
      </tr>
      <tr>
        <td>An Egomaniac?</td>
        <td>
          yes
          <cfinput name="isEgomaniac" type="radio" value="1"
    checked="#myActor.isEgomaniac#" >
              
          no
          <cfinput name="isEgomaniac" type="radio" value="0" checked="#NOT
    myActor.isEgomaniac#">
          </td>
      </tr>
    </TABLE>
    <BR>
      <input name="doEdit" type="submit" value="Save Changes">
    </cfform>
    </body>
    </html>
```

This form can be seen in Figure 49.1.

Figure 49.1

Actor Update Form.

Listing 49.2 shows the same form, now using an XForms XML form:

Listing 49.2 `actorFormXML.cfm`—Actor Update Form, Using `<cfform type="XML">`

```
<!---
  actorFormXML.cfm
  Actor Form, now using format="XML" Ken Fricklas (kenf@fricklas.com)
  Modified: 11/1/2007
  Listing 49.2
--->
<!DOCTYPE HTML PUBLIC " -//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd" >
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" >
<title>Sample Form</title>
</head>

<body>
<cfparam name="URL.ActorID" default="1" >
<!--- note: update code left out for brevity --->
<cfquery name="myActor" datasource="ows" >
  select * from Actors where ActorID = #URL.ActorID#
</cfquery>
<!--- this time, format="XML" is added.  We'll use the gray CSS skin. --->
<cfform action=" #cgi.script_name#" method="post" format="xml" name="xActorForm"
skin="gray" >
  <cfformitem type="html" >
    <H1>Edit Actor</H1>
  </cfformitem>
  <cfinput type="hidden" name="ActorID" value=" #myActor.ActorID#" >
  <cfinput name="NameFirst" type="text" value=" #myActor.NameFirst#" required="yes"
label="First Name" >
  <cfinput name="NameFirstReal" type="text" value=" #myActor.NameFirstReal#"
required="no" label="Real First Name" >
  <cfinput name="NameLast" type="text" value=" #myActor.NameLast#" required="yes"
label="Last Name" >
  <cfinput name="Age" type="text" value=" #myActor.Age#" required="yes"
validate="integer" range="1,110" message="Age must be a number between 1 and 110."
label="Age?" >
  <cfformgroup type="horizontal" label="A total babe?" >
    <cfinput name="isTotalBabe" type="radio" value="1"
checked="#myActor.IsTotalBabe#" label="yes" >
    <cfinput name="isTotalBabe" type="radio" value="0" checked="#NOT
myActor.IsTotalBabe#" label="no" >
  </cfformgroup>
  <cfformgroup type="horizontal" label="An Egomaniac?" >
    <cfinput name="isEgomaniac" type="radio" value="1"
checked="#myActor.isEgomaniac#" label="yes" >
    <cfinput name="isEgomaniac" type="radio" value="0" checked="#NOT
myActor.isEgomaniac#" label="no" >
  </cfformgroup>
  <cfinput name="doEdit" type="submit" value="Save Changes">
</cfform>
</body>
</html>
```

The output of this can be seen in Figure 49.2. Since we've specified a skin that uses CSS for layout, the formatting is different, but the information is identical with the HTML form. We'll learn much more about skins in the section on Skinning and Styling, later in the chapter.

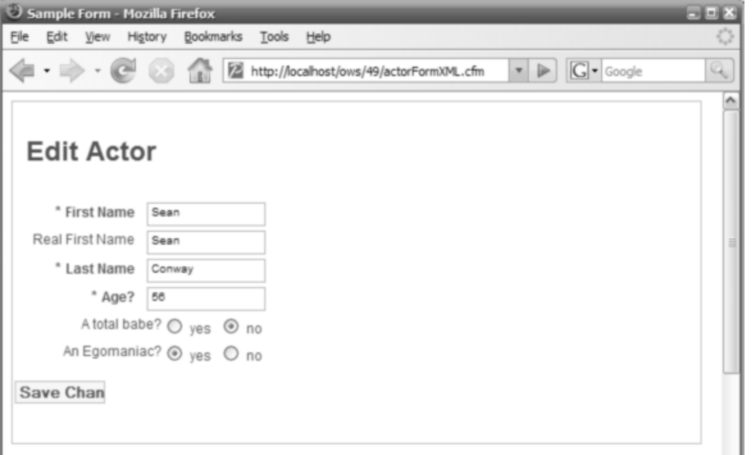


Figure 49.2

Actor Update Form (XForms).

So what's changed? `<cfform>` has a new attribute, `format`, which we've set to XML. (`format` is also used in Flash forms, with the value of "Flash".) What about the rest of the tags? You must use `<cftextarea>` and `<cfselect>` in place of the HTML equivalents – as stated before, your form can't have any HTML controls at all, as they will be ignored. You can also use `<cftree>`, `<cfcalendar>` and `<cfgrid>`.

NOTE

While `<cfslider>` does work in XML forms, many of the attributes are not passed to the control, including the `lookandfeel` and the `label`, which conflicts with the XML attribute of the same name–so the value is never visible on the slider! For these reasons, I view `<cfslider>` as not really useful with XML forms in ColdFusion as currently implemented.

NOTE

The standard size of the submit button control created by the XML skin is not big enough to hold the text "Save Changes so you must use the style= attribute to pass the required size of the control through to the HTML, or modify the skin. We give an example of this below.

The attributes for `<cfform>`, when you're using it with `format="XML"` are listed in Table 49.1.

Table 49.1 `<cfform format="XML">` Attributes

ATTRIBUTE	VALUES (DEFAULT)	NOTES
name	Name of form (`cfform_x`)	Unique name is generated if not provided. If provided, the XML for the form is put into a variable with this name whether or not the form is rendered automatically (see "`SKIN`" later).

Table 49.1 (continued)

ATTRIBUTE	VALUES (DEFAULT)	NOTES
action	Form action (CGI.SCRIPT_NAME)	The URL the form will post to; if you leave this out, it will post back to the current page.
method	"post" or "get"	"put the third XForms option, is not supported in this release.
format	XML	Must be XML for XForms generation.
skin	"basic "basiccss "basiccss_top "beige "blue "bluegray "gray "lightgray"red"silver"default" (default)	Name of the skin to use. If this is a predefined skin, it will use the skins in the CFIDE/SCRIPTS/xsl (by default) directory. See Table 49.5 for more information.
preservedata	yes or no (no)	If yes, when the form submits back to itself (e.g. CGI.SCRIPT_NAME), the values will be those submitted. Otherwise, they will be the values set in the value attributes of the <cfinput> tags.
ONSUBMIT	Javascript code	Allows you to specify Javascript code that will run when the form is submitted. This runs after any ColdFusion generated validation code is run.
ONLOAD	Javascript code	Allows you to specify Javascript code that will run when the page is loaded.
ONRESET	Javascript code	Allows you to specify Javascript code that runs when the form is reset using a reset button. Useful for forcing a reset button to clear a form instead of resetting using custom Javascript.
SCRIPTSRC	Path, (/CFIDE/scripts)	Path to the cfform.js JavaScript file that contains the client-side code necessary to do ColdFusion validation. By default, ColdFusion uses /CFIDE/scripts, or any other value set in the ColdFusion administrator.
CODEBASE	URL (/CFIDE/classes/cf-j2re-win.cab)	URL to the Java plug-ins used for cfgrid, cfslider, or cftree controls for Internet Explorer on Windows. Defaults to /CFIDE/classes/cf-j2re-win.cab
ARCHIVE	URL (/CFIDE/classes/cfapplets.jar)	URL of Java classes for cfgrid, cfslider, and cftree applets for other browsers and platforms. Defaults to /CFIDE/classes/cfapplets.jar.
STYLE	CSS Code	Style information that is passed through to the HTML.

All controls and formatting inside ColdFusion 8 XML forms either must be CFML tags, or must be contained in CFML tags that handle text data. Other HTML that is included in XML forms is discarded, which leaves us with a problem: how do we put labels and text into our forms, and how do we align things so our form looks the way we want it to?

XML Forms use the same syntax as Flash forms, using `<cfformgroup>` to group items within a form and align them, and `<cfformitem>` to add formatted text and other markup to the forms. Both `<cfformgroup>` and `<cfformitem>` require closing tags, because they act as containers that operate on other text and tags inside them.

In the code in Listing 49.2, I used `<cfformgroup>` to add a label and lay out my radio buttons:

```
<cfformgroup type="horizontal" label="A total babe?">
  <cfinput name="isTotalBabe" type="radio" value="1"
            checked="#myActor.IsTotalBabe#" label="yes">
  <cfinput name="isTotalBabe" type="radio" value="0" checked="#NOT
            myActor.IsTotalBabe#" label="no">
</cfformgroup>
```

The `type` attribute in `<cfformgroup>` can be `horizontal`, `vertical`, or `fieldset`. Horizontal, used previously, lays out the `<cfinput>` tags inside it side by side, `vertical` one above the other, and `fieldset` groups its children by drawing a box around them and putting the label (called the legend) over the upper left side of the box, as can be seen in Figure 49.3.

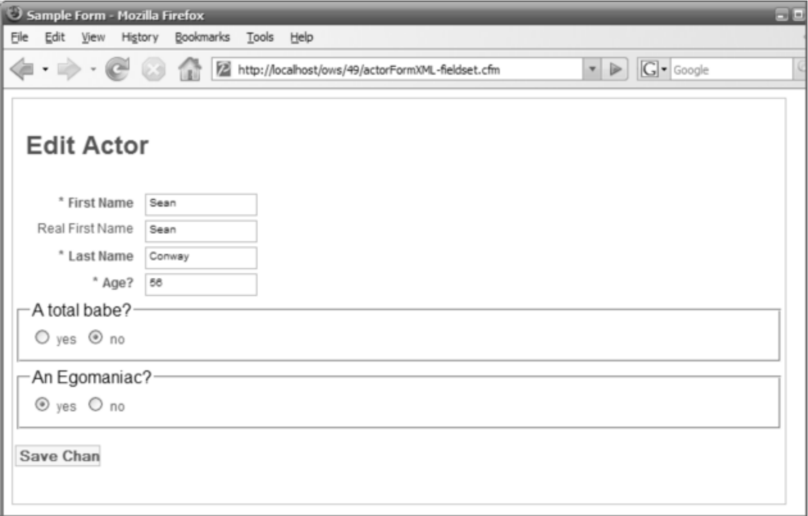


Figure 49.3
Formatting of `<cfformgroup>` `type="fieldset"`.

When you are using the `<cfformgroup>` with the `type` attribute set to `fieldset`, this translates into an HTML `<fieldset>`. You can also pass a `style` attribute to the `<cfformgroup>` tag. Any information in the `style` attribute is passed through unchanged to the HTML.

NOTE

Although the documentation indicates that the `width` attribute can be used, it doesn't really work–use `style="width:300px"` (for example) instead. This is also true for `<cfform>`; use `style` instead of `width` and `height`.

TIP

If you want to lay the controls in a `<cfformgroup>` vertically, put another `<cfformgroup>` inside it that has no label and a type of `vertical`.

The full syntax for `<cfformgroup>` is in Table 49.2.

Table 49.2 `<cfformgroup>` Syntax

ATTRIBUTE	VALUES (DEFAULT)	NOTES
`type`	horizontal, vertical, fieldset	Specifies alignment of items inside; see previous note for more information
`label`	(no default)	Label put next to items, or over border if fieldset
`style`	CSS Text	Passed through to HTML as style

What if we want other text in our form, or perhaps a horizontal line? We can use `<cfformitem>`. `<cfformitem>` does several different things, depending on its `type` attribute. If `type` is set to `HTML`, any text inside the `<cfformitem>` is passed through verbatim to the resulting form. You can put other CFML tags inside the `<cfformitem>` and generate the HTML if desired.

If `type` is set to `text`, the text inside the `<cfformitem>` is XML escaped, similar to the `XMLFormat()` function, and then passed through (for example, a left bracket '<' becomes '`<`')

If the `type` is `hrule`, this creates an `<HR>` tag. Use the `style` attribute to specify its color, width, or anything else that applied to horizontal rules.

Finally, if the `type` is anything else, the string will be passed through as an element to the XML interpreter; we'll use this later in this chapter to extend ColdFusion with our own type.

The full syntax for `<cfformitem>` is in Table 49.3

Table 49.3 `<cfformitem>` Syntax

ATTRIBUTE	VALUES (DEFAULT)	NOTES
`type`	Html, text, hrule, any other string	See previous paragraphs for more information
`style`	CSS Text	Passed through to HTML as style

The XForms Form Definition

So how exactly does an XML form work? An XML form is split into separate sections, corresponding to the values, layout, and *binding*, which means which values connect to which form items.

When ColdFusion generates a form from `<cfform format="xml">`, it puts the text of the XML form definition into a variable with the same name as the form. In the example in Listing 49.2, when the form is generated the form definition is put in the variable `xActorForm`, excerpted in Listing 49.3 (since all the input `type="text"` and radio button sets look pretty much the same).

Listing 49.3 XML from XML Actor Form

```
<form cf:archive="/CFIDE/classes/cfapplets.jar"
cf:codebase="http://java.sun.com/products/plugin/1.3/jinstall-13-
win32.cab#Version=1,3,0,0"
    cf:instance="1" cf:name="xActorForm" cf:scriptsrc="/CFIDE/scripts/"
    html:action="/ows/49/actorFormXML.cfm" html:method="post"
    html:name="xActorForm"
    xmlns:cf="http://www.macromedia.com/2004/cfform"
    xmlns:ev="http://www.w3.org/2001/xml-events"
    xmlns:html="http://www.w3.org/1999/xhtml"
xmlns:xf="http://www.w3.org/2002/XForms">
    <xf:model id="xActorForm">
        <xf:instance>
            <cf:data>
                <ActorID>8</ActorID>
                <NameFirst>Roger</NameFirst>
                <NameFirstReal>N.</NameFirstReal>
                <NameLast>Wilson</NameLast>
                <Age>35</Age>
                <isTotalBabe>1</isTotalBabe>
                <isEgomaniac>1</isEgomaniac>
            </cf:data>
        </xf:instance>
        <xf:submission action="/ows/49/actorFormXML.cfm" method="post"/>
        <xf:bind id="NameFirst"
            nodeset="//xf:model/xf:instance/cf:data/NameFirst" required="true()">
            <xf:extension>
                <cf:attribute name="type">TEXT</cf:attribute>
                <cf:attribute name="onerror">_CF_onError</cf:attribute>
            </xf:extension>
        </xf:bind>
        <xf:bind id="NameFirstReal"
        . . .
        </xf:bind>
        <xf:bind id="Age" nodeset="//xf:model/xf:instance/cf:data/Age"
required="true()">
            <xf:extension>
                <cf:attribute name="type">TEXT</cf:attribute>
                <cf:attribute name="onerror">_CF_onError</cf:attribute>
                <cf:validate type="range">
                    <cf:argument name="message">Age must be a number between 1 and
110.</cf:argument>
                    <cf:argument name="min">1.0</cf:argument>
                    <cf:argument name="max">110.0</cf:argument>
                    <cf:trigger event="onsubmit"/>
                </cf:validate>
                <cf:validate type="integer">
                    <cf:argument name="message">Age must be a number between 1 and
110.</cf:argument>
                    <cf:trigger event="onsubmit"/>
```

Listing 49.3 (continued)

```
                </cf:validate>
            </xf:extension>
        </xf:bind>
        <xf:bind id="isTotalBabe"
            nodeset="//xf:model/xf:instance/cf:data/isTotalBabe" required="false()">
            <xf:extension>
                <cf:attribute name="type">RADIO</cf:attribute>
                <cf:attribute name="onerror">_CF_onError</cf:attribute>
            </xf:extension>
        </xf:bind>
        <xf:bind id="isTotalBabe"
            nodeset="//xf:model/xf:instance/cf:data/isTotalBabe" required="false()">
            <xf:extension>
                <cf:attribute name="type">RADIO</cf:attribute>
                <cf:attribute name="onerror">_CF_onError</cf:attribute>
            </xf:extension>
        </xf:bind>
        <xf:bind id="isEgomaniac"
        . . .
        </xf:bind>
    </xf:model>
    <xf:output><![CDATA[<H1>Edit Actor</H1>]]><xf:extension/>
    </xf:output>
    <xf:input bind="NameFirst" id="NameFirst">
        <xf:label>First Name</xf:label>
        <xf:extension>
            <cf:attribute name="type">text</cf:attribute>
        </xf:extension>
    </xf:input>
    . . .
    <xf:input bind="Age" id="Age">
        <xf:label>Age?</xf:label>
        <xf:extension>
            <cf:attribute name="type">text</cf:attribute>
        </xf:extension>
    </xf:input>
    <xf:group appearance="horizontal">
        <xf:label>A total babe?</xf:label>
        <xf:extension/>
        <xf:select1 appearance="full" bind="isTotalBabe" id="isTotalBabe">
            <xf:extension>
                <cf:attribute name="type">radio</cf:attribute>
            </xf:extension>
            <xf:choices>
                <xf:item>
                    <xf:label>yes</xf:label>
                    <xf:value>1</xf:value>
                    <xf:extension>
                        <cf:attribute name="checked">checked</cf:attribute>
                    </xf:extension>
                </xf:item>
                <xf:item>
                    <xf:label>no</xf:label>
                    <xf:value>0</xf:value>
                    <xf:extension/>
                </xf:item>
```

Listing 49.3 (continued)

```
                </xf:choices>
            </xf:select1>
        </xf:group>
        . . .
        <xf:submit id="doEdit" submission="xActorForm">
            <xf:label>Save Changes</xf:label>
            <xf:extension>
                <cf:attribute name="type">submit</cf:attribute>
                <cf:attribute name="name">doEdit</cf:attribute>
            </xf:extension>
        </xf:submit>
    </form>
```

Let's look at that code a bit at a time. It starts off with the standard XML header:

```
<form cf:archive="/CFIDE/classes/cfapplets.jar"
cf:codebase="http://java.sun.com/products/plugin/1.3/jinstall-13-
win32.cab#Version=1,3,0,0"
    cf:instance="1" cf:name="xActorForm" cf:scriptsrc="/CFIDE/scripts/"
    html:action="/ows/49/actorFormXML.cfm" html:method="post"
    html:name="xActorForm"
    xmlns:cf="http://www.macromedia.com/2004/cfform"
    xmlns:ev="http://www.w3.org/2001/xml-events"
    xmlns:html="http://www.w3.org/1999/xhtml"
xmlns:xf="http://www.w3.org/2002/XForms">
```

This indicates that the data in the document is a form, and it's followed by some header information. The `xmlns:` elements define the URLs of the definitions of the XML *namespaces* where you can find the definitions of the elements in the document. The syntax `xmlns:cf="http://www.macromedia.com/2004/cfform"` indicates that anything that starts with `cf:` is defined by `http://www.macromedia.com/2004/cfform`. The `html:` elements define the elements of an XHTML form, the `xf:` elements are XForms specific, and `ev:` elements are the XML events, which define the interactions between the elements.

The rest of the document is split into two sections—the data definition and the display information. The data definition is surrounded by:

```
<xf:model id="xActorForm">…</xf:model>
```

This tells us that what's inside will be the definition of a model called "`xActorForm` which should look familiar—it's the name of our form. Therefore what this document is describing is the object, or *model*, defined by the data in our form. Inside this are the actual definition of the data, and the bindings. The data definition looks like:

```
<xf:instance>
    <cf:data>
        <ActorID>8</ActorID>
        <NameFirst>Roger</NameFirst>
        <NameFirstReal>N.</NameFirstReal>
        <NameLast>Wilson</NameLast>
        <Age>35</Age>
        <isTotalBabe>1</isTotalBabe>
        <isEgomaniac>1</isEgomaniac>
    </cf:data>
</xf:instance>
```

This indicates that we have an *instance* of a "`xActorForm`". This is where XForms gets cool—the data is defined by the attributes of our data model, instead of artificially by the way the form is laid out as in a typical form. In other words, we're defining what we've got (data) independently from the specific way we're viewing it on this page (presentation).

NOTE

In XForms, this information (or any other part of the form) can be imported from another document or URL, which allows the document to be completely broken up into separate sections, perhaps with completely different developers and sources of data. This means that the form can be reused with different data sets without modification, and without custom programming to insert all the "value=" attributes into the form. The data at the URL which provides the instance data could be generated from a ColdFusion program or directly from a database.

Next comes:

```
<xf:submission action="/ows/49/actorFormXML.cfm" method="post"/>
```

This line tells the form what to do when the form is submitted, namely `post` the data to the URL `/ows/49/actorFormXML.cfm`. The other allowed actions in an XForm are GET, corresponding to an HTML get, and PUT, which copies a file containing that instance data show above, in that format, to a specific location, possibly even the local hard disk (this isn't supported in ColdFusion however).

The glue code is next; here the document defines which of the data elements we defined are going to be displayed in which of the form controls. A typical binding from ColdFusion is the one that attaches the `NameFirst` field to its data. This looks like:

```
<xf:bind id="NameFirst"
    nodeset="//xf:model/xf:instance/cf:data/NameFirst" required="true()">
    <xf:extension>
        <cf:attribute name="type">TEXT</cf:attribute>
        <cf:attribute name="onerror">_CF_onError</cf:attribute>
    </xf:extension>
</xf:bind>
```

This element first defines its name, then shows an XPath definition of which element (nodeset) it's attached to (in the XForms definition, each element can attach to more than one control). The nodeset `//xf:model/xf:instance/cf:data/NameFirst` indicates that any data element in any instance of our model (at any depth, since it starts with "`//`") should show the data from `NameFirst` in the model, namely "Roger". It also defines `required="true()` indicating that the data element must exist, and some ColdFusion-specific extension that ColdFusion uses to pass data to the rendering engine about the error handling and display type.

The rest of the bindings are similar, so let's go on to:

```
<xf:output><![CDATA[<H1>Edit Actor</H1>]]><xf:extension/></xf:output>
```

This is the beginning of our display output. The `xf:output` is followed by a `CDATA`—character data—representation of the HTML that we passed in our `<cfforminput>` tag, which is how text data gets passed through to the output. The display output continues with all the `xf:input`, `xf:group` and similar tags. These define the actual way that various data should be displayed and input, e.g. output only, one choice from a list, etc.

A simple one of these is the Age text area:

```
<xf:input bind="Age" id="Age">
    <xf:label>Age?</xf:label>
    <xf:extension>
        <cf:attribute name="type">text</cf:attribute>
    </xf:extension>
</xf:input>
```

This defines the binding as `Age`, the label text, and a ColdFusion specific extension to tell ColdFusion how to render it. Since the display is separate from the display engine (in other words, the document doesn't "know" its going to be rendered as HTML), this could just as easily be rendered by a voice mail system as a voice mail prompt ("Please indicate Roger's age at the beep. No, his *real* age… Beep."), as by an HTML Web browser.

Table 49.4 shows all the control types for XForms.

Table 49.4 XForms Control Types

CONTROL	HTML CONTROLS	DATA TYPE
input	text	Single line
textarea	textarea	Multiple lines of text
select1	radio, select	Single value from list of choices
select	checkbox, select multiple	Multiple value from list of choices
range	slider	Value from sequence of values
trigger	image	Causes action to occur

NOTE

`SELECT1` and `SELECT` can be defined as "open which allows the user to enter a value or choose an item from the list, similar to a combo box. ColdFusion doesn't support this type of `SELECT` at this time.

? For more information on XForms, controls, and actions, see the XForms specification or the tutorial at `http://xforms.dstc.edu.au/tutorial/`. A complete listing of CFML to XML tag mappings is available in the chapter "Creating Skinnable XML Forms" in the ColdFusion Developer's Guide that is part of your ColdFusion 8 installation, or available for download at `http://www.adobe.com/support/documentation/en/coldfusion/`

Finally, the `xf:group` tags define items that should be grouped together, and how that should be visually represented.

Skinning and Styling: Rendering the Form

As mentioned previously, ColdFusion both generates the XML, and then turns that into standard HTML that your browser can read.

The `<cfform>` has an attribute of "SKIN" that you use to specify the skin that ColdFusion will use to generate the CSS and HTML code for the rendered version of the form. The rules for using SKIN are as follows:

If it's the name of an `.xsl` file in the `cf_webroot\CFIDE\scripts\xsl` directory, or a subdirectory of that directory, without the `.xsl` suffix, ColdFusion will apply the specified skin. If you specify the full qualified path to an `.xsl` file (or use a ColdFusion administrator mapped path), it will use the `.xsl` file you specify. If you omit the skin name, or use "default it uses `default.xsl` from the scripts directory. See Table 49.5 for a more concise list of the directories.

Table 49.5 Skin Format and XSL File Locations

FORMAT OF `<cfform skin="">`	FILE LOCATION
`skin="basic"`	Default directory or its subdirectories
`skin="c:\OWS\Skins\mySkin.xsl"`	Absolute path specified
`skin="mySkin.xsl"`	Current template directory
`skin=""xsl/mySkin.xsl"`	Relative path; `.xsl` subdirectory
`skin="http://www.macromedia.com/skins/mySkin.xsl"`	URL
`skin="default"` or not specified	`default.xsl` from default `scripts/xsl` directory
`skin="none"`	Not rendered

When you specify one of the standard skins, they will include a reference to the CSS style sheet files that define the classes used in the skins. These CSS files are by default contained in the directory `{web root}/CFIDE/scripts/css`. The CSS files are named `{skin}_style.css` where `{skin}` is the name of the corresponding XSL skin, for example `blue_style.css` corresponds to the `blue.xsl` skin. The `basic_style.css` is used by `the basic.xsl` skin. (The `basiccss.xsl` skin is the exception; it uses both `basic2_style.css` and `css_layout.css`.) These files contain classes to format all the control types; these files are very complete, and even contain empty classes for completeness.

You can modify these style sheets to make modifications to the classes to customize your application's layout. For example, in Figure 49.2, the submit button is too narrow to contain all the text for the "Save Changes" button. Looking at the source, we see this uses the style `.cfButton`, defined in the `{webroot}/CFIDE/scripts/css/style_lightgray.css` (we used the skin "light gray") as:

```
.cfButton{
  background-color: #f7f7f7;
  border:1px solid #cabba9;
  width: 80px;
  color: #48585f;
  font-weight:bold;
  margin-bottom:10px;
  margin-right:10px;
  margin-top:10px;
}
```

This shows that the width of a button is only 80 pixels! By removing the `width: 80px;` line, or changing it to a larger value, we can fix the formatting of this submit button. Here I've reformat-

ted the style sheet to eliminate the width, and make a more dramatic border with a dashed, 2-pixel border.

```
.cfButton{
  background-color: #f7f7f7;
  border:2px dashed #cabba9;
  color: #48585f;
  font-weight:bold;
  margin-bottom:10px;
  margin-right:10px;
  margin-top:10px;
}
```

The results of this can be seen in Figure 49.4.

Figure 49.4

XML Form with Reformatted Submit Button (partial).

NOTE

If you define the `skin` attribute as `none,` or if ColdFusion can't find the XSL file with the given name, it won't render the page at all; this allows you to create XForms that can be rendered by external rendering engines such as those mentioned at the beginning of this chapter.

NOTE

We could instead manually add the style to each button using the attribute style="width:auto;border:2px dashed #cabba9 but this is more useful, since each button will not need to be individually styled.

"Great!" you're probably thinking, "now what's in those files?" What's in those files is XSL, the Extensible Stylesheet Language.

XSL: The Extensible Stylesheet Language

XSL is an XML document type that defines *transformations* that define how to translate one variation of XML to another, in this case taking the XForms XML data and turning it into HTML. The great thing about having the XSL being used from a separate file is that it allows you to define multiple skins, and apply them to the same form. This means that a form can change based on a user's

preference, in the same way as modifying a style sheet. You can also change the actions, validations, and transformations, and customize them to the user. This enables you implement forms that are based on the same data, rendered with the same code, but provide entirely different functionality depending on the type of user who is using the page.

This book doesn't have the space to define XSL in depth, but I'll describe the basics of how it works and how you can use it to create and extend the XSL that's been provided by ColdFusion to define your own libraries.

NOTE

The complete definition of XSLT can be found at `http://www.w3.org/TR/xslt`.

XSL works by doing XPath searches, which return groups of matches called *nodesets*, and applying rules to them. The XSL language has three parts, *XPath*, the language use for referencing the various parts of an XML document, *XSLT*, or XSL Transformations, which is a language for describing the transformation of one XML document into another (in this case, XHTML), and *XSL*, the Extensible Stylesheet Language, which is XSLT plus a set of objects and properties that allow you to format the document.

The XSL documents we'll be dealing with are also called XSL stylesheets; their XML document root is `<xsl:stylesheet>`, or sometime `<xsl:transform>`, which is turns out is exactly the same thing.

Here's a very short example of an XSL stylesheet that converts an XML document into HTML.

First, here's the overly simply document we want to translate:

```
<?xml version="1.0"?>
<italic>This is in italics</italic> and this isn't.
```

All we want to do is convert this to an HTML document, with the `<italic>` converted to an HTML `<EM>` tag. This style sheet will do that:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:template match="italic">
  <EM><xsl:apply-templates/></EM>
</xsl:template>

</xsl:stylesheet>
```

How does that work? Each of the `<xsl:template>` tags in the document match text in the document. `match="italic"` in the above matches the `<italic>` in the document, and anything that doesn't match is passed through by the `<xsl:apply-templates>`, which tells the system to apply any template handlers that follow this one (if you leave that out, it'll stop processing here, and it applies the templates in the order they are in the document.) Since there's no deeper template—there aren't any more in the file at all—the default handler, which doesn't do anything at all, handles them by just passing through the text.

Therefore, the output of this stylesheet applied to this XML file would be:

```
<EM>This is in italics</EM> and this isn't.
```

The most important elements of XSL (as far as the ColdFusion 8 XSL stylesheets are concerned) and what they do are in Table 49.6. All XSL elements start with xsl.

Table 49.6 XSL Elements

XSL ELEMENT	DESCRIPTION
`<xsl:template match=xpath-pattern name=template-name>`	Match some part of the input document and run what's inside.
`<xsl:apply-templates>` current item.	Continue processing more templates on the
`<xsl:call-template name=template-name>`	Call another template by the name specified in the `<xsl:template>` and continue processing there.
`<xsl:if test=boolean-expression>`	Contents are only evaluated if the expression is true. For else functionality, use `<xsl:choose>`.
`<xsl:choose>, <xsl:when test=boolean-expression>,<xsl:otherwise>`	These work like `<cfswitch>`, except each case defined by `<xsl:when>` contains its own conditional expression, providing a if/elseif/else functionality.
`<xsl:for-each select=expression>`	Process each item matched by the loop expression.
`<xsl:text>`	Inserts literal text.
`<xsl:value-of select=string-expression>`	Inserts the value of the expression, like `evaluate()` in CFML.
`<xsl:param name=value>`	Allows values to be passed into template. All ColdFusion XSL templates are passed `HTTP_USER_AGENT`, `SCRIPTSRC`, and `CONTEXTPATH`.
`<xsl:with-param name=value>`	Passes in the value to a receiving `<xsl:param>` in another template.
`<xsl:include href=path-or-URL>`	Includes another template. Works like `<cfinclude>`, in that no variable protect is provided and the file is included and processed at the point and in the order included. If a template is redefined, the last one defined will be used, overwriting any previous templates with the same name.
`<xsl:import href=path-or-URL>`	Same as `<xsl:include>`, except that if any templates in the included file have the same name as one already defined, they are ignored.

Table 49.6 (continued)

XSL ELEMENT	DESCRIPTION
`<xsl:element name=value>`	Lists the names of attribute sets (`<xsl:attribute-set>`) to be copied into the result tree. Standard attributes are defined with `<xsl:attribute>` tags. Mostly this is used with `name="input` which creates an HTML `<input>` tag.
`<xsl:attribute name=value>`	Defines an attribute within an `<xsl:element>`

All of the expressions in Table 49.6 are either XPath expressions that match some part of the document, or variables referenced by `${varname}`.

There is an XSL file for every skin provided with ColdFusion, as well as several utility files that are included by the skins. These files are shown in Table 49.7.

Table 49.7 Skin Format and XSL File Locations

XSL FILE	DESCRIPTION
basic.xsl, basiccss.xsl, basiccss_top.xsl, color.xsl	Form formats for the standard file types. basiccss.xsl lays out using CSS, basic.xsl lays out using tables, and basiccss_top.xsl lays out using CSS, but with the label above the form items rather than to the left. The color.xsl files (for example, lightgray.xsl) also use table based layout.
_cfformvalidation.xsl	ColdFusion validation rules (required, integer, range, etc.)
_formelements.xsl	Standard form elements
_group_horizontal_css.xsl, _group_horizontal_css_top.xsl, _group_horizontal_table.xsl, _group_vertical_css.xsl, _group_vertical_css_top.xsl, _group_vertical_table.xsl, _group_fieldset.xsl	cfformgroup transformations, CSS or table version. Versions with _top are used in basiccss_top.xsl.

Let's look at one of the XSL files provided as part of the ColdFusion installation to see how they work. These files are located on your Web server at `{webroot}/CFIDE/scripts/XSL`. Listing 49.4 shows the `basic.xsl` file that comes with ColdFusion.

Listing 49.4 `basic.xsl`—Header

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright (c) 1995-2005 Macromedia, Inc. All rights reserved. -->

<xsl:stylesheet version="1.0" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:cf="http://www.macromedia.com/2004/cfform"
```

Listing 49.4 (continued)

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xf="http://www.w3.org/2002/xforms" xmlns:html="http://www.w3.org/1999/xhtml"
exclude-result-prefixes="xsi cf xsl xf html">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
  <xsl:output omit-xml-declaration="yes"/>
```

This part of the file contains the definition of the stylesheet and defines the name spaces that are used, the same as in our XML form. The two `<xsl:output>` lines define the fact we're generating an XML file, and that we won't need the `<?xml …>` statement that normally appears at the top of any XML file.

Listing 49.4 (cont'd) `basic.xsl`—Parameter Definitions

```
  <xsl:param name="HTTP_USER_AGENT"/>
  <xsl:param name="SCRIPTSRC"/>
  <xsl:param name="CONTEXTPATH"/>
```

The parameters `HTTP_USER_AGENT`, `SCRIPTSRC`, and `CONTEXTPATH` are next; `<xsl:param>` statements must always be the first thing after the stylesheet definition.

```
  <!-- include cfform javascript generation -->
  <xsl:include href="_cfformvalidation.xsl" />
The next thing the template does is include the form validation templates, which are
used in the <xsl:call-template> tags below.
  <!-- start form -->
  <xsl:template match="/">
    <xsl:variable name="formName" select="//form/@cf:name"/>
```

The first `<xsl:match="/">` tells the template to match the top level, which matches all the elements in the template. Everything in here—the rest of the file—will be run for the entire file. The next line defines a local variable called `formName`, which we can refer to by `$formName` later in the code. The `select="//form/@cf:name"` matches any form elements found at any level, followed by anything that starts with `@cf:name`. Now we start to use all those `cf:xxx` elements that were in Listing 49.3, the XML output of our form. Our form statement looked like:

```
<form cf:archive="/CFIDE/classes/cfapplets.jar"...
    cf:instance="1" cf:name="xActorForm" cf:scriptsrc="/CFIDE/scripts/"
    ...>
```

Therefore, this matches the `cf:name` attribute, which was "`xActorForm`". ColdFusion's XML form generation works mostly by leaving a lot of these types of placeholders in the form, so when the XSL starts matching element it can tell what the form's author originally intended to do, and render it back to the desired HTML.

Listing 49.4 (cont'd) JavaScript Generation

```
<!-- generate the correct javascript, based on xf:bind, to validate onSubmit -->
<xsl:call-template name="onSubmitValidation" />
```

This template, `onSubmitValidation`, is located in the `_cfformvalidation.xsl` file that we included earlier. I'll leave the code out of this document for brevity, but this template creates the script source and starts building the standard form validation JavaScript, the same code used in a plain-

Jane <cfform>. As additional elements are found in the document, their validation code is added to this function invocation. In addition, a variable named `$SCRIPTSRC` is filled in with the value of the `scriptsrc attribute of the <cfform>`, if any, or the default from the `<xsl:param>`.

Listing 49.4 (cont'd) Link to Stylesheet

```
<xsl:element name="link">
<xsl:attribute name="href"><xsl:value-of
select="$SCRIPTSRC"/>css/basic_style.css</xsl:attribute>
<xsl:attribute name="rel">stylesheet</xsl:attribute>
<xsl:attribute name="type">text/css</xsl:attribute>
<xsl:attribute name="media">all</xsl:attribute>
</xsl:element>
```

This code generates the stylesheet link, using the `$SCRIPTSRC` we just generated.

Listing 49.4 (cont'd) Create the Form

```
<xsl:element name="div">
  <xsl:attribute name="class">cfform</xsl:attribute>
    <xsl:if test="/form/@html:width or /form/@html:height or /form/@html:style">
      <xsl:attribute name="style">
        <xsl:if test="/form/@html:width">
          width: <xsl:value-of select="/form/@html:width"/>;
        </xsl:if>
        <xsl:if test="/form/@html:height">
          height: <xsl:value-of select="/form/@html:height"/>;
        </xsl:if>
        <xsl:if test="/form/@html:style">
          <xsl:value-of select="/form/@html:style"/>
        </xsl:if>
      </xsl:attribute>
    </xsl:if>
    <!--
    generate the correct root form tag, based on the submission definition in the
model
    -->
    <xsl:element name="form">
       <xsl:attribute name="id"><xsl:value-of select="$formName"/></xsl:attribute>
       <xsl:attribute name="action">
         <xsl:value-of select="/form/xf:model/xf:submission/@action"/>
       </xsl:attribute>
       <xsl:attribute name="method">
         <xsl:value-of select="/form/xf:model/xf:submission/@method"/>
       </xsl:attribute>
       <xsl:choose>
         <xsl:when test="/form/@html:onsubmit">
           <xsl:attribute name="onsubmit">
             <xsl:value-of select="/form/@html:onsubmit"/>
           </xsl:attribute>
         </xsl:when>
         <xsl:otherwise>
           <xsl:attribute name="onsubmit">
             <xsl:value-of select="concat(concat('return _CF_check', $formName),
                                  '(this);')"/>
           </xsl:attribute>
```

Listing 49.4 (cont'd) (continued)

```
        </xsl:otherwise>
      </xsl:choose>

      <!-- loop over the form children elements -->
      <xsl:for-each select="//form/@html:*">
        <xsl:if test="local-name() != 'width' and local-name() != 'height'
             and local-name() != 'validate' and local-name() != 'validation'
             and local-name() != 'type' and local-name() != 'onsubmit'
             and local-name() != 'style'">
          <xsl:attribute name="{local-name()}">
            <xsl:value-of select="."/>
          </xsl:attribute>
        </xsl:if>
      </xsl:for-each>
```

This code matches the form elements, passing through any JavaScript that the user wished to add via `onsubmit, style information, height` and `width`. Note the use of `<xsl:value-of>` to extract the elements we want.

Listing 49.4 (cont'd) Add the Form's Child Elements

```
        <!-- loop over the form children elements -->
        <xsl:for-each select="*[not(self::xf:model)]">
          <xsl:call-template name="vertical"/>
        </xsl:for-each>
```

The child elements are processed by calling the "vertical" template for anything inside the form. That complex `select chooses any nodes that aren't the form element itself (not self).`

Listing 49.4 (cont'd) Server-Side Validation

```
<!-- generate the correct hidden fields, based on xf:bind, to trigger server side
validation -->
<xsl:call-template name="onServerValidation" />
```

The system now adds hidden fields to provide server side validation for those users who don't have JavaScript enabled, another nice thing ColdFusion's XML forms do for you.

Listing 49.4 (cont'd) `<cfformgroup>` Processing

```
        </xsl:element>
      </xsl:element>
    </xsl:template>
    <xsl:template match="xf:model"/>
    <xsl:template match="xf:extension"/>
<!-- *************************************************************************
       Supported Groups
*************************************************************************-->
    <!-- passthrough match for any groups not defined -->
    <xsl:template match="xf:group">
      <xsl:apply-templates select="*"/>
    </xsl:template>

    <!-- include groups that will be supported for this skin-->
```

Listing 49.4 (cont'd) (continued)

```
        <xsl:include href="_group_vertical_table.xsl" />
        <xsl:include href="_group_horizontal_table.xsl" />
        <xsl:include href="_group_fieldset.xsl"/>
```

The next part of the close ends the initial processing, and then ignores any `model` and `extension` tags by matching them and not doing anything with them. It then matches any group attributes, created by `<cfformgroup>` tags, and applies any remaining templates to them. The next three includes define the processing of those groups. These templates match any elements inside them, and wrap them in table elements or a fieldset for HTML display.

Listing 49.4 (cont'd) Form Elements Processing

```
<!-- ****************************************************************************
       Form Elements
****************************************************************************-->
  <!-- include the default rules for form elements.-->
  <xsl:include href="_formelements.xsl" />
</xsl:stylesheet>
```

Finally, we include the form elements themselves, which are defined in `_formelements.xsl`. These configure the standard form elements, and may actually be added to as we'll see in the next section.

Writing Extensions to XForms Using XSL

There are several things to know if you want to extend XForms in ColdFusion:

- `<cfformgroup>` and `<cfformitem>` allow user-defined types, which you can access via a generated id called `appearance`.
- If you want to define additional control types, use an existing form control types and add whatever additional attributes you want to match to the `<cfinput> tags. Unrecognized attributes to <cfinput> are` passed through to the XSL untouched, and you can override existing input type handling from `_formelements.xsl` by modifying that file, or including your own.
- Include your own XSL files last, which will cause them to override any previous templates with the same name. Use narrower select criteria (match only input tags with an attribute of `myType`, for example) so the default control handling will still work.
- `Modify _cfformvalidation.xsl` to add custom JavaScript or server-side validation.

Let's modify our form to provide a color chooser custom control. The control will simply stick a hidden field into our form and put three image tags into our form that will fill in the hidden field when clicked. We need to be able to provide the name of the form control as an attribute, and also provide a label to the form.

We want the form to look like Figure 49.5.

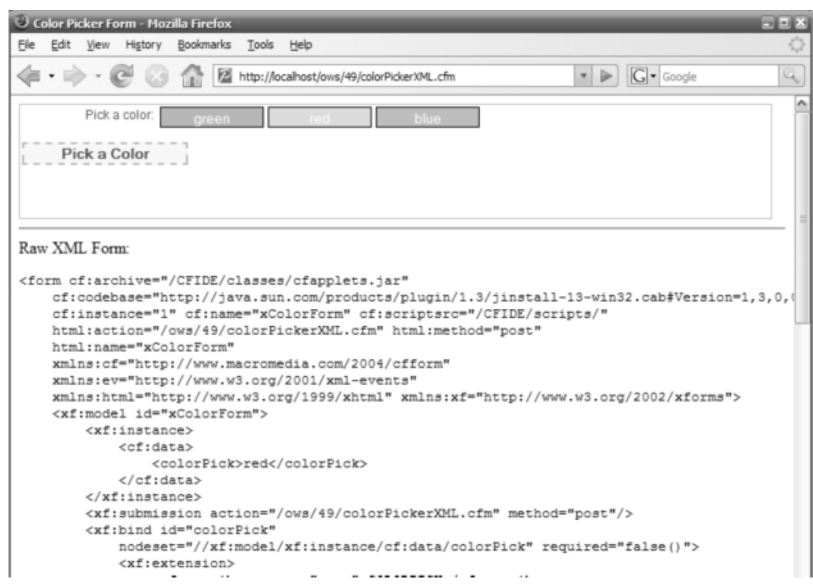


Figure 49.5

Color Picker Custom Form Control.

The HTML source we want to generate will look like Listing 49.5.

Listing 49.5 `ColorPicker.html`—Color Picker Control

```
<link href="/CFIDE/scripts/css/custom_style.css" rel="stylesheet">
<form name="xColorForm">
<div class="cfform">
  <label for="colorPick">Pick a color:</label>
  <!-- hidden class is defined in custom_style.css -->
   <input id="colorPick" name="colorPick" class="hidden" value="red">
   <input type="button" value="green" id="g1"
      style="background-color:#33FF99; color:white; border:1px solid black;
width:100px; height:20px;"

onClick="document.xColorForm.colorPick.value='green';this.style.backgroundColor
='green';r1.style.backgroundColor='#FFCCCC';b1.style.backgroundColor='#99CCCC';retur
n false"/>
  <input type="button" value="red" id="r1"
       style="background-color:#FFCCCC; color:white; border:1px solid black;
width:100px; height:20px;"

onClick="document.xColorForm.colorPick.value='green';this.style.backgroundColor
='red';g1.style.backgroundColor='#33FF99';b1.style.backgroundColor='#99CCCC';return
false"/>
  <input type="button" value="blue" id="b1"
      style="background-color:#99CCCC; color:white; border:1px solid black;
width:100px; height:20px;"
      onClick="document.xColorForm.colorPick.value='blue';this.style.backgroundColor
='blue';r1.style.backgroundColor='#FFCCCC';g1.style.backgroundColor='#33FF99';return
false"/>
  <input type="submit" value="Pick!">
</div>
</form>
```

We could define a hidden style right in the file, but instead I'm going to copy the `basic.css` file to a `custom_style.css`, and define the `hidden` style simply as:

Listing 49.5 (cont'd) `custom_style.css` (excerpt)

```
.hidden {
   display:none;
}
```

In order to generate this code, we'll use a `<cfinput type="colorPick">` command in our `<cfform>`, as can be seen in Listing 49.6.

Listing 49.6 `colorPickerXML.cfm`—XML Form for Color Picker

```
<!---
  colorPickerXML.cfm
  Color Picker Form
  Ken Fricklas (kenf@fricklas.com)
  Modified: 11/1/2007
  Listing 49.6
--->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                       "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Color Picker Form</title>
</head>

<body>
<cfif isdefined("form.subButton")>
   You chose <cfoutput>#form.colorPick#</cfoutput>
</cfif>
<cfform action="#cgi.script_name#" method="post" format="xml" name="xColorForm"
skin="custom">
  <cfinput type="colorPick" name="colorPick" value="red" label="Pick a color:">
  <cfinput type="submit" name="subButton" value="Pick a Color" style="width:160px">
</cfform>
<hr>
Raw XML Form:
<cfoutput>#HTMLCodeFormat(xColorForm)#</cfoutput>
</body>
</html>
```

We're dumping out the form XML to see what code is generated at the end of the template, as shown in Listing 49.7.

Listing 49.7 Color Picker XML Output

```
<form cf:archive="/CFIDE/classes/cfapplets.jar"
    cf:codebase="http://java.sun.com/products/plugin/1.3/jinstall-13-
win32.cab#Version=1,3,0,0"
    cf:instance="1" cf:name="xColorForm" cf:scriptsrc="/CFIDE/scripts/"
    html:action="/ows/49/colorPickerXML.cfm" html:method="post"
    html:name="xColorForm"
    xmlns:cf="http://www.macromedia.com/2004/cfform"
```

Listing 49.7 (continued)

```
        xmlns:ev="http://www.w3.org/2001/xml-events"
        xmlns:html="http://www.w3.org/1999/xhtml"
         xmlns:xf="http://www.w3.org/2002/xforms">
        <xf:model id="xColorForm">
            <xf:instance>
                <cf:data>
                    <colorPick>red</colorPick>
                </cf:data>
            </xf:instance>
            <xf:submission action="/ows/49/colorPickerXML.cfm" method="post"/>
            <xf:bind id="colorPick"
                nodeset="//xf:model/xf:instance/cf:data/colorPick" required="false()">
                <xf:extension>
                    <cf:attribute name="type">COLORPICK</cf:attribute>
                    <cf:attribute name="onerror">_CF_onError</cf:attribute>
                </xf:extension>
            </xf:bind>
        </xf:model>
        <xf:input bind="colorPick" id="colorPick">
            <xf:label>Pick a color:</xf:label>
            <xf:extension>
                <cf:attribute name="type">colorpick</cf:attribute>
            </xf:extension>
        </xf:input>
        <xf:submit id="subButton" submission="xColorForm">
            <xf:label>Pick a Color</xf:label>
            <xf:extension>
                <cf:attribute name="type">submit</cf:attribute>
                <cf:attribute name="name">subButton</cf:attribute>
            </xf:extension>
        </xf:submit>
    </form>
```

As Listing 49.7 shows, it generated `<cf:attribute name="type">colorpick</cf:attribute>`, which is what we want to look for when processing the input fields. Now it's time to get our hands dirty and write some XSL. All we need to do is look for the input with a type of `colorpick`, and add the HTML we previously wrote, filling in the values for the form and control names.

The easiest way I've found to pick out this type requires some modifications to the `_formelements.xsl` file, so I've copied it to a local file called `_cpFormElements.xsl` (`cp` for color picker!). The modifications to this file occur in the test for the input tag, near `<xsl:element name="input">`. The existing code in `_formelements.xsl` looks like Listing 49.8.

Listing 49.8 `_formelements.xsl` (excerpt)

```
    <xsl:for-each select="xf:extension/cf:attribute">
      <xsl:if test="@name != 'width' and @name != 'validate' and @name = 'validation'">
        <xsl:attribute name="{@name}"><xsl:value-of select="text()"/></xsl:attribute>
      </xsl:if>
    </xsl:for-each>
```

This code simply finds any `cf:attributes` that were passed in, and passes them back out unchanged. Our `type="colorPick"` is one of these, so we don't want to ignore it. Instead we'll build a template specifically to generate our code, and call that. The changed code does the following:

- Checks to see if the name is `width`, `validate`, or `validation`, as it used to, and ignores those (same as before).
- If it's none of those, but it is `type`, check to see if it's `colorpick`. If so, set the class to `hidden` to hide the existing control which will be filled in from the color picker. Then it calls our custom template, passing it the ID of the control to use in the JavaScript.
- If it's any other `type` attribute, just pass it through.
- And finally, if it didn't match any of those—it's a different attribute than `type`, `width`, `validate`, or `validation`—then pass it through again.

The modified code is in Listing 49.9.

Listing 49.9 `_cpFormElements.xsl` (excerpt)

```
<xsl:for-each select="xf:extension/cf:attribute">
  <xsl:choose>
    <xsl:when test="@name = 'width' or @name = 'validate' or @name = 'validation'">
      <!-- ignore -->
    </xsl:when>
    <xsl:when test="@name = 'type'">
      <xsl:choose>
        <!-- test for 'colorpick' in the text of the attribute -->
        <xsl:when test="text() = 'colorpick'">
          <!-- mark hidden, then call the colorpick in colorPick.xsl -->
          <xsl:attribute name="class">hidden</xsl:attribute>
          <xsl:call-template name='colorpick'>
            <xsl:with-param name="id"><xsl:value-of select="$id"/></xsl:with-param>
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <!-- its another type attribute, so pass it through -->
          <xsl:attribute name="'type'"><xsl:value-of
                          select="text()"/></xsl:attribute>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:when>
    <xsl:otherwise>
      <!-- its some other attribute, so pass it through unchanged -->
      <xsl:attribute name="{@name}"><xsl:value-of select="text()"/></xsl:attribute>
    </xsl:otherwise>
  </xsl:choose>
</xsl:for-each>
```

The color picker itself is pretty short. We just build the input tags using `<xsl:element>` and `<xsl:attribute>`. Text handling in XSL is a little clunky, so I create variables to hold both parts of the JavaScript text to make my life easier (lots of string concatenation!). I use an `<xsl:param>` to get

the id value from the calling template, `_cpFormElements.xsl`, in the code shown in Listing 49.9. This is in Listing 49.10.

Listing 49.10 `colorPick.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright (c) 1995-2005 Macromedia, Inc. All rights reserved. -->
<!--
  colorPick.xsl
  Color Picker XSL to generate javascript color picker
  Ken Fricklas (kenf@fricklas.com)
  Modified: 11/1/2007
  Listing 49.10
-->
<xsl:stylesheet version="1.0"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:cf="http://www.macromedia.com/2004/cfform"
xmlns:xf="http://www.w3.org/2002/xforms"
xmlns:html="http://www.w3.org/1999/xhtml"
exclude-result-prefixes="xsi cf xsl xf html">

<!-- <cfinput type="colorpick" > -->
<xsl:template name="colorpick">
  <!-- get the id value of the control to put in the javascript -->
  <xsl:param name="id"/>
  <!-- get the form name -->
  <xsl:variable name="formName" select="//form/@cf:name"/>
  <!-- create 2 variables to make the javascript code below less repetitive -->
  <xsl:variable name="jstext1" select="concat('document.',$formName, '.', $id,
'.value=')"/>
  <xsl:variable name="jstext2" select="';return false'"/>
  <!-- create variables to hold the ids of the color elements, unique to this form
instance -->
  <xsl:variable name="r1id" select="concat('r1_',$formName, '_', $id)"/>
  <xsl:variable name="g1id" select="concat('g1_',$formName, '_', $id)"/>
  <xsl:variable name="b1id" select="concat('b1_',$formName, '_', $id)"/>
  <!-- build the inputs -->
  <xsl:element name="input">
    <xsl:attribute name="type">button</xsl:attribute>
    <xsl:attribute name="value">green</xsl:attribute>
    <xsl:attribute name="id"><xsl:value-of select="$g1id"/></xsl:attribute>
    <xsl:attribute name="style">background-color:#33FF99; color:white; border:1px
solid black; width:100px; height:20px;</xsl:attribute>
    <xsl:attribute name="onClick">
      <xsl:value-of select="$jstext1"/>'green';this.style.backgroundColor
='green';<xsl:value-of select="$r1id"/>.style.backgroundColor='#FFCCCC';<xsl:value-
of select="$b1id"/>.style.backgroundColor='#99CCCC'<xsl:value-of select="$jstext2"/>
    </xsl:attribute>
  </xsl:element>
  <xsl:element name="input">
    <xsl:attribute name="type">button</xsl:attribute>
    <xsl:attribute name="value">red</xsl:attribute>
    <xsl:attribute name="id"><xsl:value-of select="$r1id"/></xsl:attribute>
    <xsl:attribute name="style">background-color:#FFCCCC; color:white; border:1px
solid black; width:100px; height:20px;</xsl:attribute>
```

```
    <xsl:attribute name="onClick">
```

Listing 49.10 (continued)

```
      <xsl:value-of select="$jstext1"/>'red';this.style.backgroundColor
='red';<xsl:value-of select="$g1id"/>.style.backgroundColor='#33FF99';<xsl:value-of
select="$b1id"/>.style.backgroundColor='#99CCCC'<xsl:value-of select="$jstext2"/>
    </xsl:attribute>
  </xsl:element>
  <xsl:element name="input">
    <xsl:attribute name="type">button</xsl:attribute>
    <xsl:attribute name="value">blue</xsl:attribute>
    <xsl:attribute name="id"><xsl:value-of select="$b1id"/></xsl:attribute>
    <xsl:attribute name="style">background-color:#99CCCC; color:white; border:1px
solid black; width:100px; height:20px;</xsl:attribute>
    <xsl:attribute name="onClick">
      <xsl:value-of select="$jstext1"/>'blue';this.style.backgroundColor
='blue';<xsl:value-of select="$r1id"/>.style.backgroundColor='#FFCCCC';<xsl:value-of
select="$g1id"/>.style.backgroundColor='#33FF99'<xsl:value-of select="$jstext2"/>
    </xsl:attribute>
  </xsl:element>
</xsl:template>
</xsl:stylesheet>
```

The last step is putting it all together. All we need to do is create our `custom xsl` file, which I'm calling `custom.xsl`. This is just a copy of the `lightgrey.xsl` file we exhaustively examined earlier in the chapter, with one minor change: we need to include our new `colorpick.xsl` file. Here are the last few lines of `custom.xsl`, shown in Listing 49.11.

Listing 49.11 `custom.xsl` (excerpt)

```
<!-- include the default rules for form elements.-->
  <xsl:include href="_cpformelements.xsl" />
  <xsl:include href="colorpick.xsl"/>
</xsl:stylesheet>
```

That's it—we've implemented a custom control. XSL and XForms are new technologies that require a good deal of learning, but the power in them is well worth it. XForms adds a new, powerful tool to your arsenal of ways to extend ColdFusion.

CHAPTER 50 Internationalization and Localization

IN THIS CHAPTER

Why Go Global? E381
What Is Globalization? E382
Going Global E383
Better G11N Practices E420

In this chapter, we'll look at how ColdFusion can help you build truly global ColdFusion applications. Before we begin, let's consider some fundamental globalization concepts.

Why Go Global?

Taking the high road, I suppose we can say that Web applications need to *go global* in order to deal with humanity's vast diversity—truth be told, it's all about money. If you've been paying attention to Internet statistics for the past few years, you already know that much of the new growth in the Internet is occurring outside the United States (in fact, North America is already at almost 70% internet penetration).

Table 50.1 summarizes some important Internet usage statistics and pretty much speaks for itself. And if these statistics aren't impressive enough, you might consider the "backyard globalization" that's occurring within the U.S. (and elsewhere). According to an article in the *San Antonio Express-News*, Hispanics in the U.S. now outnumber Canadians in Canada. That's 38.8 million people in a *growing* marketplace with an estimated $675 billion in annual purchasing power—certainly something to think about the next time you're designing a ColdFusion application.

Table 50.1 Global Internet Usage

REGION	INTERNET USAGE	GROWTH (2000-2007)	PENETRATION
Africa	33,334,800	638.4 %	3.6%
Asia	398,709,065	248.8%	10.7%
Europe	314,792,225	199.5%	38.9%
Middle East	19,424,700	491.4%	10.0%
North America	233,188,086	115.7%	69.7%
Latin America/Caribbean	96,386,009	433.4%	17.3%

Source: Internet World Stats Web site (`http://www.internetworldstats.com/stats.htm` March 2007)

Undoubtedly you can clearly see the "why" of going global, but what exactly does globalization mean?

What Is Globalization?

That's an easy question to answer—*globalization* is simply deploying an I18N-ready application across several locales, no matter what some harebrained economists think. What's I18N? We're getting ahead of ourselves a bit here, so let's first define some terminology so we can actually understand the globalization concept.

"Secret" Globalization Language Revealed

Close your windows and lock your doors, I'm going to let you in on one of the most guarded "secrets" in the world of software globalization by defining the terms that folks in the industry use to communicate among themselves.

- **Locale** is the most fundamental part of globalization. Locales are languages and other cultural norms (calendars; date, number, and currency formatting; spelling; writing system direction; and so forth) that are specific to a geographic region. The French used in Quebec is not exactly the same as the French used in Paris. Both HTML and XML define locales rather plainly as "language-country"—that is, only as a basic language identifier (though it can encompass such things as "en-scouse," the English Liverpudlian dialect known as "Scouse"). Java, and by extension ColdFusion, include other cultural information that is locale specific.
- **Internationalization or I18N** (I18N is an abbreviation for the 18 letters between the *i* and *n* in *internationalization*) refers to the design and development of an application so that it functions in at least two locales. You can think of I18N as making an application language or locale neutral.
- **Localization or L10N** (L10N is an abbreviation for the 10 letters between the l and n in localization) describes the process after I18N, of adapting an application to a specific locale. L10N might be best thought of as the process of "skinning" a locale-neutral I18N application into a specific locale.
- Finally, **globalization or G11N** (that's right, another abbreviation) is sometimes used as a synonym for I18N. But to me, it's the actual application implementation (L10N) across several locales after I18N—in other words, globalization is both I18N and L10N.

Now that we know what globalization really means, let's look at an overview of how we can accomplish it.

Dancing the Globalization Jig

How you go about globalizing your ColdFusion application depends on whether you have an existing code base. (And let me emphasize here that making an existing application I18N means you have a tough row to hoe.) The process entails these basic steps:

- First you review your existing application components to identify obstacles to the I18N process. Application components include your ColdFusion code, the application's display tier (including Flex, Flash, HTML, JavaScript, and so on), ColdFusion tags, middleware, and your back-end database. An I18N obstacle is an area where the component needs either amending to make it I18N, or replacement by an I18N-capable version.
- The next step is to actually amend or replace non-I18N application components. Wholesale replacement of existing components is by far the easiest thing to do. For instance, if your database doesn't support Unicode (the ins and outs of Unicode are discussed shortly), simply replace it with one that does. Amending existing components is another story. It's a task that's often described as "mind numbing," "brutal," "death by a thousand cuts," and "torturous," as well as a few choice adjectives that the editor wouldn't let me use here. We'll get into these devilish details later on in this chapter.
- Next, you localize the application by providing translated text resources and display tier layouts, graphics, and so on.
- Finally, you document the whole mess in the appropriate locale/language.

The bottom line: Considering the amount of work required to make an existing application I18N, it's a darned good idea to design/code your application from the ground-up for I18N. If you do that, all you need do is localize and document. Much simpler, isn't it?

Now that the background information's out of the way, we can get down to the real nitty-gritty of creating a G11N ColdFusion application.

Going Global

Let's start with the most fundamental part of G11N, locales.

Locales

Among the first things to consider when making a ColdFusion application G11N is what language your application's users want to use and, possibly, where the users are located. Knowing users' locale helps you better tailor your application's language response to them. In globalization, *locales* relate to users' languages and cultural norms, such as sorting conventions; formatting of currency, time and dates, and numbers; and even the spelling of common words (*colour* versus *color*, for instance). Put more simply, a locale is a language as used in a specific country or a region.

Locales are probably the most important piece of G11N—you absolutely need to get them right—and luckily for us, ColdFusion really shines in this area in comparison to previous versions of ColdFusion. Since ColdFusion MX 7 ColdFusion natively supports all the 130-odd locales that core Java does!

As a really masterful "ease-of-use" enhancement, in ColdFusion you can reference these locales using *standard* Java-style locale notation. You can refer to English as used in New Zealand simply as `en_NZ`, rather than `English (New Zealand)`. Besides all the typing and spelling errors this will save

you, it helps streamline and standardize locale usage, not to mention making synchronization with Java I18N objects easier.

Since locales are so important, we're going to take a closer look at them in this section, including the following:

- How can we determine a user's locale?
- Why do we need to maintain a user's locale choice?
- Are there any locale resources beyond what ColdFusion offers?
- What's the best Java library to support G11N in ColdFusion?
- What can we do about locale-based collation (sorting)?

Determining a User's Locale

It's critically important to match a user's locale to the locales that your application supports. Matching what the user wants and what your application can actually deliver is often called *language negotiation*. So how do we do that? The quick-and-dirty answer is to simply ask them to choose from among the supported locales, maybe using a simple `HTML form select` as the very first thing they see when entering the application. The quick-and-dirty way, however, doesn't make for the best user experience; it's intrusive and disruptive, wastes users' time on things outside the real purpose of the application, often makes a bad first impression, and frankly, it's just not considered "cool." In general, it's better to transparently determine a user's locale, initialize the application to use that locale, and then offer the user a way to manually change locales as part of the application's navigation interface.

TIP

Using national flag graphics as navigation aids to allow users to swap locales is generally considered bad form. For starters, it doesn't scale well; what might work with flags for 2 locales probably won't work for 52. This technique also tends to upset some folks when used with languages that cross many locales, such as English (some Brits and Aussies don't appreciate their language being represented by the U.S. flag) and, even more so, Chinese. Resist the urge to get cute.

How do we "transparently determine a user's locale"? It would be ideal if the user's ISP or browser told us precisely where the user was located—from that information we could determine their likely locale. One way to accomplish this involves "Geolocation," where a user's IP address is used to look up (usually via a copy of the WHOIS database) their country. Determining locale is a common enough need that several projects, commercial and open-source, have been developed to solve this problem.

For instance, the cleverly named `geoLocator` CFC does precisely this, using the open-source Java IP (InetAddress) Locator project (`http://javainetlocator.sourceforge.net/`) as its IP/country lookup engine. The CFC's `findLocale` method takes as arguments the user's IP, `CGI` variable `http_accept_language`, and a fallback locale, and returns the most likely locale (or the fallback locale if it can't decide) for that user's combination of IP (country) and `http_accept_language` (which reflects the user's actual locale choices for their browser). The `geoLocator` CFC can be

downloaded free from the Adobe Exchange beta (`http://www.adobe.com/cfusion/exchange//`), where you'll also find details about the CFC.

So why don't we just use the `CGI` variable `http_accept_language` and forget the CFC? Many reasons: Older browsers don't support it, not every user has bothered to set their language/locale preferences, and some user-supplied `http_accept_language` variables are obviously made-up languages/locales (Klingon, for example). Also, since `http_accept_language` can be a list of language/locale preferences, parsing these can become problematic (especially coming from browsers on Apple computers, which produce some of the longest `http_accept_language` lists I've ever seen). The `geoLocation` method is more robust and has some added benefits; it's also useful for other things besides determining a user's locale. You can use it for country-level Web traffic analysis, screening international orders (for instance, "we won't sell betel nut to anybody living in Timbuktu"), helping to determine and price products in local currencies, and so on.

TIP

It's considered good practice to display a user's locale choices in the language of that locale (the choice for French in French, Thai in Thai, and so forth).

Locale Stickiness

Now that we know a user's locale, what do we do with it? Well, the first thing is not to forget it. Say you have a Web application supporting three locales, Thai, Russian, and U.S. English (as the default locale). The `geoLocator` CFC determines that a user in Bangkok has a `th_TH` (Thai language in Thailand) locale. This user gets the home page of the Web site in Thai, with correctly formatted Thai dates and numbers, and so on. The user then navigates to a subsection of the Web site and only sees U.S. English. The application has promptly forgotten their locale and reverted to the default.

This might seem to be a rather trivial issue, but it's an important part of developing a G11N ColdFusion application. There are several approaches to fixing this: the monolingual Web site (more on this in the later section "Better I18N Practices" section), which is more of a high-level design choice than a ColdFusion coding technique; saving the locale to shared scope variables (usually `SESSION` scope); or passing locale as part of the URL string (for example `index.cfm?locale=fr_CA`). Pick one technique; just please don't forget your user's locale.

We'll examine more uses for a user's locale later on, but next let's look at what happens when we need a locale that ColdFusion doesn't support.

CLDR: The Common Locale Data Repository

As stated earlier, ColdFusion derives its locale information from core Java. While this will provide enough locale coverage to satisfy most ColdFusion G11N applications, there will be occasions where it's not sufficient—say, when you need to support Farsi or Vietnamese. For those situations, you'll either need to do your own locale research (and from my own personal experience, I can quite easily say "bah, humbug" to that idea), or you can look elsewhere for some sort of standardized locale resources. These days, "elsewhere" is the *Common Locale Data Repository (CLDR)*.

Originally a project sponsored by the Free Standards Group's OpenI18N team (`http://www.openi18n.org/`), the CLDR project was handed off to the Unicode Consortium (`http://www.unicode.org/cldr/`) in early 2004. CLDR's locale resources, as of version 1.2, cover 232 locales, including 72 languages and 108 territories. There are a further 63 draft locales (covering an additional 27 languages and 28 territories) in the process of being developed. Compare that to the 130 or so locales provided by core Java, and you can understand the real significance of the CLDR. Specifically, the CLDR provides information concerning number/date/time formatting, currency values, as well as support for measurement units and text sorting order (collation). If you find yourself working with a client whose locale or language is not in the CLDR, get in touch with SETI (`http://www.seti.org/`); you might very well be dealing with an alien.

You're probably asking yourself just how to take advantage of the CLDR. The short answer is to find a tool or component that is based on the CLDR. Let's take a quick peek at IBM's ICU4J, which is, as of version 3.2, based on the CLDR.

IBM's ICU4J

One of the truly "big deals" of ColdFusion MX's move to Java was the ease of integrating Java libraries into ColdFusion applications. For G11N applications, the mother of all Java libraries has to be IBM's open-source International Components for Unicode for Java, a.k.a. ICU4J (`http://www.icu-project.org/`). The ICU4J library fills in many of the gaps in core Java's I18N functionality, such as providing non-Gregorian calendars, beefier number formatting including scientific notation and spell-out, speedier locale-based collation, international holidays, and of course all 360 CLDR locales. (We'll discuss a couple of these items in later sections.) Plain and simple, *if you do serious G11N work, you might eventually need to use this library*.

TIP

Much of the ICU4J goodness has already been encapsulated in ColdFusion CFCs. You can find many of these in the `Adobe Exchange beta` (`http://www.adobe.com/cfusion/exchange/`) by searching for ICU4J. They're also available on my shop's Web site (`http://www.sustainableGIS.com/things.cfm`) or on the CFCZone Web site (`http://www.cfcZone.org/`).

Listing 50.1 shows a simple comparison between core Java and ICU4J using Farsi locale (`fa_IR`, the Persian or Farsi language as used in Iran). The first thing to note is that core Java methods were used instead of ColdFusion LS functions. Why? Simply because Farsi is not one of the supported ColdFusion locales. ColdFusion behaves differently than core Java, in that ColdFusion throws an error (`coldfusion.runtime.locale.CFLocaleMgrException`) rather than using a fallback locale as core Java does. Notice that the `getDisplayName` method with a `Locale` or `ULocale` (for ICU4J) as its argument simply displays the localized name for that locale. Another major difference is the use of ICU4J's `ULocale` class rather than core Java's `Locale`.

Listing 50.1 `compareFarsiLocales.cfm`—Comparison of ICU4J/Core Java for Farsi Locale

```
<cfprocessingDirective pageencoding="utf-8">
<!---
```

Listing 50.1 (continued)

```
this example assumes that you have downloaded the current version of ICU4J
from http://icu-project.org/ extracted it from it's archive and copied it to
coldfusion_install_location\wwwroot\WEB-INF\lib. You may need to stop and re-start
your ColdFuson server service in order for it to pick up the new jar.
--->
<cfsilent>
<!---
compares Farsi locale date formatting and name display using core java and icu4j
NOTE: made verbose for clarity
 --->
<cfscript>
// full date format, common to both core java and icu4j
fullFormat=javacast("int",0);
// core java
farsiLocale=createObject("java","java.util.Locale");
farsiLocale.init("fa","IR");
coreJavaDateFormat=createObject("java","java.text.DateFormat");
coreJavaDF=coreJavaDateFormat.getDateInstance(fullFormat,farsiLocale);
//////////////////////////////////////////////////////////////////////
// icu4j magic
farsiUlocale=createObject("java","com.ibm.icu.util.ULocale");
farsiUlocale.init("fa_IR"); // note the nifty init locale syntax
icu4jDateFormat=createObject("java","com.ibm.icu.text.DateFormat");
icu4jDF=icu4jDateFormat.getDateInstance(fullFormat,farsiULocale);
</cfscript>
</cfsilent>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>locale comparison</title>
<meta content="text/html; charset=UTF-8" http-equiv="content-type">
</head>

<body>
<!--- output what we've done --->
<cfoutput>
<b>core Java</b>: #farsiLocale.getDisplayName(farsiLocale)#
#coreJavaDF.format(now())#
<br><br>
<b>ICU4J</b>: #farsiULocale.getDisplayName(farsiULocale)# #icu4jDF.format(now())#
</cfoutput>
</body>
</html>
```

We'll need to see some output from this example (shown in Figure 50.1) in order to understand another important distinction between ColdFusion /core Java and ICU4J. Since it doesn't have any locale resource data for the `fa_IR` locale, core Java falls back on the default locale for the server (in this case, `en_US`) and produces "Persian (Iran)" for the localized name. Although the dates are exactly the same (produced using the default Gregorian calendar), the output formats are quite different. ICU4J formats the date display using the Farsi locale resource data; that is, besides localized Farsi date part names, it also uses Arabic-Indic digits rather than European digits.

Figure 50.1

Comparison of ICU4J/core Java output for Farsi locale

core Java: Persian (Iran) Wednesday, July 25, 2007

ICU4J: فارسی (ایران) چهارشنبه ۲۵ ژوئیهٔ ۲۰۰۷ میلادی

Is there any benefit to using ICU4J with locales that are supported by ColdFusion /core Java? In some cases there is. For example, let's compare ColdFusion to ICU4J for a locale supported by both: `ar_AE` or Arabic (United Arab Emirates). This comparison is also a good example of the benefits of Java-style locale syntax. Listing 50.2 offers this simple example. Things to note are

- The simplicity that ColdFusion brings to G11N
- A single function, `setLocale`, sets ColdFusion 's locale for that page
- The `getLocaleDisplayName` function returns a localized name for this locale similar to ICU4J's `getDisplayName` function
- The `lsDateFormat` returns a formatted date for this locale similar to ICU4J's `format` method—and this is where we find another fly in the locale ointment.

Listing 50.2 `compareCFLocales.cfm`—Comparison of ICU4J/ColdFusion for Arabic Locale

```
<cfprocessingDirective pageencoding="utf-8">
<!---
this example assumes that you have downloaded the current version of ICU4J from
http://icu-project.org/ extracted it from it's archive and copied it to
coldfusion_install_location\wwwroot\WEB-INF\lib. You may need to stop and re-start
your ColdFuson server service in order for it to pick up the new jar.
--->
<cfsilent>
<!---
compares arabic locale date formatting and name display using ColdFusion and icu4j
made verbose for clarity
 --->
<cfscript>
// ColdFusion, yup that's all there is to it
oldLocale=setLocale("ar_AE");
//////////////////////////////////////////////////////////////////////
// full date format
fullFormat=javacast("int",0);
arabicUlocale=createObject("java","com.ibm.icu.util.ULocale");
arabicUlocale.init("ar_AE"); // nifty init syntax
icu4jDateFormat=createObject("java","com.ibm.icu.text.DateFormat");
icu4jDF=icu4jDateFormat.getDateInstance(fullFormat,arabicULocale);
</cfscript>
</cfsilent>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>locale comparison</title>
<meta content="text/html; charset=UTF-8" http-equiv="content-type">
</head>
```

Listing 50.2 (continued)

```
<body>
<!--- output what we've done --->
<cfoutput>
<b>ColdFusion </b>: #getLocaleDisplayName("ar_AE","ar_AE")#
#lsDateFormat(now(),"full")#
<br><br>
<b>ICU4J</b>: #arabicULocale.getDisplayName(arabicULocale)# #icu4jDF.format(now())#
</cfoutput>
</body>
</html>
```

Figure 50.2 shows the output from this example. Although ColdFusion certainly gets the localized date parts (month and day of week) correct, it doesn't fully support Arabic-Indic digits for the numeric parts (year and day of month) of the date format; ICU4J, however, does. In general, for locales supported by ColdFusion /core Java, all Arabic locales in ColdFusion will yield date/time and numeric formatting incorrectly using European instead of Arabic-Indic digits. Note that this is an issue with the underlying core Java, and **not** with ColdFusion per se.

Figure 50.2

Comparison of ICU4J/core Java output for Arabic (UAE) locale

ColdFusion MX 8: العربية (الإمارات) 25 يوليو, 2007

ICU4J: العربية (الامارات العربية المتحدة) الأربعاء, ٢٥ يوليو, ٢٠٠٧

It should be noted that there exists some differences between core Java and ICU4J locale data, some of which are indeed minor. For example, ColdFusion /core Java returns named time zones (ICT for machines using Bangkok, Thailand, time zones), whereas ICU4J returns time zone information as offsets from GMT (GMT+07:00). Also, for the `da_DK`, Danish (Denmark) locale, ColdFusion /core Java returns `30. januar 2005` (no day part name), whereas ICU4J returns `søndag 30 januar 2005` (day part name, but no period after day of month part). In many cases, the meaning of the output display is still relatively clear, but again, the devil is in the details. The locale resource used by your application will depend entirely on the locales you need to support, on whether the Unicode Consortium's CLDR has any official meaning to you or your users, or perhaps on other factors such as non-Gregorian calendars. For a listing of the locales supported by ICU4J as well as differences between it and ColdFusion /core Java, please see Appendix C, "Supported Locales," and Appendix D, "Locale Differences," online.

The final piece of the locale puzzle we'll look at is collation, or sorting.

Collation

Collation is a peculiar thing. It's more or less a universal user requirement, and getting it wrong will certainly make users think less of your application. But getting it right across many locales will also certainly go unnoticed; most users think sorting is quite trivial and do it routinely almost unconsciously. Further complications arise because collation is not consistent for the same characters; for instance, people of German, French, and Swedish nationality sort the same characters differently.

Collation is not even consistent within the same language, as in so-called phone-book collation as opposed to sorting in dictionaries and book indices. And that's just the alphabet-based scripts—Asian ideograph collation can be either phonetic or based on the appearance (strokes) of the characters. Then there are the special cases based on user preferences: ignore/consider punctuation, case (*A* before/after *a*), and so on. You're looking at thousands of years of human collation baggage, so yes, it's going to be complex, even if users do think it's pretty minor. You can read more about the Unicode Consortium's take on collation at `http://www.unicode.org/reports/tr10/`.

As a rule of thumb, your application should first take advantage of your database's collation functionality. Quite a bit of research time and effort was put into this. Most of today's "big iron" databases can handle substantial collation complexity and even "cast" result sets to a collation other than that table/database's default. See Listing 50.3 for an example using Microsoft SQL Server's `COLLATE` clause. The subsequent discussion deals with cases where we have to sort within a ColdFusion page, as in Query-of-Query or when sorting a list or an array.

NOTE

Fine-tuning collation/sorting to a given locale is more important than many developers think. Most users would think an application plain stupid if it couldn't even sort their alphabet correctly.

Listing 50.3 `castCollation.cfm` Casting Collation with Microsoft SQL Server

```
<!---
snippet showing MS SQL Server syntax to cast from default collation,say
SQL_Latin1_General_Cp1250_CS_AS (case & accent sensitive) to
SQL_Latin1_General_Cp1250_CI_AS (case insensitive, accent sensitive)
this should produce a resultset ordering that ignores case
--->
<cfquery name="getTaxRoll" datasource="municipalINFO">
       SELECT title+' '+firstName+' '+Lastname as taxPayer
       FROM taxRoll
       ORDER BY COLLATE SQL_Latin1_General_Cp1250_CI_AS
</cfquery>
```

Suppose we have this scenario:

- Application serving German locale (`de_DE`)
- Requirement to sort an array of names
- Users bitterly complaining that results aren't being sorted correctly

Let's examine what's happening here to see what we can do about shutting up those darned users. The application is quite logically using the `arraySort` function. The problem is that the sorted results aren't at all what the user expects. Names with umlauts (Ä, Ë, Ü) are sorting together as a group *after* the unadorned characters (A, E, U), rather than as most German users would expect, which would be more along the lines of AÄEËUÜ (the commonly used German phone-book or DIN-2 collation).

Why is this happening? Because all of ColdFusion 's collation functionality is based on sorting sequential Unicode codepoints (see Table 50.2 for an example). This will work for users in most

locales; after all, *a* < *b* is true for both lexigraphical (dictionary) and Unicode orders. However, it obviously won't work for languages/locales with collation orders that differ from the Unicode code-point order.

Table 50.2 Some Unicode Codepoint Values

CHARACTER	DECIMAL VALUE
A	41
E	45
U	55
Ä	196
Ë	203

As usual, the solution to this conflict for G11N issues in ColdFusion is to make use of the underlying Java functionality—specifically, core Java's `java.text.Collator` class or ICU4J's `com.ibm.icu.text.Collator` class. Either of these classes allows you to perform locale-sensitive string comparison, although the ICU4J class handles collation considerably better (see `http://oss.software.ibm.com/icu/charts/performance/collation_icu4j_sun.html` for details). Listing 50.5 provides a look at using ICU4J to solve this problem, but before we can make sense of this example, we'll have to examine how core Java and ICU4J actually handle collation.

In Java (both plain Java and ICU4J), collation complexity is handled using three parameters: `locale`, `strength`, and `decomposition`. The `locale` parameter is obvious; a specific locale's collation data is used to order sorts (and searches). The `strength` parameter is used across locales (although exact strength assignments vary from locale to locale) and determines the level of difference considered significant in comparisons. There are four basic strengths:

- `PRIMARY`. Significant for base letter differences; *a* versus *b*.
- `SECONDARY`. Significant for different accented forms of the same base letter (*o* versus *ô*).
- `TERTIARY`. Significant for case differences such as *a* versus *A* (but, again, differs from locale to locale).
- `IDENTICAL`. All differences are considered significant during comparison (control characters, precomposed and combining accents, etc.).

ICU4J adds a fifth strength, `QUATERNARY`, which distinguishes words with/without punctuation.

Let's take an example from the Java docs (`http://java.sun.com/j2se/1.4.2/docs/api/index.html`). In Czech, *e* and *f* are considered primary differences; *e* and *è* are secondary differences; *e* and *E* are tertiary differences; and *e* and *e* are identical. Got that?

The decomposition parameter is just what it sounds like: Characters are decomposed for comparison. There are three basic decompositions (only two for ICU4J):

- NO_DECOMPOSITION. Characters are not decomposed; accented and plain characters are the same. This is the fastest collation but will only work for languages without accented (and so on) characters.
- CANONICAL_DECOMPOSITION. Characters that are canonical variants are decomposed for collation; that is, accents are handled.
- FULL_DECOMPOSITION. Not only accented characters, but also characters that have special formats are decomposed (this decomposition doesn't exist in ICU4J; CANONICAL_DECOMPOSITION is used instead). Basically, un-normalized text is properly handled.

TIP

The i18nSort.cfc wraps up both the core Java and ICU4J versions of locale collation, including functions to sort queries. You can find it in the usual places (mentioned previously).

Now that we understand how collation works in core Java and ICU4J, let's consider the example code in Listing 50.4.

Listing 50.4 icu4jSort.cfm—ICU4J-Based Locale Array Sorting Function

```
<!---
authors:hiroshi okugawa <hokugawa@macromedia.com>
       paul hastings <paul@sustainableGIS.com>
date: 8-feb-2004
notes: this method handles sorting string arrays using locale based collation.
originally part of i18nSort.cfc. note that this code has been made verbose for
clarity.
 --->
<cffunction name="icu4jSort" output="No" returntype="array" hint="returns array
sorted using ICU4J collator">
<cfargument name="toSort" type="array" required="yes">
<cfargument name="sortDir" type="string" required="no" default="Asc">
<cfargument name="thisLocale" type="string" required="no" default="en_US">
<cfargument name="thisStrength" type="string" required="no" default="TERTIARY">
<cfargument name="thisDecomposition" type="string" required="no"
default="FULL_DECOMPOSITION">
<cfscript>
var icu4jCollator=createObject("Java","com.ibm.icu.text.Collator");
var uLocale=createObject("Java","com.ibm.icu.util.ULocale");
var tmp="";
var i=0;
var strength="";
var decomposition="";
var thisCollator="";
var locale=uLocale.init(arguments.thisLocale);
// Arrays object to handle sort
var Arrays = createObject("java", "java.util.Arrays");
//set up the collation options
//strength of comparison
```

Listing 50.4 (continued)

```
switch (arguments.thisStrength){
//handles base letters 'a' vs 'b'
       case "PRIMARY" :
              strength=icu4jCollator.PRIMARY;
              break;
//handles accented chars
       case "SECONDARY" :
strength=icu4jCollator.SECONDARY;
              break;
//handles accented chars, ignores punctuation
       case "QUATERNARY" :
              strength=icu4jCollator.QUATERNARY;
              break;
//all differences, including control chars are considered
       case "IDENTICAL" :
              strength=icu4jCollator.IDENTICAL;
              break;
//includes case differences, 'A' vs 'a'
       default:
strength=icu4jCollator.TERTIARY;
}
//decompositions, only 2 for icu4j
//fastest sort but won't handle accented chars, etc.
if (arguments.thisDecomposition EQ "NO_DECOMPOSITION")
       decomposition=icu4jCollator.NO_DECOMPOSITION;
else //compromise, handles accented chars but not special forms
decomposition=icu4jCollator.CANONICAL_DECOMPOSITION;
//set collator to required locale
thisCollator=icu4jCollator.getInstance(locale);
thisCollator.setStrength(strength);// set strength
thisCollator.setDecomposition(decomposition);//set decomposition
tmp=arguments.toSort.toArray();
//do the array sort based on this collator
Arrays.sort(tmp,thisCollator);
if (arguments.sortDir EQ "Desc") { //need to swap array?
       arguments.toSort=arrayNew(1);
       for (i=arrayLen(tmp);i GTE 1; i=i-1) {
              arrayAppend(arguments.toSort,tmp[i]);
       }
} else arguments.toSort=tmp;
return arguments.toSort;
</cfscript>
</cffunction>
```

The first thing to note (again) is the use of ICU4J's `Ulocale` class rather than core Java's `Locale` class. The next point is the use of core Java's `Arrays` class; we're using it because it can accept a `Collator` object that we begin to build by sorting out (pun intended) what strength and decomposition to use for this `Collator`. We then build the `Collator` for this locale:

```
thisCollator=icu4jCollator.getInstance(locale)
```

We next have to turn the ColdFusion `Array` into a Java `Array` (in order to use the `Arrays` object's nifty sorting methods), using:

```
tmp=arguments.toSort.toArray()
```

Now we're ready to actually do the sort using the `Arrays` object, quite simply:

```
Arrays.sort(tmp,thisCollator)
```

The last thing we have to handle is the direction of the sort (ascending or descending), swapping the array around if the calling page required descending sort direction.

What happens if the locale we're interested in isn't one of the locales for which ICU4J has actual collation data? ICU4J will silently fall back on the Unicode Collation Algorithm (UCA), which should suffice for many of these locales. You can read more about how the UCA works at `http://www.unicode.org/reports/tr10/`. You can also construct your own collation using ICU4J's `com.ibm.icu.text.RuleBasedCollator` class. Besides creating new collations, this class also allows you to combine existing collations or customize individual collations to suit specific needs.

NOTE

By now you might be starting to suspect that G11N ColdFusion code isn't exactly rocket science, and you're right. You can pretty much use any style or framework that you're comfortable with. As long as you follow most of the principles/information laid out in this chapter, you should be good to go.

The preceding discussion has given you a good handle on the ins and outs of locales, so let's examine the next G11N issue, the always-fun task of character encoding.

Character Encoding

In my experience, many (perhaps too many) ColdFusion developers get into some kind of trouble over character encoding. This section is going to provide you with the one single answer to all your character encoding problems; it goes like this: *"Just use Unicode."* Let's review some of the more important aspects of character encoding as they apply to ColdFusion.

Not Unicode? Not So Smart

I suppose it would be useful to see what ColdFusion has to say about character encoding. Quoting from the *Developing ColdFusion Applications* documentation: "*Character encoding* maps each character in a character set to a numeric value that can be represented by a computer. These numbers can be represented by a single byte or multiple bytes." Great, but what that doesn't mention is that it's not unusual for a language to have more than one encoding. For example, English has both 8-bit ISO-8859-1 or Latin-1, and 7-bit ASCII; Japanese has Shift-JIS, EUC-JP, and ISO-2022-JP encodings; and, well, we won't get into the Chinese encodings. Furthermore, not all characters for a given language are represented in every encoding used for that language. For instance, the Euro symbol (?) isn't found within the ISO-8859-1 encoding. (The ISO encoding came before the Euro was established as the default currency in the EU.)

If this weren't enough variety, some character sets appear to be equivalent (at least to some folks) but are in fact not. Many developers think ISO-8859-1 and Windows-1252 are the same character set, when in fact Windows-1252 (also called Windows Western or Windows Latin-1) is more like a superset of ISO-8859-1. The mistake of copying and pasting characters from Word documents into HTML forms using ISO-8859-1 encoding highlights this issue pretty nicely. This is particularly

troublesome if no encoding metadata is available for a chunk of text. G11N projects are prone to this misstep owing to the need for translations, often done by non-IT professionals who quite often wouldn't know a character encoding if it fell on their heads.

Let's summarize some things about character sets:

- Undeniably, there are a lot of character sets floating around (see the IANA's page on character sets, `http://www.iana.org/assignments/character-sets`.) I stopped counting at 75.
- The same character encoding can be used in different languages.
- Many languages are covered by several character sets.

That kind of wild variety is one of the things I loathe as a ColdFusion G11N developer. Matching the correct encoding to a language is quite difficult when there are multiple possible encodings for a language; you're bound to get it wrong once in a while. In fact, getting it wrong happens so often that a Japanese term, *mojibake* (), literally "ghost characters" or "disguised characters," has crept into the G11N vernacular to describe this situation. The term is used to designate the nonsense text that occurs because of the original text's being corrupted by bad or missing character encoding. For instance, becomes this mojibake—`$BJ8;z2=$1(J`—when the character encoding is incorrect (this was taken from some email correspondence). Encoding has to match end-to-end, and getting that 100 percent correct 100 percent of the time isn't trivial.

文字化け

Unicode

A lot of variety means a lot of choices, and that's not always a good thing. So what can we do to simplify things? You already know the answer to that: "*Just use Unicode.*" So what's so hot about Unicode?

- It's a standard (synchronized with the ISO 10646 standard).
- It's Internet ready (XML, Perl, Java, JavaScript, and so on all support Unicode).
- It's multilingual (see `http://www.i18nguy.com/unicode/char-count.html`).
- It travels well (text in any language can be easily exchanged globally).
- It offers monolithic text processing (and that, of course, saves you money in development and support costs, time to market, and so forth).
- It has wide industry support (Macromedia, IBM, Microsoft, HP, Sun, Oracle, and more), making it vendor neutral where pretty much nothing else is.
- It continually evolves (it's now version 5.0.0).
- It's possible to convert from legacy code pages (see `http://www.unicode.org/Public/MAPPINGS/`).

- It's more or less apolitical (see the member list at `http://www.unicode.org/unicode/consortium/memblist.html`).
- The W3C is recommending it for I18N HTML content.

NOTE

For the real skinny on Unicode, visit `www.unicode.org` or `www.macchiato.com`.

Internally, ColdFusion uses Unicode (UCS-2), which is efficient to process because its fixed width (2 bytes per character), but economical bandwidth usage requires single-byte encoding. To me, Unicode smells *in*efficient. However, the twin goals of development simplification and long-term code management are much more important than any superficial bandwidth inefficiency.

Now before you start complaining, "Hey, that smells inefficient to me, too!" stop and consider the nature of UTF-8—a multibyte encoding in which a character can be represented by from *one* to *three* or perhaps *four* bytes. That might sound uneconomical, but bear these facts in mind:

- The vast majority of text transmitted on the Internet can be represented by ASCII, which UTF-8 encodes as 1 byte (7-bit).
- UTF-8 encodes non-ASCII characters such as those used in Western Europe and Arabic countries as 2 bytes.
- Most Asian characters are encoded as 3 bytes.

UTF-8 encoding is therefore as efficient as it needs to be (despite urban myths to the contrary).

So "Just use Unicode", introduce some simplicity to the G11N process, and make UTF-8 your application's sole encoding. Using Unicode simplifies things tremendously. You only have to deal with one encoding on the front end and back end. You will always *know* the data's encoding, no matter what happens to it. And, of course, you'll be on the same page with ColdFusion.

No need to take my word for it—the latest W3C working draft on authoring I18N XHTML and HTML documents actually recommends using UTF-8 or other Unicode encoding: "Choose UTF-8 or another Unicode encoding for all content." (See `http://www.w3.org/TR/i18n-html-tech-char/`.)

Next, let's take a look at putting Unicode to some actual use in *resource bundles*.

Resource Bundles

What's a resource bundle? When Java folks begin making an application I18N, they always talk about "isolating locale-specific data" and for the most part are referring to text data. The accepted technique for this is to create `ResourceBundle` objects backed by `properties` files consisting of key/value pairs (see Listing 50.5 for an example).

The concept is rather straightforward; a "key" (from our example, `go`) has a "value" (`Go`) assigned to it. Dissecting the `properties` filename, `test_en_US.properties` (shown in the example's comments), we can see its locale (`en_US`) as well as the resource bundle name (`test`). Java `properties` files use

escaped ASCII for languages with characters beyond ISO-8859-1 encoding (see the later section "Resource Bundle Tools" for more on this); Listing 50.6 shows an example for Thai (`th_TH`) locale. The value for the key `go` is replaced by escaped ASCII encoding for the Thai word for *Go* (`\u0E44\u0E1B`).

You've probably caught on to the fact that both `properties` files contain the same keys with different values per locale. Instead of hard-coding text in applications, we can now use resource bundle keys that will have their values substituted on a per-locale basis when the page is processed.

Listing 50.5 `test_en_US.properties`—en_US Locale Resource Bundle Example

```
#Resource Bundle: test_en_US.properties - File automatically generated by RBManager
at Mon Dec 08 18:08:52 GMT+07:00 2003
#Mon Dec 08 18:08:52 GMT+07:00 2003
go=Go
cancel=Cancel
```

Listing 50.6 `test_th_TH.properties`—th_TH Locale Resource Bundle Example

```
#Resource Bundle: test_th_TH.properties - File automatically generated by RBManager
at Mon Dec 08 19:06:07 GMT+07:00 2003
#Mon Dec 08 19:06:07 GMT+07:00 2003
go=\u0E44\u0E1B
cancel=\u0E22\u0E01\u0E40\u0E25\u0E34\u0E01
```

NOTE

Java I18N is certainly a good role model for ColdFusion G11N work. I'm not ashamed to admit that many of the ideas in this chapter are derived from Java I18N work–the Java world has been at this G11N game a lot longer than many of us ColdFusion developers.

Now let's go through a simple example converting some ColdFusion code with hard-coded text to make use of resource bundles.

Using a Resource Bundle

Suppose we have a simple login form (Listing 50.7) that we want to use across all the locales supported by our application. For this exercise, the first thing we need to do is to pick through the code and isolate the text that needs replacing with resource bundle keys. So far, so good.

Listing 50.7 `noni18nLogin.cfm`—Non-I18N Login Form

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Please login</title>
  <style type="text/css" media="screen">
    TABLE {
    font-size : 85%;
    font-family : "Arial,Helvetica,sans-serif";
  }
  </style>
</head>
<body text="#330000">
<form action="authenticate.cfm" method="post" name="loginForm" id="loginForm">
```

Listing 50.7 (continued)

```
<table cellpadding="5" cellspacing="5" border="0">
<caption>
<font size="+1" color="#FF0000"><b>Please login</b></font>
</caption>
<tr>
  <td align="right">user name:</td>
  <td><input type="text" name="userName" size="10" maxlength="20"></td>
</tr>
<tr>
  <td align="right">password:</td>
  <td><input type="password" name="password" size="10" maxlength="20"></td>
</tr>
<tr valign="top" bgcolor="Silver">
  <td colspan="2" align="center">
  <input type="submit" value="login">
    <font face=""></font>
  <input type="reset" value="clear">
  </td>
</tr>
</table>
</form>
</body>
</html>
```

Let's also suppose our application design dictates a couple of things: The application's resource bundles will all be initialized at the same time and loaded into a ColdFusion structure in the `APPLICATION` scope. For this example, let's call it `APPLICATION.loginRB`. Also, each user's locale is detected using the `geoLocator` CFC discussed previously and stored in a `SESSION` scope variable, `SESSION.locale`.

TIP

It's a very good idea to logically separate your resource bundles into smaller files based on your application's modules.

Next, Listing 50.7 shows what our original non-I18N login form would look like after we replace its static text with ColdFusion-flavored resource bundle keys (and in light of the application design outlined just above). To illustrate what's happening, let's dissect one key:

```
APPLICATION.loginRB[SESSION.locale].loginFormTitle
```

The `APPLICATION.loginRB` indicates which resource bundle we want to use. `SESSION.locale` indicates which locale this user is in and acts as a key into the `APPLICATION.loginRB` structure. And `loginFormTitle` is the exact resource bundle key for which we want to substitute localized text.

Listing 50.8 `i18nlogin.cfm`—I18N Login Form

```
<!---
NOTE: these bits WOULD NOT normally be used in this page but rather
in an initialization routine. The example assumes you have downloaded & installed
the rbJava.cfc.
 --->
<cfscript>
  rB=createObject("component","rbJava");
  geoL=createObject("component","geoLocator");
  i18nUtil=createObject("component","i18nUtil");
```

Listing 50.8 (continued)

```
    loginRB=structNew();
    loginRB["en_US"]=rB.getResourceBundle("loginRB","en_US");
    loginRB["th_TH"]=rB.getResourceBundle("loginRB","th_TH");

  // figure out the user's locale
  session.locale=geoL.findLocale(CGI.remote_addr,CGI.http_accept_langauge,"en_US");
  // is this a BIDI locale?
  if (i18nUtil.isBIDI(session.locale)
    SESSION.writingDir="rtl";
  Else
    SESSION.writingDir="ltr";
  </cfscript>
  <cfprocessingdirective pageencoding="utf-8">
  <cfcontent type="text/html; charset=utf-8">

  <cfoutput>
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
  <html dir="#SESSION.writingDir#" lang="#session.language#">
  <head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <meta http-equiv="Content-Language" content="#session.language#">
  <title>#APPLICATION.loginRB[SESSION.locale].loginFormTitle#</title>
  <style type="text/css">
    TABLE {
    font-size : 85%;
    font-family : "Arial Unicode MS,Arial,Helvetica,sans-serif";
    }
  </style>
  </head>
  <body text="##330000">
  <form action="authenticate.cfm" method="post" name="loginForm" id="loginForm">
  <table cellpadding="5" cellspacing="5" border="0">
  <caption>
    <font size="+1" color="##FF0000">
    <b>#APPLICATION.loginRB[SESSION.locale].loginFormTitle#:</b>
    </font>
  </caption>
  <tr>
    <td align="right">#APPLICATION.loginRB[SESSION.locale].userNameLabel#:</td>
    <td><input type="text" name="userName" size="10" maxlength="20"></td>
  </tr>
  <tr>
    <td align="right">#APPLICATION.loginRB[SESSION.locale].passwordLabel#:</td>
    <td><input type="password" name="password" size="10" maxlength="20"></td>
  </tr>
  <tr valign="top" bgcolor="Silver">
    <td colspan="2" align="center">
    <input type="submit" value="#APPLICATION.loginRB[SESSION.locale].loginButton#">
      
    <input type="reset" value="#APPLICATION.loginRB[SESSION.locale].clearButton#">
    </td>
  </tr>
  </table>
  </form>
  </body>
  </html>
  </cfoutput>
```

The original static English text became part of the `en_US` locale resource bundle, which you can see in Figure 50.3. The same login form, using the same code from in Listing 50.8 for users in the Thai locale (`th_TH`), is shown in Figure 50.4. This is quite a bit easier than trying to maintain separate forms files, one per locale.

Figure 50.3
The en_US locale login form

Now that we know what a resource bundle is, let's look at what it's not.

Figure 50.4
The th_TH locale login form

What *Isn't* a Resource Bundle?

Listing 50.9 is an example of what a resource bundle is not. Let me put to rest the notion of using ColdFusion code in lieu of "proper" resource bundles. There are several reasons not to do this; chief among these are:

- It mixes code and text like the bad old spaghetti code days.
- It requires some knowledge of ColdFusion to manage these files—and you do not want ColdFusion developers handling the translation of, say, information about brain surgery.
- It doesn't lend itself to using any of the nifty resource bundle–management tools (see "Resource Bundle Tools" coming up) that are commonplace in the G11N world.

So using ColdFusion code instead of resource bundles is a bad habit—it might work with small files for a few languages but will eventually break down as your G11N applications become more complex and cover more locales. If you're just beginning G11N work, don't start out with this method no matter how tempting it looks. And if you're already using this approach, think about quitting while you're ahead. Mingling code and text in this way is *not* a good idea.

Listing 50.9 `notRB.cfm`—Not a Resource Bundle

```
<cfset loginRB=structNew()>
<cfset loginRB.en_US.loginFormTitle="Please login">
<cfset loginRB.en_US.userNameLabel="user name">
<cfset loginRB.en_US.passwordLabel="password">
<cfset loginRB.en_US.loginButton="login">
<cfset loginRB.en_US.clearButton="clear">
```

Resource Bundle Flavors

There are actually a two kinds of resource bundles that can be used with ColdFusion. The first is what might be termed "CFMX UTF-8," where the resource bundle is constructed similar to a traditional INI file. A variable's text value is written out using UTF-8–encoded human-readable text. It's simple to implement, relying solely on ColdFusion code to parse the files. Reading it requires nothing more complex than Notepad (which to my mind makes it unsuitable for larger, more complex applications).

TIP

There are ready-made CFCs for handling resource bundles available in the previously mentioned places.

The second flavor of resource bundle is the "proper" Java-style resource bundle as outlined earlier. These resource bundles require the use of core Java classes, which entail some overhead but have the benefits of being "standard" and having a wealth of ready-made (mostly open-source) tools to manage them. You can further subdivide this resource bundle flavor into two subflavors, depending on how you're able (or want) to access these files. "Pure" resource bundles are accessed using the Java `ResourceBundle` class. This class provides automatic determination of resource bundle from locale, and automatic fallback locales (if it can't find a resource bundle for a given locale, it truncates that locale back to the language identifier and searches again; if it can't find that resource bundle, it falls back to the base one, usually `en_US`). The class does, however, require that all resource bundles be located somewhere on a Java `classpath`, which makes for some complexity in shared-hosts environments. The other subflavor uses the Java `PropertyResourceBundle` class to access resource bundles. It provides none of the automatic features of the `ResourceBundle` class but does have the advantage of locating your resource bundles anywhere, although you must explicitly load each resource bundle. Table 50.3 summarizes the pros and cons of the resource bundle types.

Now let's have a look at some tools to manage resource bundles.

Table 50.3 Resource Bundle Flavor Comparison

RESOURCE BUNDLE FLAVORS	PRO	CON
ColdFusion UTF-8	Human readable Easy to manage (Notepad, etc.) Simple to implement in ColdFusion Quite fast	Complex resource bundles quickly become hard to manage Can't easily use standard resource bundle tools
Java `ResourceBundle` class	Pure standard Java resource bundle solution Handles resource bundle from standard tools Self-determines resource bundle for locale Handles complex resource bundle quite easily	Not human readable Requires that resource bundle be somewhere in `classpath` Requires `createObject` permission Some overhead in using Java object

Table 50.3 (continued)

RESOURCE BUNDLE FLAVORS	PRO	CON
Java `PropertyResourceBundle` class **(a)**	Resource bundle can be anywhere Pure standard Java resource bundle solution Handles resource bundle from standard tools Handles complex resource bundle quite easily	Not human readable Requires caller to determine resource bundle from locale Requires `createObject` permission Some overhead in using Java object

(a) See `http://www.sustainablegis.com/unicode/resourceBundle/javaRB.cfm` for an example.

Resource Bundle Tools

It's a fact of life that large, complex G11N applications usually generate large, complex resource bundles. Trying to manage these with Notepad and Post-its isn't very realistic. You have to manage the creation/editing of the resource bundle keys, manage the creation/editing of resource bundles per locale, manage keys that have been translated into certain locales, and so on. Luckily, the Java I18N world has developed several resource bundle management tools that we can use for this task. Foremost among these (and also my favorite) is ICU4J's pure-Java Resource Bundle Manager (`RB Manager`). Among the things `RB Manager` can do to help solve day-to-day L10N problems include the following:

- Handles editing multiple language files
- Provides sophisticated resource bundle search functionality
- Checks resource bundle keys for duplicates and for proper format
- Provides a grouping of resources; individual translations are easier to find
- Provides that each language file will only display a list of resources that are untranslated (wonderful for tracking what still needs to be translated)
- Keeps track of statistics such as number of resources, untranslated items, and so on
- Handles importing and exporting of translation data into multiple formats such as XLIFF, TMX, ICU, and more
- Use of the `RB Manager` application cuts down on development, translation, and debugging time in any internationalized setting

You can find a complete tutorial for `RB Manager` in the download file.

Figure 50.5 shows a typical resource bundle for English and Arabic languages being managed using `RB Manager`. In this example, the view provides a list of all resource bundle keys and their English and Arabic translations. You can download a free copy of `RB Manager` from ICU4J's site, `http://www.icu-project.org/download/rbmanager.html`.

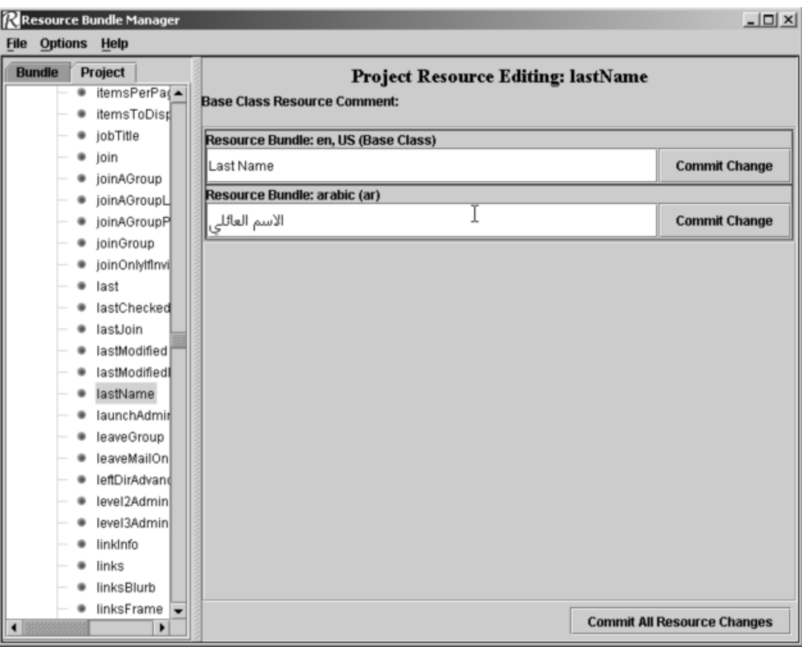


Figure 50.5

Figure 50.5 ICU4J's Resource Bundle Manager

In addition to `RB Manager`, there are other resource bundle management tools available that are more-or-less free (please review each application's licensing):

- Attesoro (`http://ostermiller.org/attesoro/`) is another pure-Java solution that can produce proper Java resource bundles.
- BabelFish (`http://www.solyp.com/2975.html`), also a Java program, has an interesting feature: it has links to machine translation sites (I should state for the record that I'm entirely against everyday machine translation for any content that even resembles anything important).
- Zaval Java Resource Editor (`http://www.zaval.org/products/jrc-editor/`). Yes, it's another Java program.
- I18nEdit (`http://www.cantamen.de/i18nedit.php?lang=en`) is another Java-based resource editor; most noteworthy is the nifty built-in Unicode character picker for those days when you're too lazy to load another locale.
- `native2ascii` is a command-line tool that will convert a file with native-encoded characters (the caveat being that the "native" encoding must be one of the Java-supported ones) to one with Unicode-encoded characters. It's found in the `bin` directory of your Java JRE/JDK installation.

Our next stop on the ColdFusion G11N tour deals with mailing addresses.

Addresses

Living outside the United States, one of my pet peeves is the assumption by many sites that users' addressing schemes are similar to their own. A prime example of this is the State field. Most countries do not have State as part of their addressing scheme, and ColdFusion developers' adding it to their applications or, even worse, requiring it, will only confuse and possibly annoy these users. Developers need either to intimately understand a locale's addressing scheme (very possible through localization research) or to build flexibility into their address-capture routines and storage.

Developers should also not assume that postal codes (ZIP codes) confine themselves to a particular format or length. For example, Japanese postal codes can have a format such as 460-0002 (Aichi), whereas Canadian ones come in the form V2B 5S8 (Kamloops, British Columbia). Even the placement of the postal code in a mailing address can vary widely. In Laos, the postal code is to the left of the locality (01160 XAYSETHA), and in Japan it's to the left of the country (460-0002 JAPAN).

Let's look at a brief example of these ideas. Listing 50.10 shows a table design (Microsoft SQL Server data types) to hold worldwide customer information for a spatial data set product. This simple table design comes from my years of dealing with a global customer base. Its flexibility is its most important point.

Listing 50.10 `galacticCustomer.txt`—Galactic Customer Table Design

```
[CustomerID] [int] IDENTITY (1, 1) NOT NULL
[Salutation] [nvarchar] (100) NULL --- not fixed, as customer prefers
[FirstName] [nvarchar] (100) NOT NULL
[LastName] [nvarchar] (200) NOT NULL
[eMail] [varchar] (50) NULL --- may not have email
[PurchaseDate] [datetime] NOT NULL
[Organization] [nvarchar] (200) NULL --- company, government office, etc.
[Address] [ntext] NULL --- nTEXT will hold anything customer provides
[City] [nvarchar] (150) NULL --- may not have a city
[Locality] [nvarchar] (200) NULL --- state/province/etc. may or may not have
[Country] [varchar] (35) NOT NULL --- minimally have this, pulled from our SELECT
[PostalCode] [varchar] (40) NULL --- may or may not have
[Phone] [varchar] (50) NULL -- plenty of room
[Fax] [varchar] (50) NULL -- plenty of room
[FreeCustomer] [bit] NOT NULL --- local schools, etc. on charity list
[timestamp] [timestamp] NULL --- edit/full text indexing flag
```

NOTE

In Microsoft SQL Server's T-SQL DDL, `NOT NULL` means required data, whereas `NULL` means not required.

At first glance, there's nothing particularly remarkable about this design; however, take note of a few items. Many columns that you might normally compel a user to supply are not required, and many columns might seem overly large to someone dealing with just one locale. For example, `City` isn't required because in some cases there isn't an identifiable city in an address. `Address`, on the other hand, is an `NTEXT` data type capable of holding a huge amount of freeform text that might include streets, lanes, subdistricts, districts, and even directions. Notice also that SQL Server's Unicode data types (`NVARCHAR` and `NTEXT`) are used to allow the customer to supply their own

language version of name, address, and so on. For more information on address formats, see `http://www.upu.int/post_code/en/addressing.shtml`.

Date/Time

Addressing nuances might frustrate users, but date formatting certainly frustrates ColdFusion developers. If the ColdFusion Support Forums are any indication, even within one single locale, dates often make developers punch drunk. Even though dates are basically a simple combination of day, month, and year, there's an extensive and often confusing variety of date formats across locales. For example, 12/10/56 could be interpreted in a number of ways. In Thailand (which has a short date format of day/month/year), 12/10/56 would be taken to mean October 12, 1956. In the United States (which has a short date format of month/day/year) that date would be December 10, 1956. A similar date in Japan (where the short date format is year/month/day) would be hopelessly broken: October 56, 1912. Keeping date formats straight among locales is critical to developing G11N applications.

Our next date/time formatting issue is all the various calendars in use throughout the world.

Calendars

Besides date formatting, developers should not forget the types of calendars in use within a given locale. This can be critical; a month in one calendar might not cover the exact same time span in another. Weeks and weekends don't always start on the same day across locales using different calendars, or even within the same calendar as in the case of Europe versus the U.S.

Out of more than 40 calendars in use around the world today, we'll examine the six most common (the "big six"), and throw in one rare calendar just for added flavoring. The "big six" discussed here are, of course, supported by the ICU4J library. The reason we're discussing these at all is to give ColdFusion developers some background information so that you're not operating in a vacuum with these calendars behaving like some sort of mysterious "black box."

NOTE

Examples of the following calendars can be found at http://www.sustainablegis.com/projects/icu4j/calendarsTB.cfm

Gregorian Calendar

Pope Gregory XII introduced the Gregorian calendar in 1582 as an adaptation of the Julian calendar (named after Julius Caesar), when the 10-day difference between the actual time of year and traditional time of year on which calendar events occurred became intolerable. This calendar was constructed to give a closer approximation to the tropical year, which is the actual length of time it takes for the Earth to complete one orbit around the Sun.

The actual changeover from Julian to Gregorian calendar resulted in quite an interesting "month." When England and her colonies made the change to the Gregorian in 1752 (not all countries adopted this calendar at the same time), it created a month of September something like what is

shown in Table 50.4. This move provoked widespread riots—yes, you do indeed need to pay attention to calendars.

The Gregorian calendar is in common use in Christian countries (even though some of them hadn't adopted this calendar until the early part of the twentieth century). This is the calendar most ColdFusion developers are familiar with, so I won't go into any more detail (but you can read more about this calendar here: `http://scienceworld.wolfram.com/astronomy/GregorianCalendar.html`).

Table 50.4 Result of Julian to Gregorian Calendar Changeover

SEPTEMBER 1752

SUN	MON	TUE	WED	THU	FRI	SAT
		1	**2**	**14**	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Buddhist Calendar

Behaving similarly to the Gregorian, the Buddhist calendar is identical to the Gregorian in all respects except for the year and era (B.C., A.D., etc.). Years are numbered since the birth of the Buddha in 543 B.C. (Gregorian), so that 1 A.D. (Gregorian) is equivalent to 544 B.E. (Buddhist Era) and 2005 A.D. is 2548 B.E. Quick and dirty is to simply add 543 years to the Gregorian year to arrive at the Buddhist year, and subtract 543 years to go the other way. In predominantly Buddhist countries such as Thailand (where I live these days) the Buddhist calendar is the civil calendar (the official one in general use by most folks and, of course, the government). This calendar is often used elsewhere for religious purposes.

Chinese Calendar

The traditional Chinese calendar is a *lunisolar* calendar (interestingly the same type as the Hebrew calendar). Months start with a new moon, with each month numbered according to solar events. Why? It guarantees that month 11 will always contains the winter solstice. How? Leap months are inserted in certain years. These leap months are numbered the same as the month they follow (how's that for complication?). Which month is a leap month? It depends entirely on the movements of the sun and moon.

Distinct from the Gregorian calendar, the normal Era field differs from other calendars in that it holds a 60-year cycle number rather than the usual B.C./A.D. Right now, in 2005, we're in the 78th cycle, which began in 1983 A.D. Years are counted sequentially, numbering from the 61st year of the reign of Huang Di (more or less 2637 B.C.), which is designated year 1 on the Chinese calendar—yes, that's right, this calendar system is over 4,000 years old. Let's look at an example:

星期三 20x78-9-13

where 20 is the year in the current cycle, 78 is the cycle for this calendar (Era in other calendars), 9 is the month, and 13 is the day.

TIP

CFCs for handling these ICU4J-based calendars are available in the usual places (mentioned previously).

Hebrew Calendar

The Hebrew calendar is also lunisolar, which gives it what some folks would call "a number of interesting properties." Distinct from the Gregorian calendar, months start on the day of each new moon (the ICU4J library actually makes an approximation of this). The solar year (which, as everyone knows, is 365.24 days) is not an even multiple of the lunar month (approximately 29.53 days), so an extra leap month is inserted in 7 out of every 19 years (this is beginning to sound interesting). And just to make sure everybody's paying attention, the start of a year can be delayed by up to 3 days in order to prevent certain holidays from falling on the Sabbath (as well as to prevent illegal year lengths). As the cherry on the ice cream, the lengths of certain months can vary depending on the number of days in the year. And finally, years are counted since the creation of the world (A.M. or *anno Mundi*), believed to have taken place in 3761 B.C. Hurts my head, too—and is a compelling reason to make use of the ICU4J library and let the smart guys at IBM worry about this sort of thing.

Islamic Calendar

The Islamic calendar is also known as Hijri because it starts at the time of Mohammed's journey or *hijra* to Medinah on Thursday, July 15, 622 A.D. It is the civil calendar used by most of the Arab world and is the religious calendar of the Islamic faith. This calendar is a strict lunar calendar; an Islamic year of 12 lunar months therefore does not exactly correspond to the solar year used by the Gregorian calendar system. An Islamic year averages about 354 days, so viewed from the Gregorian, each subsequent Islamic year starts about 11 days earlier.

The *civil* Islamic calendar uses a *fixed* cycle of alternating 29- and 30-day months, with a leap day added to the last month of 11 out of every 30 years. That makes the calendar predictable, so it is used as the civil calendar in a number of Arab countries.

The Islamic *religious* calendar, however, is based on the actual observation of the crescent moon. This sounds predictable and simple enough, but that observation varies based on where you are when you look (your geography), when you look (sunset varies by season), moon orbit "eccentricities," and even the weather (too cloudy and you obviously can't see the moon). All this makes it impossible to calculate in advance, so the start of a month in the religious calendar might differ from the civil calendar by up to 3 days.

Japanese Calendar

The Japanese calendar, sometimes called the Japanese Emperor Era calendar, is identical to the Gregorian calendar except for the year and era. Each Emperor's ascension to the throne begins a

new era. Each new era's years are numbered starting with 1 (the year of ascension). What could be simpler? The "modern" eras began as follows:

- *Meiji*. January 8, 1868 A.D.
- *Taisho*. July 30, 1912 A.D.
- *Showa*. December 25, 1926 A.D.
- *Heisei*. January 7, 1989 A.D. (current era)

Persian Calendar

A Persian (or perhaps Iranian) calendar is the formal calendar in general use in Iran. It's also known as the solar Hijri calendar and sometimes as the Jalali calendar. I've also seen it described as the Shamsi calendar; quite frankly, I have no idea which is correct, so I'll stick with Persian.

The Persian calendar has a starting point that matches the Islamic calendar but is otherwise unrelated. The origin of this calendar can be traced back to the eleventh century when a group of astronomers (including the famous poet Omar Khayyam) created what was then called the Jalali calendar, with the "modern" version being adopted in 1925 A.D. Since it's one the few calendars designed in the era of accurate positional astronomy, it's probably the most accurate solar calendar around today (we'll see why in a bit).

Like the Gregorian calendar, this calendar consists of 12 months; the first 6 are 31 days in length, the next 5 are 30 days, and the final month is 29 days in a normal year and 30 days in a leap year. To put it mildly, the Persian calendar uses a very complex leap-year structure; years are grouped into cycles that begin with 4 normal years, after which every 4th subsequent year in the cycle is a leap year. These cycles are in turn grouped into "grand" cycles of either 128 years (composed of cycles of 29, 33, 33, and 33 years) or 132 years (containing cycles of 29, 33, 33, and 37 years). A "great grand" cycle is composed of 21 consecutive 128-year grand cycles and a final 132 grand cycle, for a total of 2,820 years. The pattern of normal and leap years, which began in 1925, will not repeat until the year 4745.

Each 2,820-year great grand cycle contains 2,137 normal years of 365 days, and 683 leap years of 366 days. The average year length over the great grand cycle is 365.24219852 days, which is so close to the actual solar tropical year of 365.24219878 days that the Persian calendar accumulates an error of only 1 day in every 3.8 million years.

If this isn't enough information for you, you might have a look at this site: `http://www.tondering.dk/claus/cal/node6.html`.

Calendar CFC Usage

Space doesn't permit me to post any of the code for the preceding calendar CFCs (each runs to over 1100 lines of code). What I will do instead is introduce some of the functions from these CFCs in order to help you to start thinking about using calendars in your G11N applications. (Note that many of these functions have had `i18n` added to their function name in order not to conflict with existing ColdFusion functions.)

The following are functions related to calendar math:

- `i18nDateAdd` returns a `datetime` object with units of time added. This should be used instead of ColdFusion's `dateAdd` function. Why? If you examine the output from the various calendars shown above, you will see that the same unit of time isn't equivalent across calendars. Adding 2 years to a date of 3-Feb-2005 for an Islamic calendar results in a date 709 days in the future; for the Hebrew calendar, it results in a date 739 days in the future; and for the Buddhist calendar it's 730 days.
- `18nDateDiff` returns the difference in date parts between two dates. For the same reasons outlined for `i18nDateAdd`, this method should be used instead of ColdFusion's `dateDiff` function.
- `i18nDateParse` parses a date string formatted as `FULL`, `LONG`, `MEDIUM`, `SHORT` style into a valid date object.
- `i18nIsWeekend` returns a `boolean` indicating whether input date falls on a weekend according to a given calendar. Weekends do not begin on the same day of the week across all calendars.
- `weekStarts` returns the first day of week for a given calendar. Weeks do not start on the same day across calendars, or even across locales within the same calendar.
- `18nDaysInMonth` returns the number of days in given month.
- `i18nDayOfWeek` returns the day of week for a given date.
- `is24HourFormat` returns `0` if not 24-hour time format, `1` if 24-hour time format in 0-23 style, or `2` if 24-hour time format in 0-24 style.
- `i18nIsLeapYear` returns true or false if a given year is a leap year.
- `getEras` returns a locale-based era (A.H., A.D., B.C., etc.).

TIP

It's not usually a good idea to use your own custom date/time formats in G11N applications. You're better off leaving that up to the standard locale-formatting functions.

These functions were designed mainly for use in page layout logic:

- `isDayFirstFormat` determines whether a given locale uses day-month or month-day format; mainly used in page layouts.
- `getDateTimePattern` returns locale `datetime` pattern string (for example, `mm-dd-yy`) for a given locale.
- `getDatePartOrder` is metadata method; returns date part order (day-month-year, month-day-year, etc.) for a given calendar/locale combination.
- `getTimeDelimiter` returns time delimiter (`:/.`) for a given calendar/locale combination.

The following functions are specific to individual calendars:

- `getCycle` returns the cycle for a passed date (Chinese calendar).
- `getCycleYear` returns the year in a given cycle for a passed date (Chinese calendar).
- `getExtendedYear` returns the extended year for this calendar; that is, years since start of the Chinese calendar.
- `getCycleMonth` returns the month in a cycle year for a passed date (Chinese calendar).
- `getCycleDay` returns the day in a cycle month for a passed date (Chinese calendar).
- `isLeapMonth` returns true/false if a given month is a leap month (`ADAR 1`) in the Hebrew calendar.
- `getEmperorEra` returns a string indicating the Japanese emperor era in which a given date falls (Japanese calendar).

Hopefully, the preceding sections have given you a firm grounding in the role of calendars in G11N. Now, let's look at one final time-related G11N issue: time zones.

Time Zones

If your application involves a global base of users, you're likely to run into issues concerning time zones. It's often the case that the application server is in one time zone while the users are in others (even non-G11N applications are affected by this). Toss daylight savings time (DST) into the mix, and things can become complicated rather quickly. Why are time zones so complicated? In theory, a time zone is an area on the Earth's surface between two meridians spaced by 15 degrees of longitude (the x-axis, if you will) where the same time is adopted. Realistically, for administrative and sometimes political reasons, state or country borders often define the time zone instead of exact geographic position. For example, Table 50.5 shows the various time zone equivalents for the Asia/Bangkok (GMT+0700). These are all the same physical time zone, but simply named differently.

TIP

A CFC encapsulating the core Java time-zone functionality (timezoneCFC) is available in the usual places (mentioned previously).

Table 50.5 Asia/Bangkok (GMT+0700) Time Zone Equivalents

Antarctica/Davis
Asia/Bangkok
Asia/Hovd
Asia/Jakarta
Asia/Krasnoyarsk
Asia/Phnom_Penh
Asia/Pontianak
Asia/Saigon

Table 50.5 (continued)

Asia/Vientiane
Etc/GMT-7
Indian/Christmas VST

For our G11N applications, the ideal would be to allow users their own time zones and simply cast our application `datetimes` to/from the server's time zone. To further simplify things, we might also always store our application's `datetime` values in the GMT time zone. You could conceivably do this in pure ColdFusion code, but it is simpler to use either core Java's `java.util.TimeZone` class or ICU4J's `com.ibm.icu.util.TimeZone` class (handling DST changes alone in CFMX code would be quite messy especially as there is no built-in mechanism to determine when a given time zone's DST begins/ends).

NOTE

You need to understand that for a ColdFusion server, **all** datetimes are considered to be in *that server's time zone*. Consider a server in a time zone that has Daylight Savings Time (DST). A GMT datetime of 2006-04-02 02:01:00.0 on those servers would *never* exist, it would automagically get flopped over to 2006-04-02 03:01:00.0 (because at one time that was when DST kicked in) because ColdFusion doesn't see your GMT time zone, *only* the server's DST time zone. So for an hour or so twice a year, your datetimes would be wrong by an hour.

An example of this method can be found at
`http://www.sustainablegis.com/projects/tz/testTZCFC.cfm`,

TIP

If your application must support multiple time zones, it's probably a good idea to maintain your `datetime` data in GMT time zone rather than the server's or client's time zone.

Our next stop on the ColdFusion G11N tour is the topic of databases.

Databases

As far as G11N applications go, the most important factor is whether or not the database is Unicode capable. In this day and age it is rather difficult to find many popular or "big iron" databases that do *not* support Unicode. The last holdout among these was MySQL, which finally supported Unicode with the release of version 4.1. The following is a brief review of Unicode-capable databases that you can use with ColdFusion. Consult the database's documentation for details.

Microsoft Access

Microsoft Access, within its limitations, is a suitable database for G11N applications; it supports Unicode, provided you use the Access for Unicode driver supplied with ColdFusion. You need to be aware of some *quirks* associated with that driver, for instance it uses 1/0 instead of true/false for Boolean values and it follows JET 4.0's reserved words (see `http://support.microsoft.com/?kbid=248738`).

Microsoft SQL Server

Microsoft SQL Server has been Unicode capable since version 7. It provides three data types to handle Unicode text: `NVARCHAR`, `NCHAR`, and `NTEXT`. (The `N` comes from the SQL-92 specification and stands for "national" data types). Be aware that the limits for the `VARCHAR` and `CHAR` data types (8000 bytes) apply to both the standard and the Unicode variants, which effectively halves the Unicode size limits (4000 Unicode characters). If you use Unicode data (which, of course, you should be doing at all times), also be mindful that Microsoft SQL Server requires that all Unicode text passed to it be assigned an `N` prefix (see `http://support.microsoft.com/kb/239530/EN-US/` for more information):

```
SELECT someColumn
FROM someTable
WHERE Greeting = N'Hello!'
```

If you use the `<cfqueryparam>` tag (which is a very good idea) you will need to turn on Unicode support via ColdFusion Administrator's Advanced option for that DSN, as shown in Figure 50.6. As noted earlier in Listing 50.3, SQL Server can "cast" collations using the `COLLATE` clause, which should be your first line of attack when it comes to sorting data.

Figure 50.6
DSN Unicode support option in ColdFusion Administrator

TIP

Always use your database's JDBC driver if available.

MySQL

The release of MySQL version 4.1 brings Unicode support as UTF-8 or UCS-2. You can assign a character set and/or collation to the server, database, table, and column. For example:

```
CREATE DATABASE dayLateDollarShort DEFAULT CHARACTER SET utf8
```

would assign the UTF-8 encoding to all `CHAR` and `VARCHAR` columns in that database. Similar to Microsoft SQL Server, you can "cast" collations using the `COLLATE` clause.

In terms of database connections, you can set the client connection character set (where ColdFusion is MySQL's "client") either within MySQL itself or via the MySQL DSN's connection string option (in the Advanced option section of that DSN in the ColdFusion Administrator) using:

```
useUnicode=true&characterEncoding=utf8
```

PostgreSQL

PostgreSQL has had full Unicode support since version 7.1. Its current version is 8.0, which is also its first native Windows version. Unlike MySQL, you can only set character encoding at the database level:

```
CREATE DATABASE postGISUnicode WITH ENCODING 'UNICODE'
```

Collation is also fixed at the database level—or actually at the "cluster level"; one instance of PostgreSQL can only have one locale.

Oracle

Oracle has supported Unicode since version 7. Oracle handles I18N issues via National Language Support (NLS), which provides database utilities, error messages, sort orders, date/time and numeric/currency formatting, and so on, adapted to relevant native languages. Oracle covers about 67 territories (locales) with 46 languages.

Oracle provides Unicode support through UTF-8 (AL31UTF8 in Oracle-talk), although the character sets differ from version 7 (AL24UTFFSS) to version 8 (AL31UTF8). AL31UTF8 handles ASCII as single-byte encoding. Similar to Microsoft SQL Server, Oracle's Unicode data types are `nchar`, `nvarchar2`, and `nclob`. Provided that its NLS parameters (`NLS_Language, NLS_Territory`) are initialized properly (server-side initialization parameters, client-side environment variables, or through the `ALTER SESSION` parameter), there are no serious I18N issues involving Oracle.

Display

Most ColdFusion developers tend to turn up their noses at so-called "design" issues like page display and layout. Display is, however, an important G11N topic, especially in locales with right-to-left (RTL) writing systems such as Arabic or Hebrew—what some folks refer to as the BIDI (bi-directional) locales. You need to understand that not just the text is RTL; the whole concept of a "page" in these locales is RTL. Let's look at an example.

NOTE

In case you're wondering why these languages' writing systems are considered BIDI, it's because things like numbers are written left-to-right. That is, the most significant digit is leftmost, so the number 100 (one hundred) is written in Arabic or Hebrew as 100 rather than 001. Also, note that "languages" do not have a direction; their writing systems do.

Figure 50.7 is the desktop for a fully internationalized virtual office application (HyperOffice, `http://www.hyperoffice.com/`) for a user in the `en_US`, English (United States) locale. This page is laid out left-to-right (LTR), with the most important objects (menu, user name, and so on) on the left side of the page. If you look closely at the arrow icons, even these graphics are LTR (they point from the left to the right)—the devil is indeed in the details.

If we log in to this application as a user in the `ar_AE`, Arabic (United Arab Emirates) locale—one of the BIDI locales—you will see something like Figure 50.8. The most important objects are now on the right side of the page; the arrow icons and other graphical details are RTL as well.

As you can see, it's simply not enough to consider text handling alone; you must be concerned about every aspect of the page in locales with an RTL writing system. For more information on RTL page layout, you can visit the World Wide Web Consortium or W3C Web site (`http://www.w3.org/International/questions/qa-scripts.html`) or Tex Texin's Web site (`http://www.i18nguy.com/markup/right-to-left.html`).

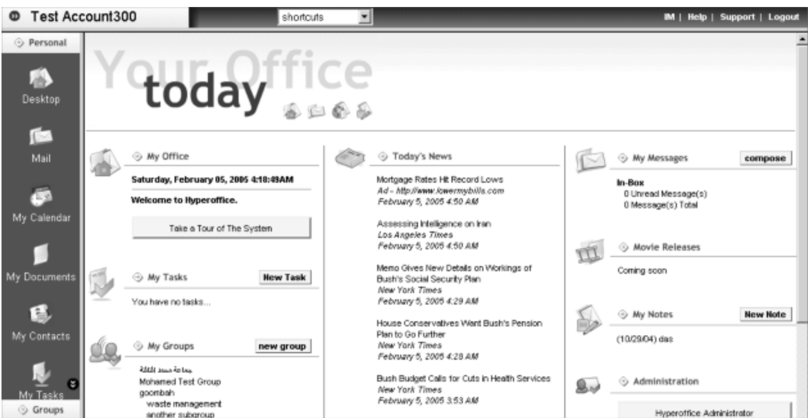


Figure 50.7

LTR page layout

Figure 50.8

RTL page layout

So how do we go about developing a page layout to handle directionality of writing system? Leaving graphics out of it, it's actually rather easy. Recall the following line in the code of Listing 50.8:

```
<html dir="#SESSION.writingDir#" lang="#SESSION.language#">
```

That's pretty much it. It's most often recommended to set the page's writing direction in the `<html>` tag using its `dir` attribute. That's because it will also set all of the page's HTML object's directionality, as well, while leaving you with the option of changing the directionality for individual HTML objects as needed. For the page's text, this setting will have the most effect on directionally neutral text (numbers, punctuation, and so on), since most of your Unicode text will have inherent directionality (certainly another reason to "Just use Unicode").

If your page layout design tends to HTML frames, you will have to use special logic to arrange the frames in their proper sequence (see Listing 50.11 for a simple example). On the other hand, if you design with cascading style sheets (CSS), there's no special logic required. Starting with version 2, CSS has a direction property similar to the HTML `dir` attribute (see `http://www.w3.org/TR/`

`CSS21/visuren.html#direction` for more information). CSS 3 goes a step further, adding the block-progression property to specify vertical flow (top-to-bottom) or horizontal flow (LTR or RTL), as well as a writing-mode property to act as shorthand for specifying both direction and block-progression (see `http://www.w3.org/TR/css3-text/#Progression` for specifics).

Listing 50.11 `frameLayout.cfm`—RTL Frame Layout Logic

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<cfoutput><html dir="#SESSION.writingDir#"></cfoutput>
<head>
  <title>Bubba's Triassic Desktop</title>
</head>
<!-- frames -->
<cfif SESSION.writingDir EQ "ltr">
<!--- menu left--->
<frameset cols="20%,*">
  <frame name="menu" src="menu.cfm">
  <frame name="desktop" src="desktop.cfm">
</frameset>
<cfelse>
<!--- menu right --->
<frameset cols="80%,*">
  <frame name="desktop" src="desktop.cfm" >
  <frame name="menu" src="menu.cfm">
</frameset>
</cfif>

</html>
```

Next we look at ColdFusion 's text searching as it applies to G11N applications.

Text Searching

Using the built-in Verity text search engine was problematic in ColdFusion versions prior to CFMX 7. For starters, it didn't support Unicode. It also only supported a few languages (mainly in line with the locales that ColdFusion previously supported). That made Verity's application across locales "uneven" (works in some locales, not in others) and therefore complex. It forced many G11N developers to turn to other solutions, such as Microsoft's Index Server or the open-source Lucene project. ColdFusion has changed all of this with the introduction of the Unicode character set for Verity collections, as well as new Verity languages (see Table 50.6).

The code to access the new Verity G11N functionality is more or less the same familiar code. To programmatically build a Verity Unicode collection, all you need do is set the language option to "`uni`":

```
<cfcollection
  action = "create"
  collection = "unicodeTest"
  path ="#collectionLocation#"
  language = "uni">
```

The same applies to searching a collection:

```
<cfsearch
  collection="unicodeTest"
  name="test"
  criteria="#searchPhrase#"
  language="uni">
```

This is incredibly economical; one simple code change opens up your Verity text-searching to the G11N world as well as simplifies your application by doing away with the need for third-party search engines. You'll need to download the "Verity International Search Packs" from http://www.adobe.com/support/coldfusion/verity_reg/register/index.cgi

Table 50.6 Supported Verity Languages in ColdFusion

ASIAN LANGUAGE PACK					
Japanese	Korean	Chinese	Traditional Chinese		
MULTILANGUAGE LANGUAGE PACK					
Unicode					
WESTERN EUROPEAN LANGUAGE PACK					
Bokmal	Finnish	Italian	Spanish	Danish	French
Nynorsk	Swedish	Dutch	German	Portuguese	
EASTERN EUROPEAN/MIDDLE EASTERN LANGUAGE PACK					
Arabic	Hebrew	Greek	Polish	Turkish	
Bulgarian	Russian	Czech	Hungarian	Russian2	

Next let's briefly review the new I18N bits that ColdFusion brings to the table.

What's New in ColdFusion Internationalization

ColdFusion MX 6 firmly rooted ColdFusion in the Java world, making it Unicode capable and opening up Java's huge I18N libraries to ColdFusion developers. ColdFusion MX 7 took this process one giant step further by aligning it's locales to core Java's locales, literally opening up the world to ColdFusion applications. ColdFusion continues this trend with the main I18N-related changes taking place in the JDK 6 (which is now the default JDK for ColdFusion).

The following CLDR (1.4) based locales are now supported:

- zh_SG—Chinese (Simplified), Singapore
- en_MT—English, Malta
- en_PH—English, Philippines
- en_SG—English, Singapore

- el_CY—Greek, Cyprus
- id_ID—Indonesian, Indonesia
- ga_IE—Irish, Ireland
- ms_MY—Malay, Malaysia
- mt_MT—Maltese, Malta
- pt_BR—Portuguese, Brazil
- pt_PT—Portuguese, Portugal
- es_US—Spanish, United States

This actually marks the first time that core Java has made use of the CLDR for its locale data.

Another interestingI18N change that JDK 6 provides is support for the *Japanese Imperial Calendar*. You can make use of it for date conversion and formatting by simply using the JP variant, *ja_JP_JP* for your locale as show in Listing 50.12, which should produce something similar to this:

平成19年7月26日

Listing 50.12 `jec.cfm`—Japanese Imperial Calendar

```
<cfscript>
// yes it's that simple! Don't you just love coldfusion?
setLocale("ja_JP_JP");
writeoutput("today in Japanese Imperial calendar: #lsDateFormat(now(),"FULL")#");
</cfscript>
```

The final ColdFusion I18N related change I'll mention is the addition of optional locale arguments for the following functions (these are discussed in a bit more detail below) allowing you to override the current locale:

- DayOfWeekAsString
- LSCurrencyFormat
- LSDateFormat
- LSEuroCurrencyFormat
- LSIsCurrency
- LSIsDate
- LSIsNumeric
- LSNumberFormat
- LSParseCurrency

- LSParseDateTime
- LSParseEuroCurrency
- LSParseNumber
- MonthAsString

Our final stop on this tour is a brief overview of the G11N-relevant tags and functions.

Relevant ColdFusion Tags/Functions

The following tables (Table 50.7 and Table 50.8) provide a list of the G11N-relevant ColdFusion tags and functions. The majority of these should be familiar to developers from ColdFusion MX 7.

`CharsetDecode` and `CharsetEncode` are functions `you should` find useful in situations where you're forced to convert string data to/from its binary representation. `CharsetEncode` is intended as a replacement for the `ToString` function. `CharsetDecode` is a "shortcut" function for the process of string-to-binary conversion. (In ColdFusion MX 6.1 you had to first set the string to Base64 and then use the `ToBinary` function to convert the string to binary data.) Both of these functions allow you to control the encoding/decoding process more finely.

Table 50.7 ColdFusion G11N Tags

FUNCTION	PARAMETER	USE
`cfcontent`	`type`	Specifies the encoding in which to return the results to the client browser.
`cffile`	`charset`	Specifies how to encode data written to a file, or the encoding of a file being read.
`cfheader`	`charset`	Specifies the character encoding in which to encode the HTTP header value.
`cfhttp`	`charset`	Specifies the character encoding of the HTTP request.
`cfhttpparam`	`mimeType`	Specifies the MIME media type of a file; can also include the file's character encoding.
`cfmail`	`charset`	Specifies the character encoding of the mail message, including the headers.
`cfmailpart`	`charset`	Specifies the character encoding of one part of a multipart mail message.
`cfprocessingdirective`	`pageEncoding`	Identifies the character encoding of the contents of a page to be processed by ColdFusion MX.

NOTE

In ColdFusion the localization functions, those that begin with `LS` like `LSDateFormat`, `LSNumberFormat`, and so on also take an optional locale argument to use instead of the current locale. This would be helpful for displaying data for multiple locales on the same page.

Table 50.8 ColdFusion G11N Functions

FUNCTION	PARAMETER	USE
`GetLocale`	-	Returns the current locale setting.
`GetLocaleDisplayName`	-	Returns the name of a locale in the language of a specific locale. The default value is the current locale in the locale's language.
`LSCurrencyFormat`	-	Converts numbers into a string in a locale-specific currency format using either the current server locale or locale argument.
`LSDateFormat`	-	Converts the date part of a date/time value into a string in a locale-specific date format using either the current server locale or locale argument..
`LSEuroCurrencyFormat`	-	Converts a number into a string in a locale-specific currency format using either the current server locale or locale argument.
`LSIsCurrency`	-	Determines whether a string is a valid representation of a currency amount in the current locale.
`LSIsDate`	-	Determines whether a string is a valid representation of a date/time value in the current locale.
`LSIsNumeric`	-	Determines whether a string is a valid representation of a number in the current locale.
`LSNumberFormat`	-	Converts a number into a string in a locale-specific numeric format using either the current server locale or locale argument.
`LSParseCurrenc` y	-	Converts a string that is a currency amount in the current locale into a formatted number using either the current server locale or locale argument.
`LSParseDateTime`	-	Converts a string that is a valid date/time representation into a date-time object using either the current server locale or locale argument.
`LSParseEuroCurrency`	-	Converts a string that is a currency amount in the current locale or locale argument into a formatted number. Requires `Euro` as the currency for all countries that use the Euro.
`LSParseNumber`	-	Converts a string that is a valid numeric representation in the current locale or locale argument into a formatted number.
`LSTimeFormat`	-	Converts the time part of a date/time value into a string in a locale-specific date format using either the current server locale or locale argument..

Table 50.8 (continued)

FUNCTION	PARAMETER	USE
`SetLocale`	-	Specifies the locale setting.
`CharsetDecod e`	`encoding`	Converts a string in the specified `encoding` to a binary object.
`CharsetEncode`	`encoding`	Converts a binary object to a string in the specified `encoding`.
`GetEncoding`	-	Returns the character encoding of text in the Form or URL scope.
`SetEncoding`	`charset`	Specifies the character encoding of text in the Form or URL scope. Used when the character set of the input to a form, or the character set of a URL, is not in UTF-8 encoding.
`ToBase64`	`encoding`	calculates the Base64 representation of a string.
`ToString`	`encoding`	Returns a string encoded in the specified character encoding.
`URLDecode`	`charset`	Decodes a `URL`-encoded string.
`URLEncodedFormat`	`charset`	Generates a `URL` encoded string.
`CharsetDecode`	`encoding`	Converts a string in the specified `encoding` to a binary object.
DayOfWeekAsString	locale	Returns day of week formatted for specified locale.

Better G11N Practices

We'll wrap up this chapter by providing a set of "better" practices for developing G11N applications in ColdFusion. "Better" means better than "good" but not quite ready to be the "best." Why not "best practices"? My own modesty aside, mainly because ColdFusion is so new and there are so many ways to pitch globalized woo.

As shown throughout this chapter, the introduction of CFCs and Unicode in ColdFusion, as well as the fact that it's easier access to Java I18N libraries, certainly improved our ability to develop G11N applications. Yet G11N concepts are still relatively new to ColdFusion application developers, and I expect there is still quite a bit more to evolve as the community's "heavy thinkers" turn their brains to this field. What follows is what I think are better G11N practices now that ColdFusion has come to town.

What Not to Do

Before beginning on what you ought to be doing, let me consolidate some of the points made in the previous sections about what *not* to do.

Don't use a database that doesn't support Unicode. In this day and age, it must be some kind of dinosaur anyway, so why bother?

Don't make your application or it's database monoculture. Here are some things to do to make your application more culturally sensitive:

- Require almost nothing cultural in your forms or database.
- Non-English terms, names, and so forth are sometimes longer than their English equivalent or consist of more than one word. Keep this in mind when designing databases, input forms, and reports.
- Mailing address forms and database designs should be flexible in order to handle the varied address styles in use around the world (see `http://www.upu.int/post_code/en/addressing.html` for examples). Postal codes (ZIP codes in the U.S.) should not assume numeric-only data. The Street part of your address design should allow for more than one street part. Your application logic should not assume that address identifiers are always house numbers/street addresses. Also, it should not assume any patterns (left side odd, right side even) or regular sequences (house #3 might not come after house #1).
- Don't assume global measurement units are the same ones printed on your cereal box. Your application should separate measurement units from measurement values (that is, as separate database columns or form fields). And for developers in metric countries, the whole world isn't yet metric. For applications dealing with apparel (clothing, shoes, and so on) be aware that sizes vary wildly from locale to locale.
- Always store date/time data as UTC (Universal Time Coordinate) or Greenwich Mean Time. It's a generally good idea to keep your date time data as UTC, especially if your servers or users are physically located in many time zones.

Don't presume a Gregorian calendar system for your date/time data. There are at least six other major calendar systems in popular use today (Buddhist, Chinese, Hebrew, Islamic, Japanese and Persian, as discussed in this chapter). While it's sometimes acceptable to use the Gregorian calendar with localized date formats, this isn't true in all locales—it's not A.D. 2007 to everyone. As noted in the "Calendars" section, this is especially critical for date-based calculations; one person's 30 days might not be another's month.

Don't ignore CSS. Although CSS use can be complicated owing to browser quirks and the lack of a CSS police force, its use can greatly simplify G11N page layouts across locales, especially for applications that need to support BIDI text.

Don't assume text or graphic directionality. For instance, people in Arabic/Hebrew cultures look at a page of text and graphics differently from people in Canada.

Don't mix text/text presentation and application code. This is, by far, the dreariest part of making an existing ColdFusion application I18N. You must search out each wayward bit of hard-coded text and replace it with ColdFusion variables of one sort or another.

Don't fail to use UTF-8 encoding. *Just use Unicode*. It's the default encoding for ColdFusion and so offers the path of least resistance to ColdFusion globalization.

TIP

If you want to save yourself the trouble of showering your ColdFusion pages with `<cfprocessingdirective>` tags, it's wise to use an editor such as Dreamweaver, which supports a BOM (byte order mark) so that the ColdFusion server will automatically understand your page's encoding. However if there's a chance of other users using software which might remove the BOM, it's often wiser to liberally use `<cfprocessingdirective>` tags. I should also point out that the popular Eclipse development platform doesn't support BOM in it's file editing.

Let's next examine a design issue: monolingual versus multilingual Web sites.

Monolingual or Multilingual Web Sites

Some folks in the G11N field often break down Web application design to a choice between so-called monolingual or multilingual designs. In its simplest form, a *monolingual* design opts for a distinct URL for each language served by the application. For example, a Web site originally in English, say `www.iWantYourMoney.com`, might through some mechanism redirect its French users to `fr.iWantYourMoney.com`, its Thai users to `th.iWantYourMoney.com`, and so on. Such a design is obviously not well suited to static HTML text (translation and HTML file management would quickly become a nightmare). It would, however, work quite nicely as a ColdFusion-driven Web site. The downside to this approach is when an application has to serve locales rather than a single language; for example, French in Canada versus French in France. This design would either have to add increasingly more URLs to the mix it must maintain, or it would have to develop special mechanisms to handle locale within a monolingual site (something akin to a multilingual design to serve locales instead of languages within a monolingual site).

A *multilingual* design would serve all supported languages and/or locales from a single URL, `www.iWantYourMoney.com`. And this usually makes heavy use of resource bundles to deliver locale-specific text. Given the choice, this is my personal preference for the following reasons:

- Much more sensitive to locale. You can just as easily serve `fr` as you can `fr_FR` or `fr_CA`.
- Simplifies the application by removing the need to redirect to another URL, which also makes it more palatable to users.
- Helps simplify load-balancing schemes.
- Reduces potential issues with site structure varying across locales. (This is a pet peeve of mine. I find it particularly annoying to see a Web site's structure in one language be a thin shadow of itself in another; government Web sites are often culprits).

As you might imagine, there are shades of these two design colors, and variations within each. There's really no right or wrong way to handle this, as long as you follow the globalization practices outlined in this chapter.

Locale Stickiness

As noted earlier in the chapter, it's very important for your ColdFusion application to remember a user's locale preference. The monolingual design is one sure way not to forget; users are basically "stuck" in a domain that's devoted to their language/locale. Stickiness in the multilingual design can be maintained via

- URL variables such as `index.cfm?locale=fr_CA`, which requires some mechanism to rewrite the URL variable string to append the locale on each page request
- Cookies, but these are subject to users turning them off, proxy servers trashing them, and so on
- `SESSION` variables

A frequent choice for maintained stickiness is the `SESSION` variable because it is usually the easiest to understand and maintain in an application's code. The user's locale choice is simply "there," and nothing special need be done in the application to maintain it across page requests. The downside to using these variables is the added complexity in load balancing and handling expired sessions. Of course, another option to maintain stickiness is to use `SESSION` variables with URL variables as a fallback mechanism.

HTML

ColdFusion always overrides HTML-based character encoding `META` tags (specifically `CONTENT-TYPE`). However, it's usually a good idea to include these, properly identified, in your ColdFusion pages. Why? The main reasons are search engines and speech synthesizers (without some kind of hint, most text-to-speech software cannot tell one language from another). The three most important HTML tags from this perspective are `<HTML>` and the `CONTENT-LANGUAGE META` and `CONTENT-TYPE META` tags.

As noted earlier in the Display section, the `<HTML>` tag has two important G11N attributes:

- The `lang` attribute, which specifies the base language of an element's attribute values and text content (note that you can apply the `lang` attribute to many HTML elements). Acceptable values for `lang` follow ISO639 and ISO3166, using a format such as `primary-language-code-subcode`. This is normally a two-letter country code but might also include language versions (for instance, "`en-cockney`", the Cockney version of English).
- The `dir` attribute, which specifies the base direction of directionally neutral text. Like the `lang` attribute, `dir` can be applied to many HTML elements such as tables, inputs, and so forth. Acceptable values for `dir` attribute are `LTR` (left-to-right) and `RTL` (right-to-left).

The `CONTENT-LANGUAGE META` tag specifies the primary human language(s) for a page. For example, the following tag indicates that the page has English as used in the United States, Thai as used in Thailand, and French on this page:

```
<META HTTP-EQUIV="CONTENT-LANGUAGE" CONTENT="en-US,th-TH,fr">
```

The CONTENT-TYPE META tag specifies the content type, such as text/html, and the character set used on this page. This is one tag that I habitually include. The following META tag indicates that this page is plain text/HTML and uses a UTF-8 character set:

```
<META HTTP-EQUIV="CONTENT-TYPE" CONTENT="text/html; charset=UTF-8">
```

See Listing 50.8 for an example of these HTML tags. Even though ColdFusion will ignore them, it is recommended to include these three HTML tags with appropriate attribute values.

CFML

Besides the HTML tags listed just above, your application should include the following in your application.cfm:

```
<!--- url and form encoding to UTF-8. --->
<cfset setEncoding("URL", "UTF-8")>
<cfset setEncoding("Form", "UTF-8")>
<!--- output encoding to UTF-8 --->
<cfcontent type="text/html; charset=UTF-8">
```

Your applications should also include:

```
<cfprocessingdirective pageEncoding="utf-8">
```

at or near the top of each and every one of your applications' templates, unless you can be 100 percent sure that each of them is properly encoded as UTF-8, including the byte order mark (BOM). Since the strict definition of UTF-8 encoding doesn't actually mention using a BOM, you might be better off including the <cfprocessingdirective> tag.

Resource Bundles

To completely separate text and text presentation from application code, your ColdFusion application should make use of resource bundles (discussed earlier in the chapter). The resource bundle "flavor" you use depends entirely on your application's logic and how and where it can be deployed, although it is strongly recommended to use the Java-style resource bundle owing to the excellent set of free management tools available.

Your application should separate resource bundle files into logical groupings (for instance, menu.properties, deskTop.properties, loginForm.properties) and then again into locale-specific files within those groupings (menu_en_US.properties, menu_th_TH.properties, and so forth). It is recommended that resource bundles be loaded into shared-scope structure variables with locale as key—for instance, APPLICATION.menu.en_us or, if you prefer, APPLICATION.menu["en_US"].

There are generally two approaches used to initialize these resource bundles: "on demand," and "fire and forget."

"On demand" initialization, in which the application loads each locale's resource bundles as needed (that is, when a user from that locale visits the Web site). This approach might be appropriate in situations where the developer knows or suspects the application will have an uneven distribution of locale users, say 1 million Americans, 500,000 Italians, 10,000 Brazilians, and 3 guys in New Jersey

that speak Zulu. (Of course, there are plenty of Zulu or IsiZulu speakers outside New Jersey, but for the purpose of this discussion let's assume there are only three and they live in Hoboken.) Server resources are only used as and when needed—if the three Zulu speakers in New Jersey never visit the Web site, the application never loads that locale's resource bundle. Listing 50.13 shows an example of this arrangement.

Listing 50.13 `onDemandRB.cfm`—"On Demand" Resource Bundle Loading

```
<cfscript>
// uses rbJava CFC to handle java style resource bundles
if (NOT structKeyExists(APPLICATION.commonBundle,"#SESSION.userLocale#")) {

APPLICATION.commonBundle[SESSION.userLocale]=rB.getResourceBundle("common.thisAppCom
mon",SESSION.userLocale,markDebug);

APPLICATION.adminBundle[SESSION.userLocale]=rB.getResourceBundle("admin.thisAppAdmin
",SESSION.userLocale,markDebug);

APPLICATION.appBundle[SESSION.userLocale]=rB.getResourceBundle("applications.thisApp
Applications",SESSION.userLocale,markDebug);

APPLICATION.globalBundle[SESSION.userLocale]=rB.getResourceBundle("global.thisAppGlo
bal",SESSION.userLocale,markDebug);

APPLICATION.groupBundle[SESSION.userLocale]=rB.getResourceBundle("groups.thisAppGrou
ps",SESSION.userLocale,markDebug);

APPLICATION.toolsBundle[SESSION.userLocale]=rB.getResourceBundle("tools.thisAppTools
",SESSION.userLocale,markDebug);
}
</cfscript>
```

This code assumes the use of the `rbJava.CFC` discussed previously, and also assumes that the relevant ColdFusion structures have been created to hold the resource bundles. The `rbJava.CFC` returns a structure holding the resource bundle's key/value pairs. The code first checks to see if one of the resource bundle structures has a key for this user's locale. If not, it then proceeds to load the various resource bundles into the appropriate structures using this user's locale as a key.

The dot-syntax naming scheme (`common.thisAppCommon`) for the resource bundle properties file is required for the `getBundle` method, which uses a `directory.fileName` notation to find the correct resource bundle property file.

The `markDebug` option for that CFC's `getResourceBundle` function indicates whether or not to mark the returned resource bundles with asterisks (`*`) to aid in debugging. (It helps pick out text supplied from the resource bundle versus static application text leftover from the I18N process; that is, bugs.)

"*Fire and forget*" simply loads all supported-locale resource bundles when the application is initialized, rather than waiting for a user to initialize a new resource bundle for a particular locale. This method of initialization might best be applied when the developer knows that locale usage is evenly distributed or doesn't want to bother with any dynamic loading of resource bundles.

Listing 50.14 provides a simple example of this technique. The array of application-supported locales (`APPLICATION.supportLocales`) might be supplied as an application parameter or dynamically determined by parsing the locales available for any given resource bundle.

Listing 50.14 `ffRB.cfm`—"Fire and Forget" Resource Bundle Loading

```
<cfscript>
// uses rbJava CFC to handle java style resource bundles
for (i=1; i LTE arrayLen(APPLICATION.supportLocales); i=i+1) {
  //verbose for readability
  thisLocale=APPLICATION.supportLocales[i];

APPLICATION.commonBundle[thisLocale]=rB.getResourceBundle("common.thisAppCommon",thi
sLocale,markDebug);

APPLICATION.adminBundle[thisLocale]=rB.getResourceBundle("admin.thisAppAdmin",thisLo
cale,markDebug);

APPLICATION.appBundle[thisLocale]=rB.getResourceBundle("applications.thisAppApplicat
ions",thisLocale,markDebug);

APPLICATION.globalBundle[thisLocale]=rB.getResourceBundle("global.thisAppGlobal",thi
sLocale,markDebug);

APPLICATION.groupBundle[thisLocale]=rB.getResourceBundle("groups.thisAppGroups",this
Locale,markDebug);

APPLICATION.toolsBundle[thisLocale]=rB.getResourceBundle("tools.thisAppTools",thisLo
cale,markDebug);
}
</cfscript>
```

Just Use Unicode

If your application needs to support more than a few languages—especially any of the CJK (Chinese-Japanese-Korean) languages—it needs to use Unicode as its character encoding. This isn't rocket science and, as emphasized earlier, there are no real adverse side effects to using Unicode. The only serious issue arises over legacy data using codepage encodings, and in the long run you'll be better off biting the bullet and doing the conversion early on in the application's life cycle. *Just use Unicode.*

When ColdFusion Isn't Enough

Out of the preceding sections, you'll want to recall one of the guiding principles for developing G11N applications: These applications should be *generic*. A G11N application must be distilled down to its essence (internationalized) before it becomes specialized (localization). That old saw "Build once, run anywhere" really does apply to G11N applications.

Making generic applications in the face of all this complexity is difficult. It becomes even more difficult if you are faced with the situation where you need to support locales that ColdFusion doesn't (or where core Java provides poor support, see Appendix D, online). Figure 50.9 shows a graph of

locales supported by various versions of ColdFusion, with the final value in the graph showing locales supported by ColdFusion backed by the ICU4J library. If you're at this G11N business long enough, you will eventually run into that 90+ locale difference. It becomes a question then of handling the unsupported locales as special cases, and maintaining the simplicity and speed offered by the native ColdFusion G11N tags and functions for the rest of your locales. Or you can switch to the complete use of ICU4J's G11N functions and deal with the added code complexity, but maintain a more generic application.

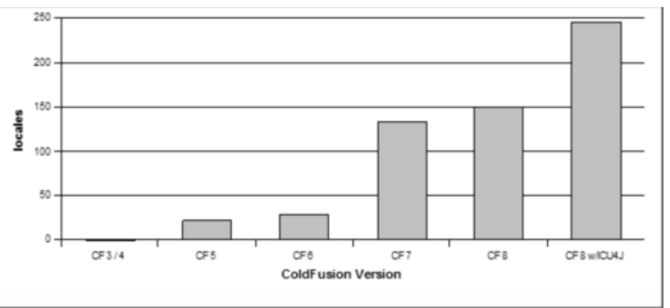


Figure 50.9

ColdFusion locale support by version

There is no right or wrong answer for this question of generic applicability. It depends a great deal on the individual developer and/or the policies of the development shop, and how long both have been in the G11N business. My own shop has more or less slid into the ICU4J-only style. We've developed enough applications in locales that ColdFusion previously didn't support (such as Thai and Arabic) that we have a rather large set of essentialICU4J-based tools that we can't live without. Developers new to G11N applications have a tough decision to make and, quite frankly, I can't offer any critical advice—other than to measure out your locale support and see if that warrants one style or the other.

You're probably thinking to yourself, "Well that was a mouthful". Perhaps it was but G11N concepts are not trivial, even though the code for it very often is. This chapter covered the major G11N issues including locales, character encoding, databases and text/code separation via resource bundles. It also provided information to help you over the rough spots and code, download resources to help you easily deal with the more common G11N needs. Finally it set out some "better" G11N practices that you should follow.

"So long and thanks for all the fish."

CHAPTER 51 Error Handling

IN THIS CHAPTER

Catching Errors as They Occur E429
Throwing and Catching Your Own Errors E462

Catching Errors as They Occur

Unless you are a really, really amazing developer—and an incredibly fast learner—you have seen quite a few error messages from ColdFusion, both during development and after deployment of your applications. ColdFusion is generally very good about providing diagnostic messages that help you understand what the problem is. Error messages are a developer's friends. These clever little helpers selflessly provide hints and observations about the state of your code so you can fix it as soon as possible. It's an almost romantic relationship.

Unfortunately, your users won't see error messages through the rose-colored glasses coders tend to wear. They will call or page you or flood your in-box with unfriendly email until you get the error message (which they perceive as some kind of ugly monster) under control.

Wouldn't it be great if you could have your code watch for certain types of errors and respond to them on the fly, before the user even sees them? After all, as a developer, you often know the types of problems that might occur while a particular chunk of code does its work. If you could teach your code to recover from predictable problems on its own, you could lead a less stressful and healthier (albeit somewhat lonelier) coding lifestyle.

This and more is what this chapter is all about. ColdFusion provides a small but powerful set of structured exception-handling tags, which enable you to respond to problems as they occur.

What Is an Exception?

When ColdFusion displays an error message, it's responding to an exception. Whenever a CFML tag or function is incapable of doing whatever your code has asked it to do—such as connect to a database or process a variable—it lets ColdFusion know what exactly went wrong and says why. This process of reporting a problem is called *raising an exception*. After the tag or function raises an exception, ColdFusion's job is to respond to it. Out of the box, ColdFusion responds to nearly all exceptions in the

same way, by displaying an error message that describes the exception. ColdFusion's logs also note the fact that the exception occurred. If you want, you can use its exception-handling tags to respond to an exception in some different, customized way. When you do this, you are telling ColdFusion to run special code of your own devising, instead of displaying an error message as it normally would. This process of responding to an exception with your own code is called *catching an exception*.

NOTE

After your code has caught an exception, ColdFusion no longer considers itself responsible for displaying any type of error message, and suppresses the error message.

NOTE

Although subtle distinctions exist between the exact meanings of each term, for now you can consider exception, exception condition, error, and error condition to all mean the same thing: the state that occurs whenever an operation can't be completed. For purposes of this discussion, also consider raising and throwing to be synonyms as well. Raising an exception, throwing an error, and throwing an exception all mean pretty much the same thing.

As you learned in Chapter 19, "Introducing the Web Application Framework," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 1: Getting Started*, you can customize the look and feel of ColdFusion's error messages. If all you want to do is change the way error messages look, you can skip this chapter and just use what you learned in Chapter 19. But if you want to take active recovery steps when an error occurs, read on.

Introducing `<cftry>` and `<cfcatch>`

ColdFusion provides two basic CFML tags for handling exception conditions:

- The `<cftry>` tag—A paired tag you place around the portions of your templates that you think might fail under certain conditions. The `<cftry>` tag doesn't take any attributes. You simply place opening and closing `<cftry>` tags around the block of code you want ColdFusion to attempt to execute.
- The `<cfcatch>` tag—Used to catch exceptions that occur within a `<cftry>` block. `<cfcatch>` takes only one attribute, `type`, as shown in Table 51.1. `type` tells ColdFusion what type of problem you are interested in responding to. If that type of problem occurs, ColdFusion executes the code between the `<cfcatch>` tags. Otherwise, it ignores the code between the `<cfcatch>` tags.

Table 51.1 `<cfcatch>` Tag Attributes

ATTRIBUTE	DESCRIPTION
`type`	The type of exceptions to catch or respond to. It can be any of the types shown in Table 51.2. For instance, if you are interested in trying to recover from errors that occur while executing `<cfquery>` or another database-related operation, you would use a `<cfcatch>` of `type="Database"`.

Table 51.2 Predefined Exception Types

EXCEPTION TYPE	DESCRIPTION
`Any`	Catches any exception, even those you might not have any way of dealing with (such as an "out of memory" message). Use this exception value if you need to catch errors that don't fall into one of the other exception types in this table. If possible, use one of the more specific exception types listed in this table.
`Application`	Catches application-level exception conditions. Your own CFML code reports these exceptions, using the `<cfthrow>` tag. In other words, catch errors of `type="Application"` if you want to catch your own custom errors.
`Database`	Catches database errors, which could include errors such as inability to connect to a database, an incorrect column or table name, a locked database record, and so on.
`Expression`	Catches errors that occur while attempting to evaluate a CFML expression. For instance, if you refer to an unknown variable name or provide a function parameter that doesn't make sense when your template actually executes, an exception of type `Expression` is thrown.
`Lock`	Catches errors that occur while attempting to process a `<cflock>` tag. Most frequently, this type of error is thrown when a lock can't be obtained within the `timeout` period specified by the `<cflock>` tag. This usually means that some other page request that uses a lock with the same name or scope is taking a long time to complete its work.
`MissingInclude`	Catches errors that arise when you use a `<cfinclude>` tag but the CFML template you specify for the `template` attribute can't be found.
`Object`	Catches errors that occur while attempting to process a `<cfobject>` tag or `createObject()` function, or while attempting to access a property or method of an object returned by `<cfobject>` or `createObject()`.
`Security`	Catches errors that occur while using one of ColdFusion's built-in security-related tags.
`Template`	Catches general application page errors that occur while processing a `<cfinclude>`, `<cfmodule>`, or `<cferror>` tag.
`SearchEngine`	Catches errors that occur while performing full-text searches or other Verity-related tasks, such as indexing. For details about Verity, see Chapter 39, "Full-Text Searching," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 2: Application Development*.

TIP

You can also provide your own exception types for the `type` attribute of the `<cfcatch>` tag. For details, see the "Throwing and Catching Your Own Errors" section, later in this chapter.

Basic Exception Handling

The easiest way to understand exception handling is to actually go through the process of adding `<cftry>` and `<cfcatch>` to an existing template. Listings 51.1 and 51.2 show the effects of ColdFusion's exception-handling tags, using a before-and-after scenario.

A Typical Scenario

Say you have been asked to create a page that lets users select from two separate drop-down lists. The first drop-down provides a list of films; the second shows a list of film ratings. Each drop-down list has a Go button next to it, which presumably takes the user to some type of detail page when clicked.

Here's the catch: For whatever reason, you know ahead of time that the database tables populating these drop-down lists (the Films and FilmsRatings tables) won't always be available. There might be any number of reasons for this. Perhaps you are connecting to a database server that is known to crash often, that goes down during certain times of the day for maintenance, or that is accessed via an unreliable network connection.

Your job is to make your new drop-down list page operate as gracefully as possible, even when the database connections fail. At the very least, users shouldn't see an ugly database error message. If you can pull off something more elegant, such as querying some kind of backup database in the event that the normal database tables are not available, that's even better. Your yearly review is coming up, and you might get a hefty raise if you can pull off this project with aplomb. Let's see whether ColdFusion can help you get that raise.

A Basic Template, Without Exception Handling

Listing 51.1 is the basic template, before the addition of any error handling. This template works just fine as long as no problems occur while the user is connecting to the database. It runs two queries and displays two drop-down lists, as shown in Figure 51.1. If an error occurs, though, the user sees an ugly error message and can't get any further information.

Figure 51.1
This is what the drop-down page looks like, as long as no errors occur.

NOTE

The examples in this chapter assume the creation of an `Application.cfc` file that sets the `APPLICATION.dataSource` variable and also turns on Session and Client management (as discussed in Chapter 18, "Planning an Application," in Vol. 1, *Getting Started*). For your convenience, the appropriate `Application.cfc` file for this chapter may be downloaded online.

Listing 51.1 `ChoicePage1.cfm`—A Basic Display Template, Without Any Error Handling

```
<!---
 Filename: ChoicePage1.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Provides navigation elements for films and ratings
--->

<html>
<head><title>Film Information</title></head>
<body>
<h2>Film Information</h2>

<!--- Retrieve Ratings from database --->
<cfquery name="getRatings" datasource="#APPLICATION.dataSource#">
 SELECT RatingID, Rating
 FROM FilmsRatings
 ORDER BY Rating
</cfquery>
<!--- Retrieve Films from database --->
<cfquery name="getFilms" datasource="#APPLICATION.DataSource#">
 SELECT FilmID, MovieTitle
 FROM Films
 ORDER BY Films.MovieTitle
</cfquery>


<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">
 <!--- Display Film names in a drop-down list --->
 <P>Films:
 <cfselect query="getFilms" name="filmID" value="filmID"
  display="MovieTitle"/>

 <!--- Display Rating names in a drop-down list --->
 <P>Ratings:
 <cfselect query="getRatings" name="ratingID" value="RatingID"
  display="Rating"/>

</cfform>

</body>
</html>
```

This template doesn't contain anything new yet. Two `<cfquery>` tags named `getRatings` and `getFilms` run, and the results from each query appear in a drop-down list that enables the user to select a film or rating. If any of this looks unfamiliar, review Chapter 14, "Using Forms to Add or Change Data," in Vol. 1, *Getting Started*.

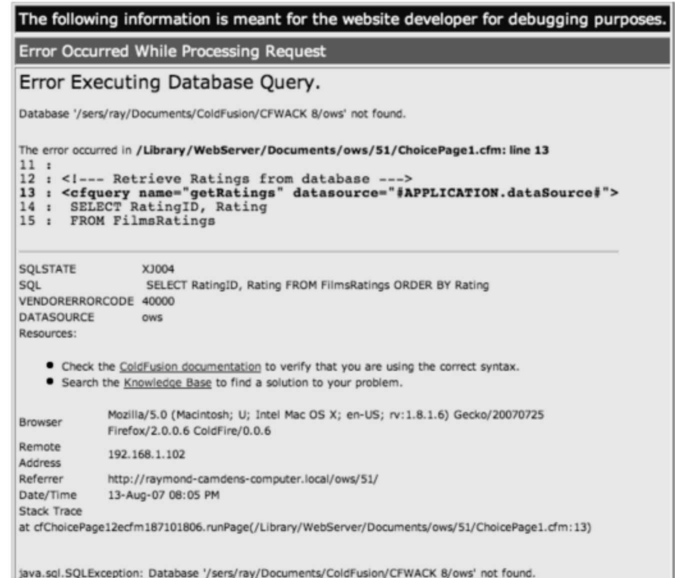
If for whatever reason getRatings and getFilms can't execute normally, an error message appears, leaving the user with nothing useful (other than a general impression that you don't maintain your site very carefully). For instance, if you go into the ColdFusion Administrator and sabotage the connection by providing an invalid filename in the Database File field for the ows data source, you'll see the error shown in Figure 51.2.

Figure 51.2

If an error occurs while connecting to the database, the default error message is displayed.

Film Information

The web site you are accessing has experienced an unexpected error.
Please contact the website administrator.



Adding <cftry> and <cfcatch>

To add ColdFusion's structured exception handling to Listing 51.1, simply wrap a pair of opening and closing <cftry> tags around the two <cfquery> tags. Then add a <cfcatch> block that specifies an exception type of Database, just before the closing </cftry> tag. The code inside the <cfcatch> tag will execute whenever either of the <cfquery> tags raises an exception.

Listing 51.2 is a revised version of Listing 51.1. The code is almost exactly the same, except for the addition of the <cftry> and <cfcatch> blocks, which display a basic "Sorry, we are not able to connect" error message. Now, if any problems occur when connecting to the database, the error message appears as shown in Figure 51.3. This really isn't much better than what the user saw in Figure 51.2, but it's a start.

Figure 51.3

You can use a <CFCATCH> block in its simplest form to display context-specific error messages.

Listing 51.2 ChoicePage2a.cfm—Adding a Simple <cftry> Block to Catch Database Errors

```
<!---
 Filename: ChoicePage2a.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Provides navigation elements for films and ratings
--->

<html>
<head><title>Film Information</title></head>
<body>
<h2>Film Information</h2>

<cftry>
 <!--- Retrieve Ratings from database --->
 <cfquery name="getRatings" datasource="#APPLICATION.dataSource#">
 SELECT RatingID, Rating
 FROM FilmsRatings
 ORDER BY Rating
 </cfquery>

 <!--- Retrieve Films from database --->
 <cfquery name="getFilms" datasource="#APPLICATION.dataSource#">
 SELECT FilmID, MovieTitle
 FROM Films
 ORDER BY Films.MovieTitle
 </cfquery>

 <!--- If any database errors occur during above query, --->
 <cfcatch type="Database">
 <!--- Let user know that Films data can't be shown right now --->
 <cfoutput>
<i>Sorry, we are not able to connect to our real-time database at
the moment, due to carefully scheduled database maintenance.</i><br>
 </cfoutput>

 <!--- Stop processing at this point --->
 <cfabort>
 </cfcatch>
```

Listing 51.2 (continued)

```
</cftry>

<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">
 <!--- Display Film names in a drop-down list --->
 <P>Films:
 <cfselect query="getFilms" name="filmID" value="FilmID"
  display="MovieTitle"/>

 <!--- Display Rating names in a drop-down list --->
 <P>Ratings:
 <cfselect query="getRatings" name="ratingID" value="RatingID"
  display="Rating"/>

</cfform>

</body>
</html>
```

Even though it's rather simplistic, take a close look at Listing 351.2. First, the `<cftry>` block tells ColdFusion that you are interested in trapping exceptions. Within the `<cftry>` block, all of the queries needed by the page execute. If any type of problem occurs—anything from a crashed database server to a mere typo in your SQL code—the code inside the `<cfcatch>` block executes, displaying the static message shown in Figure 51.3. Otherwise, the `<cfcatch>` block is skipped entirely.

NOTE

It's important to note that the `<cfcatch>` block in Listing 51.2 includes a `<cfabort>` tag to halt all further processing. If not for the `<cfabort>` tag, ColdFusion would continue processing the template, including the code that follows the `<cftry>` block. This would just result in a different error message, because the `<cfselect>` tags would be referring to queries that never ended up running.

All this code does so far is display a custom error message without taking any specific action to help the user, so Listing 51.2 really isn't any better than the templates in Chapter 18, which used the `<cferror>` tag or `onError()` method to customize the display of error messages. The advantages of using `<cftry>` and `<cfcatch>` will become apparent shortly. In practice, you will often use `<cftry>` and `<cfcatch>` along with `<cferror> or onError()`.

Understanding What Caused the Error

When it catches an exception, ColdFusion populates a number of special variables that contain information about the problem that actually occurred. These variables are available to you via the special `CFCATCH` scope. Your code can examine these `CFCATCH` variables to get a better understanding of what exactly went wrong, or you can just display the `CFCATCH` values to the user in a customized error message.

Table 51.3 lists the variables available to you within a `<cfcatch>` block.

Table 51.3 `CFCATCH` Variables Available After an Exception Is Caught

VARIABLE	DESCRIPTION
`CFCATCH.Type`	The type of exception that was caught. If the exception isn't a custom one, it will be one of the exception types listed in Table 51.2.
`CFCATCH.Message`	The text error message that goes along with the exception that was caught. Nearly all ColdFusion errors include a reasonably helpful `Message` value; this is the message that shows up at the top of normal error messages. For instance, the value of `CFCATCH.Message` for the exception shown in Figure 51.2 is `Error Executing Database Query`.
`CFCATCH.Detail`	Detail information that goes along with the caught exception. Most ColdFusion errors include a helpful `Detail` value. For the error shown in Figure 51.2, it's the value of `CFCATCH`.
`CFCATCH.SqlState`	Available only if `type` is `Database`. A standardized error code that should be reasonably consistent for the same type of error, even between different database systems.
`CFCATCH.Sql`	Available only if `type` is `Database`. The SQL statement that was used in the query.
`CFCATCH.queryError`	Available only if `type` is `Database`. The error message returned from the database.
`CFCATCH.where`	Available only if `type` is `Database`. If the query had used `<cfqueryparam>` tags, the where value will contain the name-value pairs used in the tags.
`CFCATCH.NativeErrorCode`	Available only if `type` is `Database`. The native error code reported by the database system when the problem occurred. These error codes are not usually consistent between database systems.
`CFCATCH.ErrNumber`	Available only if `type` is `Expression`. This code identifies the type of error that threw the exception. We recommend that you don't use this value to check for specific errors, because the values are not documented and have been known to change from version to version of ColdFusion. It's generally better to examine the text of the `CFCATCH.Detail` or `CFCATCH.Message` values.
`CFCATCH.MissingFileName`	Available only if `type` is `MissingInclude`. The name of the ColdFusion template that could not be found.
`CFCATCH.LockName`	Available only if `type` is `Lock`. The name of the lock, if any, that was provided to the `<cflock>` tag that failed.
`CFCATCH.LockOperation`	Available only if `type` is `Lock`. At this time, this value will always be `Timeout`, `Create Mutex`, or `Unknown`.
`CFCATCH.ErrorCode`	Available only when you throw your own exceptions with `<cfthrow>`. The value, if any, that was supplied to the `errorCode` attribute of the `<CFTHROW>` tag that threw the exception.

Table 51.3 (continued)

VARIABLE	DESCRIPTION
`CFCATCH.ExtendedInfo`	Available only when you throw your own exceptions with `<cfthrow>`. The value, if any, that was supplied to the `extendedInfo` attribute of the `<cfthrow>` tag.
`CFCATCH.TagContext`	An array of structures that contains information about the ColdFusion templates involved in the page request when the exception occurred. This value is used primarily for creating your own debugging templates, not for exception handling as discussed in this chapter. For details, see the ColdFusion documentation.

Listing 51.3 is a slightly revised version of Listing 51.2. This time, the message shown to the user includes information about the exception, by outputting the value of `CFCATCH.SqlState`. One possible result is shown in Figure 51.4; this figure was taken after changing a column name in one of the queries to an unknown name.

Figure 51.4
You can use the `CFCATCH` variables to examine or display diagnostic information.

NOTE
If you modified the DSN to force an error in the earlier template, you should fix it now, since we are demonstrating a new problem, a bad SQL statement.

Listing 51.3 `ChoicePage2b.cfm`—Displaying the SQL Error Code for a Database Error

```
<!---
 Filename: ChoicePage2b.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Provides navigation elements for films and ratings
--->

<html>
<head><title>Film Information</title></head>
<body>
<h2>Film Information</h2>
```

Listing 51.3 (continued)

```
<cftry>
 <!--- Retrieve Ratings from database --->
 <!--- RatingIck is not a real column. --->
 <cfquery name="getRatings" datasource="#APPLICATION.dataSource#">
 SELECT RatingID, Rating
 FROM FilmsRatings
 ORDER BY RatingIck
 </cfquery>
 <!--- Retrieve Films from database --->
 <cfquery name="getFilms" datasource="#APPLICATION.dataSource#">
 SELECT FilmID, MovieTitle
 FROM Films
 ORDER BY Films.MovieTitle
 </cfquery>

 <!--- If any database errors occur during above query, --->
 <cfcatch type="Database">
 <!--- Let user know that the Films data can't be shown right now --->
 <cfoutput>
 <i>Sorry, we are not able to connect to our real-time database right now,
 because of SQL Error Code <b>#CFCATCH.ErrorCode#</b>.
 We're taking care of it, of course. Please try again soon!</i><br>
 </cfoutput>

 <!--- Stop processing at this point --->
 <cfabort>
 </cfcatch>
</cftry>

<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

 <!--- Display Film names in a drop-down list --->
 <P>Films:
 <cfselect query="getFilms" name="filmID"
value="FilmID" display="MovieTitle"/>

 <!--- Display Rating names in a drop-down list --->
 <P>Ratings:
 <cfselect query="getRatings" name="ratingID"
value="RatingID" display="Rating"/>

</cfform>

</body>
</html>
```

NOTE

This template only outputs the value of `CFCATCH.SqlState`. Of course, if you want to give the user more information, you are free to use the other `CFCATCH` variables from Table 51.3 that are relevant to the type of exception.

Writing Templates That Work Around Errors

As shown in Figure 51.4, you can use `<cftry>` and `<cfcatch>` to respond to errors by displaying a customized error message. But that's really not so different from the way you learned to use `<cferror>` and `onError()` in Chapter 18. The real value of `<cftry>` and `<cfcatch>` is the fact that they let you create templates that actively respond to or work around exception conditions. In other words, they let you write Web pages that continue to be at least somewhat helpful to your users, even after an error has occurred.

Working Around a Failed Query

Take a look at Listing 51.4, which is similar to the other listings you have seen so far in this chapter. This version isolates each `<cfquery>` within its own `<cftry>` block. Within each `<cftry>` block, if any database errors occur, the user sees a static message, but the template execution doesn't stop.

Listing 51.4 `ChoicePage3a.cfm`—Using `<cftry>` and `<cfcatch>` to Display Information

```
<!---
 Filename: ChoicePage3a.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Provides navigation elements for films and ratings
--->

<html>
<head><title>Film Information</title></head>
<body>
<h2>Film Information</h2>
<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

 <!--- Attempt database operation --->
 <cftry>
 <!--- Retrieve Films from database --->
 <!--- MovieTitleBad is not a valid column. To fix, change to MovieTitle --->
 <cfquery name="getFilms" datasource="#APPLICATION.DataSource#">
 SELECT FilmID, MovieTitle
 FROM Films
 ORDER BY Films.MovieTitleBad
 </cfquery>

 <!--- If any database errors occur during above query, --->
 <cfcatch type="Database">
 <!--- Let user know that the Films data can't be shown right now --->
 <p><i>Sorry, we can't show a real-time list of Films right now.</i></p>
 </cfcatch>
 </cftry>


 <!--- Attempt database operation --->
 <cftry>
<cfquery name="getRatings" datasource="#APPLICATION.DataSource#">
 SELECT RatingID, Rating
 FROM FilmsRatings
 ORDER BY Rating
```

Listing 51.4 (continued)

```
  </cfquery>

  <!--- If any database errors occur during above query, --->
  <cfcatch type="Database">
  <!--- Let user know that the Ratings data can't be shown right now --->
  <p><i>Sorry, we can't show a real-time list of Ratings right now.</i></p>
  </cfcatch>
  </cftry>


  <!--- If, after all is said and done, we were able to get Film data --->
  <cfif isDefined("VARIABLES.getFilms")>
  <!--- Display Film names in a drop-down list --->
  <p>Films:
  <cfselect query="getFilms" name="filmID" value="FilmID"
  display="MovieTitle"/>
  <cfinput type="submit" name="go" value="Go">
  </cfif>

  <!--- If, after all is said and done, we were able to get Ratings data --->
  <cfif isDefined("VARIABLES.GetRatings")>
  <!--- Display Ratings in a drop-down list --->
  <p>Ratings:
  <cfselect query="getRatings" name="ratingID" value="RatingID"
  display="Rating"/>
  <cfinput type="submit" name="go" value="Go">
  </cfif>

 </cfform>

 </body>
 </html>
```

If for some reason the `getFilms` query fails to run properly, a "Not able to get a real-time list of Films" message is displayed. Template execution then continues normally, right after the first `<cftry>` block. Because the `<cfquery>` couldn't complete its work, the `getFilms` variable doesn't exist; otherwise, everything is fine.

At the bottom of the template, an `isDefined()` check, which ascertains whether the corresponding query actually exists, protects the display of each drop-down list. If the `getFilms` query is completed without error, the first `<cfif>` test passes and the Films drop-down list is displayed. If not, the code inside the `<cfif>` block is skipped.

Since the first query causes an error, the user sees a friendly little status message, as shown in Figure 51.5, instead of the Films drop-down list. The second query can execute successfully, however, and the Ratings drop-down list still displays normally. You can easily modify the code to correct the bad SQL in the first query to see how the template changes. Also feel free to "break" the second query.

The result is a page that remains useful when a problem with the database partially disables it. In a small but very helpful way, it's self-healing.

Listing 51.5 shows a slightly different way to structure the code. Almost all the code lines are the same as Listing 51.4, just ordered differently. Instead of placing both `<cfselect>` tags together at the end of the template, Listing 51.5 outputs each `<cfselect>` in the corresponding `<cftry>` block, right after the query that populates it. If one of the queries fails, the `<cfswitch>` block immediately takes over, thus skipping over `<cfselect>` for the failed query.

The result is a template that behaves in exactly the same way as Listing 51.4. Depending on your preference and on the situation, placing code in self-contained `<cftry>` blocks as shown in Listing 51.5 can make your templates easier to understand and maintain. On the other hand, in many situations, the approach taken in Listing 51.4 is more sensible because it keeps the database access code separate from the HTML generation code. Use whichever approach results in the simplest-looking, most straightforward code.

Figure 51.5 ColdFusion's exception-handling capabilities enable your templates to recover gracefully from error conditions.

Listing 51.5 `ChoicePage3b.cfm`—An Alternative Way of Structuring the Code in Listing 32.4

```
<!---
 Filename: ChoicePage3b.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Provides navigation elements for films and ratings
--->

<html>
<head><title>Film Information</title></head>
<body>
<h2>Film Information</h2>

<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

  <!--- Attempt database operation --->
  <cftry>
  <!--- Retrieve Films from database --->
  <!--- MovieTitle2 is a bad column. Remove the "2" to fix it. --->
  <cfquery name="getFilms" datasource="#APPLICATION.DataSource#">
  SELECT FilmID, MovieTitle2
  FROM Films
```

Listing 51.5 (continued)

```
    ORDER BY Films.MovieTitle
    </cfquery>

    <!--- Display Film names in a drop-down list --->
    <p>Films:
    <cfselect query="getFilms" name="filmID" value="FilmID"
  display="MovieTitle"/>
    <cfinput type="submit" name="go" value="Go">

    <!--- If any database errors occur during above query, --->
    <cfcatch type="Database">
    <!--- Let user know that the Films data can't be shown right now --->
    <p><i>Sorry, we can't show a real-time list of Films right now.</i></p>
    </cfcatch>
    </cftry>


    <!--- Attempt database operation --->
    <cftry>
    <!--- Retrieve Ratings from database --->
    <cfquery name="getRatings" datasource="#APPLICATION.DataSource#">
    SELECT RatingID, Rating
    FROM FilmsRatings
    ORDER BY Rating
    </cfquery>

    <!--- Display Rating names in a drop-down list --->
    <p>Ratings:
    <cfselect query="getRatings" name="ratingID" value="RatingID"
   display="Rating"/>
    <cfinput type="submit" name="go" value="Go">

    <!--- If any database errors occur during above query, --->
    <cfcatch type="Database">
    <!--- Let user know that the Ratings data can't be shown right now --->
    <p><i>Sorry, we can't show a real-time list of Ratings right now.</i></p>
    </cfcatch>
    </cftry>

  </cfform>

  </body>
  </html>
```

Working Around Errors Silently

Listing 51.6 is another version of Listing 51.4. This time, no action of any type is taken in the `<cfcatch>` block (not even the display of an error message). Because nothing needs to be placed between the `<cfcatch>` and `</cfcatch>` tags, they can be coded using the shorthand notation of `<cfcatch/>` alone. The trailing slash is just an abbreviated way of providing an empty tag pair.

TIP

You can use this trailing-slash notation whenever you're using a CFML tag that requires a closing tag, but you don't actually need to place anything between the tags. Most of the templates in this chapter use this shorthand in their `<cfselect>` tags.

The result is a template that recovers silently from database errors, without even acknowledging that anything has gone wrong. If the `getFilms` query can't execute, the Films drop-down list just doesn't appear. As far as the user is concerned, no error has occurred. The template simply displays what it can and gracefully omits what it can't.

Listing 51.6 `ChoicePage3c.cfm`—Handling Errors Silently

```
<!---
 Filename: ChoicePage3c.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Provides navigation elements for films and ratings
--->

<html>
<head><title>Film Information</title></head>
<body>
<h2>Film Information</h2>

<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

  <!--- Attempt database operation --->
  <cftry>
  <!--- Retrieve Films from database --->
  <!--- MovieTitle2 is a bad column. Remove the "2" to fix it. --->
  <cfquery name="getFilms" datasource="#APPLICATION.dataSource#">
  SELECT FilmID, MovieTitle2
  FROM Films
  ORDER BY Films.MovieTitle
  </cfquery>
  <!--- Display Film names in a drop-down list --->
  <p>Films:
  <cfselect query="getFilms" name="filmID" value="FilmID"
display="MovieTitle"/>
  <cfinput type="submit" name="go" value="Go">

  <!--- Silently catch any database errors from above query --->
  <cfcatch type="Database"/>
  </cftry>

  <!--- Attempt database operation --->
  <cftry>
  <!--- Retrieve Ratings from database --->
  <cfquery name="getRatings" datasource="#APPLICATION.dataSource#">
  SELECT RatingID, Rating
  FROM FilmsRatings
  ORDER BY Rating
  </cfquery>

  <!--- Display Ratings in a drop-down list --->
  <P>Ratings:
  <cfselect query="getRatings" name="ratingID" value="RatingID"
display="Rating"/>
  <cfinput type="submit" name="go" value="Go">

  <!--- Silently catch any database errors from above query --->
```

Listing 51.6 (continued)

```
  <cfcatch type="Database"/>
  </cftry>

</cfform>

</body>
</html>
```

Writing Templates That Recover from Errors

Listings 51.4 to 51.6 created templates that serve your users by skipping blocks of code that rely on failed queries. Depending on the situation, you can often go a step further, creating templates that continue to provide the basic functionality they are supposed to (albeit in some type of scaled-back manner), even when a problem occurs.

For instance, Listing 51.7 creates yet another version of the drop-down page example. It looks similar to Listing 51.4, except for the first `<cfcatch>` block, which has been expanded. The idea here is to use a backup copy of the Films database table, which exists for the sole purpose of providing a fallback when the primary database system can't be reached.

Listing 51.7 `ChoicePage4.cfm`—Querying a Backup Text File When the Primary Database Is Unavailable

```
<!---
 Filename: ChoicePage4.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Provides navigation elements for films and ratings
--->
<html>
<head><title>Films</title></head>
<body>
<h2>Film Information</h2>

<cftry>
  <!--- Retrieve Films from live database --->
  <cfquery name="getFilms" datasource="#APPLICATION.dataSource#">
  SELECT FilmID, MovieTitles
  FROM Films
  ORDER BY Films.MovieTitle
  </cfquery>

  <!--- If any database errors occur during above query, --->
  <cfcatch type="Database">

  <!--- Location of our emergency backup file --->
  <cfset backupFilePath = expandPath("FilmsBackup.xml")>

  <!--- Read contents of the WDDX/XML in from backup file --->
  <cffile action="read" file="#backupFilePath#" variable="wddxPacket">

  <!--- Convert the XML back into original query object --->
  <cfwddx action="WDDX2CFML" input="#wddxPacket#" output="getFilms">

  <!--- Let user know that emergency version is being used --->
```

Listing 51.7 (continued)

```
      <p><i>NOTE:
      We are not able to connect to our real-time Films database at the moment.
      Instead, we are using data from our archives to display the Films list.
      Please try again later today for an up to date listing.</i></p>
      </cfcatch>
    </cftry>

    <!--- Attempt database operation --->
    <cftry>
      <!--- Retrieve Ratings from database --->
      <cfquery name="getRatings" datasource="#APPLICATION.dataSource#">
      SELECT RatingID, Rating
      FROM FilmsRatings
      ORDER BY Rating
      </cfquery>

      <!--- Silently catch any database errors from above query --->
      <cfcatch type="Database"/>
    </cftry>


    <!--- Create self-submitting form --->
    <cfform action="#CGI.script_name#" method="post">

      <!--- Display Film names in a drop-down list --->
      <p>Films:
      <cfselect query="getFilms" name="filmID" value="FilmID"
     display="MovieTitle"/>
      <cfinput type="submit" name="go" value="Go">
      <!--- If, after all is said and done, we were able to get Ratings data --->
      <cfif isDefined("getRatings")>
        <!--- Display Ratings in a drop-down list --->
        <p>Ratings:
        <cfselect query="getRatings" name="ratingID"
        value="RatingID" display="Rating"/>
        <cfinput type="submit" name="go" value="Go">
      </cfif>

    </cfform>

    </body>
    </html>
```

If the `<cfquery>` named `GetFilms` fails for whatever reason, the code in the `<cfcatch>` block will execute in an attempt to save the day. The idea behind this code is to read information about films from a previously saved XML file called `FilmsBackup.xml`. The backup file is in WDDX format, which is a special type of XML used to convert any type of data (such as a query record set) quickly and easily into a simple string format that can be saved to disk.

NOTE

We don't have space to explain WDDX in detail here, but if you're curious, open the `FilmsBackup.xml` file in a text editor and take a look at the format. You'll find it fairly self-explanatory.

NOTE

This process (converting a variable in the server's RAM memory into a string) is called serialization. The reverse process (parsing the XML-formatted string back into the original variable in the server's memory) is called deserialization. For more information and examples regarding XML, WDDX, serialization, and deserialization, see the online chapter 47, "Using WDDX," online. The ColdFusion documentation also contains a few simple examples.

First, the `<cffile>` tag is used to read the contents of the `FilmsBackup.xml` file into a string variable called `wddxPacket`. At this point, the variable holds the XML code that represents the backup film data. Now the `<cfwddx>` tag is used to convert the XML code into a normal query object called `getFilms`. You can use this query object just like the results of a normal `<cfquery>` tag. In other words, the rest of this template can continue working as if the `<cfquery>` at the top of the page hadn't caused an error.

The result is a version of the template that still provides the basic functionality it's supposed to, even when the first `getFilms` query fails. Assuming that the backup file can be processed correctly, the user is presented with the expected drop-down list of films. The user also sees a message that alerts him or her to the fact that the list is populated not from the live database, but rather from an archived backup copy, as shown in Figure 51.6.

Figure 51.6

This version of the template uses a backup text file if the usual database isn't accessible.

Nesting `<cftry>` Blocks

You can nest `<cftry>` blocks within one another. There are generally two ways you can nest the blocks, depending on the behavior you want.

If you nest a `<cftry>` within another `<cftry>` block (but not within a `<cfcatch>`), this doubly protects the code inside the inner `<cftry>`. This type of nesting typically follows this basic form:

```
<cftry>
 <cftry>
 <!--- Important code goes here --->

 <cfcatch></cfcatch>
 </cftry>
 <cfcatch></cfcatch>
</cftry>
```

If something goes wrong, ColdFusion will see whether any of the <cfcatch> tags within the inner <cftry> are appropriate (that is, whether a <cfcatch> of the same type as the exception itself exists). If so, the code in that block executes. If not, ColdFusion looks to see whether the <cfcatch> tags within the outer <cftry> block are appropriate. If it doesn't find any appropriate <cfcatch> blocks there either, it considers the exception uncaught, so the default error message displays.

Alternatively, you can essentially create a two-step process by nesting a <cftry> within a <cfcatch> that belongs to another <cftry>. This second type of nesting is typically structured like this:

```
<cftry>
 <!--- important code here --->
 <cfcatch>
 <cftry>
 <!--- fallback code here --->
 <cfcatch>
 <!--- last chance code here --->
 </cfcatch>
 </cftry>
 </cfcatch>
</cftry>
```

If the code in the outer <cftry> fails, the inner <cftry> attempts to deal with the situation in some other way. If the code in the inner <cftry> also fails, its <cfcatch> tags can catch the error and perform some type of last-ditch processing (such as displaying an error message).

Listing 51.8 is an example that includes both of these forms of <cftry> nesting. It builds upon Listing 51.7 by adding two <cftry> blocks within the larger <cftry> block that begins near the top of the template.

Listing 51.8 ChoicePage5.cfm—Nesting <cftry> Blocks Within Each Other

```
<!---
 Filename: ChoicePage5.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Provides navigation elements for films and ratings
--->

<html>
<head><title>Films</title></head>
<body>
<h2>Film Information</h2>

<cftry>
  <!--- Location of our emergency backup file --->
  <cfset backupFilePath = expandPath("FilmsBackup.xml")>

  <!--- Retrieve Films from live database --->
  <!--- MovieTitle2 is an invalid column name --->
  <cfquery name="getFilms" datasource="#APPLICATION.dataSource#">
  SELECT FilmID, MovieTitle2
  FROM Films
  ORDER BY Films.MovieTitle
  </cfquery>
```

Listing 51.8 (continued)

```
      <!--- If the backup file has never been created --->
      <cfif not fileExists(backupFilePath)>
        <!--- Now we'll make sure that our backup file exists --->
        <!--- If it doesn't exist yet, we'll try to create it --->
        <cftry>

        <!--- Convert the query to WDDX (an XML vocabulary) --->
        <cfwddx action="CFML2WDDX" input="#getFilms#" output="wddxPacket">

        <!--- Save the XML on server's drive, as our backup file --->
        <cffile action="write" file="#backupFilePath#" output="#wddxPacket#">

        <!--- Silently ignore any errors while creating backup file --->
        <!--- (the worst that happens is the backup file doesn't get made) --->
        <cfcatch type="any"/>
        </cftry>
      </cfif>

      <!--- If any database errors occur during above query, --->
      <cfcatch type="Database">

        <cftry>
        <!--- Read contents of the WDDX/XML in from backup file --->
        <cffile action="read" file="#backupFilePath#" variable="wddxPacket">
        <!--- Convert the XML back into original query object --->
        <cfwddx action="WDDX2CFML" input="#wddxPacket#" output="getFilms">

        <!--- Let user know that emergency version is being used --->
        <p><i>NOTE:
        We are not able to connect to our real-time Films database at the moment.
        Instead, we are using data from our archives to display the Films list.
        Please try again later today for an up to date listing.</i></p>

        <!--- If any problems occur while trying to use the backup file --->
        <cfcatch type="Any">
        <!--- Let user know that the Films data can't be shown right now --->
        <i>Sorry, we are not able to provide you with a list of films.</i><br>
        </cfcatch>
        </cftry>

      </cfcatch>
    </cftry>

    <!--- Attempt database operation --->
    <cftry>
      <!--- Retrieve Ratings from database --->
      <cfquery name="getRatings" datasource="#APPLICATION.DataSource#">
      SELECT RatingID, Rating
      FROM FilmsRatings
      ORDER BY Rating
      </cfquery>

      <!--- Silently catch any database errors from above query --->
      <cfcatch type="Database"/>
```

Listing 51.8 (continued)

```
</cftry>


<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

 <!--- If, after all is said and done, we were able to get Film data --->
 <cfif isDefined("getFilms")>
   <!--- Display Film names in a drop-down list --->
   <p>Films:
   <cfselect query="getFilms" name="filmID" value="FilmID" display="MovieTitle"/>
   <cfinput type="submit" name="go" value="Go">
 </cfif>

  <!--- If, after all is said and done, we were able to get Ratings data --->
  <cfif isDefined("getRatings")>
    <!--- Display Ratings in a drop-down list --->
    <p>Ratings:
    <cfselect query="getRatings" name="ratingID"
    value="RatingID" display="Rating"/>
   <cfinput type="submit" name="go" value="Go">
  </cfif>

</cfform>

</body>
</html>
```

The first of the nested `<cftry>` blocks attempts to create the `FilmsBackup.xml` file if it hasn't been created already, or if it has been deleted for some reason. A new version of the backup data is serialized using `<cfwddx>`, then written to the server's drive using `<cffile>`. If any errors occur during this file-creation process, they will silently be ignored (no error message is shown). In other words, if the backup file doesn't exist, this nested `<cftry>` will attempt to create it. But if `<cftry>` can't do so, it just moves on.

NOTE

Since the first nested `<cftry>` sits within an outer `<cftry>` that catches all exceptions of `TYPE="Database"`, the nested `<cftry>` will be skipped altogether if `<cfquery>` causes a database error.

NOTE

If you want to test this code, delete the `FilmsBackup.xml` file from the directory you're using for this chapter's examples, then visit this template in your browser. Provided the query itself doesn't have a problem, the backup file should be re-created.

The second nested `<cftry>` is within the outer `<cftry>` tag's `<cfcatch>` block. This is the portion of the code that attempts to read the backup file if the `<cfquery>` fails (as introduced in Listing 51.7). Remember, because of where it's now nested, this inner `<cftry>` block will only be encountered if there is a problem retrieving the live film data from the database. This inner `<cftry>` tells ColdFusion that if the database query fails and there is a problem with the emergency `<cffile>` and `<cfwddx>` code, then the template should just give up and display a "Sorry, we are not able to provide you with a list of films" message.

In other words, this version of the template has double protection: It has a fallback plan if the database query fails, and it knows how to degrade gracefully even if the fallback plan fails. This is quite an improvement over the original version of the template (refer to Listing 51.1), which could do no better than display a user-unfriendly error message when something went wrong (refer to Figure 51.2).

Deciding Not to Handle an Exception

When your code catches an exception with `<cfcatch>`, it assumes all responsibility for dealing with the exception in an appropriate way. After your code catches the exception, ColdFusion is no longer responsible for logging an error message or halting further page processing.

For instance, look back at Listing 51.8 (`ChoicePage5.cfm`). The `<cfcatch>` tags in this template declare themselves fit to handle any and all database-related errors by specifying a `type="Database"` attribute in the `<cfcatch>` tag. No matter what type of database-related exception gets raised, those `<cfcatch>` tags will catch the error.

The purpose of the first `<cfcatch>` block in Listing 51.8 is to attempt to use the backup version of the database when the first `<cfquery>` fails. This backup plan is activated no matter what the actual problem is. Even a simple syntax error or misspelled column name in the SQL statement will cause the `FilmsBackup.xml` file to be used.

It would be a better policy for the backup version of the database to be used only when the original query failed due to a connection problem. Other types of errors, such as syntax errors, should probably not be caught and dealt with in the same way.

ColdFusion provides the `<cfrethrow>` tag for exactly this type of situation. This section explains how to use `<cfrethrow>` and when to use it.

Exceptions and the Notion of Bubbling Up

Like the error or exception constructs in many other programming languages, ColdFusion's exceptions can do something called *bubbling up*.

Let's say you have some code that includes several layers of nested `<cftry>` blocks and that an exception has taken place in the innermost block. If the exception isn't caught in the innermost `<cftry>` block, ColdFusion looks in the containing `<cftry>` block, if any, to see whether it contains any appropriate `<cfcatch>` tags. If not, the exception continues to "bubble up" through the layers of `<cftry>` blocks until it's caught. If the error bubbles up through all the layers of the `<cftry>` blocks (that is, if none of the `<cfcatch>` tags in the `<cftry>` blocks choose to catch the exception), ColdFusion then displays its default error message.

Using `<cfrethrow>`

The `<cfrethrow>` tag is basically the opposite of `<cfcatch>`. After `<cfcatch>` catches an error, `<cfrethrow>` can uncatch the error. It's then free to bubble up to the next containing `<cftry>` block, if any.

The <cfrethrow> tag takes no attributes and can be used only inside a <cfcatch> tag. Typically, you will decide to use it by testing the value of one of the special CFCATCH variables shown previously in Table 51.3.

For instance, with Apache Derby, if the path to the file defined in the datasource is incorrect, the CFCATCH.SQLState variable will be set to XJ004. If you want your code to handle only errors of this specific type, letting all other errors bubble up normally, you would use code similar to the following within a <cfcatch> block:

```
<cfif CFCATCH.SQLState neq "XJ004">
 <cfrethrow>
</cfif>
```

Listing 51.9, a revision of Listing 52.8, uses the basic <cfif> test shown in the previous snippet to ensure that the backup data file (FilmsBackup.xml) is used only if the database exception has a SQLState value of XJ004.

Listing 51.9 ChoicePage6.cfm—Using <cfrethrow> to Process Only Database Errors of Code XJ004

```
<!---
 Filename: ChoicePage6.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Provides navigation elements for films and ratings
--->

<html>
<head><title>Films</title></head>
<body>
<h2>Film Information</h2>

<cftry>
  <!--- Location of our emergency backup file --->
  <cfset backupFilePath = expandPath("FilmsBackup.xml")>

  <!--- Retrieve Films from live database --->
  <cfquery name="getFilms" datasource="#APPLICATION.dataSource#">
  SELECT FilmID, MovieTitle
  FROM Films
  ORDER BY Films.MovieTitle
  </cfquery>

  <!--- Now we'll make sure that our backup file exists --->
  <!--- If it doesn't exist yet, we'll try to create it --->
  <!--- Also, re-create it whenever server is restarted --->
  <cftry>
  <!--- If the server has just been restarted, or --->
  <!--- if the backup file has never been created --->
  <cfif not isDefined("APPLICATION.getFilmsBackupCreated")
  or not fileExists(backupFilePath)>

    <!--- Convert the query to WDDX (an XML vocabulary) --->
    <cfwddx action="CFML2WDDX" input="#getFilms#" output="wddxPacket">

    <!--- Save the XML on server's drive, as our backup file --->
```

Listing 51.9 (continued)

```
    <cffile action="write" file="#backupFilePath#" output="#wddxPacket#">

    <!--- Remember that we just created the emergency file --->
    <!--- This will be forgotten when server is restarted; thus, backup --->
    <!--- file will be refreshed on first successful query after restart --->
    <cfset APPLICATION.getFilmsBackupCreated = True>
  </cfif>

  <!--- Silently ignore any errors while creating backup file --->
  <!--- (the worst that happens is the backup file doesn't get made) --->
  <cfcatch TYPE="Any"/>
  </cftry>

  <!--- If any database errors occur during above query, --->
  <cfcatch type="Database">

    <!--- Unless this is SQL Error XJ004, un-catch the exception --->
    <cfif CFCATCH.SQLState neq "XJ004">
      <cfrethrow>
    <!--- If it's SQL Error, XJ004, attempt to get data from txt file --->
    <cfelse>
      <cftry>

      <!--- Read contents of the WDDX/XML in from backup file --->
      <cffile action="read" file="#backupFilePath#" variable="wddxPacket">

      <!--- Convert the XML back into original query object --->
      <cfwddx action="WDDX2CFML" input="#wddxPacket#" output="getFilms">

       <!--- Let user know that emergency version is being used --->
       <p><i>NOTE:
       We are not able to connect to our real-time Films database at the
       moment. Instead, we are using data from our archives to display
       the Films list. Please try again later today for an up to date
       listing.</i></p>

      <!--- If any problems occur while trying to use the backup file --->
      <cfcatch type="Any">
        <!--- Let user know that the Films data can't be shown right now --->
        <i>Sorry, we are not able to provide you with a list of films.</i><br>
      </cfcatch>
      </cftry>

    </cfif>
  </cfcatch>
</cftry>

<!--- Attempt database operation --->
<cftry>

  <!--- Retrieve Ratings from database --->
  <cfquery name="getRatings" datasource="#APPLICATION.dataSource#">
  SELECT RatingID, Rating
  FROM FilmsRatings
  ORDER BY Rating
```

Listing 51.9 (continued)

```
  </cfquery>

  <!--- Silently catch any database errors from above query --->
  <cfcatch type="Database"/>
</cftry>


<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

  <!--- If, after all is said and done, we were able to get Film data --->
  <cfif isDefined("getFilms")>
    <!--- Display Film names in a drop-down list --->
    <p>Films:
    <cfselect query="getFilms" name="filmID" value="FilmID" display="MovieTitle"/>
    <cfinput type="submit" name="go" value="Go">
  </cfif>

  <!--- If, after all is said and done, we were able to get Ratings data --->
  <cfif isDefined("getRatings")>
    <!--- Display Ratings in a drop-down list --->
    <p>Ratings:
    <cfselect query="getRatings" name="ratingID"
    value="RatingID" display="Rating"/>
    <cfinput type="submit" name="go" value="Go">
  </cfif>

</cfform>

</body>
</html>
```

To test the exception-handling behavior of this template, try editing the DSN and editing the database folder. Now visit Listing 51.9 with your Web browser. That should cause error `XJ004`, so the backup version of the `Films` table will be used (refer to Figure 51.6). Now database folder and sabotage the first `<cfquery>` in some other way—for instance, by changing one of the column names to something invalid. That should cause the `<cfrethrow>` tag to fire, which in turn causes the error to be raised again. Because there are no other `<cfcatch>` tags to handle the reraised exception, ColdFusion displays its usual error message (which you could customize with `<cferror>`, as discussed in Chapter 19).

You are free to have as many `<cfif>` tests as you need to make the decision whether to rethrow the error. For instance, if you wanted your `<cfcatch>` code to handle only errors of types `XJ004`, `S1005`, and `S1010` instead of only `XJ004`, you could replace the `<CFIF>` test in Listing 32.19 with something like the following:

```
<cfif listFindNoCase("XJ004,S1005,S1010", CFCATCH.SQLState) EQ 0>
 <cfrethrow>
</cfif>
```

NOTE

Listing 51.9 also contains a simple check for the presence of an application variable called `APPLICATION.GetFilmsBack-upCreated`. If this variable doesn't exist, it creates a new version of the backup file, as if the backup file didn't exist. It then sets the `APPLICATION.GetFilmsBackupCreated` variable to True. In other words, this version of the template attempts to create a fresh version of the backup template whenever the server is restarted. The idea is to keep the backup file somewhat current (not months or years old), without negatively affecting performance. If you wanted, you could replace this simple test with code that refreshed the age of the backup file once per day, or according to other similar logic.

Examining Exception Messages to Improve Usability

In Chapter 21, "Interacting with Email," online, you learned how to create a simple Web page that people can use to check their email. In that chapter, the ColdFusion template that does most of the work is called `CheckMail.cfm`. When users visit this page for the first time, they are prompted for the host name of their POP mail server, plus their username and password.

One of the usability problems with that example is that it will display an ugly and potentially confusing error message if the user provides an incorrect mail server address, or accidentally types the username or password incorrectly. It would be a lot more helpful if the application would just let the user know what the problem is, encouraging them to check their input and try again.

Listing 51.10 uses a `<cftry>` block to do just that. The `<cfpop>` tag, which is what will throw an exception if the email can't be retrieved (for whatever reason), is surrounded by `<cftry>` tags. If the `<cfpop>` operation throws an exception, it's caught by the large `<cfcatch>` block underneath. Within the `<cfcatch>` block, the text of the `CFCATCH.Detail` variable is examined to attempt to determine what exactly went wrong.

The result is a template that simply asks users to verify their login credentials if their messages can't be retrieved, as shown in Figure 51.7.

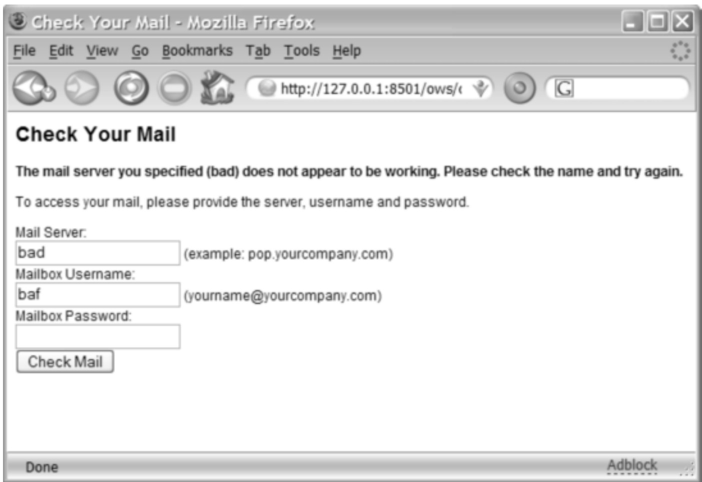


Figure 51.7

If a POP server can't be reached, the exception can be caught so that the user will see a sensible message.

Listing 51.10 `CheckMail.cfm`—Catching Advanced Exceptions Raised

```
<!---
 Filename: CheckMail.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Uses exception handling while retrieving email with <CFPOP>
--->

<!--- Include UDF function library, which includes the --->
<!--- TryToParsePOPDateTime() function used near the end of this page --->
<cfinclude template="POPDateFunctions.cfm">

<html>
<head><title>Check Your Mail</title></head>
<body>

<!--- Simple CSS-based formatting styles --->
<style>
 BODY { font-family:sans-serif;font-size:12px}
 TH { font-size:12px;background:navy;color:white}
 TD { font-size:12px;background:lightgrey;color:navy}
</style>
<h2>Check Your Mail</h2>

<!--- If user is logging out, --->
<!--- or if user is submitting a different username/password --->
<cfif isDefined("URL.logout") or isDefined("FORM.popServer")>
  <cfset structDelete(SESSION, "mail")>
</cfif>

<!--- If we don't have a username/password --->
<cfif not isDefined("SESSION.mail")>
  <!--- Show "mail server login" form --->
  <cfinclude template="CheckMailLogin.cfm">
</cfif>


<!--- If we need to contact server for list of messages --->
<!--- (if just logged in, or if clicked "Refresh" link) --->
<cfif not isDefined("SESSION.mail.getMessages") or isDefined("URL.refresh")>

  <!--- Attempt to contact mail server --->
  <!--- If attempt is unsuccessful, try to catch the most common errors --->
  <cftry>

  <!--- Contact POP Server and retrieve messages --->
  <cfpop action="GetHeaderOnly" name="SESSION.mail.getMessages"
  server="#SESSION.mail.popServer#" username="#SESSION.mail.username#"
  password="#SESSION.mail.password#" maxrows="50">

  <!--- If an error occurs... --->
  <cfcatch type="APPLICATION">
    <!--- If the problem seems to be related to javax.mail, then --->
    <!--- examine the message further and try to deal with the problem --->
    <cfif CFCATCH.detail contains "javax.mail">

      <!--- If the username/password were rejected by mail server --->
```

Listing 51.10 (continued)

```
        <cfif CFCATCH.detail contains "javax.mail.AuthenticationFailedException">
          <!--- Explain what happened --->
          <cfoutput>
          <p><b>The username and password you provided were not accepted by
          your mail server. Keep in mind that the username and password may be
          case-sensitive, so make sure you are getting the capitalization
          exactly right.</b></p>
          </cfoutput>

        <!--- If the mail server cannot be found or can't be connected to --->
        <cfelseif CFCATCH.Detail contains "java.net.UnknownHostException">
          <!--- Explain what happened --->
          <cfoutput>
          <p><b>The mail server you specified (#SESSION.mail.popServer#) does
          not appear to be working. Please check the name and try again.</b></p>
          </cfoutput>

        <!--- If some error occurs... --->
        <cfelse>
          <!--- Display a generic error message --->
          <cfoutput>
          <p><b>We're sorry, we weren't able to retrieve your messages at
          this time. The problem may be with your mail server,
          or some other network issue.</b></p>
          </cfoutput>

          <!--- Create a message to store in the log --->
          <cfset logMsg = "Couldn't get mail from #SESSION.mail.popServer#."
          & "Message: #CFCATCH.Message# Detail: #CFCATCH.Detail#">

          <!--- Log the problem. --->
          <!--- Appends to CheckMailErrorLog.log in CFusion/Logs folder --->
          <cflog file="CheckMailErrorLog" text="#logMsg#">

        </cfif>

        <!--- Discard login information from SESSION scope --->
        <!--- This will force user to re-provide credentials --->
        <cfset structDelete(SESSION, "mail")>

        <!--- Show "mail server login" form again --->
        <cfset structDelete(FORM, "popServer")>
        <cfinclude template="CheckMailLogin.cfm">

        <!--- Stop processing at this point --->
        <cfabort>

      <!--- If the problem does not seem to be related to javax.mail, --->
      <!--- then just rethrow the error and let it expose itself normally. --->
      <cfelse>
        <cfrethrow>
      </cfif>
   </cfcatch>

   </cftry>
```

Listing 51.10 (continued)

```
</cfif>


<!--- If no messages were retrieved... --->
<cfif SESSION.mail.getMessages.recordCount eq 0>
  <p>You have no mail messages at this time.<br>

<!--- If messages were retrieved... --->
<cfelse>
  <!--- Display Messages in HTML Table Format --->
  <table border="0" cellPadding="2" cellspacing="2" cols="3" width="550">
  <!--- Column Headings for Table --->
  <tr>
  <th width="100">Date Sent</th>
  <th width="200">From</th>
  <th width="200">Subject</th>
  </tr>

  <!--- Display info about each message in a table row --->
  <cfoutput query="SESSION.mail.getMessages">
    <!--- Let user click on Subject to read full message --->
    <cfset linkURL = "CheckMailMsg2.cfm?msgNum=#MessageNumber#">

    <tr valign="baseline">
    <!--- Show parsed and formatted Date and Time for message--->
    <!--- If it can't be parsed, it will just be displayed as is --->
    <td>
    #tryToFormatPOPDateTime(SESSION.mail.getMessages.Date)#
    </td>
    <!--- Show "From" address, escaping brackets --->
    <td>#htmlEditFormat(From)#</td>
    <td><strong><a href="#linkURL#">#Subject#</a></strong></td>
    </tr>
  </cfoutput>
  </table>

  <!--- "Refresh" link to get new list of messages --->
  <b><a href="CheckMail.cfm?refresh=Yes">Refresh Message List</a></b><br>
  <!--- "Log Out" link to discard SESSION.Mail info --->
  <A HREF="CheckMail.cfm?Logout=Yes">Log Out</A><BR>
</cfif>

</body>
</html>
```

NOTE

Most of the code in Listing 51.10 is unchanged from the version in Chapter 21, so only the `<cftry>` block near the top of the listing will be discussed here. For a discussion of the rest of the code, especially the use of the `<cfpop>` tag, please refer to Chapter 21.

The `<cftry>` and `<cfcatch>` blocks in this listing rely on the fact that ColdFusion will throw predictable error messages when a `<cfpop>` operation fails. Internally, `<cfpop>` uses the `javax.mail` package provided by Java; when a problem occurs, the Java runtime engine throws an exception,

which is caught by ColdFusion internally. In a `<cfcatch>` block, the text of the Java exception will be available in the `CFCATCH.Message` and `CFCATCH.Detail` variables (refer to Table 51.3).

NOTE

Don't worry about the fact that the `javax.mail` package is used internally by ColdFusion. You don't need to know how to use `javax.mail`, or any other Java package, to use ColdFusion. I only mention it here to help you understand the origin of the error messages.

To create this listing, the first thing I did was to find out what error messages would be displayed if I entered an incorrect username or password, or entered an incorrect mail server address. I simply used the original version of the `CheckMail.cfm` page from Chapter 21 and purposefully entered incorrect values in the login form. I found that if I supplied an incorrect mail server address (for instance, `fake.mailserver.net`), I would receive an error message that read:

```
This exception was caused by: javax.mail.MessagingException:
Connect failed; nested exception is: java.net.UnknownHostException:
fake.mailserver.net.
```

If I entered a correct mail server but an incorrect username or password, I would receive an error message that read:

```
This exception was caused by: javax.mail.AuthenticationFailedException: password
rejected
```

Both of the error messages contain the string `javax.mail`, so the first thing the `<cfcatch>` block in Listing 51.10 does is to check to see if the `CFCATCH.Detail` variable includes `javax.mail`. If it does, it checks to see if it contains `javax.mail.AuthenticationFailedException`; if so, it displays an error message to check the username and password. If not, it checks to see if the message detail contains `java.net.UnknownHostException`; if so, it displays an error message to check the mail server name.

If neither of these two messages were found, the `<cfelse>` blocks executes, the assumption being that some other minor mail-related exception has occurred. A generic error message is displayed, and the problem is logged with the `<cflog>` tag. Any exceptions that contain the string `javax.mail` will be logged in a separate log file called `CheckMailErrorLog.log` in the `ColdFusion8/logs` folder. This will help you debug other types of mail server-related problems.

No matter which of these three `javax.mail`-related messages is displayed, the code then proceeds to deletes the `mail` structure from the `SESSION` scope, which effectively logs the user out of the Web-based mail application. The `<cfinclude>` tag is then used to show the login form again so that the user can try logging in again. Finally, the `<cfabort>` is used to halt any further processing.

It's worth noting that if the `CFCATCH.Detail` variable doesn't include the string `javax.mail`, it's assumed that some other more serious or unforeseen error has occurred, possibly not related to mail at all. In such a case, the `<cfrethrow>` tag is used to allowed it to resurface on its own.

Using Exception Handling In UDFs

In Chapter 24, "Building User-Defined Functions," in Vol. 2, *Application Development*, you learned how to create your own user-defined functions (UDFs) using ColdFusion's `<cffunction>` tag.

Because you write UDFs using ordinary CFML code, you are free to use `<cftry>` and `<cfcatch>` to catch exceptions that might occur while a UDF does its work.

Creating the UDFs

Listing 51.11 shows how you can use exception handling within a UDF. This listing creates a user-defined function called `TryToParsePOPDateTime()`. The idea behind this function is to provide a sensible remedy to the fact that ColdFusion's own `ParseDateTime()` function, which is meant to be able to convert the date strings included in mail messages to CFML date objects, will sometimes fail to parse the date correctly. Basically, the problem is that different systems format the date differently. Some of the date formats can be parsed by `ParseDateTime()` if the `"POP"` option is used; some can't be parsed by that function but can be parsed by `LSParseDateTime()`, and others can be parsed by `ParseDateTime()` without the `"POP"` option. The function created in this listing accepts a date (which is presumably from the `Date` column of a mail message retrieved with `<CFPOP>`) as a string and tries to convert the date using each of the three strategies.

Listing 51.11 `POPDateFunctions.cfm`—Using Exception Handling Within a UDF

```
<!---
 Filename: POPDateFunctions.cfm
 Author: Nate Weiss (NMW)
 Purpose: A small library of functions related to POP dates
--->

<!---
 TryToParsePOPDateTime(date_string)
 Function that tries to parse a date string from an incoming email message.
 It attempts to parse the string using ParseDateTime() with "POP" option,
 then LSParseDateTime(), then ParseDateTime() without "POP" option. If all
 attempts fail, this function returns the original date string, unmodified.
--->
<cffunction name="tryToParsePOPDateTime" output="false" returnType="date">
 <!--- Required argument: DateString --->
 <cfargument name="dateString" type="string" required="Yes">

  <!--- Local variable to hold this function's result --->
  <!--- Start off by just setting it to original date string --->
  <cfset var result = ARGUMENTS.dateString>

  <!--- Attempt 1 --->
  <cftry>
  <cfset result = parseDateTime(ARGUMENTS.dateString, "POP")>

  <!--- If attempt 1 fails... --->
  <cfcatch type="Expression">

    <!--- Attempt 2 --->
    <cftry>
    <cfset result = lsParseDateTime(ARGUMENTS.dateString)>

    <!--- If attempts 2 fails... --->
    <cfcatch type="Expression">
      <!--- Attempt 3 --->
```

Listing 51.11 (continued)

```
          <cftry>
          <cfset result = parseDateTime(ARGUMENTS.dateString)>

          <!--- If attempt 3 fails, give up. --->
          <!--- Result will just be the original date string, --->
          <!--- from the initial <CFSET> at top of this function. --->
          <cfcatch type="Expression" />
          </cftry>

        </cfcatch>
        </cftry>

      </cfcatch>
      </cftry>

      <!--- Return the result --->
      <cfreturn result>
    </cffunction>


    <!---
     TryToFormatPOPDateTime(date_string)
     Attempts to parse a date, then formats it if possible
    --->
    <cffunction name="tryToFormatPOPDateTime" output="false" returnType="string">
      <!--- Required argument: DateString --->
      <cfargument name="dateString" type="string" required="Yes">

      <!--- Attempt to parse the date --->
      <cfset var result = tryToParsePOPDateTime(ARGUMENTS.dateString)>

      <!--- If it was parsed successfully, format it with DateFormat() --->
      <cfif isDate(result)>
        <cfset result = "<b>#dateFormat(result)#</b><br>#timeFormat(result)#">
      </cfif>

      <!--- Return the result --->
      <cfreturn result>
    </cffunction>
```

As you can see, this listing isn't much more than three separate attempts to set the `result` variable with the value returned by the `parseDateTime()` and `lsParseDateTime()` functions. Each attempt is protected by its own `<cftry>` block, so that if the first attempt fails, the second attempt executes, and so on. Even if all three attempts fail, the function still does something reasonably useful: It returns the original date string.

NOTE

As a learning exercise, you could adapt the innermost `<cfcatch>` block so that it parsed the date on its own, using string functions like `getToken()`, `len()`, and the like.

This listing also includes a function called `tryToFormatPOPDateTime()`, which calls the new `try-ToParsePOPDateTime()` function internally, then formats it using `dateFormat()` and `timeFormat()` if

possible. If tryToParsePOPDateTime() wasn't able to return a parsed date object, tryToFormatPOP-DateTime() returns whatever string was passed to it. In other words, you can pass any string to this second function. If it can be parsed as a string, a formatted version will be returned; if not, the original string is returned.

Using the UDFs

If you look back at Listing 51.11, you will see that it's already using these functions. The code from Listing 51.12 is included as a UDF library with the <cfinclude> tag at the top of the listing. Then, near the end of the listing, the try ToFormatPOPDateTime() function is used to display each message's date in the most sensible way possible. Any dates that can't be parsed by the UDFs are simply displayed to the user exactly as they were included in the original mail message.

Throwing and Catching Your Own Errors

You can throw custom exceptions whenever your code encounters a situation that should be considered an error within the context of your application, perhaps because it violates some type of business rule. So custom exceptions give you a way to teach your ColdFusion code to treat certain conditions—which ColdFusion wouldn't be capable of identifying as problematic on its own—as exceptions, just like the built-in exceptions thrown by ColdFusion itself.

Introducing <cfthrow>

To throw your own custom exceptions, use the <cfthrow> tag. The exceptions you raise with <cfthrow> can be caught with a <cftry>/<cfcatch> block, just like the exceptions ColdFusion throws internally. If your custom exception isn't caught (or is caught and then rethrown via the <cfrethrow> tag), ColdFusion simply displays the text you provide for the message and detail attributes in a standard error message.

Table 51.4 lists the attributes you can provide for the <cfthrow> tag. All of them are optional, but it's strongly recommended that you at least provide the message attribute whenever you use <cfthrow>.

Table 51.4 <cfthrow> Tag Attributes

ATTRIBUTE	DESCRIPTION
type	A string that classifies your exception into a category. You can provide either type="APPLICATION", which is the default, or a type of your own choosing. You can't provide any of the predefined exception types listed in Table 51.1. You can specify custom exception types that include dots, which enables you to create hierarchical families of custom exceptions. See the next section, Custom Exception Families."
message	A text message that briefly describes the error you are raising. If the exception is caught in a <cfcatch> block, the value you provide here is available as CFCATCH.Message. If the exception isn't caught, ColdFusion displays this value in an error message to the user. This value is optional, but it's strongly recommended that you provide it.

Table 51.4 (continued)

ATTRIBUTE	DESCRIPTION
`detail`	A second text message that describes the error in more detail, or any background information or hints that will help other people understand the cause of the error. If the exception is caught in a `<cfcatch>` block, the value you provide here is available as `CFCATCH.Detail`. If the exception isn't caught, ColdFusion displays this value in an error message to the user.
`errorCode`	An optional error code of your own devising. If the exception is caught in a `<cfcatch>` block, the value you provide here is available as `CFCATCH.ErrorCode`.
`extendedInfo`	A second optional error code of your own devising. If the exception is caught in a `<cfcatch>` block, the value you provide here will be available as `CFCATCH.ExtendedInfo`.
`object`	Allows you to throw an exception object defined in Java. The object must first be created by `<cfobject>` or `createObject()` before it can be used in `<cfthrow>`.

Throwing Custom Exceptions

Say you're working on a piece of code that performs some type of operation (such as placing an order) based on a `ContactID` value. As a sanity check, you should verify that `ContactID` is actually valid before going further. If you find that `ContactID` isn't valid, you can throw a custom error using the `<cfthrow>` tag, like so:

```
<cfthrow
 message="Invalid Contact ID"
 detail="No record exists for that Contact ID.">
```

This exception can be caught using a `<cfcatch>` tag, as shown in the following. Within this `<cfif>` block, you can take whatever action is appropriate in the face of an invalid contact ID (perhaps inserting a new record into the Contacts table or using `<cfmail>` to send a message to your customer service manager):

```
<cfcatch type="APPLICATION">
 <cfif CFCATCH.message eq "Invalid Contact ID">
 <!--- recovery code here --->
 </cfif>
</cfif>
```

This `<cfcatch>` code catches the Invalid Contact ID exception because the exception's type is `APPLICATION`, which is the default exception type used when no `type` attribute is provided to `<cfthrow>`. If you want, you can specify your own exception type, like so:

```
<cfthrow
 type="InvalidContactID"
 message="Invalid Contact ID"
 detail="No record exists for that Contact ID.">
```

This exception will be caught by any `<cfcatch>` tag that has a matching `type`, like this one:

```
<cfcatch type="InvalidContactID">
```

```
 <!--- recovery code here --->
</cfcatch>
```

NOTE

You can also use the `<cfabort>` tag with the `showError` attribute to throw a custom exception. If the exception is caught, the text you provide for `showError` becomes the `CFCATCH.Message` value. However, the use of `<cfthrow>` is preferred because you can provide more complete information about the error you are raising, via `type`, `detail`, and other attributes shown in Listing 51.5.

Creating Custom Exception Families

You can create hierarchical families of exceptions by including dots (periods) in the `type` attribute you provide to the `<cfthrow>` tag. If you do so, you can catch whole groups of exceptions using a single `<cfcatch>` tag.

For instance, you could throw an error such as the following:

```
<cfthrow
 type="OrangeWhipStudios.InternalData.InvalidContactID"
 message="Invalid Contact ID"
 detail="No record exists for that Contact ID.">
```

You could catch the previous exception with a matching `<cfcatch>` tag, like this one:

```
<cfcatch type="OrangeWhipStudios.InternalData.InvalidContactID">
```

If no `<cfcatch>` matches the `TYPE` exactly, ColdFusion looks for a `<cfcatch>` tag that matches the next most specific type of exception, using the dots to denote levels of specificity. So this `<cfcatch>` tag would catch the error thrown by the previous `<cfthrow>`:

```
<cfcatch type="OrangeWhipStudios.InternalData">
```

If no `<cfcatch>` such as the previous one was present, this `<cfcatch>` would catch the error, which would also catch other errors in the `OrangeWhipStudios` hierarchy, such as `OrangeWhipStudios.InternalData.InvalidMerchID` and `OrangeWhipStudios.DatabaseNotAvailable`:

```
<cfcatch TYPE="OrangeWhipStudios">
```

Custom Exceptions and Custom Tags

Listing 51.12 is an example of a ColdFusion template that includes several `<cfthrow>` tags that know how to report helpful diagnostic information when problems arise. It's a revision of the `<cf_PlaceOrder>` custom tag from Chapter 22, "Online Commerce," online. This version performs all the same operations as the original version; the main difference is the addition of several sanity checks at the top of the template, which ensure that the various ID numbers passed to the tag make sense. If not, custom exceptions are thrown, which the calling template can catch with `<cfcatch>` if it wants. If the calling template doesn't catch the exceptions thrown by the tag, ColdFusion displays the exception to the user in the form of an error message.

➜ This template relies on many of the templates from Chapter 22. To test this template, save it as `PlaceOrder.cfm` in the same folder as the examples from Chapter 22.

Listing 51.12 `PlaceOrder.cfm`—Throwing Your Own Exceptions with `<cfthrow>`

```
<!---
 Filename: PlaceOrder.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Processes a user's user, charging credit card, etc
 Please Note Relies upon <cf_ProcessPayment> custom tag
--->

<!--- Tag Parameters --->
<cfparam name="ATTRIBUTES.processor" type="string" default="PayflowPro">
<cfparam name="ATTRIBUTES.merchList" type="string">
<cfparam name="ATTRIBUTES.quantList" type="string">
<cfparam name="ATTRIBUTES.contactID" type="numeric">
<cfparam name="ATTRIBUTES.creditCard" type="string">
<cfparam name="ATTRIBUTES.creditExpM" type="string">
<cfparam name="ATTRIBUTES.creditExpY" type="string">
<cfparam name="ATTRIBUTES.creditName" type="string">
<cfparam name="ATTRIBUTES.shipAddress" type="string">
<cfparam name="ATTRIBUTES.shipCity" type="string">
<cfparam name="ATTRIBUTES.shipCity" type="string">
<cfparam name="ATTRIBUTES.shipState" type="string">
<cfparam name="ATTRIBUTES.shipZIP" type="string">
<cfparam name="ATTRIBUTES.shipCountry" type="string">
<cfparam name="ATTRIBUTES.htmlMail" type="boolean">
<cfparam name="ATTRIBUTES.returnVariable" type="variableName">
<cftry>
  <!--- Make sure the MerchList and QuantList Attributes make sense --->
  <cfif (listLen(ATTRIBUTES.merchList) EQ 0)
  OR listLen(ATTRIBUTES.merchList) neq listLen(ATTRIBUTES.quantList)>
    <!--- If not, throw an error --->
    <cfthrow message="Invalid MerchList or QuantList attribute"
    detail="Both must have same number of list elements, and can't be empty.">
  </cfif>

  <!--- Quick query to verify the ContactID is valid --->
  <cfquery name="getCount" datasource="#APPLICATION.dataSource#">
  SELECT Count(*) AS ContactCount
  FROM Contacts
  WHERE ContactID = #ATTRIBUTES.contactID#
  </cfquery>

  <!--- If any of the MerchIDs are not valid, throw custom error --->
  <cfif getCount.contactCount neq 1>
    <cfthrow type="ows.MerchOrder.InvalidContactID"
    message="Invalid Contact ID"
    detail="The ContactID you provided (#ATTRIBUTES.contactID#) is not valid."
    errorcode="1">
  </cfif>

  <!--- Quick query to verify that all MerchIDs are valid --->
  <cfquery name="getCount" datasource="#APPLICATION.dataSource#">
  SELECT Count(*) AS ItemCount
  FROM Merchandise
  WHERE MerchID IN (#ATTRIBUTES.merchList#)
  </cfquery>
```

Listing 51.12 (continued)

```
      <!--- If any of the MerchIDs are not valid, throw custom error --->
      <cfif getCount.ItemCount neq listLen(ATTRIBUTES.merchList)>
        <cfthrow type="ows.MerchOrder.InvalidMerchID"
        message="Invalid Merchandise ID"
        detail="At least one of the MerchID values you supplied is not valid."
        errorcode="2">
      </cfif>

      <!--- If any database problems came up during above validation steps --->
      <cfcatch type="Database">
        <cfthrow type="ows.MerchOrder.ValidationFailed"
        message="Order Validation Failed"
        detail="A database problem occured while attempting to validate the order.">
      </cfcatch>
    </cftry>

    <!--- Begin "order" database transaction here --->
    <!--- Can be rolled back or committed later --->
    <cftransaction action="begin">
      <cftry>
        <!--- Insert new record into Orders table --->
        <cfquery datasource="#APPLICATION.dataSource#">
        INSERT INTO MerchandiseOrders (
        ContactID,
        OrderDate,
        ShipAddress, ShipCity,
        ShipState, ShipZip,
        ShipCountry)
        VALUES (
        #ATTRIBUTES.contactID#,
        <cfqueryparam cfsqltype="CF_SQL_TIMESTAMP"
        value="#dateFormat(Now())# #timeFormat(Now())#">,
        '#ATTRIBUTES.shipAddress#', '#ATTRIBUTES.shipCity#',
        '#ATTRIBUTES.shipState#', '#ATTRIBUTES.shipZip#',
        '#ATTRIBUTES.shipCountry#'
        )
        </cfquery>

        <!--- Get just-inserted OrderID from database --->
        <cfquery datasource="#APPLICATION.dataSource#" name="getNew">
        SELECT MAX(OrderID) AS NewID
        FROM MerchandiseOrders
        </cfquery>

        <!--- For each item in user's shopping cart --->
        <cfloop from="1" to="#listLen(ATTRIBUTES.merchList)#" index="i">
          <cfset thisMerchID = listGetAt(ATTRIBUTES.merchList, i)>
          <cfset thisQuant = listGetAt(ATTRIBUTES.quantList, i)>

           <!--- Add the item to "OrdersItems" table --->
           <cfquery datasource="#APPLICATION.dataSource#">
           INSERT INTO MerchandiseOrdersItemsf
           (OrderID, ItemID, OrderQty, ItemPrice)
           SELECT
           #getNew.NewID#, MerchID, #thisQuant#, MerchPrice
```

Listing 51.12 (continued)

```
      FROM Merchandise
      WHERE MerchID = #thisMerchID#
      </cfquery>
    </cfloop>

    <!--- Get the total of all items in user's cart --->
    <cfquery datasource="#APPLICATION.dataSource#" name="getTotal">
    SELECT SUM(ItemPrice * OrderQty) AS OrderTotal
    FROM MerchandiseOrdersItems
    WHERE OrderID = #getNew.NewID#
    </cfquery>

    <!--- Attempt to process the transaction --->
    <cf_ProcessPayment
    processor="#ATTRIBUTES.processor#"
    orderID="#getNew.NewID#"
    orderAmount="#getTotal.OrderTotal#"
    creditCard="#ATTRIBUTES.creditCard#"
    creditExpM="#ATTRIBUTES.creditExpM#"
    creditExpY="#ATTRIBUTES.creditExpY#"
    creditName="#ATTRIBUTES.creditName#"
    returnVariable="chargeInfo">

    <!--- If the order was processed successfully --->
    <cfif chargeInfo.IsSuccessful>
      <!--- Commit the transaction to database --->
      <cftransaction action="Commit"/>
    <cfelse>
      <!--- Rollback the Order from the Database --->
      <cftransaction action="RollBack"/>
    </cfif>

    <!--- If any errors occur while processing the order --->
    <cfcatch type="Any">
      <!--- Rollback the Order from the Database --->
      <cftransaction action="RollBack"/>
      <!--- Throw a custom exception --->
      <cfthrow type="ows.MerchOrder.OrderFailed"
      message="Order Could Not Be Completed"
      detail="The order (ID #getNew.NewID#) was rolled back from the database."
      errorcode="3">
    </cfcatch>
  </cftry>
</cftransaction>

<!--- If the order was processed successfully --->
<cfif chargeInfo.IsSuccessful>
  <!--- Send Confirmation e-mail, via Custom Tag --->
  <cf_SendOrderConfirmation
  orderID="#getNew.NewID#"
  useHTML="#ATTRIBUTES.htmlMail#">
</cfif>

<!--- Return status values to callling template --->
<cfset "Caller.#ATTRIBUTES.returnVariable#" = chargeInfo>
```

The first use of `<cfthrow>` just performs a bit of simple data validation on the `merchList` and `quantList` attributes. The custom tag needs these two lists to have the same number of elements; additionally, neither list should be allowed to be empty. If the `listLen()` function reports that the lists contain different numbers of elements, or if the lists are empty, an exception is raised with a `message` that reads `Invalid MerchList or QuantList attribute`.

Next, the `<cfthrow>` tag is used again to raise an exception of type `ows.MerchOrder.InvalidContactID` if the `contactID` attribute passed to the custom tag isn't a valid contact ID number. First, the `getCount` query counts the number of records in the Contacts table with the given `contactID`. If the ID number is legitimate, the query returns a `ContactCount` of `1`. If not, the custom exception is thrown.

Similar logic is used to ensure that all the merchandise IDs in the `MerchList` attribute are legitimate. If the `ItemCount` value returned by the second `GetCount` query is the same as the number of items in the list, all the merchandise IDs must be legitimate (and must not contain any duplicate values). If not, a custom exception of type `ows.MerchOrder.InvalidMerchID` is thrown.

The whole top portion of the template (the portion that performs the sanity checks) is also wrapped in its own `<cftry>` block, so a custom exception of type `ows.MerchOrder.ValidationFailed` is thrown if any database errors occur while performing either of the `GetCount` queries.

NOTE

Listing 51.12 has also added a `<cftry>` block around the main portion of the template, where the order is actually processed. This `<cftry>` tag uses `<cftransaction>` to roll back any changes to the database if a problem occurs. See the next section, "Exceptions and Database Transactions," for an explanation.

Listing 51.13 is a revised version of the `StoreCheckout.cfm` template, also from Chapter 22. It calls the `<cf_PlaceOrder>` custom tag from Listing 51.12, catching any of the custom exceptions the custom tag might throw. Because all of the exceptions that the tag throws start with `ows.MerchOrder`, a single `<cfcatch>` of `type="owsMerchOrder"` can catch them all.

➜ This template relies on many of the templates from Chapter 22. To test this template, save it as `StoreCheckout.cfm` in the same folder as the examples from Chapter 28, "ColdFusion Server Configuration," in Vol. 2, *Application Development*. Make sure that you've also saved Listing 51.12 as `PlaceOrder.cfm` in the same folder.

Listing 51.13 `StoreCheckout.cfm`—Catching Custom Exceptions Thrown by a Custom Tag

```
<!---
 Filename: StoreCheckout.cfm
 Created by: Nate Weiss (NMW)
 Purpose: Places an order for all items in user's shopping cart
 Please Note Relies upon <CF_PlaceOrder> custom tag
--->

<!--- Show header images, etc., for Online Store --->
<cfinclude template="StoreHeader.cfm">

<!--- Get current cart contents, via Custom Tag --->
<cf_ShoppingCart action="List" returnVariable="GetCart">
```

Listing 51.13 (continued)

```
<!--- Stop here if user's cart is empty --->
<cfif getCart.recordCount eq 0>
 There is nothing in your cart.
 <cfabort>
</cfif>

<!--- If user isn't logged in, force them to now --->
<cfif not isDefined("SESSION.auth.isLoggedIn")>
  <cfinclude template="LoginForm.cfm">
  <cfabort>
</cfif>

<!--- If user is attempting to place order --->
<cfif isDefined("FORM.isPlacingOrder")>

  <cftry>
    <!--- Attempt to process the transaction --->
    <cf_PlaceOrder
    processor="JustTesting"
    contactID="#SESSION.auth.contactID#"
    merchList="#valueList(getCart.MerchID)#"
    quantList="#valueList(getCart.Quantity)#"
    creditCard="#FORM.creditCard#"
    creditExpM="#FORM.creditExpM#"
    creditExpY="#FORM.creditExpY#"
    creditName="#FORM.creditName#"
    shipAddress="#FORM.shipAddress#"
    shipState="#FORM.shipState#"
    shipCity="#FORM.shipCity#"
    shipZIP="#FORM.shipZIP#"
    shipCountry="#FORM.shipCountry#"
    htmlMail="#FORM.htmlMail#"
    returnVariable="orderInfo">

    <!--- If any exceptions in the "ows.MerchOrder" family are thrown... --->
    <cfcatch type="ows.MerchOrder">
    <p>Unfortunately, we are not able to process your order at the moment.
    Please try again later. We apologize for the inconvenience.</p>
    <cfabort>
    </cfcatch>
  </cftry>

  <!--- If the order was processed successfully --->
  <cfif orderInfo.isSuccessful>

    <!--- Empty user's shopping cart, via custom tag --->
    <cf_ShoppingCart
    ACTION="Empty">

    <!--- Display Success Message --->
    <cfoutput>
    <h2>Thanks For Your Order</h2>
    <p><b>Your Order Has Been Placed.</b><br>
    Your order number is: #OrderInfo.OrderID#<br>
    Your credit card has been charged:
```

Listing 51.13 (continued)

```
        #lsCurrencyFormat(OrderInfo.OrderAmount)#</p>
        <p>A confirmation is being e-mailed to you.</p>
        </cfoutput>

        <!--- stop here. --->
        <cfabort>
      <cfelse>
        <!--- Display "Error" message --->
        <font color="red">
        <p><strong>Your credit card could not be processed.</strong><br>
        Please verify the credit card number, expiration date, and
        name on the card.</p>
        </font>

        <!--- Show debug info if viewing page on server --->
        <cfif CGI.remote_addr eq "127.0.0.1">
          <cfoutput>
          Status: #orderInfo.Status#<br>
          Error: #orderInfo.ErrorCode#<br>
          Message: #orderInfo.ErrorMessage#<br>
          </cfoutput>
        </cfif>
      </cfif>
    </cfif>


    <!--- Show Checkout Form (Ship Address/Credit Card) --->
    <cfinclude template="StoreCheckoutForm.cfm">
```

This listing is the same as the original version from Chapter 22, except for the addition of the `<cftry>` and `<cfcatch>` blocks. Please see Chapter 22 for a complete discussion.

Exceptions and Database Transactions

Listing 51.12 uses `<cftry>` and `<cfcatch>` together with the `<cftransaction>` tag so that the database transaction can be rolled back if any errors occur. The `<cftry>` block is wrapped around all the database interactions, as well as the call to the `<cf_ProcessPayment>` custom tag.

If any database errors or syntax errors occur while the order is being processed (even inside `<cf_ProcessPayment>`), the `<cfcatch>` tag near the end of Listing 51.12 catches them. The `<cfcatch>` tag uses `<cftransaction>` with `action="Rollback"`, which has the effect of undoing all the database changes made since the beginning of the first `<cftransaction>` tag. This greatly minimizes the risk that any erroneous or inconsistent order information will get recorded in the `MerchandiseOrders` or `MerchandiseOrdersItems` tables.

In many respects, you can think of `<cftransaction>` as the database equivalent of `<cftry>` because they are both about recovering gracefully when problems arise. Used together, they become even more powerful.

➜ It's generally preferable to keep transactions contained within stored procedures whenever possible. However, when that isn't possible, you can use `<cftry>` and `<cftransaction>` together as shown here. For more information, see Chapter 41, "More About SQL and Queries," and Chapter 42, "Working with Stored Procedures," both online.

CHAPTER 52

Using the Debugger

IN THIS CHAPTER

Overview E472

Installing the Debugger E473

Configuring ColdFusion and the Debugger E474

Using the Debugger E481

ColdFusion 8 introduces a powerful new developer tool in the new Debugger, an interactive step debugger which finally allows a CFML developer to walk through the execution of their code, observing what lines of code it executes, what other CFML files (include files, custom tags, or CFCs) it executes, and the value of all variables (in all scopes) at a given point during execution. You can even change the value of variables on the fly within the debugger, such as to alter the runtime flow of execution of your code.

More than just watching the code that you run as a developer, in your browser, it's important to note that the debugger can watch the execution of any CFML, however it's requested, including calls from Ajax or Flex clients, or from other servers issuing web service requests or CFHTTP equivalents. There are also other kinds of requests that can happen within ColdFusion that have no traditional browser: scheduled tasks, requests made via the ColdFusion gateways, and events in the `Application.cfc` such as `onSessionEnd` and `onApplicationEnd`.

With all these kinds of requests, there often is no browser to which you can easily send back traditional debugging statements. And even with traditional browser requests, there are still times when CFML code can't display any output, such as when output has been disabled in a CFC or method, or within a `CFSILENT` tag pair, and so on.

This is where a debugger becomes so valuable, where you use a tool to watch the execution of some code while it runs live, regardless of how it was requested by a user. And yes, you can even use the debugger to observe the execution of requests made by another user. Of course, there is all the requisite security to ensure the debugger is used only by authorized developers.

The debugger is part of the new suite of Adobe ColdFusion 8 Extensions for Eclipse, which add new functionality to the Eclipse IDE. Whether you're new to debugging or feel lukewarm about the concept, and even if you may be hesitant about using Eclipse as an editor ("I'm a DreamWeaver/CF Studio/HomeSite fan"), this chapter will help you feel comfortable and confident using the Debugger as a tool in your CFML developer toolkit. It will introduce debugging and the debugger, how to install and configure it, and how to use it.

The initial discussion of understanding, installing, and configuring the debugger (and ColdFusion) may seem tedious, but it's generally a one-time effort. Once configured, you will be able to use the debugger quite easily and quickly.

Overview

Traditional Forms of Debugging

As developers, we spend most of our day debugging code: whether it's new code we're writing, old code we're updating, or someone else's code we've inherited. When trying to resolve a problem in a given CFML template or CFC, you often need to understand the flow and function of the code, and to do that you have a couple of choices.

First, you can try to just statically review the source code, relying on your skills at reading and interpreting the code to determine what it should be doing, eye-balling potentially many lines of code, and guessing at or tracking manually what the values of various variables will be at any given point. This can be a complicated effort, but some developers are especially well-suited to the task.

More commonly we instead let ColdFusion just run the page and then we use either of two techniques to have it report what's going on in the execution of the code. The ColdFusion debugging output, which is displayed at the end of a page request (if enabled in the ColdFusion Administrator), can show high-level information about that request, including how long it took to run, what files it called (includes, custom tags, CFCs), what queries it executed (their duration, record count, SQL, and such), the variables and values in many variable scopes, and much more. This feature was discussed in Chapter 17, "Debugging and Troubleshooting," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 1: Getting Started*.

Still, there are times when either that level of detail is not enough, or perhaps debugging output is not enabled (and/or cannot be). In that case most developers will resort to placing `CFOUTPUT` and `CFDUMP` tags strategically in the program code both to depict the flow of control ("what lines of code am I running?") and to determine the value of variables at a given point. This is a tried and true method, and for many developers they've known no other alternative.

Introducing Step Debugging

Developers who come from some other programming environments, however, are often surprised to find CFML developers having only these techniques at their disposal. Such developers have often had the advantage of using a step debugging tool to help them in this very sort of situation. With a step debugger, a developer can actually observe the lines of code as they are executed, watching each line begin and end, as well as watching the value of any and all variables in play at that point in the code.

Consider the code fragment in Figure 52.1, which depicts some code about to be executed. For now, let's not focus on the editor in which it's appearing. As we observe the code, we can wonder whether the first if test will be true, and if so, whether its embedded if or else test will be true, and

in either case we may wonder what goes on in the `setMessage` method that they call. If we knew the value of the variables being tested, then of course we'd know which test would be true.

Figure 52.1

Sample CFML code being debugged

```
29        <!--- set status message using user defined function--->
30        <cfif not compareNoCase(form.artID,0)>
31           <cfset status=setMessage("#form.artName# was added.")>
32        <cfelse>
33           <cfset status=setMessage("#form.artName# was updated.")>
34        </cfif>
35        <cflocation url="manage_art.cfm" addtoken="yes">
36     <cfelse>
37        <cfset status=setMessage("There was a problem. Please try again.")>
38     </cfif>
39  </cfif>
```

A step debugger could tell us all that information, and more. We could tell the debugger to stop execution at a given line of code, such as our line 28 above, and while there we could view all the currently defined variables (in any and all scopes). More important, we could ask the debugger to step through the code, running whatever the next line would be (line 30), and then step again, in which case we'd run either line 31 or 33, depending on the condition evaluated. No more guessing!

Beyond that, while observing this code (without a debugger) we may wonder other things: how did we get to this point in the code? Was this file called as an include file, custom tag, or CFC method? Did we run an `Application.cfm` or `Application.cfc` file before getting here? And if the code was large and complex, we may even wonder about the `setMessage` function itself: what does it do? Where is it defined (in this program, or included from another)? And so on.

All these things and more can be discovered instantly with the ColdFusion 8 debugger. Let's look first at how to install and configure it, then we'll show how to use the debugger to answer the questions just posed.

Installing the Debugger

The ColdFusion 8 debugger is built on top of Eclipse (and uses the same debugging interface as Adobe Flex Builder and other Eclipse plug-ins). It's offered as a free tool (one of many) within Adobe `ColdFusion 8 Extensions for Eclipse`, which is a zip file available at `http://www.adobe.com/support/coldfusion/downloads.html#cfdevtools`. These extensions are not to be confused with the community-driven CF-oriented extension for Eclipse called CFEclipse (available at `cfeclipse.org`).

If you're not yet familiar with Eclipse or have not installed it (or CFEclipse), please see the introduction offered in Chapter 2, "Choosing a Development Environment," in Vol. 1, *Getting Started.*

Even if you don't care to use Eclipse as your day-to-day editor, it's still very worthwhile to implement the extensions and use the debugger (and its many other features, including the Ajax and Flex wizards, CFC/ActionsScript wizards, RDS tools, a Services browser, Log file viewer, and more).

Installation of the Eclipse extensions is discussed in the ColdFusion manual, *Installing and Using ColdFusion*, in the chapter "Installing Integrated Technologies," available online at `http://livedocs.adobe.com/coldfusion/8/htmldocs/othertechnologies_11.html`. Other resources discussing installation of the Extensions include `http://www.forta.com/blog/index.cfm/2007/5/30/Installing-ColdFusion-8-Eclipse-Extensions` and `http://www.adobe.com/devnet/coldfusion/articles/debugger.html`.

With Eclipse and the Adobe ColdFusion 8 Extensions installed, you're ready to configure ColdFusion and the debugger.

Configuring ColdFusion and the Debugger

In order to use the debugger against a given implementation of ColdFusion 8, you must configure that ColdFusion 8 server to permit debugging. (Note that the debugger only works with ColdFusion 8. If you're interested in debugging ColdFusion 6 or 7 servers, look into FusionDebug, a commercial debugger that functions very similarly to the ColdFusion 8 debugger, available at `www.fusiondebug.com`.)

Configuring the ColdFusion Administrator

You enable a server to permit debugging by way of settings in the ColdFusion Administrator. The setting for the ColdFusion 8 debugger is separate from the traditional debugging feature also enabled in the Administrator. There are also important settings on other Administrator screens.

Enable Debugger Settings

The first and most important settings to enable for debugging are on a page in the Debugging & Logging section of the Administrator, in a new page called Debugger Settings. See Figure 52.2. There, you must select the Allow Line Debugging option and click Submit Changes to enable debugging against this server. Finally, you must restart the server upon enabling (or disabling) it for the setting to take effect. (More on this is offered later in the section, Special Considerations for MultiServer Deployment.)

Figure 52.2

ColdFusion Administrator Debugger Settings Page

While on that page, notice the available Debugger Port setting, which you need not change unless it's a port that's unavailable on your server. The setting Maximum Simultaneous Debugging Sessions also need not be changed unless you expect to have many developers debugging against a single shared server. Finally, I'll explain the Start Debugger Server button at the end of the article. For now, just know that you don't need to click the button to start debugging. The first debugging request will cause the debugger to start. (And note that when debugging has started, the button will change to Stop Debugger and will also have a Restart Debugger button appear next to it.)

Configuring RDS to Secure the Server

Once the Allow Debugging option is enabled, which permits a server to be debugged, the next significant step is controlling who may debug code on the server. Adobe chose to build debugger security on top of the long-standing RDS (Remote Developer Services) feature, for providing secured access from developers using editors trying to access the ColdFusion server.

In order for a ColdFusion server to be debugged, the debugging developer will need to connect to the server using credentials specified in the RDS settings of the server, and as will be specified by the developer in configuring the editor (in Eclipse, in the case of the CF 8 debugger). The section below, "Configuring Eclipse," will explain using these RDS credentials in Eclipse.

The ColdFusion server administrator controls whether RDS is enabled (it can be disabled at installation), and s/he also chooses the credentials (if any) that will be required by developers, as configured on the Administrator page Security>RDS. Note that ColdFusion 8 Enterprise and Developer editions add a powerful new option regarding RDS, allowing multiple user accounts, so that different developers can each be given their own RDS username/password pair authorizing them to access the server from development tools like Eclipse and the debugger. RDS security is discussed further in Chapter 65, "Security in Shared and Hosted Environments," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 3: Advanced Application Development*.

Increase Maximum Request Timeouts

While debuggers are powerful in enabling you to stop on a line of code and inspect variables, etc., there is a down-side. ColdFusion is often configured to forcefully timeout requests that take too long, as configured in the Administrator page, Server Settings>Settings. If Timeout Requests After (Seconds) is checked, then ColdFusion will try to terminate requests lasting longer than the specified time limit. When debugging code on a server using the step debugger, it may be wise to either increase the time limit or disable this feature entirely (in which case requests are free to run as long as they need to).

Special Considerations for MultiServer Deployment

A final discussion regarding ColdFusion configuration for the debugger revolves around a difference between installing ColdFusion in the standalone (Server) mode versus the Multiserver (or multi-instance) or J2EE deployment mode. In the former, when you enable the aforementioned setting, Allow Line Debugging, ColdFusion will automatically modify the `jvm.config` file used to control the server, placing there the appropriate information needed to enable debugging.

In the Multiserver or J2EE modes, however, you must manually place the following information into the `jvm.config` file on the existing `java.args` line provided here:

```
Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=#portNum#
```

`#portnum#` is the port number entered in the Debugger Port setting, discussed above. Be sure to make a copy of the file before editing it. Any mistake in the file could prevent ColdFusion restarting.

Yet another concern when you run the Multiserver edition of ColdFusion is that by default all the instances share a single `jvm.config`. This is a problem when using the debugger, as only one instance at a time can use the debugger port specified above. You may find that at times when an instance goes down, the lock on the debugger port remains tied to one or the other instances. In this case, you should explore giving each instance its own `jvm.config`, with its own designation of the debugger port. This is discussed in the Adobe technote at `http://www.adobe.com/go/tn_18206`.

Upon successful restart of ColdFusion, you're ready to proceed to configuring Eclipse.

Configuring Eclipse

In order to connect a developer's Eclipse installation to a ColdFusion server, a few configuration settings must be made in the Eclipse environment, including RDS, and possibly debug mappings and debugger settings, as well as defining an Eclipse debugging configuration for ColdFusion. The following steps presume you have successfully installed the Adobe ColdFusion 8 Extensions for Eclipse, as discussed above.

Configuring RDS

As explained above, a ColdFusion server must enable RDS in order to use the debugger, and a developer must configure their Eclipse environment to connect to the server via RDS. To configure an RDS server in Eclipse, select Window > Preferences > ColdFusion > RDS Configuration. (Note that you are to choose ColdFusion in the Preferences pane, not CFEclipse, which may also appear. These are two very different plug-ins for Eclipse.)

Here you specify the connection information to point to the ColdFusion server against which you want to perform debugging. See Figure 52.3.

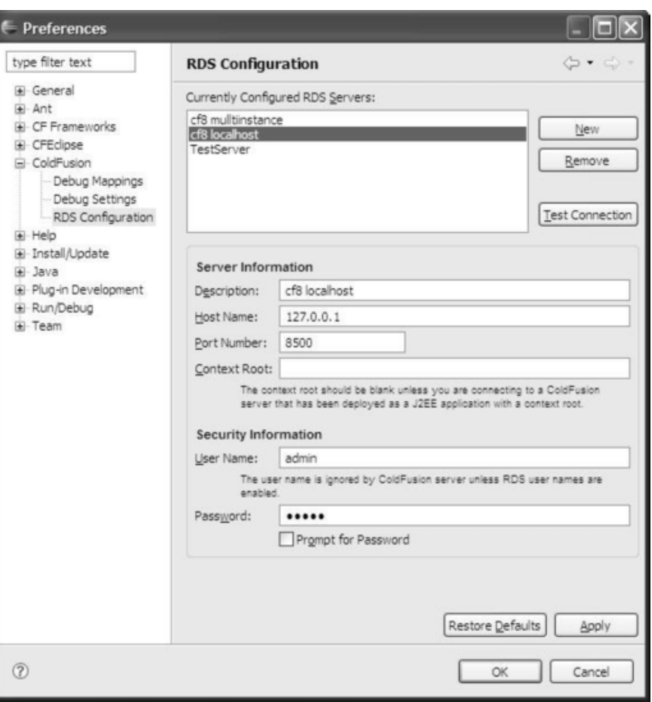
If a server is already defined in this display (perhaps for using it with other Adobe ColdFusion Extensions), simply select the server and observe the settings in Server Information.

To create a new server configuration, click the New button. The fields provided identify the server to connect to. The Description is used to provide a name used within Eclipse. The Host Name and Port Number values are those you would use to connect to the server to be debugged. This is the port number used to reach the web server configured for handling ColdFusion requests, such as port 80 or 8500. (This is not the Debugger port mentioned earlier in the Administrator Debugger Settings page.)

Note that like a browser, RDS connects to your intended server over HTTP, so you can point to any server (even a remote one) as long as it's accessible via the URL/port you specify and as long as RDS is enabled for that server.

Figure 52.3

Eclipse ColdFusion Plug-in RDS Configuration Page



For debugging against your local machine, you would likely use `localhost` for the HostName and either `80` or `8500` for the port. (If you have difficulty understanding what values to place here, simply use whatever values you would use to specify the URL to the Administrator for the intended server.)

The Context Root field should be left blank unless you've deployed ColdFusion using the J2EE deployment mode, in which case you would specify here the context root of the deployed J2EE web app.

Finally the User Name and Password fields are for providing the RDS security authentication credentials required to access the server to be debugged, as discussed above in, "Configuring RDS to Secure the Server." If only an RDS password was defined in the ColdFusion Administrator, then the User Name field is ignored here. Use it when you have configured ColdFusion to use multiple user accounts for RDS.

Click Apply to save any changes you may have made on this screen. Finally, to test a new or existing connection, choose the Test Connection button. If you can successfully connect to the server via RDS, you should be able to debug against it. You can also use other RDS-based features of the Adobe ColdFusion Extensions such as the RDSDataView, RDSFileView, Service Browser, and other features, such as those available under the Eclipse option Window > Show View > Other > ColdFusion.

Debug Mappings

If ColdFusion and Eclipse are running on the same computer, you can skip the Mappings setting (Window > Preferences > ColdFusion > Debug Mappings). Otherwise, use this setting to map the path to files as Eclipse will see them (on your local machine typically) to the path to the same files as they will be found by the ColdFusion server. For instance, if you edit files in c:\inetpub\wwwroot\someapp\ but later upload them to a remote or central server where they're identified to that server as d:\webs\someapp, then these would be the two values you'd specify on this page for client and server mappings.

To add a new set of mappings, first choose the server against which you will be debugging (in the RDS Server list, populated per the previous steps described above), then click Add Mapping and follow the instructions to fill in the Eclipse and ColdFusion paths, and click Apply to save the settings. See Figure 52.4. (Again, though, if you're running Eclipse and ColdFusion on the same computer, then you don't need add a debug mapping.)

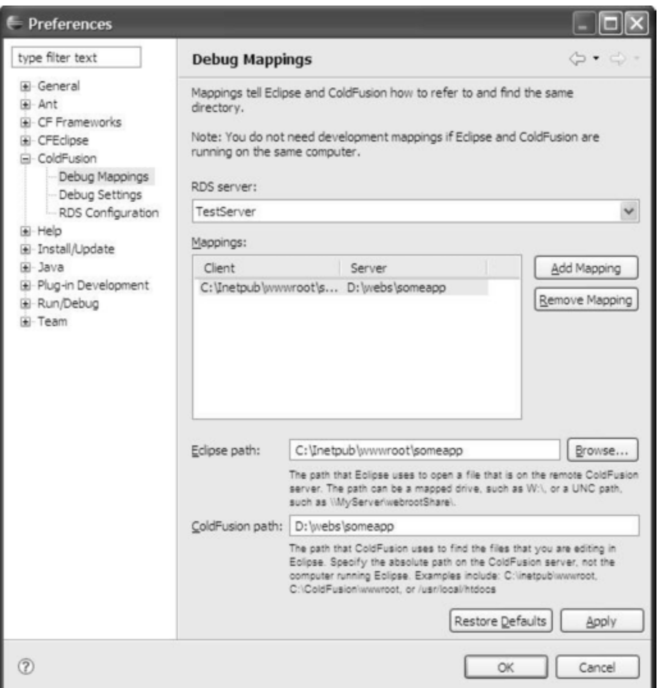


Figure 52.4

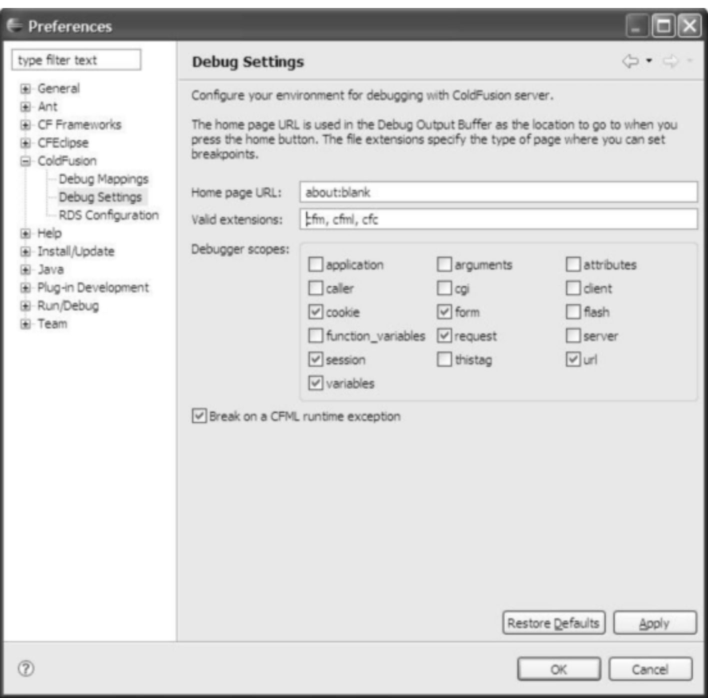
Eclipse ColdFusion Plug-in Debug Mappings Page

Debug Settings: Variables Shown and Stop on Error

The third and final segment of the Eclipse settings for RDS is the Debug Settings page at Window > Preferences > ColdFusion > Debug Settings. There are four settings configurable on the page, with the final two being most important. See Figure 52.5:

Figure 52.5

Eclipse ColdFusion Plug-in Debug Settings Page



First, you can specify (if desired) a Home Page URL, which will be used to point to the page that appears in the Browser pane of the Debug Output Buffer of the debugger (a feature discussed later) when you click the Home button there (not something you'd likely used that often). Also, you can specify the extensions of the types of files that you want to debug.

More important for most is the Debugger Scopes setting, where you can configure which variable scopes you want the Debugger to display when you're displaying variables (discussed later). Though it's tempting to select many scopes to see all possible information within the debugger, each will add more burden to (and slow execution of) the debugger. Here's an important tip: you can always view any variable in any scope using the available Add Watch Expression feature, discussed later.

The fourth and final setting on this page is Break on a CFML Runtime Exception. With this feature enabled, the debugger should stop on a template you are debugging, at the line which causes a ColdFusion error. When it works, it's a powerful feature (since it precludes your needing to anticipate where code in the program might cause an error). In my own testing, I've found it to sometimes not stop on the error, though. In that case, debugging simply stops and the user making the request gets the same error message that they'd normally get.

Manage Debug Configurations

The final step in being able to debug ColdFusion code is to create a new Eclipse debugging configuration for ColdFusion, which simply indicates the ColdFusion server that you want to connect to

(using the RDS settings defined above). In Eclipse, choose the menu command Run > Debug, which opens the Debug window to create, manage, and run debugging configurations. See Figure 52.6:

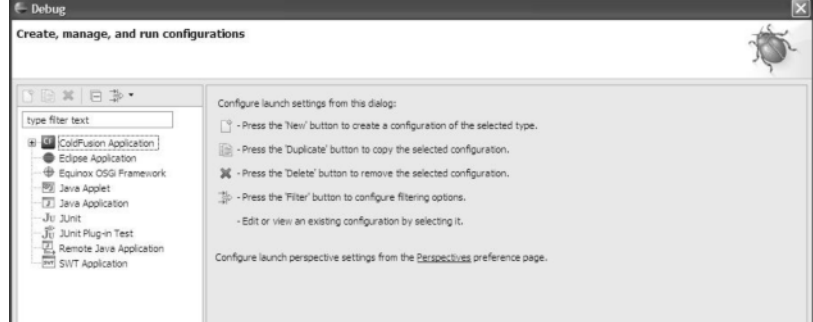


Figure 52.6

Eclipse Debug Configurations Page

Here one could define debug configurations for different kinds of Eclipse applications. To create a new ColdFusion configuration, double-click the option shown for ColdFusion Application, or as directed on screen select the ColdFusion Application option and press the New button. The icon to press is depicted in the instructions shown on screen as in the figure above.

This creates a new configuration, as shown in Figure 52.7:

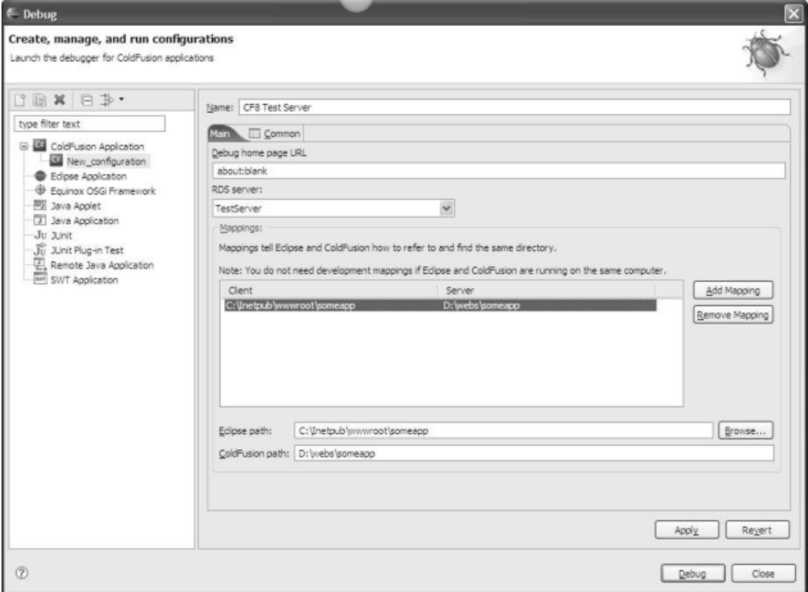


Figure 52.7

Adding a New ColdFusion Debug Configuration

Here you can specify a name for the debugging configuration, which can be anything you'd like, such as `CF8 localhost` or `CF8 Test Server`. It's simply a name you'll use later when launching a debugging session and typically would be a name to identify the server against which you mean to do debugging. Then select the appropriate server you want to connect to for this debugging session, using the RDS Server dropdown list of options. These are those defined above in the section, "Configuring RDS".

That's all you really need to do. You need not set the Debug Home Page URL. It's used only for the Debug Output Buffer feature, discussed later.

Finally, note as well that the option offered here for Mappings is simply an alternative interface to the Debug Mappings preferences page discussed in the previous section. Any values you entered there will be shown here, and vice-versa.

Having entered a name and chosen an RDS server, you could click Debug to cause Eclipse to attempt to start a ColdFusion debugging session for the selected server, but let's do that in the next section. For now, just click Apply and then Close to save these settings.

Ok, that really was a lot of introduction to get to debugging. I'm sure some are wondering why it's so complicated. It's not really. Again, most of the steps to this point are one-time configuration settings. Once set, you'll never do them again. But they did need explanation, as a lot of the documentation and other resources out there run through the steps rather superficially, leaving a lot of open questions. I hope I've answered them for you.

Using the Debugger

We're now finally ready to debug a CFML page (`.cfm` or `.cfc` file). In this section we'll learn how to set breakpoints (places where control should stop while debugging), start a debugging session, step through code, and observe (or change) variables, among other things.

To recap, in order for a CFML page to be debugged, three things are necessary: the ColdFusion server on which the code is running must be configured to permit it, a developer must have Eclipse open and the debugger started (typically with a breakpoint set for the file in question, though the Break on a CFML Runtime Exception can stop on a line even without a breakpoint being set.)

Opening a file (within a Project)

You'll most typically start debugging by setting a breakpoint, and to do that, you need to open the file. Actually, in order to set a breakpoint on a file, the file must be defined as being part of an Eclipse project.

If you've already been using Eclipse for a while and already have a project defined for your code, just open the file as you normally would and skip to the next section.

NOTE

If you find that when you open a CFML file in Eclipse it opens instead in Dreamweaver or another editor, use Window>Preferences>General>Editor>File Associations and select CFM (and/or CFC) file, and in the pane for associated editors, use the Add button to select Text Editor, or CFML Editor if you've installed CFEclipse.

Defining a New Project

If you're new to Eclipse, you may not yet have defined a project, which for our purposes is simply a way for Eclipse to focus on a subset of your file system.

To see if you have any projects already defined, open the Navigator, using Window > Show View > Other > General > Navigator. If you have a project that points to the file you're interested in, double click the file to open it.

To define a new project, use the menu command File > New > Project > General > Project and click Next. Choose a project name (any descriptive name), deselect the Default Location option, and use the Browse button to point to the directory where your code is located (such that when browsing the code, ColdFusion will find it in that directory), then click Finish. (If you have CFEclipse installed, you can choose to create a new CFEclipse project instead. Files can be debugged from that just the same.)

In Figure 52.8, I have defined a new project pointing to the ColdFusion 8 `Getting Started Experience` application (downloadable at `http://www.adobe.com/devnet/coldfusion/eula_cf8gettingstarted.html`). You could for your own project choose instead to point at your Web site `docroot`, for instance.

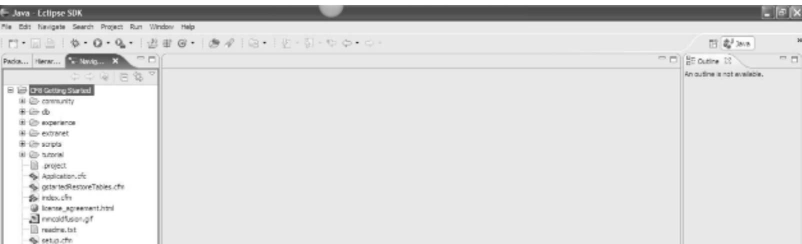


Figure 52.8

Viewing the ColdFusion 8 Getting Started Experience as an Eclipse Project

Switching to the Debug Perspective

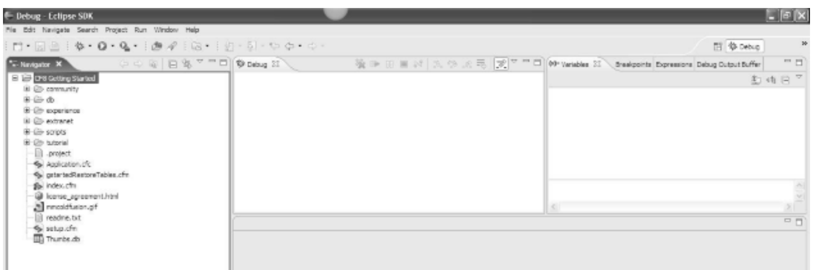
If you've installed and opened Eclipse for the first time, then you may notice (as in Figure 52.8) that the top right corner of Eclipse shows an icon labeled Java. If you have installed CFEclipse, it may say "CFEclipse." These icons represent what Eclipse calls perspectives. They define a pre-configured layout of the screen to show tabs and window panes that are appropriate for a given kind of development work.

In our case, we want to switch to what Eclipse calls the Debug perspective. You can switch perspectives by clicking on any icons shown in that top right corner of the screen, or on the double-arrow icon to their right, or by using Window > Open Perspective. If a perspective you desire is not listed, it may be under the available Other option shown in that list, but in our case the Debug option is listed, so choose that.

You'll notice that the interface changes (see Figure 52.9) and now we see panes that are related to doing debugging. I'll explain these panes and their use in the remainder of this chapter.

Figure 52.9

The Eclipse Debug Perspective



Opening a File

When we switch to the Debug perspective, the Eclipse Navigator remains displayed on the screen, so if we want to debug a page we need to open it. Drill down in the project directory tree to find the file and double-click to open it. The file will appear in an editor like that shown originally in Figure 52.1 above.

You'll notice that line numbers are shown, which may not be enabled on your display. These are very helpful, especially for debugging. To enable the display of line numbers, use Window > Preferences > General > Editors > Text Editors and choose the option, Show Line Numbers.

Setting a Breakpoint

With a file open, you can identify a line of code on which you want the debugger to stop when it reaches that line during execution of the request. This is called setting a breakpoint. You can right-click on the area to the left of the line (either the line number or the gray area left of that) and choose Toggle Breakpoint.

When you do that, a blue dot will appear to the left of the line (and line number), depicting that the breakpoint has been set. Note that it only makes sense to set a breakpoint on a line with CFML (tags or expressions), though Eclipse won't stop you from trying.

You could also set a breakpoint using the keyboard, by selecting the line and pressing control-shift-b. However once you've set the breakpoint, if you repeat the action, the breakpoint (and the blue dot) will be removed.

The breakpoint(s) you set will also be depicted in the breakpoints tab (see the top right of Figure 52.9), where they are displayed with their filename and line number.

Note finally that breakpoints remain enabled across editing sessions. In other words, if you close Eclipse and re-open it, the breakpoints you set previously will remain enabled. That said, breakpoints are specific to your editor, so if another developer opens the file, they will not see breakpoints you have set.

This may lead to a question: if breakpoints remain enabled across editing sessions, does that mean that a request will always be intercepted once a breakpoint is set? Not quite. Setting a breakpoint is the first step. The far more important step is to start a debugging session.

Starting a Debugging Session

Even with breakpoints set, the debugger doesn't do anything until you begin a debugging session. To do that, you use the Run > Debug command, which will open the same interface that we saw in Figure 52.7. There we defined the debug configurations we planned to use, now we will use them. Select the one that for the location of the file you have opened and press the Debug button in the lower right corner.

TIP

Note that you can also start a debugging session by clicking the icon (just above the file Navigator) that looks like a bug. If you've not previously used any of your debugging configurations, you'll be presented the same dialogue as above to select one. If you have previously selected a debugging configuration, however, clicking this button will instead just restart that previously selected debugging configuration. And when you've selected more than one debugging session, you will then be able to select one of them quickly by clicking the down arrow icon just to the left of the bug icon, which will present a list of previously selected debugging configurations. Choosing one will start that debugging configuration. Note that the menu offered also offers a Debug option which also opens the interface in Figure 52.7.

Assuming that all is configured properly, this should initiate a debugging session between your editor and the ColdFusion server. Figure 52.10 shows the kind of information that might be displayed in the debug window, which shows the debugger ready to accept requests for files in the selected debugging configuration. (I've selected to start my debugging configuration named `cf8 localhost`. Technically, the value shown on the first line (with the ColdFusion icon to its left) is the name of the selected debugging configuration, and the next line below it is the name of the RDS configuration used within that configuration.) The debugger is now ready for us (or someone) to execute our desired requests in order to be intercepted and debugged.

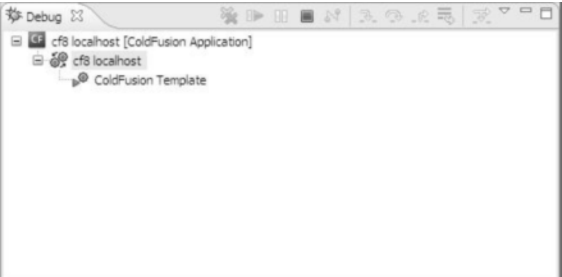


Figure 52.10

A successfully started Eclipse debugging session

When Starting a Debug Session Fails

Your attempt to start a debugging session can fail for a number of reasons. What are some of the things that could go wrong?

- **ColdFusion isn't started**: You'll get a popup saying, "Connection refused: connect," followed by another starting with "Unable to connect to the RDS server…"

- **RDS wasn't enabled in the ColdFusion server**: You'll get a popup starting with, "Unable to connect to the RDS server…" and continuing with an error message including "JRun Servlet Error." Again, the ColdFusion 8 debugger works only via RDS, so the server being debugged must have RDS enabled. You can learn how to enable it at http://www.adobe.com/go/tn_17276.
- **The RDS authentication information provided was invalid**: You'll get a popup starting with, "Unable to connect to the RDS server…" and continuing with an error message, "Unable to authenticate using the current username and password information." Correct the RDS connection information as shown above in the section, "Configuring RDS." (In both this case and the last case, you may have solved these problems sooner by using the test connection discussed there when you created the connection.)
- **Allow Line Debugging wasn't enabled in the ColdFusion Administrator**: You'll get a popup starting with, "Unable to connect to the RDS server…" and continuing with an error message "ColdFusion Line Debugger was not enabled." Return the ColdFusion Administrator, as discussed in the section "Enable Debugger Settings," and ensure that Allow Line Debugging is checked. You do not need to restart ColdFusion for this change to take effect.

Browsing a Page to Be Debugged

With a debugging session enabled, it's now time to run your page to experience debugging. This step can actually be more confusing for some developers than any of the configuration features to this point. Most debuggers have some set of configuration that must be done, as above, but then with most debuggers the next step is to run the desired program in some sort of special debugging window. That's not the case with the ColdFusion 8 debugger.

Instead, you simply run the CFML page request just as you normally would, whether from a browser, or via a Flex or Ajax client, or as a web service call from some other environment, and so on. There's no special debugging window in which you must run your request. (There are available internal browsers provided by Eclipse, as well as the Adobe ColdFusion 8 Extensions for Eclipse and CFEclipse as well, but you don't need to use those.)

NOTE

> Indeed, since the debugger intercepts any request for a CFML page, it can even intercept some requests that have no browser or client at all, such as if you debugged the `OnSessionEnd` method of `Application.cfc` (which would be executed when a session ended), or in a ColdFusion Event Gateway like the `DirectoryWatcher`, which would be executed when there was a change in the directory it was observing.)

If you've got an Eclipse debugging session enabled against a server, with a breakpoint set in a file, then when that CFML code is executed, whether by you or anyone else, the debugger will stop on that line of code.

NOTE

That may be another surprise, that the debugger intercepts any request for a CFML page, regardless of whether you request it or someone else does. This means that in an environment with multiple users making requests, you could intercept someone else's request. That could be desirable or unexpected. Just be aware.

What You, the Developer, See

Figure 52.11 shows how the Eclipse environment will reflect that you're stopped at a breakpoint.

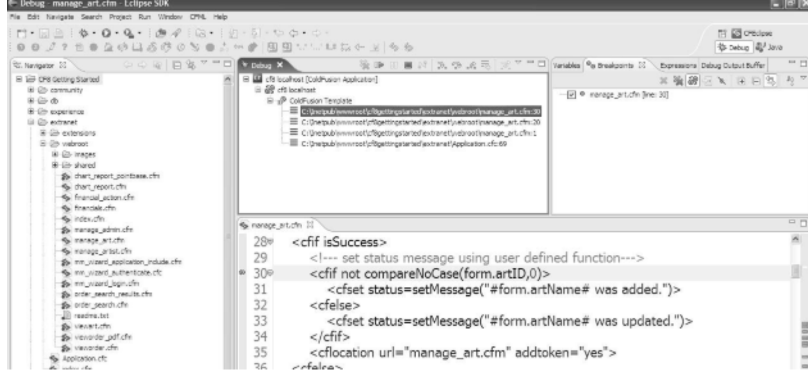


Figure 52.11

Execution stopped at a breakpoint

First, notice that the Debug window at the top now reflects more information than we saw in Figure 52.10. It shows the complete path to the file being debugged and the line number at which execution has stopped. This complete path can help if you ever wonder what file you're looking at/debugging.

Note also that in this case we see other lines there, which reflect what some may call the stack trace—or how the code we're stopped on was called by other code. For instance, if we were stopped in a CFC, include file, or custom tag, it would show the file (and line number) of the code that called this file. In our case, we can see, among other things, that our execution also included running the `Application.cfm` in the directory one level above our current one.

Second, in the code window, you can see that where Figure 52.1 had showed only a blue dot next to the line on which a breakpoint had been set, now we see also a blue arrow overlaid atop it. (This may be too small to observe in the screenshot.) The arrow is a current instruction pointer, which reflects where the debugger has stopped execution. It shows the line that is about to execute. We'll learn about stepping through that and other code shortly.

(You may have observed that in this screenshot, as in Figure 52.1, the CFML code shown is color-coded, with CFML tags and attributes in red, attribute values in blue, CFML comments in gray. These are features enabled by CFEclipse. If you've not installed that, you won't see that coloring, but all features of the debugger work the same.)

What the User Sees

While we sit here looking at the debugger, observing the fact that the debugger has intercepted the request allowing to look at the information above (and more to be discussed soon), you may—and should—wonder what things look like to the user making the request. If it's a regular browser user, what they'll see will appear as if their request is hung, as if the server is not responding, and in fact it's not. We've held up the request's processing so there is not yet a response for the user to see.

Just as when a user's request is waiting for any response from the server, they will see various indicators that they are waiting, which will change depending on the browser being used. See Figure 52.12 for an example in Firefox, where I've highlighted with arrows the 3 ways that that browser shows that a page is waiting to complete loading. This is the page that triggered the debugging session in Figure 52.10. It's the submission of a form in the Getting Started Experience application:

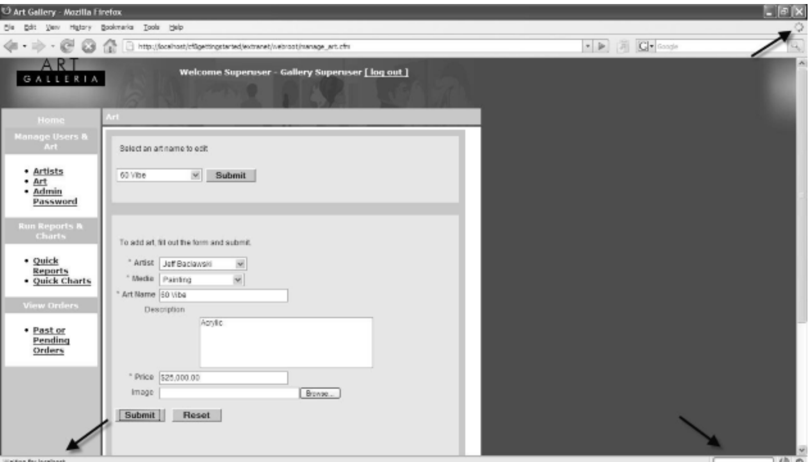


Figure 52.12
User waits for page execution to finish

You may wonder, if the page had started to generate output before the breakpoint, whether that output would at least be sent to the browser. It is not. As with a normal ColdFusion request, the page output is buffered internally and all sent only when the page completes (unless a CFFLUSH causes it to be sent sooner). If you're interested in being able to see what partial page output has been created in the output buffer, that's a feature enabled by a feature of the debugger, discussed below under, "Observing the Debug Output Buffer."

Of course, this discussion has assumed we were referring to a normal browser request calling the page. If the page had been an Ajax or Flex client request, those two would show the request waiting, typically using some sort of prompt to the user (which might normally flash by in a split second under normal circumstances).

Beware Page Timeouts

It's worth noting that it's possible that you could leave the browser wait so long that it would consider the page abandoned and could show a message indicating that it stopped waiting.

Readers familiar with ColdFusion, however, may wonder how ColdFusion will respond to our making a request take so long while we sit in the debugger watching the execution of the request (and later, stepping through it, watching variables, and so on.)

The concern may not be as important as it seems. It's true that ColdFusion offers an option in the Administrator (Server Settings > Settings) to Timeout Requests After (Seconds), where you can enter a value for the number of seconds that you want ColdFusion to attempt to cancel the request if it exceeds that duration. And fortunately, this is an option which you can enable or disable (using its checkbox), so that when you're doing debugging against a server, you could do so.

But in my testing, it doesn't seem that time spent sitting at breakpoints is accumulated against that server timeout limit. Perhaps it's only that the actual time spent running the code is charged against that limit, which would be very clever on the Adobe engineering team's part, and much appreciated.

If this does turn out to be an issue, and if for some reason you can't get the Admin console setting changed, keep in mind that there's yet another option to prevent timeouts on a page-by-page basis. When the normal execution of a page may be legitimately expected to exceed the server timeout, you can use the `CFSETTING` tag and its `RequestTimeout` attribute, to specify a number of seconds that the page should be permitted to run before timing out.

Stepping Through Code

We've stopped the debugger on the line of code set as a breakpoint. There are many things we can do at the point, as we'll see in upcoming sections, but often the sole reason for using the debugger is to observe the flow of execution through the code (and among the files that the code may call upon). There are many beneficial reasons to step through the code.

Perhaps you're trying to understand why certain code is (or is not) being executed. Or maybe this is new code that you've been asked to maintain (or debug), and you want to use the debugger to become more familiar with its operation (versus statically reading the source code by hand.) And the fact that the debugger can open files that are called upon (such as includes or CFC methods) means that it's also an excellent tool to know where your code is going (and what files are being opened. Recall that Figure 52.10 showed you the full path to the file being debugged.) Finally, you may be new to ColdFusion in general, and the debugger is an excellent tool to help you become familiar with how CFML works.

You can step through the code on a line-by-line basis, or you can run until a next breakpoint is reached. And if you've stepped into some other file, you can have the debugger complete execution of code in that file and stop at the next line in the caller after line which called the other file.

Stepping Line by Line: Step Over, Step Into

The most common situation is that you want to have the debugger just proceed to execute the next line in the flow of execution. Here, though, you have two choices.

The first is to step over the line, which means simply to execute it and set the current instruction pointer to the next line that would follow it. That can be done using Run > Step Over, or the F6 key, or using the Step Over icon listed among the icons atop the debug window, as depicted in Figure 52.13:

Figure 52.13
The various stepping icons

The second option is step into (Run > Step Into, or the F5 key, or using the Step Into icon shown in Figure 52.13. If the line of code about to be executed would call another file (such as a CFINCLUDE, custom tag call, or CFC method invocation), you could use step into to have the debugger open the file and stop on the first line of CFML code in that newly opened file.

Note that you don't need to tell the debugger where the file is. That's one of the very powerful features of the debugger: it figures out the file to be opened by the same logic that ColdFusion would use. This is a great feature when you have any doubt about what file is being opened in such instances (which may not always be obvious to you, due to ColdFusion mappings and other mechanisms by which ColdFusion may find and open a named file.)

Note that if you don't want to follow the flow of execution into the file that would be opened by the line of code about to be executed, you don't have to. You can use Step Over in that case, and the code within the file will be executed (just as during normal execution of the page) and the current instruction pointer will be set at the next line in this page (unless there are no more lines to execute in which case it will terminate, or if you've stepped into a file, it will return control to the caller).

If you do choose to use Step Into on a tag that opens a new file, note that once you're inside that new file, you have the same options discussed above, so you can use Step Over and Step Into, as well as a new option, Step Return, discussed next.

Caution

> You should be careful about using Step Into when you don't need to (meaning, don't use it on tags or expressions which would not otherwise have any reason to open a new file). For one thing, it can cause the request to take longer than it would with a Step Over. More important, on some tags, using Step Into will actually try to open an underlying file unexpectedly. Some CFML tags, like `CFDUMP`, `CFSAVECONENT`, and a few more, are actually CFML pages that ColdFusion executes (see the `\ColdFusion8\wwwroot\WEB-INF\cftags\` directory.)

Step Return and Resume

If you do step into a new file, and you decide you no longer want to step line-by-line through the code, you don't need to laboriously make your way through the remaining lines of code. That's what Step Return is for (Run > Step Return, or F7, or the appropriate icon in Figure 52.13.) That option completes execution of the file that's been stepped into (unless there's another breakpoint later in the execution of this file that's been stepped into) and returns control to the line of code that follows whatever called it. All the code is executed, of course. You just don't step through it.

Similarly, if at any point while debugging a request you decide that you no longer want to step through the code (whether you've stepped into a file or not), you can have the debugger proceed to

execute the entire remainder of the request using Resume (Run > Resume, or F8, or using the appropriate icon). Unless there's another breakpoint in the flow of execution, the debugger will finish and the completed page will be shown to the browser (or whatever made the request for the CFML page).

The final feature listed in Figure 52.13, Terminate, is discussed in the final section of this chapter, on stopping the debugger.

Observing Variables and Expressions

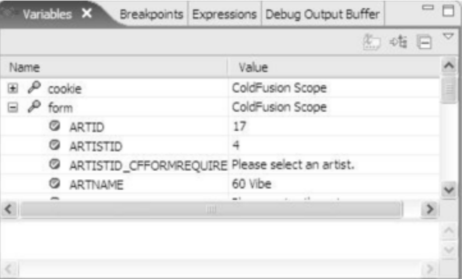
For some developers, just being able to have a granular, line-by-line depiction of the flow of execution through their pages is miraculous enough. But there's more. While you're at a breakpoint, you can also observe the value of any variable, in any of ColdFusion's scopes. This includes all the usual scopes (form, URL, session, request, and so on), as well as things like the arguments scope when in a function, the this scope as created by a CFC, and so on.

The Variables Pane

The first place to look is the Variables pane, which appears to the left of the Breakpoints pane as shown in Figure 52.11. If you select it, you can traverse the tree of available scopes, expanding them to see the name and value of any variables in the selected scope. See Figure 52.14, where I've selected the Form scope. Since this code is processing a form submission, you can see all the form fields that were passed to the request.

Figure 52.14

Display of Form variables



The depiction of variables in this pane is of course not limited to the simple variable values shown above. If the code had executed a query, we could expand the query to see all its related variables (`recordcount`, `columnlist`, etc., and even each row and column in the resultset). And if we created an instance of a CFC, that also could be observed, including all its metadata. See Figure 52.15 to observe some of that data.

TIP

Here are a couple of tips that could make working with the Variables pane more effective. First, if the value of the variable you wish to observe is too long to be displayed in the Value column provided, while you could select the column divider to make it wider, a simpler solution is that if you select the variable, you'll notice (as shown in Figure 52.15) that the value is also displayed in a Details window at the bottom of the Variables pane. This can be more easily scrolled to view the full value.

Also, while in the Variables pane, you can of course scroll up and down in this list to see all the available variables in the selected scope. But note that you can also use an available Find feature, which can search through all the scopes to find a variable of a given name. Right-click the variables pane and choose Find, or left-click on it and use Ctrl-F.

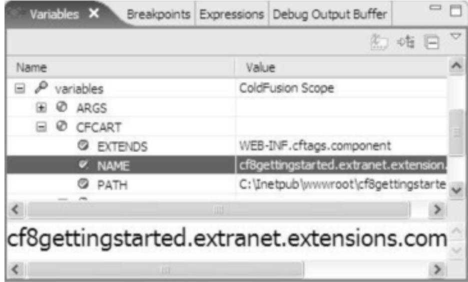


Figure 52.15
Display of a CFC instance

Remember, too, that the list of scopes shown is limited to those you chose when you set up the Debug Settings preference, as discussed previously in the section of the that name. You may recall from that discussion that you should be careful about adding too many scopes, as that can slow the execution of the debugger. Another option is that you can instead ask the debugger to show you some specific variable using the Expressions pane instead, discussed next.

The Expressions Pane

Consider that your interest in observing a variable value is to be able to watch how it changes over lines of code as you step through it, or perhaps over multiple requests for the page. While you could traverse the Variables pane to find and display your desired variable each time, you could instead use the available Expressions pane. This has the benefit that you choose to name what variable you want to watch (including one in a scope that is not being displayed in the Variables pane because of Debug Settings preferences.)

To add a new expression to the pane, right-click on it and choose Add Watch Expression, then provide the name of the variable. You don't need to use the pound signs (but it's ok if you do). The variable you name will be displayed in the pane (in the order in which you add them). When you're stopped at a breakpoint, its value will be shown—assuming the variable is defined at that point in the code. If it's not, the variable will be shown in red. (If you ever find that a variable which should be defined is not displaying, right-click it and choose Reevaluate Watch Expression.)

Note that you're not limited to naming variables with simple values. You can even name an entire scope and all its values will be displayed. You can also create a new Watch Expression by right-clicking on a variable in the Variables pane and choosing Create Watch Expression.

Another benefit of the Expressions pane is that, like breakpoints, the items you create here remain across debugging sessions and editor restarts. And like the variables pane, you can use the Find command to search among them.

Changing Variables on the Fly

Here's a feature that will amaze some readers. Did you know that you can change the value of a variable on the fly, while the program is being debugged? Yes, you can. While at a breakpoint, you can right-click on a variable name in either the Variables and Expressions panes and choose Change Value. It's really that simple. The value will be changed for the remainder of the execution of the request (or until you or some other code changes it again.)

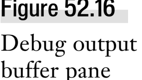
Observing the Debug Output Buffer

Finally, the last feature to discuss is the Debug Output Buffer pane. I mentioned it previously as a way that you can view the generated output of the CFML code (whether you're generating HTML, XML, JSON, or whatever.)

The pane appears to the right of the Expressions pane. Within that, select the Server Output Buffer tab (I'll discuss the Browser tab in a moment).

Server Output Buffer

The Server Output Buffer is itself comprised of two panes. See Figure 52.16:



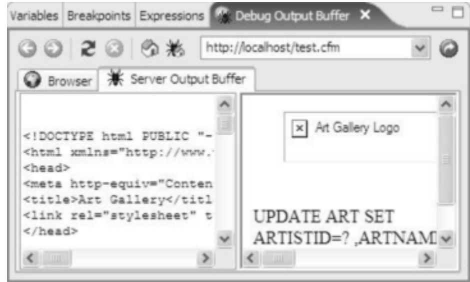


Figure 52.16
Debug output buffer pane

The left pane shows the raw generated browser code (HTML, in this case) which the code has generated to this point. You could scroll down and see that it's not a complete page, since the request is in the middle of generating the content. Further testament of that is in the right pane, which shows an attempt to render that HTML in a built-in browser.

As you step through the code, you'll be able to observe the building of the page as it might appear in the real browser. This is just a simple rendering of the HTML in a temporary file created by the editor. It's not really executing in the context of the URL of the server, so some things (like the image shown) may not render correctly.

Also, don't be misled by the URL displayed atop the pane. That's not the URL of the page being debugged (unless you typed a URL there). That address bar is really there for the sake of the other window in this pane, the Browser tab.

Browser

I had mentioned previously that to trigger the debugger you could make a request from any browser, and I'd mentioned that there were some internal browsers available within Eclipse. This is one of them. If you wanted to, you could make your page requests using this address bar and the result will be shown in the Browser tab of this pane (not in the Server Output Buffer pane. Indeed, it wouldn't really make sense to use that address bar while you're viewing the result of a page being debugged.)

It's a real browser. In fact, the URL that's shown (unless you entered one yourself) is the very value of the Home Page URL that was mentioned briefly in the discussion of Debug Settings, Figure 52.5. (So that's what that's for!) If you click the home page icon atop that Browser pane, that URL will be loaded. And what about the bug-like icon to the right of that? That URL will come from the Debug Configuration page (Figure 52.7) where it was available as the Debug home page URL. Now you see why I ignored them at the time, and to be honest, many will never bother to use this internal browser.

That said, I should clarify that some might not care for it because the window is too small. But here's a tip: as with all the panes discussed here, you can resize this one and even detach it so it floats as a window over top the Eclipse interface.

Stopping the Debugger

Finally, when you're done debugging pages, you may wonder how you stop debugging. There are several ways, each with slightly different purposes.

Terminating Debugging Within the Debugger

Well, first, if you close Eclipse, that will terminate your debugging session, so fortunately, you can't leave it running by mistake (well, not just by closing it without stopping it.)

You can also stop the debugging session while you remain in the editor. There are a few ways. You can use the Terminate button, shown in Figure 52.13. That stops the debugger and lets the request run to completion. It's also available as Run > Terminate.

You can also right-click in the Debug pane (where the filename and line number of what's being debugged is shown), and there choose Terminate. That will stop the debug session and leave an indication in the Debug pane that you had used that debug configuration. That way you could right-click and choose Relaunch as another way to start a session. You can also right-click and use Terminate and Relaunch, if for instance you had made changes in the debug configuration. Finally, there is also an option there, Terminate and Remove, which will remove the debugging session from the debug pane.

Forcing Termination from the Server

I mentioned above that closing the editor would terminate any debugging sessions. But what if you instead mistakenly left the Editor open and a debug session running while you left for lunch—or for

the day? This could be a real problem. Users might run a request for a page you're debugging, and at least the first of those would have their request hang waiting for a debugging activity that might not happen for hours.

Here's where it's important to note that there's one final way to stop debugging which can be effected from the server, rather than from the editor. Recall the discussion of the Admin console page for Debugging & Logging > Debugger Settings. I mentioned the available Start Debugger Server button, which once debugging begins it becomes a Stop Debugger button, and an associated Restart Debugger Button appears.

This is the situation where you would use those buttons. You could either use Stop Debugger (and then Start Debugger) or just use Restart Debugger. That will terminate all current debugging sessions, thus freeing up the requests that were pending waiting for a debugging session to complete. Even if you'd used Stop Debugger, the first request for a debugging session from a developer would restart the Debugger Server anyway.

Think of the Stop or Restart as like a forced termination of debugging from the server. All developers who were performing debugging will have their debug sessions terminated, so it's a bit brute force, but they can easily restart them. The good news is that they do need to manually restart them, so the person who left their editor (and debugger) running when they left will no longer be the cause for intercepted requests.

If you wonder how to stop all debugging from happening any more on the server, clearly that's not what the Stop Debugger Server does (since the next request for a debug session will start it up). Instead, turn off (uncheck) the option on that Admin page for Allow Line Debugging. That takes effect immediately, and ColdFusion would no longer permit debugging requests against that server.

It's great to see how in so many ways the Adobe Engineering team has thought ahead to help not only provide a debugger but one that addresses some common challenges that CFML developers would face. For many, the biggest challenge of using a debugger is simply remembering that it's there. I hope this discussion has motivated you to consider it and to use it.

Coding Standards 1
Documentation 8
Version Control 8

CHAPTER 53

Managing Your Code

Coding Standards

When developing an application with Adobe ColdFusion, remember that *how* you code can sometimes have as much of an effect on your application's performance as *what* you code. Opinions differ as to which coding methodology is the best choice and how one can—and should—perform certain CFML operations. In truth, the best methodology is simply to have a methodology. Chapter 54, "Development Methodologies," online, discusses some of the more prevalent ColdFusion development paradigms in greater detail.

Before you implement a specific approach, remember that your choices as a developer can hurt your application's performance. Even though you can code exactly the same functionality a hundred different ways, you should consider some general guidelines when writing your code to keep your application performing at the best level possible.

The coding standards covered here outline some issues you should be cautious about as you develop any application with ColdFusion. This chapter describes some of the most common things you can do as a developer to improve application performance. As you gain experience and confidence, you can add your own suggestions to this list.

The guidelines are as follows:

- Separate database queries, logic, and processing-oriented code from presentation-oriented code.
- Avoid overuse of subqueries in a select statement.Use `<cfqueryparam>`
- Don't reassign variable values with each request.
- Don't use if statements to test for the existence of variables.
- Use `<cfparam>` at the top of your templates.

- Always scope your variables.
- Use `<cfswitch>` and `<cfcase>` in place of `<cfif>`.
- Don't overuse the # sign.
- Avoid overuse of the `<cfoutput>` tag.
- Reduce white space in your output.
- Comment, comment, comment.

Separate Processing-oriented Code from Presentation-oriented Code

Much of the code you'll write in ColdFusion templates can be classified as either business/processing-oriented or presentation-oriented. Business/processing-oriented code includes queries, parameter validation, variable definition and manipulation, and calculations, which don't involve generating output. Presentation-oriented code is code that results in or formats output.

Your code will be more reusable, portable, and easier to maintain if you separate the business from the presentation. It's often best to segregate your business/processing-oriented code into CFCs. At the very least, you should put business processing logic at the top part of your CFML templates followed by the presentation-oriented code.

This will make your code easier to port it to a different type of presentation interface (e.g., from HTML to WML). It will also make it easier to skin your app—give it a new look and feel.

Avoid Overuse of Subqueries in a Select Statement

Using subqueries within a select statement can sometimes cause a database to create an inefficient execution plan for the overall query. The database has no way to know what the results of the subquery contained in your SQL statement will be, so it's forced to examine the table as many times as there are possible subquery results.

If you can break a large query containing a nested query into two or three smaller queries, you will generally see page execution time improve.

USE `<cfQueryparam>`

It's good to get in the habit of using `<cfqueryparam>` when you're using `<cfquery>`. There are two purposes served. If your database supports bind variables (as SQL Server does, for example), use of `<cfqueryparam>` enables ColdFusion to use bind variables in the SQL statement. This will enhance performance.

This tag also enables you to enforce data typing to some degree and this will somewhat mitigate SQL hacks, making your database a bit more secure.

Don't Reassign Variable Values with Each Request

Developers commonly define certain variables—a data source, for example—inside the `application.cfc` file. There's nothing wrong with doing this, but keep in mind when working with your `application.cfc` file that it loads with each and every page request. Because of this, you should avoid forcing ColdFusion to reassign values to all of the variables inside the `application.cfc` if they are already present in `application.cfc`.

One way to avoid this unnecessary processing is to encapsulate variable assignments within a `<cfif>` statement. This way, you can check whether a variable value has already been assigned instead of asking ColdFusion to perform this work again. The following example assumes that you are assigning a value to the variable `application.datasource` within your `application.cfc` page. It demonstrates the concept of testing for the existence of this variable and writing a value for the variable only if one isn't already present:

```
<cfif not IsDefined("APPLICATION.DataSource")>
 <cfset APPLICATION.datasource = "MyDSN">
</cfif>
```

Note that if you assign a group of application variables in the same location in your app (a good idea!), and if you're concerned about race conditions, you should lock the whole block; but you needn't test for the existence of each variable as they're all defined together. Just test for the existence of one and if it doesn't exist, then none of them do. In that case you go ahead and assign all of them.

Don't Use If Statements to Test for the Existence of Variables

If you know you need to evaluate a specific variable inside a ColdFusion template, you likely must check to see whether that variable is already present. If it isn't, you should create it and assign it a default value. You can do this with the `<cfif>` statement, as shown in the following code:

```
<cfif not IsDefined("myCat")>
  <cfset myCat = "Carbon">
</cfif>
```

This method of testing for the existence of a variable will work, but it's not the most efficient way. The ColdFusion Markup Language provides an even easier method of testing for the existence of variables and assigning default values. The following example accomplishes the same thing as the previous `<cfif>` evaluation, but much more efficiently:

```
<cfparam name="myCat" default="Carbon">
```

Use `<cfparam>` at the Top of Your Templates

You can avoid a lot of business-logic problems by checking for required parameters at the top of your templates. Use `<cfparam>` to check for the existence of `FORM`, `URL`, `ATTRIBUTE`, and any other variables that your template requires. You can also use `<cfparam>` to validate data types.

Always Scope Your Variables

Forgetting to include the scope with each reference to a variable is easy, and it can get you into a lot of trouble. This can become a particular problem if your application uses a lot of `<cfinclude>`s. The problem arises when you inadvertently use the same variable name in more than one scope, e.g., you use both `FORM` and local variables with the same names but they are processed differently in your application. When you do this—and neglect to explicitly identify the scope—your app could wind up processing one of the variables the wrong way. This also makes your code harder to follow and maintain.

Wrong:

```
<cfif orderID GTE 10>
  …
</cfif>
```

Right:

```
<cfif FORM.orderID GTE 10>
  …
</cfif>
```

Note that when you are creating local variables in a template, it's not essential that you specify the `VARIABLES` scope, because that is what will be used by default, and any ColdFusion developer will see that they are local variables. It's when you refer to the variables later in your code that you must be sure to scope them.

Use `<cfswitch>` and `<cfcase>` in Place of `<cfif>`

If you want to have several pieces of code run depending on the value of a single parameter, you might assume that you should just create multiple `<cfif>` statements to check for the value of the desired parameter, and then run the code inside your `<cfif>` if that value is met. To demonstrate how this would work, let's think about a task that many developers replicate when first working with ColdFusion. Suppose you are returning a list of users from a database. You want to list these users in alphabetical order by either first or last name, depending on the client's preference. To accomplish this, you must allow the user to pass in a simple parameter, perhaps on the URL, telling your application to order the results by either last or first names. You could accomplish this as shown in the following example:

```
<cfquery name="ListTheUsers" dataSource="MyDSN">
SELECT User.FirstName, User.LastName
FROM tbl_Users
<cfif URL.OrderBy EQ "LastName">
 ORDER BY LastName DESC
</cfif>
<cfif URL.OrderBy EQ "FirstName">
 ORDER BY FirstName DESC
</cfif>
</cfquery>
```

Though that code will accomplish your goal, ColdFusion provides a more efficient way to complete this type of evaluation: using the <cfswitch> and <cfcase> tags, as shown here:

```
<cfquery name="ListTheUsers" dataSource="MyDSN">
SELECT User.FirstName, User.LastName
FROM tbl_Users
<cfswitch expression="URL.OrderBy">
 <cfcase value="LastName">
 ORDER BY LastName DESC
 </cfcase>
 <CFCASE value="FirstName">
 ORDER BY FirstName DESC
 </cfcase>
 <cfdefaultcase>
 ORDER BY LastName DESC
 </cfdefaultcase>
</cfswitch>
</CFQUERY>
```

Both of these examples produce the same results, but using the <cfswitch>/<cfcase> tags doesn't force evaluation of logic unless the specific watched-for conditions are met, making this method more efficient than a long list of <cfif> statements.

Don't Overuse the # Sign

As you get started with ColdFusion, one of the characters you will type most often is the # sign. Surrounding a variable, function call, or variable-function combination with the # sign flags that bit of code so that ColdFusion knows it's a variable to evaluate. Otherwise, the ColdFusion engine would have no way of distinguishing what you wanted it to evaluate from other text within your CFML templates.

When you use a variable as a parameter for a standard CFML tag, the # signs are not necessary. Take the following example:

```
<cfif #VARIABLES.MyName# EQ "John">
 <!--- Some conditional ColdFusion code --->
</cfif>
```

This is the same thing as:

```
<cfif VARIABLES.MyName EQ "John">
 <!--- Some conditional ColdFusion code --->
</cfif>
```

Notice how having the # signs around the MyName variable makes no difference in how the code is processed.

Now, let's assume that you want to run a special bit of code if in fact MyName is John. The following example demonstrates this:

```
<cfif VARIABLES.MyName EQ "John">
 <cfset FormAction = "updateData.cfm?MyName=#URLEncodedFormat(Variables.MyName)#">
</cfif>
```

Notice how when you want to use the MyName variable as part of a URL, you must use # signs to let ColdFusion know that you want this variable evaluated so that the string John will be passed to the page you're calling. Without the # signs here, ColdFusion passes the string MyName to the next page. By the way, when you include variables values in a URL, be sure to URL-encode them as well; this will reduce errors when the URL variables need to be processed at some later point in your application.

Avoid Overuse of the <cfoutput> Tag

In general, use as few <cfoutput> tags per page as possible. Overusing the <cfoutput> tag can put undue strain on the ColdFusion parser, and under heavy load conditions this may slow your application. Try to combine code that requires <CFOUTPUT> statements into a single <cfoutput> block as much as possible. This example demonstrates overuse of <cfoutput>:

```
<input type="text" VALUE="<cfoutput>#VARIABLES.Variable1#</cfoutput>">
<input type="text" VALUE="<cfoutput>#VARIABLES.Variable2#</cfoutput>">
<input type="text" VALUE="<cfoutput>#VARIABLES.Variable3#</cfoutput>">
```

This code will work as you would expect it to, but the multiple output statements create a lot of extra work for ColdFusion's parser. Here is a much better way to achieve the same result:

```
<cfoutput>
 <input type="text" value="#VARIABLES.Variable1#">
 <input type="text" value="#VARIABLES.Variable2#">
 <input type="text" value="#VARIABLES.Variable3#">
</cfoutput>
```

This said, beware of going to the opposite extreme. It's also a bad idea to force ColdFusion to parse a lot of content that doesn't include any CFML. In other words, if you have large blocks of HTML or text that contain only a few CFML expressions, you should use individual <cfoutput> tags, because ColdFusion must process every token within the body of the <cfoutput> tag.

Reduce White Space in Your Output

The more blank spaces your code produces, the more useless output users have to download. Your code should be easy to read and maintain, but avoid producing output with a lot of unnecessary white space. ColdFusion provides several useful tags for accomplishing this.

One option is to block chunks of CF code in <cfprocessingdirective>:

```
<cfprocessingdirective suppressWhiteSpace="Yes">
…
</cfprocessingdirective>
```

This doesn't affect the white space in your HTML, just in your CFML code.

Another choice is <cfsetting>:

```
<cfsetting enableCFoutputOnly="Yes" />
…
<cfsetting enableCFoutputOnly="No" />
```

This tag prevents *any* output from making it to the browser that isn't inside a `<cfoutput>85 </cfoutput>` block.

Another choice is `<cfslient>`. Its effects are even more dramatic:

```
<cfslient>
<cfoutput>Even I am not visible</cfoutput>
</cfsilent>
```

As you can gather from this snippet, `<cfsilent>` prevents output of everything within its start and end tags.

If you follow the advice offered in previous chapters, you're probably writing some of your code in ColdFusion components. If you don't explicitly set the `output` attribute in your `<cfcomponent>` to `No`, then any code in the constructor portion (the part that creates instance data) will be treated like CFML in a regular template and will result in white space. Unless you plan on outputting data in the constructor (not a good idea!), code your components like this:

```
<cfcomponent output="No">
  <cfset myFirstName="Joe" />
  <cfset myLastName="Shmo" />
  …
</cfcomponent>
```

Comment, Comment, Comment

Nothing is more frustrating than having to dig into another developer's code to resolve a problem or add functionality, only to discover that they haven't commented any of it.

Trying to follow the flow of uncommented code is like finding your way around an unfamiliar city without a map. You have no idea what the person before you was doing or what their thought process was in developing the code. This can make your job extremely difficult.

Be conscientious when developing your own applications, and comment your CFML code effectively. Place comments that will make it easy for someone to come along and pick up work right where you left off. Make your comments clear, concise, easy to understand, and relevant. Don't waste time restating the obvious, either. Good comments provide insight into *why* something is being done. For example, the comment below is of no practical value:

```
<!--- Set MyVar to part of YourVar which is needed in case
statement below --->
<cfset MyVar = Right(VARIABLES.YourVar, 3)>
```

Anyone can tell that `MyVar` is being set to part of the value `YourVar`. Whoever has to maintain this code will gain more insight from a comment like this:

```
<!--- Set MyVar to file extension portion of YourVar --->
<cfset MyVar = Right(VARIABLES.YourVar, 4)>
```

A few weeks after a job, you may not remember how you accomplished a specific task within your code. Having comments for guidance will help you recall what you were thinking when you wrote the code.

Every template you write should include a comment block at the top. It should identify the name of the module, author, date, input parameters (including data type, scope, whether or not they're required), output produced (return values, including data type and scope), and a log of changes to the module. The changes should include the author's name and date and brief description of what was done.

Documentation

By nature, the first thing any ColdFusion developer wants to do when presented with a project is open Adobe Dreamweaver and start coding away. That's understandable, because writing the code is the fun part—and is why most of us became ColdFusion developers.

Still, you must consider a few other things if you really want your project to succeed. One of the most important is documenting every step clearly and completely.

Project documentation is a deeply personal process. There really is no right or wrong in this regard. So long as you are aware that you must document your project completely, from start to finish, how you choose to do it isn't that important.

Table 53.1 outlines some key steps along the path to complete documentation of a project of any size to ensure that you have a complete record.

Table 53.1 Steps to Complete Documentation of a Project

ITEM	DESCRIPTION
List of Key Players	Make sure you have a record of all the people involved in the project, along with a description of their specific roles and responsibilities.
Application Diagram	One of the first steps in building a successful ColdFusion application is to lay out that application on paper before you begin to code. This way, by the time you start development you will already have solved many problems.
Problem and Solution Guide	As you start developing your application, you will undoubtedly encounter some unexpected and perhaps complex problems. Keep a good record of what these problems were and how you solved them. These documents both add to your understanding of the current application and provide a valuable reference should you run into similar problems in the future.
List of Resources	Keep a good record of all resources involved in the development of a specific application: in this instance, all CFML templates, their functions in the application, and their archived locations, as well as any data sources or third-party software used in conjunction with the application.

Version Control

In the development of any ColdFusion application, the ability to control access to the templates you're developing, to track changes, and to roll back to earlier versions if necessary is important to your successfully completing the application on time.

Developers often work in teams, sometimes dispersed over large distances. It's important to know that what you are working on doesn't affect what someone else has already done or will be doing soon.

Often, a developer will experiment with code, only to find that a change has just resulted in a template that no longer functions. Worse, someone can change a different template and cause your template to stop working properly. In that case, it's extremely helpful to be able to roll back to a stable version of the code.

Implementing some form of version control speaks to all these issues. It lets you know which developer made changes to any given template, regardless of where they are located. You can revert to previous versions of code if you encounter an unexpected error, and you can allow large groups of people to work on the same project without having to worry about their stepping on each other's work.

Planning for Version Control

Before you jump into implementing version control, ask yourself some questions about how you'll be designing, implementing, and managing your project. And consider all the people involved in working on it.

You can save yourself some time by answering the following questions before you implement a specific version-control solution:

- **Should I define this as a single project?** If the application you are developing is of any size, consider dividing the development into multiple subprojects beneath the banner of a larger, all-encompassing project. For example, you may be developing an administrative section of the site, as most developers do at some point. This type of functionality is a good candidate for a subproject that could reside in a layer below the larger application project.
- **Who are you working with on this project?** If you're implementing source control, you probably aren't the sole developer. Clearly define the role and responsibilities of each person involved with the development process so you can determine which members of the team need access to which resources. Try to avoid giving team members access to resources they aren't directly responsible for or on which they shouldn't be working.
- **Where will the source files reside?** Determine where you want to store the project database and source files. Choose a location convenient for all involved.

After you've answered all these questions, you can begin to think about what kind of version control you want to implement.

Version-Control Systems

There are a number of good third-party tools that can be used to provide version control for ColdFusion code. Most of these tools share some key functionality:

- The capability to control access to source files
- The capability to log changes to files as they occur
- Maintenance of old source files to provide rollback capabilities
- An easy way to compare an older version of the source to the current version

Having a version-control system in place is particularly important when your development team is large or dispersed. And version control is essential if your project is likely to have a long life span, where several builds of the application might exist at any given time.

Most version-control software systems have similar core functionality, but the degree to which they let you manipulate the source code, track changes, and roll back coding errors depends on their quality and price. The products available run the gamut from freeware tools to extremely pricey, all-encompassing solutions. CVS and Subversion (SVN) are among the more popular version control systems used by ColdFusion developers.

It's beyond the scope of this book to review version control software but here are a few features you should consider when selecting a package:

- What is the nature of the software's "sharing" model?
- Does the software support atomic commits?
- Does the software track file or directory renames?
- Does the software run on your client and server platforms?
- Different version control tools support different models controlling access to files. Some provide simple locking mechanism such that two people cannot access the same file simultaneously. Others support a "copy-modify-merge" type approach in which each user gets his (or her) copy which can be modified. The software changes from different users are merged together. When conflicts occur, users are required to confirm changes.Atomic commits refers to a version control system's ability to save all of a set of changes to the system repository; like in a database commit, either all the changes or none of them are made. You'll find that not all version control software supports atomic commits.
- Some version control software does not track file system type changes to source code, such as file renames. This may be an important feature for you to consider.
- In many team development settings, the team agrees to use a particular set of tools, such as a file editor, web server, database server and so on. You may want to consider the nature of support for the version control system in the tools you use, particularly your file editor. Version control systems that can be integrated into your file editor may enable team members to work a bit more efficiently.

This chapter presented a number of suggestions that should get you started thinking about the quality of the code you write and the approach you take to writing code in general. You've also been exposed to the notion of version control software and learned of a few parameters to consider when evaluating version control software for your team.

➜ If you'd like to learn more about coding standards and frameworks for ColdFusion, you should be sure to read Chapter 54.

CHAPTER 54 Development Frameworks

IN THIS CHAPTER

Why Use a Development Methodology or Framework? E507
The Model Viewer Controller Design Pattern E508
Fusebox E510
Mach-II E519
Model-Glue: Unity E532
Issues to Consider with All Development Frameworks E542
Conclusions E542

Why Use a Development Methodology or Framework?

If you've been building Web applications for a while, you've probably run into this situation: You're asked to work on something that you originally wrote many months ago. Perhaps you're asked to change the way a complex set of forms works. As you pore over your old code, you're bewildered. You have trouble figuring out which pages call which and why you did certain things the way you did. Sound familiar?

Of course, this can happen to the most experienced developer. You should recognize that when you spend *n* hours developing an application, you will probably spend 2*n* or 3*n* hours maintaining the application over time. The predicament just described can be largely avoided—or at least significantly mitigated—by using a well-defined and well-thought-out development framework or methodology.

The term *development methodology* refers to the use of a defined, methodical approach to developing software. It implies the use of defined analysis, design, programming, testing and documenting procedures. The term development framework isn't as all-inclusive. Frameworks—the focus of this chapter—involve directives on how to write code and typically include a set of files that form the framework. These files provide some infrastructural function and may be involved in the processing of each request. Some frameworks and development methodologies may encompass variable, procedure, file-naming conventions, directory structures, and other features.

Using these types of concepts let you more easily maintain your code. It also enables a team of developers—all using the same approach—to debug and support each other's code. Take it one step further, and it's easy to imagine how a common framework used by unrelated individuals can produce code that can be understood and maintained by anyone who uses the framework.

A robust framework also lets you build large applications with many features and will scale as your application's features grow. It provides a meaningful way of organizing the user interface and program logic so as to make it easier to maintain.

This chapter takes a detailed look at three frameworks: Fusebox, Mach-II and Model-Glue. You will construct a simple contact manager application in each so you can better understand how these frameworks address common application requirements.

NOTE

The code in this chapter is set up under a directory named /ows/54, after this chapter. This directory should be installed directly in your web root in order for the examples to run without any modifications.

The Model Viewer Controller Design Pattern

Before you dig into the frameworks themselves, you should take a minute to acquaint yourself with or review the *design pattern* commonly referred to as Model-View-Controller (MVC). A design pattern, by the way, is a conceptual solution to a problem or set of problems often found in some aspect of software design and development.

In our case, we're developing rich, complex, web-based applications that involve business logic, data and a user interface. The interdependencies between these elements of your applications often lead to code that is inflexible and difficult to maintain. The coupling between UI elements and data or business logic results in a lot of inelegant copying and pasting of chunks of code when changes and additions are required in the future.

The MVC design pattern minimizes the coupling by dividing the application into three logical areas, as can be seen in Figure 54.1 and described below.

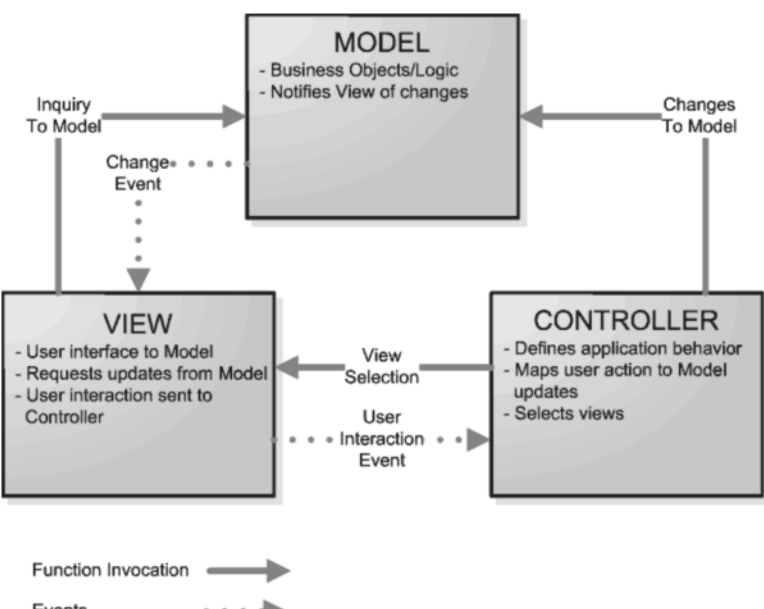


Figure 54.1

The MVC design pattern provides a flexible, event-based model that separates business logic from data and user interface.

The *model* portion of the application encapsulates data and business logic. Many of your application's components will be implemented as part of the model. The model is typically where data storage and retrieval takes place (though MVC doesn't really dictate anything regarding databases per se).

The *view* portion of the application specifies how data, provided by the model, should be presented to the user. This *presentation layer* provides no real processing per se, it just provides an interface with which the user will interact.

The *controller* portion of the application translates user interactions with the viewer into actions performed within the model. The interactions in the web application are virtually all HTTP requests that result from submitting forms, clicking on links and so on.

MVC applications are typically event-driven as can be ascertained from Figure 54.1. For example, when a user interacts with the viewer in some way, perhaps submitting a form, this is interpreted as an event and a message passes to the controller which describes how the event is handled, typically by the model. The model processes and produces the data required, passing it to the controller which determines which view will be used to present the data. The view then formats and presents the data and the cycle continues.

There are several distinct benefits produced from use of MVC. Because the user interface is completely separate from the data and business logic of the application, it is easy to employ different types of user interfaces. In our examples, we'll be using HTML but there's no reason that the views couldn't be rendered in Flash or WAP.

Similarly, it can be easier to switch databases if the model is isolated from the controller and viewer. Moving from SQL Server to mySQL, for example, shouldn't require any changes to the viewer or controller code.

Here's another benefit to the MVC design pattern worth mentioning. As your application has been divided into three distinct areas, it lends itself nicely to a division of labor. For example, you can have your less experienced folks working on the views. The more experienced people can be working in the model and controller. Your team members with more database and data modeling experience can be assigned to develop the model and others can work with them to develop the controller. Those working in the controller will design the applications events, event-handlers and interfaces between front and back end code.

There are drawbacks as well, number one being *complexity*. As with object-oriented design, there is a lot more architecting and planning required. You cannot just sit down and starting banging out sequences of `<cfquery>…<cfif>…<cfelse>…<cfoutput>`.

The flow of an application is dispersed over a variety of files. In a more traditional CF application, you might have a template that does some initial branching and business logic, does a query retrieving some data and then creates a form to display the data. Then you might use an "action" template to process the form submission which sends you back to the first page when done. Or perhaps you did the whole thing in one template.

But in an MVC app, there will typically be a lot more files involved. There will be different files for each view, different CFCs for various components in the model, one or more files providing the controller function that provides a good deal of branching type code to handle different events. Following the flow just to present the initial page in an app will be a little dizzying at first. As you'll soon see in the three frameworks presented here, there are XML configuration files involved and a host of syntax and user-defined functions to learn.

This can be a lot of overhead. In fact, in many situations, this approach is complete overkill. But, when building complex applications, with rich interfaces and a lot of "moving parts", the benefits of MVC will outweigh these costs.

Fusebox

First developed in 1998 by Steve Nelson and Gabe Roffman, Fusebox is the granddaddy of all ColdFusion development frameworks. It has evolved significantly since its inception. More recent versions represent a very significant revamping in order to take advantage of more recent features in CFML and the MVC design pattern.

This framework is based on the metaphor of the electrical box in a house—it contains independent circuits and those contain fuses that control the flow of electricity. A Fusebox application is a collection of independent *circuits*, which include *fuses*, which process *fuseactions*. A circuit is a logical grouping of functionality. It is common to have one subdirectory for each circuit.

Fuseactions are request handlers. These are high level abstractions of tasks, akin to commands, within a circuit. They are used to invoke fuses and are specific to the circuit.

Fuses are low-level, self-contained chunks of functionality, typically represented by one file.

Requests in Fusebox applications identify the circuit and the fuseaction and might look like this in a URL:

```
…&fuseAction=myCircuit.hello
```

The implementation of this electrical fuse box metaphor has evolved with ColdFusion and CFML features. Currently, it is suggested that you break your application into three main circuits – Model, View and Controller, though this is not mandatory.

XML configuration files are used to describe how your Fusebox app is configured and to define and configure each circuit.

How Fusebox 5.1 Works

Fusebox includes a set of core files, a set of "skeleton" files and your code. Later in this chapter, you'll note that the two other frameworks use analogous approaches, though different names are used. Table 54.1 describes the purpose of each set of files.

Table 54.1 File breakdown in a Fusebox application

FILE TYPE	FUNCTION AND USE
Core files	The Fusebox "plumbing" or infrastructure responsible for handling requests. These can be thought of as "read-only".
Skeleton files	Files used to start a new application. These include special xml configuration files that define the circuits and other aspects of your application. You'll tweak these.
Your code	Files that provide business logic and presentation layer of the application. You'll produce most of these from scratch.

NOTE

You'll need to download and install the Fusebox 5.1 framework from http://www.Fusebox.org. Install the core files directly under your web root if possible, in a directory named Fusebox5. If you can't put them there, you must add a mapping in the ColdFusion Administrator named Fusebox5 that points to the directory in which you installed the core files.

You should also download and install the Fusebox skeleton application. Again, put it in your root if possible. Later in this chapter, you'll be instructed on modifying the skeleton application. The modifications that you're instructed to make are found in. `\ows\54\FuseboxExample\skeleton\`

The initial examples we'll review are in the Fusebox Skeleton application, which you were instructed to download into the `webroot\skeleton` directory.

The `fusebox.xml.cfm` XML file is used to provide application configuration settings. (While the filename ends in `.CFM` when used in CF, note that Fusebox is not specific to ColdFusion per se and can be deployed in other languages.) Another XML file (`circuit.xml.cfm`) provides configuration info for each circuit; you create one `circuit.xml.cfm` file for each circuit (or directory) in your app.

If you open the `fusebox.xml.cfm` file in the `skeleton` subdirectory, you'll find the code snippet in listing 54.1

Listing 54.1 Circuits Section from `fusebox.xml.cfm`

```
<!--- Defines circuits used in this application. --->
<circuits>
   <!-- illustrates defaults for parent ("") and relative ("true") -->
   <circuit alias="m" path="model/" parent="" />
   <circuit alias="v" path="views/" parent="" />
   <circuit alias="app" path="controller/" relative="true" />
</circuits>
```

NOTE

Here's how to test that Fusebox is working on your machine. Point your browser to `http://localhost:8500/skeleton/index.cfm`. Note that you might have to change the value localhost to a domain or IP on the server on which you've installed ColdFusion. You should see a "Hello World" type display like the one in Figure 54.2. This assumes that you've installed the sample files in a directory named `ows/54/FuseboxExample/skeleton/` off your webroot. If you're not able to get this display up, make sure ColdFusion is running properly and consult the Fusebox.org site for additional installation instructions.

Note that all of the XML code in the `fusebox.xml.cfm` and `circuit.xml.cfm` files is proprietary to the FuseBox framework. This xml is part of the FuseBox dialect. This dialect is extensible. For more information on extending this dialect to add your own verbs refer to the Fusebox.org site and read the Wiki entry entitled Extending the FB5 Framework with Custom Lexicons

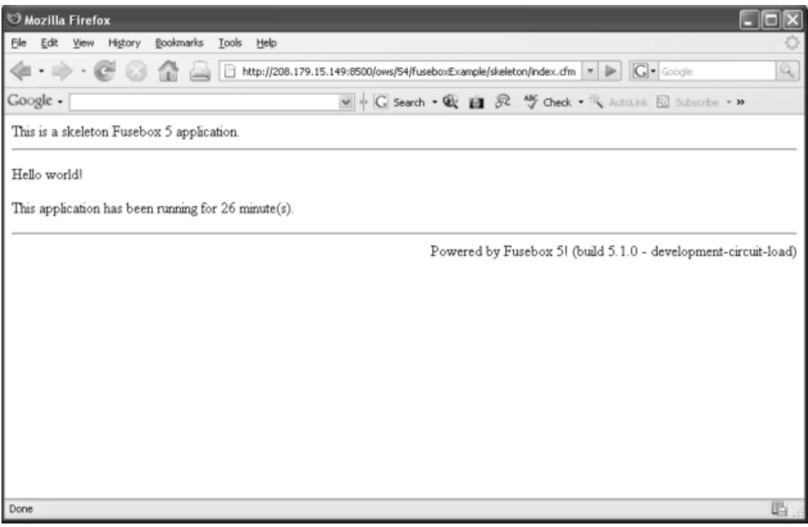


Figure 54.2

The Fusebox 5 skeleton application, if set up properly, displays "Hello World" and indicates how long the app has been running.

This XML code in this section of `fusebox.xml.cfm` defines 3 circuits for this application—`m`, `v` and `app`—which point to the relative paths, `model\, views\` and `controller\`. You'll note that there are indeed 3 subdirectories from the `skeleton\` directory named model, views and controller. Note also that each `<circuit>` tag defines an "alias" by which the circuit will be referenced in other code.

The code snippet in Listing 54.2 shows you how you define the *default fuseaction*.

Listing 54. 2 Parameters Snippet from `fusebox.xml.cfm`

```
<!--- This section defines parameters used throughout the app --->
<parameters>
   <parameter name="defaultFuseaction" value="app.welcome" />
   …
</parameters>
```

The default `fuseaction` is the request handler that is invoked when a user points their browser at the root of your application without reference to any particular `fuseaction`, i.e., it's your app's default page. The value `app.welcome` identifies a `fuseaction` named *welcome* on the *app* circuit, but you might be able to imagine other types of default actions. For example, if your application requires authentication, you might use the default `fuseaction` to point the user at a login form.

You may recall from Listing 54.1 that the `app` alias points to the physical directory, controller. Open the `circuit.xml.cfm` file in the controller subdirectory. You'll see the code snippet found in Listing 54.3.

Listing 54.3 `"welcome" fuseaction` Defined in `controller\circuit.xml.cfm`

```
<!--
   Default fuseaction for application, uses model and view circuits
   to do all of its work:
-->
```

Listing 54.3 (continued)

```
<fuseaction name="welcome">
   <do action="m.getTime" />
   <do action="v.sayHello" />
</fuseaction>
```

So, the XML code in Listing 54.3 defines the *welcome* `fuseaction`. It is comprised of two fuses – `getTime` which is in the `m` (model) circuit and `sayHello` which is in the `v` (views) circuit. At this point, I imagine you can surmise what we'll find in the `circuit.xml.cfm` files in the views and model subdirectories. You'll find this in the model directory's `circuit.xml.cfm` file:

```
<fuseaction name="getTime">
   <include template="act_get_time" />
</fuseaction>
```

And you'll find this in view directory's `circuit.xml.cfm`:

```
<fuseaction name="sayHello">
   <include template="dsp_hello" contentvariable="body" />
</fuseaction>
```

So what do `act_get_time` and `dsp_hello` do? You can see that the Fusebox XML verb, `include`, is used in the case of both `fuseactions`. The `include` verb is certainly the most commonly used verb for executing your custom CFML templates and is equivalent to using the `<cfinclude>` tag. So this Fusebox XML code roughly translates to `<cfinclude template="act_get_time.cfm">` and `<cfinclude template="dsp_hello.cfm">`. The code from `act_get_time.cfm` is displayed in Listing 54.4 and `dsp_hello.cfm` is displayed in Listing 54.5.

NOTE

The Fusebox 5.1 XML dialect provides a rich set of verbs to control the flow of your application:

- `include`: Includes specified file (typically a fuse)
- `do`: Go do (pretty much like "goto") specified fuseAction
- `xfa`: Specify an exit fuseaction (used to exit a fuse)
- `instantiate`: Creates an instance of an object
- `invoke`: Method invocation; execute a user defined function; optionally instantiates object class to
- `argument`: Used to specify method arguments when using invoke
- `parameter`: Enables you to pass privately parameters to fuseactions and fuses that you've executed with do or include such that they are privately scoped rather than globally scoped.
- `set`: Enables you to set variable values
- `relocate`: Enables you to redirect the browser to a different URL
- `if`: Introduces branching logic into XML so you can conditionally execute fuse and fuseactions, etc.
- `loop`: Enables you to perform conditional loops in the XML

Listing 54.4 `act_get_time.cfm`

```
<!--- Invoke functions in cfc, myFusebox.cfc to get time app started running --->
<cfset timeNow = now() />
<cfset startTime = myFusebox.getApplication().getApplicationData().startTime />
<cfif dateDiff("d",startTime,timeNow) gt 0>
   <cfset runTime = dateDiff("d",startTime,timeNow) & " day(s)" />
<cfelseif dateDiff("h",startTime,timeNow) gt 0>
   <cfset runTime = dateDiff("h",startTime,timeNow) & " hour(s)" />
<cfelseif dateDiff("n",startTime,timeNow) gt 0>
   <cfset runTime = dateDiff("n",startTime,timeNow) & " minute(s)" />
<cfelseif dateDiff("s",startTime,timeNow) gt 0>
   <cfset runTime = dateDiff("s",startTime,timeNow) & " seconds(s)" />
<cfelse>
   <cfset runTime = "less than a second" />
</cfif>
```

Listing 54.5 `dsp_hello.cfm`

```
<!--- Display restult of model\act_get_time.cfm --->
<cfoutput>
   <p>Hello world!</p>
   <p>This application has been running for #runTime#.</p>
</cfoutput>
```

NOTE

Looking at Listing 54.4, you may be wondering, "What is `myFusebox` and where did those functions `getApplication()` and `getApplicationData()` come from?" `myFusebox` is component created by `myFusebox.cfc` in the Fusebox5 directory. It is part of the Fusebox core files. `getApplication()` and `getApplication()` are two (of about a dozen) functions defined in that CFC. You can get a better handle on what this object does by adding `<cfdump var="#myFusebox#">` to the bottom of `dsp_hello.cfm` and by invoking `myFusebox.cfc` from your browser.

Looking at the code in Listing 54.5, you may be wondering, "Where does the rest of that page displayed in Figure 54.2 come from?" I know I was. Well if you open `views\circuit.xml.cfm`, you'll also find the code snippet displayed in Listing 54.6.

Listing 54.6 Another Snippet from `views\circuit.xml.cfm`

```
<!--
   Apply a standard layout to the result of all display fuseactions.
   This is fine for simple applications that have just one layout but
   for more complicated situations you will either need to move to
   multiple view circuits or a view circuit and a layout circuit and
   may have to explicitly call a layout fuseaction from your other
   display fuseactions.
-->
<postfuseaction>
   <include template="lay_template" />
</postfuseaction>
```

The code that displays "This is a skeleton Fusebox 5 application." and "Powered by Fusebox 5! (build 5.1.0 - development-circuit-load)", as seen in Figure 54.2, comes from the template `views\lay_template.cfm`. As the comment in Listing 54.6 indicates, the `postfuseaction` gets invoked after rendering any code produced by a `fuseaction` in this circuit. If you look in `views\lay_template.cfm`, you'll see that a variable named `body` holds the rendered content of the

fuseaction that preceded this `postfuseaction`. It's as if you used `<cfsavecontent variable="body">…</cfsavecontent>` around the code in `dsp_hello.cfm`. This behavior is built-in to Fusebox.

So…there's a lot going on in this trivial Fusebox "Hello World" example, isn't there?! The life cycle of this request is displayed in Figure 54.3. Note that when identifying how Fusebox implements MVC, the framework and associated XML pretty much fill the role of controller.

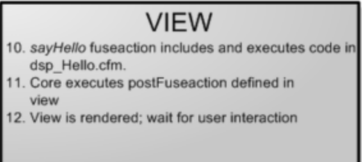


Figure 54.3

The processing flow through the simple "Hello World" Fusebox 5 example, as it relates to the MVC design pattern, is a bit convoluted.

Using Fusebox to Display a List of Contacts

We're really just scratching the surface of Fusebox so far. Let's add a bit of functionality to this simple example that will produce some insights into other aspects of building Fusebox applications. You're going to give our application the ability to display a list of contacts from the Contacts table in the OWS database.

We need to add the link and HREF value to the `dsp_Hello.cfm` template. But making this work will involve modifying `circuit.xml.cfm` files in the controller, model and view directories. We'll need to create a `fuse` (template) in the model that retrieves the data and a `fuse` in the view to display the output.

First things first—let's add a link to the default page. Open the file `views\dsp_Hello.cfm` and add this line just below `<p>This application has been running…`, before the `</cfoutput>`:

```
<p><a href="#mySelf##xfa.contactList#">Contact list</a></p>
```

Note that the variable `mySelf` is part of the Fusebox framework. It provides a reference to the currently executing template…which is actually `index.cfm` (note that all other files were included `into index.cfm` by the framework and our XML directives).

This code also introduces the concept of an *exit fuseaction*, or *xfa*. An exit `fuseaction` is a reference to a `fuseaction` that is used to exit a fuse. In our case, we're exiting this `dsp_Hello.cfm` and going somewhere else. But where should it take us? We'll add a definition for this xfa into `views\ciruit.xml.cfm` because that is the XML file for the current circuit. Add the following line below `fuseaction="sayHello"` just above the directive to include the `dsp_Hello` template:

```
<set name="xfa.contactList" value="app.getContactList" />
```

Your fuseaction section in `views\circuit.xml.cfm` should look like Listing 54.7:

Listing 54.7 Add This xfa to the `sayHello` Fuseaction in `views\circuit.xml.cfm`

```
<fuseaction name="sayHello">
   <!-- This exit fuseaction action invokes a fuse in the controller -->
   <set name="xfa.contactList" value="app.getContactList" />
   <include template="dsp_hello" contentvariable="body" />
</fuseaction>
```

When the `dsp_Hello` fuse is rendered, this xfa will point to the URL

```
http://localhost:8500/ows/54/fuseboxExample/skeleton/index.cfm?fuseaction=
app.getContactList
```

Now we need to define what the `fuseaction app.getContactList` does. As you can surmise, this `fuseaction` will be created in the `controller\circuit.xml.cfm` (recall that "app" is the alias for the controller subdirectory). Add the `fuseaction` in Listing 54.8 to `controller\circuit.xml.cfm`.

Listing 54.8 Add This Fuseaction to `controller\circuit.xml.cfm`

```
<!-- Use model to get contact data and view to display list -->
<fuseaction name="getContactList">
   <do action="m.getContacts" />
   <do action="v.listContacts" />
</fuseaction>
```

You can see that when this `fuseaction` is invoked, it in turn invokes two other `fuseactions` in succession—`getConatcts` (in `model\`) and `listContacts` (in `views\`). This means we'll be adding the corresponding `fuseactions` to the model and views circuits. These are displayed in Listing 54.9 and 54.10.

Listing 54.9 Add This Fuseaction to `model\circuit.xml.cfm`

```
<!-- Includes template to produce query result with all contacts -->
<fuseaction name="getContacts">
   <include template="qry_GetContacts" />
</fuseaction>
```

Listing 54.10 Add This Fuseaction to `views\circuit.xml.cfm`

```
<!-- This fuseaction uses query result qryAllContacts to display list -->
<fuseaction name="listContacts">
   <include template="lst_Contacts" contentvariable="body" />
</fuseaction>
```

These `fuseactions` are simple includes. `qry_GetContacts.cfm` is a very simple query that just retrieves all the contact records from the database and `lst_Contacts.cfm` simply outputs that query result. Please create these new files as you see them in Listing 54.10 and 54.11.

> Note that in a real application, you should consider developing components and methods in the model section. The Fusebox XML verb <invoke... /> is used to invoke methods on objects for this purpose. Use of components is covered in more detail in book 2, Chapter 27, "Creating Advanced ColdFusion Components," in *Adobe ColdFusion 8 Web Application Construction Kit, Volume 2: Application Development*. You should also review some of the Fusebox sample applications (e.g., Brian Kotek's Fusebox Bookstore) for examples of how to incorporate objects into your Fusebox apps.

Listing 54.11 `model\qry_GetContacts.cfm`

```
<!---
qry_getContacts.cfm
L Chalnick 7/2007
Selects all fields, all rows from contact table.
--->
<cfquery name="qryAllContacts" datasource="ows">
   select * from Contacts
</cfquery>
```

Listing 54.12 `views\lst_Contacts.cfm`

```
<!---
lst_Contacts.cfm
Leon Chalnick 7/2007
Displays list of all contacts.
--->
<h1>Contacts</h1>
<p>
<cfoutput query="qryAllContacts">
#FirstName# #LastName#<br />
</cfoutput>
</p>
```

You may have been wondering what Fusebox does with the various `circuit.xml` files that you produce in your application. Fusebox core reads these files, parses them and creates the CF code needed to execute your application logic. The first time you run your application, Fusebox must do all the parsing so it can be rather slow. But once parsed, it creates structures in memory that will result in significant performance benefits.

NOTE

> As you learn more about Fusebox you'll also note that you can configure it to operate in different modes depending on where you are in the development cycle. For example, when you're developing and testing, you need Fusebox to reparse/compile frequently, but when your app is stable and in production, you don't need it to re-parse all the time. These settings are configured in the parameters section of `fusebox.xml.cfm` with the `mode` parameter.

Fusebox still goes quite a bit deeper than we have room for in this chapter. Other important Fusebox features include the ability to employ plug-ins, the ability to enhance the XML vocabulary, globally shared files, handling application level events and many other useful features. Beyond, but related to Fusebox itself, you'll find FuseDocs (an approach to specifying and documenting Fusebox apps) and FLiP (Fusebox Lifecycle Process – a project management methodology).

Fusebox Benefits

The last few updates to Fusebox have radically changed the way it works, incorporating new ColdFusion features and adapting the framework to the MVC design pattern.

Beyond the benefits you get from any framework and the use of any MVC application architecture, newer versions offer a natural way for folks familiar with the Fusebox approach to adapt to ColdFusion's current feature set.

Many ColdFusion developers were using Fusebox back in its first few releases and quite a few still do. If you learn to use this framework, you'll certainly be in a great position to work and share code with other developers. There is a large community of Fusebox developers and the framework has been around so long that there is a robust set of ancillary and 3rd party tools that support the framework, including tools for building wire-frames (code generators for building out basic structures within an application).

Drawbacks

Understanding how Fusebox applications work isn't trivial. Fusebox has become a lot more complicated with each update to the specification and associated files and Fusebox 5.1 is light years beyond earlier versions in terms of complexity.

There are a lot of XML files to maintain as well. This provides some additional flexibility, but you'll find yourself hopping around a bit from the `fusebox.xml` to this `circuit.xml` to that `circuit.xml` to some view page and so on. Part of it is the desire to provide flexibility (e.g., not to force you into using a particular directory structure), but when MVC became all the rage in web development, it seemed as though the desire to support MVC was sort of "shoe-horned" into Fusebox.

Ironically, some people believe that Fusebox applications can be harder to debug than non-Fusebox applications. Most templates are included in the `index.cfm` template and URLs in the app always point back to it and a fuseaction. So when something isn't working, it can be harder to tell which template the offending code came from. Some may also argue that this type of coding harkens back to the days of BASIC and GOTO type commands where the program flow jumps around and the introduction of conditional logic and loops in XML makes this framework ripe for "inexperienced programmer" abuse.

You'll see that some of these drawbacks are not specific to Fusebox and apply to the other frameworks discussed here as well.

Mach-II

Mach-II was introduced in 2003, as developers were getting comfortable with ColdFusion MX and its introduction of CFCs. It was developed largely by Ben Edwards, a software engineer and Java programmer. Now in its second release (version 1.5), advances have been made in expanding its utility in large scale app development and several other areas. Mach-II is a software architecture based on the object-oriented, event/message-driven model that has been used in most modern programming languages. At Mach-II's foundation is the notion of *implicit invocation* of modules.

This approach facilitates producing code modules that are more *cohesive* and more loosely *coupled*. Cohesion refers the extent to which a module is focused on doing one "atomic" task. Coupling refers to the extent to which one module's ability to perform its function relies on other modules. In general, you want to increase the cohesiveness of your application modules and reduce the coupling. By doing so, you will produce applications that are less brittle and easier to change as business requirements change. This is one of the goals of Mach-II. Sound familiar? It should as these concepts are very much in sync with the whole Model-View-Controller pattern.

How Mach-II Works

Like Fusebox, you use an XML file to define many of the functions of your application. It is called `mach-ii.xml`. This file includes elements for defining basic application properties, listeners, event handlers, page views and plug-ins. These are described in Table 54.2. You'll get a good inkling of how Mach-ii works by reviewing this table. As with `fusebox.xml.cfm` and `circuit.xml.cfm` in Fusebox, you produce `mach-ii.xml` in conformance with the Mach-II XML dialect.

NOTE

The Mach-II framework itself (version 1.5) isn't included with the online content. You can download it for free from `www.mach-ii.com`. Follow their instructions for installing the framework. You will need the framework in order to run the examples, but you don't need to run the examples to get a feel for how they work.

Table 54.2 Sections in the `mach-ii.xml` Configuration File

SECTION NAME	DESCRIPTION
Properties	Identifies application properties, such as the application root, default event and as of 1.5, utility CFCs (classes of objects).
Listeners	Identifies the application's listeners, how they're invoked and any parameters they're passed.
Event filters	Identifies code to be optionally called in response to specified events, enabling you to add or remove functionality without affecting of your basic business objects.
Event handlers	Identifies how the application's various events are to be handled (e.g., by notifying listeners, calling event filters, announcing (spawning) new events, etc.
Page views	Identifies templates that are used to display content produced in the process of handling an event.

Table 54.3 (continued)

SECTION NAME	DESCRIPTION
Plug-ins	Identifies modules that are invoked application wide, at a specific point in the life of each request.
Includes	Identifies other Mach-ii.xml-type files that are to be included in the current file.
Modules	Identifies a self-contained application that extends and is accessible within the base application
Subroutines	Identifies subroutines which execute like in-line XML, sharing the context of the calling event handler

The Mach-II framework, as you'll soon see, more closely resembles and implements the Model View Controller design pattern than Fusebox does. A Mach-II application consists of a root directory containing an `index.cfm` file and a commonly used directory structure (`config\`, `model\`, `views\`, `filters\`, `plugins\`) that contain specific types of files. Every request your application makes points to the file `index.cfm`. This file invokes the Mach-II framework (by loading `config\mach-ii.cfm`). The framework itself consists of a variety of CFCs. These CFCs produce and send your application messages/events in accordance with your `mach-ii.xml`, in response to user interaction with your app. These are passed between the framework and your code.

The code you write consists mostly of CFCs that in the Mach-II nomenclature, are referred to as *listeners*, *filters* and *plug-ins*. You also produce simple display-oriented `.cfm` files called *page views*. These page views should be void of business logic and simply provide the user interface. The listeners, really part of the controller, listen for the occurrence of the events and provide an interface to CFCs you've developed in the model. Filters and plug-ins will be discussed a bit later.

NOTE

We'll use the example contact manager application, developed by Team Mach-II member, Matt Woodward, in order to get a feel for how Mach-II works. This app has been tweaked to employ the OWS database used throughout this book. Note that the sample code for the application is in the directory 54\ContactManagerDB. You'll need to download and install Mach-II framework 1.5 (visit `http://www.mach-ii.com` for the download) in order for this to run. I suggest installing Mach-II under your web root. If you can't install it directly in the web root, create a ColdFusion mapping named machii (no hyphen) that points to the directory where you installed it.

Assuming you've installed the sample files on your system (under the web root in the path `ows\54\ContactManagerDB\`) and installed Mach-II, pointing your browser to `http://localhost:8500/ows/54/ContactManagerDB/`should result in a page like that seen in Figure 54.4.

Looking at the directory in which you installed the example application, you'll note the use of standard Mach-II directories, described in table 54.3.

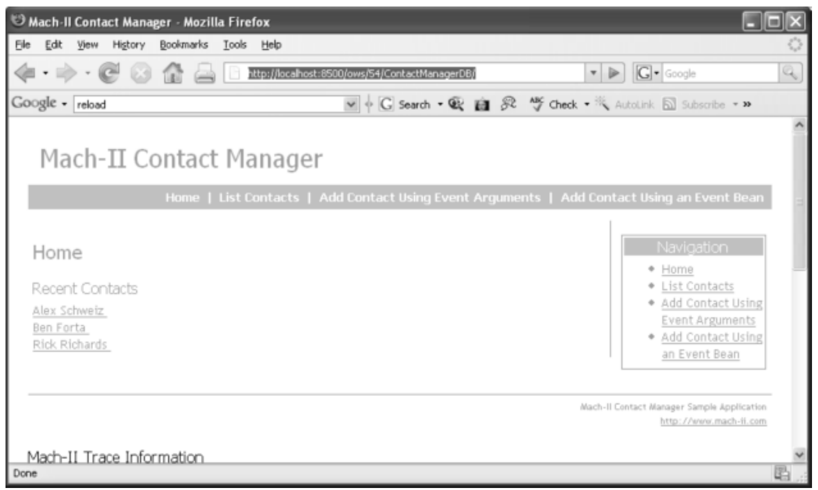


Figure 54.4 The default display from our Mach-II contact manager sample displays a list of recent contacts by default.

Table 54.3 Directories Use by Convention in Mach-II Applications

DIRECTORY	PURPOSE
config	Location for your `mach-ii.xml` configuration file(s)
filters	Location for your filter code (typically CFCs)
model	Location for your applications business objects (CFCs, typically)
views	Location for your page-view code (typically `.cfm` files)

Listing 54.13 `mach-ii.xml`

```
<mach-ii version="1.0">

   <!-- PROPERTIES section identifies app level params -->
   <properties>
      <!-- Mach-II Required -->
      <property name="applicationRoot" value="/ows/54/ContactManagerDB" />
      <property name="defaultEvent" value="showHome" />
      <property name="eventParameter" value="event" />
      <property name="parameterPrecedence" value="form" />
      <property name="maxEvents" value="10" />
      <property name="exceptionEvent" value="exceptionEvent" />

      <!-- Application Specific -->
      <property name="dsn" value="ows" />
   </properties>

   <!-- LISTENERS -->
   <listeners>
      <listener name="ContactListener"
type="ows.54.ContactManagerDB.listeners.ContactListener" />
   </listeners>
```

Listing 54.13 (continued)

```
   <!-- EVENT-FILTERS -->
   <event-filters>
      <!-- ContactBeanerFilter: ensures there is a contact object available in the
event -->
      <event-filter name="contactBeaner"
type="ows.54.ContactManagerDB.filters.ContactBeanerFilter" />

      <!-- EventArgsFilter: sets parameters passed to the filter as arguments in the
event -->
      <event-filter name="eventArgs" type="MachII.filters.EventArgsFilter" />

      <!-- RequiredFieldsFilter: used to control required fields in form submissions
-->
      <event-filter name="requiredFields" type="MachII.filters.RequiredFieldsFilter"
/>
   </event-filters>

   <!-- PLUGINS -->
   <plugins>
      <plugin name="tracePlugin" type="MachII.plugins.TracePlugin" />
   </plugins>

   <!-- EVENT-HANDLERS -->
   <event-handlers>
      <!-- Main Menu and Basic Display Events -->
      <event-handler event="showHome" access="public">
         <notify listener="ContactListener" method="getRecentContacts"
resultArg="recentContacts" />
         <view-page name="mainMenu" contentArg="content" />
         <view-page name="mainTemplate" />
      </event-handler>

      <event-handler event="listContacts" access="public">
         <notify listener="ContactListener" method="getAllContacts"
resultArg="allContacts" />
         <view-page name="contactList" contentArg="content" />
         <view-page name="mainTemplate" />
      </event-handler>

      <event-handler event="viewContact" access="public">
         <notify listener="ContactListener" method="getContact" resultArg="contact"
/>
         <view-page name="viewContact" contentArg="content" />
         <view-page name="mainTemplate" />
      </event-handler>

      <!-- Contact Create- and Update-Related Events -->
      <event-handler event="newContactEventArgs" access="public">
         <!-- One way to set event-args... -->
         <event-arg name="submitEvent" value="createContactFromEventArgs" />
         <event-arg name="submitLabel" value="Create" />
         <announce event="showContactForm" copyEventArgs="true" />
      </event-handler>
```

Listing 54.13 (continued)

```
      <event-handler event="newContactEventBean" access="public">
         <event-arg name="submitEvent" value="createContactFromEventBean" />
         <event-arg name="submitLabel" value="Create" />
         <announce event="showContactForm" copyEventArgs="true" />
      </event-handler>

      <event-handler event="showContactForm" access="public">
         <!-- use contactBeaner filter to make sure there's a contact object in the
event; that way
               we can use the same form for creating and updating contacts and not
worry about doing
               a bunch of conditional stuff on the form -->
         <filter name="contactBeaner" />
         <view-page name="contactForm" contentArg="content" />
         <view-page name="mainTemplate" />
      </event-handler>

      <event-handler event="createContactFromEventArgs" access="public">
         <!-- use requiredFields filter to check the required fields on the form -->
         <filter name="requiredFields">
            <parameter name="requiredFields"
value="firstName,lastName,address,city,state,zip" />
            <parameter name="invalidEvent" value="showContactForm" />
         </filter>
         <notify listener="ContactListener" method="createContactFromEventArgs" />
      </event-handler>

      <event-handler event="createContactFromEventBean" access="public">
         <!-- use requiredFields filter to check the required fields on the form -->
         <filter name="requiredFields">
            <parameter name="requiredFields"
value="firstName,lastName,address,city,state,zip" />
            <parameter name="invalidEvent" value="showContactForm" />
         </filter>
         <event-bean name="contact" type="ows.54.ContactManagerDB.model.Contact"
               fields="id,firstName,lastName,address,city,state,zip" />
         <notify listener="ContactListener" method="createContactFromEventBean" />
      </event-handler>

      <event-handler event="contactCreated" access="private">
         <announce event="listContacts" />
      </event-handler>

      <event-handler event="editContact" access="public">
         <notify listener="ContactListener" method="getContact" resultArg="contact"
/>
         <!-- Second way to set event-args... -->
         <filter name="eventArgs">
            <parameter name="submitEvent" value="updateContact" />
            <parameter name="submitLabel" value="Update" />
         </filter>
         <announce event="showContactForm" copyEventArgs="true" />
      </event-handler>

      <event-handler event="updateContact" access="public">
```

Listing 54.13 (continued)

```
            <!-- use requiredFields filter to check the required fields on the form -->
            <filter name="requiredFields">
               <parameter name="requiredFields"
    value="firstName,lastName,address,city,state,zip" />
               <parameter name="invalidEvent" value="editContact" />
            </filter>
            <event-bean name="contact" type="ows.54.ContactManagerDB.model.Contact"
                  fields="id,firstName,lastName,address,city,state,zip" />
            <notify listener="ContactListener" method="updateContactFromEventBean" />
         </event-handler>

         <event-handler event="deleteContact" access="public">
            <notify listener="ContactListener" method="deleteContact" />
         </event-handler>

         <event-handler event="contactDeleted" access="public">
            <announce event="listContacts" />
         </event-handler>

         <event-handler event="contactUpdated" access="public">
            <!-- add the newly created contact to the list of recent contacts -->
            <announce event="listContacts" />
         </event-handler>

         <event-handler event="exceptionEvent" access="private">
            <view-page name="exception" />
         </event-handler>
      </event-handlers>

      <!-- PAGE-VIEWS -->
      <page-views>
         <page-view name="home" page="/views/home.cfm" />
         <page-view name="mainMenu" page="/views/mainMenu.cfm" />
         <page-view name="mainTemplate" page="/views/mainTemplate.cfm" />
         <page-view name="contactList" page="/views/contactList.cfm" />
         <page-view name="contactForm" page="/views/contactForm.cfm" />
         <page-view name="viewContact" page="/views/viewContact.cfm" />
         <page-view name="exception" page="/views/exception.cfm" />
      </page-views>
   </mach-ii>
```

Looking at Figure 54.4, note that a list of recent contacts is initially displayed in response to the application's default event (which is identified in the `<properties>` section at the top of the `mach-ii.xml` file displayed in Listing 54.13). This is analogous to the default `fuseaction` back in the Fusebox example. You can see that the value of the default event is *showHome*. Let's follows this through and figure out how the page in Figure 54.4 was created.

As you might guess, the `showHome` event is handled by an entry in the event handlers section of `mach-ii.xml`, as seen in Listing 54.14. The `<notify…>` directive sends a message to something called `ContactListner` and then two `<view-page…> directives` are use to display the results.

Listing 54.14 `showHome` Event Handler Snippet from `mach-ii.xml`

```
<event-handler event="showHome" access="public">
   <notify listener="ContactListener" method="getRecentContacts"
resultArg="recentContacts" />
   <view-page name="mainMenu" contentArg="content" />
   <view-page name="mainTemplate" />
</event-handler>
```

Following through with the `ContactListener`, you'll note a reference to it in the `<listeners>` section near the top of `mach-ii.xml`, back in Listing 54.13. The following line points to `ContactListener.cfc` in the `ows\54\ContactManagerDB\model\` directory.

```
<listener name="ContactListener"
type="ows.54.ContactManagerDB.model.ContactListener">
```

You'll note that a method named `getRecentContacts` is being invoked, and that the results will be stored in a `resultArg` using the name `recentContacts`. Can you see where this is going? What will probably happen is that `ContactListener` will somehow produce a list of recent contacts and store it in a variable that will be referenced in the templates specified next in the `showHome` event handler.

That is indeed what happens, though it's a little convoluted in that `ContactListener` doesn't do this work itself; it calls another function in another object, (`contactGateway.getRecentContacts()`) to do it. We'll explain this in more detail later in the chapter.

Before we move on to other parts of this application, let's look at the template loaded by the `<view-page name="mainMenu"…>` line in `mach-ii.xml` in Listing 56.15 (note that the page, `mainMenu.cfm` is located in the `views\` subdirectory). This page is responsible for producing the listing of recent contacts and includes a link to each recent contact, if there are any.

Listing 54.15 `mainMenu.cfm`

```
<!--- get the contact query from the event and set a default empty query
just in case it doesn't exist for some reason --->

<cfset recentContacts = event.getArg("recentContacts", QueryNew("id")) />
<h2>Home</h2>
<!--- List the most recent contacts accessed. --->
<h3>Recent Contacts</h3>

<cfif recentContacts.RecordCount GT 0>
   <cfoutput query="recentContacts">
      <a href="index.cfm?event=viewContact&id=#recentContacts.contactid#">
      #recentContacts.firstName# #recentContacts.lastName#
      </a><br />
   </cfoutput>
<cfelse>
   <em>- no recent contacts -</em>
</cfif>
```

The first line contains a little surprise: `event.getArg(…)`. What's that about? Well an *event* is an object that is created by the Mach-II framework. When `ContactListener` produced the list of recent contacts, it stored them in a variable defined in the `<notify…>` directive (review Listing 54.14). You can retrieve *event arguments* with the Mach-II framework `event` method, `getArgs()`. Note that the

Mach-II framework packages up `URL` and `form` data in the `event` object for you as well. The framework also includes a `setArgs()` method on the `event` object that enables you to attach arguments to the event as it flows to the next point for processing.

So why is `QueryNew("id")` employed in the call to `event.getArg()`? This is done in the event that there is no data in the database, i.e., no recent contacts; it provides an empty query result.

You may be wondering about the `ContactListner` object we touched on a few paragraphs back. In Mach-II, you use CFCs called *listeners* as part of its controller implementation. Listeners interface with the business objects in your application model. Listeners are CFCs that extend the Mach-II framework `listener` object. Listeners are programmed to "listen" for specific events and invoke the appropriate methods in the business objects in your application model. In our contact manager example, there is only one listener as there is really only one main business object that we're dealing with. You identify listeners in their own section of the `mach-ii.xml` file and Mach-II will load them with the framework.

NOTE

Note that the loading and reloading of the Mach-II framework is controlled via the `MACHII_CONFIG_MODE` variable defined in `index.cfm`. It's currently set to 0, which indicates that the framework (and your objects) will be reloaded dynamically, when the `mach-ii.xml` is updated. So when you want to reload your objects after making some changes, for example, simply insert a space and then delete it from `mach-ii.xml` and save the file.

Mach-II Building Blocks

You've been introduced to the concept of listeners and events. There are two ways to invoke a listener, i.e. two different types of *invokers* in the Mach-II framework. `machii.framework.invokers.CFCInvoker_EventArgs` passes event arguments to the listener as separate, named and typed arguments—clearly a safe approach. Whereas the entire event is provided as a single argument when you use the `machii.framework.invokers.CFCInvoker_Event` invoker, so it provides flexibility at the expense of *type-safety*.

The Mach-II framework supports the use of event filters and plug-ins. There are some useful examples on the Mach-II website. Event filters are defined in their own section of `mach-ii.xml`. They are used when you need to perform an operation on data provided to (or by) a certain event, or when you need to conditionally abort certain event handlers.

Plug-ins are used when you need to perform operations in response to events on a more global level, e.g., before each request is processed or at the end of every event handler's process. Version 1.5 of the framework includes the `tracePlugin` plug-in. This provides Mach-II-specific debugger "trace" type output.

Further Exploration of the Contact Manager Example

So what happens when you click on one of the contacts in the "recent contacts" listing? The `<a href…>` anchor around each contact name in the list points to a URL like this

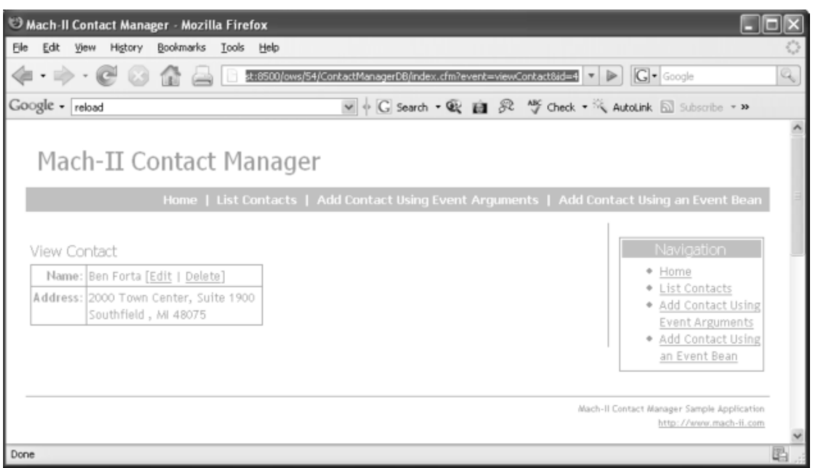
```
index.cfm?event=viewContact&id=4
```

As you can guess, clicking on the link loads `index.cfm` and triggers the event we've named `viewContact`, and will pass it the ID of the contact to be viewed.

The `index.cfm` template runs Mach-II which processes the requested URL, creating an event and handing it off to the appropriate event handler in `mach-ii.xml`. The `<event-handler>` section for `viewContact` is presented in Listing 54.16. Figure 54.5 specifies how this request will be dealt with.

Figure 54.5

When you click on a contact in the recent contacts listing, that contact's data is displayed.



Listing 54.16 From `mach-ii.xml`

```
<event-handler event="viewContact" access="public">
   <notify listener="ContactListener" method="getContact" resultArg="contact" />
   <view-page name="viewContact" contentArg="content" />
   <view-page name="mainTemplate" />
</event-handler>
```

The `<event-handler…>` invokes the `getContact` method in `ContactListener` and then uses two `<view-page…>` directives to display the results. As we saw earlier, `ContacListener` is notified of this event. This time, its method, `getContact`, is employed and the result—an instance of the requested contact—will be returned in the result argument `contact`. We'll follow this through in order to learn how the database is dealt with in Mach-II.

As you saw in Listing 54.11, we're calling the method `getContact` in the `contactListener.cfc`. You'll find this method in Listing 54.17

Listing 54.17 `getContact` Method in `ContactListener.cfc`

```
<!--- get contact - returns a specific contact object --->
<cffunction name="getContact" access="public" output="false"
 returntype="ows.54.ContactManagerDB.model.Contact"
 hint="Returns a specific contact object based on the id passed to
 this function">
   <cfargument name="event" type="MachII.framework.Event" required="true" />
```

Listing 54.17 (continued)

```
      <!--- Create an empty contact object.  This step is unnecessary
      in the event bean version of this process (below). --->
      <cfset var contact = variables.contactService.getContactBean() />

      <!--- set the id from the event to the contact object --->
      <cfset contact.setId(arguments.event.getArg("id")) />

      <!--- call the read method in the service layer and pass it the
      contact bean with the id set --->
      <cfset variables.contactService.read(contact) />

      <!--- return the contact bean --->
      <cfreturn contact />
   </cffunction>
```

The `getContact` method creates an instance of a contact using a local object variable named `contactService`…wait a second. Where did that object come from? And we still haven't seen a database query yet; this where the `contactService` comes into play. Best practices in Mach-II suggest that listeners (part of the controller) should only have access to the database via a *service layer*. The service layer is intended to further separate the business objects from controller. It acts as a go-between for the model and controller.

Where did `variables.contactService` come from though? You can see it's in the variables scope. When creating CFCs in your Mach-II apps, you must include a method called `configure`. This method is used to initialize the object and Mach-II will run it automatically when it loads the object. The `configure` method for `contactListener` is displayed in Listing 54.18.

Listing 54.18 `configure` Method in `contactListener.cfc`

```
   <!--- this configure method is called by Mach-II automatically when
    the application loads --->
   <cffunction name="configure" access="public" output="false"
      returntype="void"
      hint="Configures this listener as part of the Mach-II framework">
         <!--- We'll need access to our ContactService object in this
            Listener. The service layer exposes an API to the
            business logic/model layer of the application and should be
            the only part of the model with which the listener communicates.
            Since the service layer includes an instance of the data objects
            that communicate with the database, it needs the datasource name.
            We'll get the datasource name from the Mach-II properties we
            declared in the XML config file by using the getProperty
            function. --->
         <cfset variables.contactService = createObject("component",
       "ows.54.ContactManagerDB.model.ContactService").init(getProperty(
         "dsn")) />
   </cffunction>
```

You can see in Listing 54.18 that the `contactService` object are created when Mach-II calls the `configure()` method in `ContactListener.cfc`.

So when are we going to see the database already? We're almost there; let's crack open the `contactService` object and see how it acts as a go-between. Listing 54.19 contains the `contactService.init()` constructor method.

Listing 54.19 `init()` Method in `contactService`

```
<!--- Constructor for this CFC. In the constructor we take in
 the datasource name as an argument, create an instance of the
 ContactDAO and ContactGateway CFCs, and then we return the
 instance of ContactService. --->
<cffunction name="init" access="public" output="false"
 returntype="ContactService">
   <!--- We take the datasource in as an argument because the
    service layer and everything behind it are not and should not
    be aware of Mach-II. This makes the business logic independent
    of Mach-II, making it far more flexible and reusable. --->
   <cfargument name="dsn" type="string" required="true" />

   <!--- We'll need access to our dataobjects (gateway and DAO)
    in this Service. --->
   <cfset variables.contactGateway = CreateObject("component",
    "ContactGateway").init(arguments.dsn) />
   <cfset variables.contactDAO = CreateObject("component",
    "ContactDAO").init(arguments.dsn) />

   <cfreturn this />
</cffunction>
```

This method is invoked when the `contactService` object is created. It in turn creates two objects that it uses internally: `contactGateway` and `contactDAO`.

The DAO (data access object) is a concept right out of object-oriented programming. A DAO is used for dealing with one instance of an object and includes methods for creating reading, saving and deleting (CRUD) a single instance of the object. When you need to deal with sets of data in your database (rather than an instance), you use a gateway type object rather than the DAO and that's what the `contactGateway` object is for.

We'll see how `contactDAO` is used in `contactService` in a few paragraphs. Back in Listing 54.17, we called a method like so: `contactService.getContactBean()`. This method is displayed in Listing 54.20.

Listing 54.20 `getContactBean()` Method in `ContactService`

```
<!--- getContactBean simply returns an empty instance of the Contact bean.
 The use of a wrapper method such as this reduces the number of hard-coded
 instances of createObject("component", "Contact") so that if we ever
 refactor things or move Contact.cfc to another directory, we only have to
 change this method. --->
<cffunction name="getContactBean" access="public" output="false"
 returntype="Contact">
   <cfreturn createObject("component", "Contact").init() />
</cffunction>
```

An empty contact object is created by the contact "bean" (`model\Contact.cfc`) and returned. Now if you recall, we called `getContactBean()` back in Listing 54.17, in `contactListener`. That produced

this empty contact object. Then we populated it with the ID from the…wait a sec…where did that ID come from? Remember that Mach-II was told to invoke `getContact` in response to the event that occurred when the user clicked on a link in the recent contacts listing. You may also recall that Mach-II includes `form` and `URL` variables in the `event` object.

So the `contactListener`, having been passed the event as an argument, passes it off (in the newly created contact *bean*) to the `contactService.read()` method, as you can see back in Listing 54.17. `contactService.read()` is displayed in Listing 54.21.

Listing 54.21 `read()` Method in `contactService`

```
<!--- read populates a contact bean based on the id of the bean passed in --->
<cffunction name="read" access="public" output="false" returntype="void">
   <cfargument name="contact" type="Contact" required="true" />
   <cfset variables.contactDAO.read(arguments.contact) />
</cffunction>
```

This `contact` instance is then passed to `contactDAO.read()`, displayed in Listing 54.22, in order to be populated with the values from the database for that contact. You've been waiting to see the database? Well here it is.

Listing 54.22 `read` Function in `contactDAO.cfc`

```
<!--- READ: populates a contact bean using info from the database --->
<cffunction name="read" access="public" output="false" returntype="void"
 hint="Populates the contact bean using the ID of the object passed in">
   <cfargument name="contact" type="Contact" required="true" />

   <!--- var scope everything! --->
   <cfset var getContact = "" />

   <cfquery name="getContact" datasource="#variables.dsn#">
      SELECT *
      FROM contacts
      WHERE contactid = <cfqueryparam value="#arguments.contact.getId()#"
    cfsqltype="cf_sql_char" />
   </cfquery>

   <!--- if we got a record back, populate the bean --->
   <cfif getContact.RecordCount EQ 1>
      <cfset arguments.contact.init(getContact.contactid,
       getContact.firstName, getContact.lastName,
       getContact.address, getContact.city, getContact.state,
       getContact.zip) />
   </cfif>
</cffunction>
```

As you can see, the contact data access object as been passed an argument of type contact (i.e., a bean built by `model\contact.cfc`). It grabs the `ContactID` using the `contact.getID()` method. Then it uses `contact.init()` to populate this instance of a contact object with the values retrieved from the database. Once populated, it is returned to `contactService`, which returns it to `contactListener`.

Finally, `contactListener` produces the contact via Mach-II in the variable specified in the `resultArg` parameter in the `<notify…>` directive back in `mach-ii.xml`:

```
<notify listener="ContactListener" method="getContact"
 resultArg="contact" />
```

The event handler in `mach-ii.xml` next passes the event to the `<view-page…>` directives:

```
<view-page name="viewContact" contentArg="content" />
<view-page name="mainTemplate" />
```

If you look down to the `<page-views>` section in `mach-ii.cfm`, you'll see that these directives simply load the .cfm pages with the same names (from the views\ subdirectory). The framework makes the event object accessible to these templates. The first template simply displays the contact that comes from event. The contentArg parameter in the first `<view-page…>` directive results in the rendered HTML content being stored in a variable named content. This variable is then employed in the second `<view-page…>` directive, which generates all of the content surrounding the contact info, as seen in Figure 54.5.

You should explore the other aspects of Matt Woodward's contact manager example. Now that you have been exposed to the basic concepts and organization of a Mach-II application, finding your way through the example will be much easier.

Mach-II Benefits

Mach-II was developed to give CF developers the benefits of the MVC design pattern, OOP, and implicit invocation concepts. These help produce applications that are more flexible and easier to change as business requirements inevitably change. Applications that clearly separate user interface from business logic are also better suited for reproduction in new technical environments (e.g., when you want to take a Web-based app and make it run on PDAs) and are in a better position to scale up.

As it was developed after OO programming constructs were introduced to ColdFusion and was explicitly designed to support MVC, writing OO apps may come a bit more naturally in Mach-II than in Fusebox.

Well-designed object-oriented applications are better able to adapt to change in the future in general. Mach-II gives you a very elegant framework on which to design and build object-oriented CF applications.

Mach-II Drawbacks

The biggest drawback to any object-oriented approach to software development is complexity and the volume of code required. As you can tell from our example, following the life cycle of an event in Mach-II, like Fusebox, isn't a trivial task that beginners are ready for.

Designing an object-based application requires a lot more thought up front to properly define the objects, handlers, etc. When you start building complex, real-world applications with many and more complex business objects, you'll find yourself writing gobs of "getter and setter" type code in

your DAOs. And as more objects and more interactions between objects are introduced into your applications, you may run into complexities resulting from object dependencies. If you're not accustomed to designing object-oriented apps, these can be significant challenges.

The same goes for MVC apps in general. It's an approach that won't come naturally to developers schooled in the tenets of structured programming who may be accustomed to writing linear code made of modules that directly invoke sub-routines.

But again, most of these drawbacks are not specific to Mach-II per se as much as they are to learning OO programming and design and employing the MVC design pattern. There are a few newer tools that will be discussed in the last framework that we'll explore that will help address some of these drawbacks.

Model-Glue: Unity

Model-Glue (MG) was initially released in 2005 and is the creation of Joe Rinehart of Firemoss, LLC. It is intended to ease the development of OO applications and supports the MVC design pattern. Currently, in release 2, it has been combined to work with two other open-source frameworks – Reactor and ColdSpring – and is now referred to as Model-Glue: Unity (or MGU).

These two ancillary frameworks address some of the problems associated with object-oriented programming in ColdFusion. ColdSpring mitigates the effect of dependencies that occur between your business objects (CFCs) through what is referred to as *dependency injection*. Reactor is a database access code generator. Using Reactor means you don't have to program gobs of mind-numbing CRUD routines for each new app. We'll discuss these a bit further in the chapter.

How Model-Glue Works

NOTE

Note that as with the other frameworks, Model-Glue, ColdSpring and Reactor are not included in the materials for this chapter. They should be downloaded from their organizations' web sites (`www.model-glue.com`, www.ColdSpring.org and `www.ReactorFramework.org`).

It is suggested that you install the Model-Glue, ColdSpring and Reactor frameworks in directories in the webroot using the names ModelGlue, ColdSpring and Reactor. If you're able to do so, you won't need to create any ColdFusion mappings to them.

Note that this may not be straightforward with Reactor in particular. Getting a recent version–which you will need–involves use and knowledge of Subversion (svn–version control software). If you're not familiar with Subversion, you may wind up with a relatively deep directory structure that you don't understand. The reactor subdirectory is buried in there and contains the `reactorFactory.cfc` file. *That* is the folder that you should either move into your webroot or make a ColdFusion mapping to.

MGU is too much to digest in one-third (or so) of a chapter, so we're going to break it down. We'll first explore the Model-Glue framework without getting into Reactor; then Reactor will be discussed.

MG seems quite a bit like Mach-II at first glance. There is a directory structure that includes subdirectories for `config\`, `model\`, `views\`, `controller\`, an `index.cfm` file in the root of your application and a `model-glue.xml` file in the `config\` subdirectory. If you open that `model-glue.xml` file, you'll see sections for controllers and event handlers.

We'll get to controllers in a second but you'll note that each event handler you define has three subsections: broadcasts, views, results. You won't always need to use all three. Broadcasts are used when the event handler needs to communicate with the controller. Broadcasts send messages to the framework which looks through the controllers you've defined for one that is listening for this particular message.

Views are used to define output. When you have multiple views defined, the later ones have access to (can include) the earlier ones.

Results are used to define the application behavior upon getting certain results from earlier steps.

What typically happens in a MG application is that an event name is passed in via the URL. This generates an event. The associated event-handler broadcasts a message when this occurs. It may also define views that will be employed in handling the event and results as well. The framework identifies an associated listener function in your controller. The event is passed to the listener, which may interact in some way with the model, perhaps calling a few functions in the model's CFCs. The resulting data are often attached to the event and then displayed with the views that were part of the event handler. This flow is depicted in Figure 54.6.

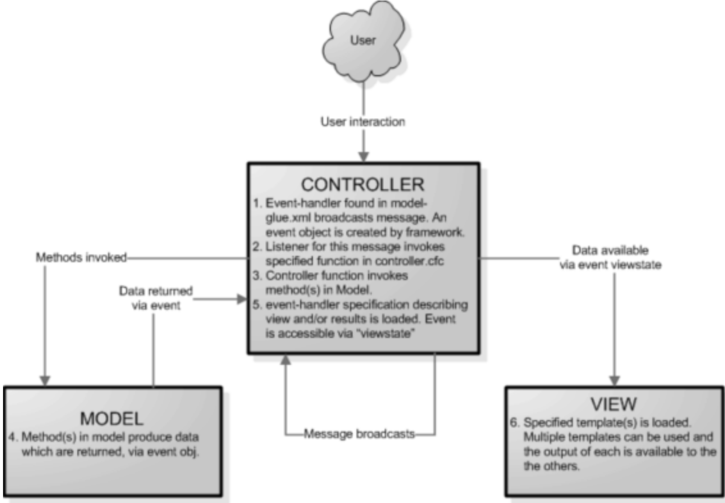


Figure 54.6

The flow of events and data in a typical Model-Glue user interaction is similar to the generic MVC design pattern.

Let's walk through an app that will display a list of contacts from the OWS contact table and allow you to view one.

A Simplified Contact Manager in Model-Glue

NOTE

The first sample Model-Glue application we'll look at is in the directory `/ows/54/MGExamples/MGContactMgr/` under your web root.

If you've installed the code samples for this chapter in the suggested location, you should be able to bring up the first Model-Glue sample by pointing to

```
http://localhost:8500/ows/54/MGContactMgr/
```

This should produce a web page like that seen in Figure 54.7. This example is largely based on techniques Ray Camden employed in his press release app from his MG presentations.

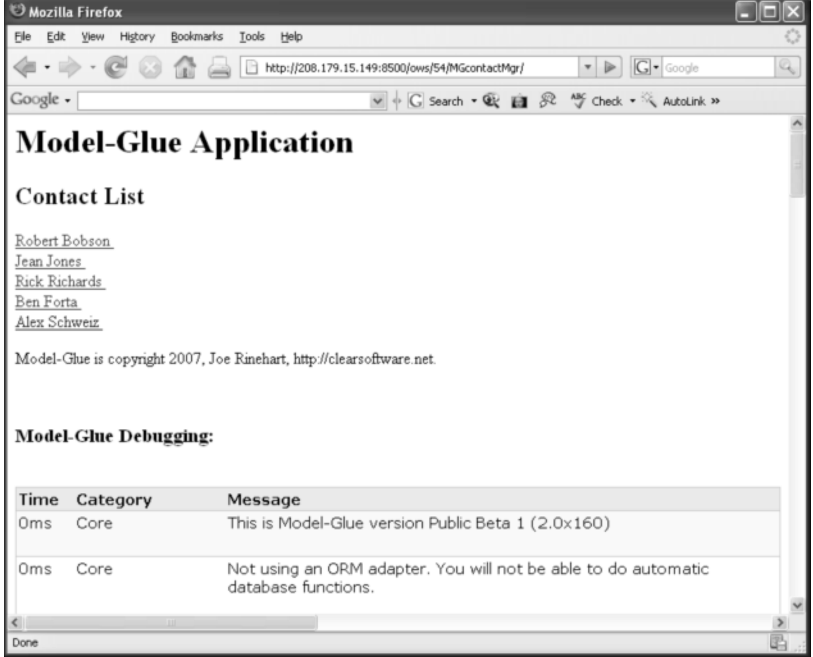


Figure 54.7 Our Model-Glue sample application's default event produces this simple contact listing

As mentioned above, the `index.cfm` file is used to launch your MG applications. You should also keep an `application.cfc` (or `.cfm`) in your application's root and be sure to use a unique application name. The `index.cfm` invokes the framework.

Let's first look into the `coldspring.xml` and `model-glue.xml` files in the `/config` folder as seen in Listings 54.23 and 54.24. There are a lot of settings we don't need to worry about now, but first note that the `coldspring.xml` provides the configuration information for your app. This file is separate and not part of the application per se. Next, you'll see that this file is broken down into different beans. Some of them are part of the MGU framework, but you add others that your application needs (in our case, we've added the `contact` and `contactGateway` beans).

NOTE

Note that we are not actually using Reactor in our example because it does not support the Derby database we use. I have, however, left the `reactorConfiguration` bean in the `coldspring.xml` file so that you can see how it might be configured, but it isn't really needed in this example.

Note there's the `modelGlueConfiguration` bean, the `reactorConfiguration` bean and then a few beans for our app. These two beans are created used by the MGU framework itself. You change the values but all the entries are there by default when you create a new MGU app.

Note how you identify the `defaultEvent` and map to the directories used for views, generated views and so on. These are values you'll need to change when you begin your own MGU apps.

Listing 54.23 `coldspring.xml`

```
<beans>
  <!-- This is your Model-Glue configuration -->
   <bean id="modelGlueConfiguration"
class="ModelGlue.unity.framework.ModelGlueConfiguration">
      <!-- Be sure to change reload to false when you go to production! -->
      <property name="reload"><value>true</value></property>
      <!-- Rescaffold is overridden by reload - if reload is false, rescaffold's
setting doesn't matter -->
      <property name="rescaffold"><value>true</value></property>
      <!-- Be sure to change debug to false when you go to production! -->
      <property name="debug"><value>true</value></property>
      <property name="defaultEvent"><value>page.index</value></property>
      <property name="reloadPassword"><value>true</value></property>
      <property
name="viewMappings"><value>/ows/54/MGcontactMgr/views</value></property>
      <property
name="generatedViewMapping"><value>/ows/54/MGcontactMgr/views/generated</value></pro
perty>
      <property
name="configurationPath"><value>config/ModelGlue.xml</value></property>
      <property
name="scaffoldPath"><value>config/scaffolds/Scaffolds.xml</value></property>
      <property name="statePrecedence"><value>form</value></property>
      <property name="reloadKey"><value>init</value></property>
      <property name="eventValue"><value>event</value></property>
      <property name="defaultTemplate"><value>index.cfm</value></property>
      <property name="defaultExceptionHandler"><value>exception</value></property>
      <property name="defaultCacheTimeout"><value>5</value></property>
      <property
name="defaultScaffolds"><value>list,edit,view,commit,delete</value></property>
   </bean>

  <!-- Reactor configuration bean (not actually used in this sample) -->
   <bean id="reactorConfiguration" class="reactor.config.config">
      <constructor-arg
name="pathToConfigXml"><value>/ows/54/MGcontactMgr/config/Reactor.xml</value></const
ructor-arg>
      <property name="project"><value>/ows/54/MGcontactMgr</value></property>
      <property name="dsn"><value>ows</value></property>
      <property name="type"><value>mssql</value></property>
      <property
```

Listing 54.23 (continued)

```
name="mapping"><value>/ows/54/MGcontactMgr/model/data</value></property>
      <property name="mode"><value>development</value></property>
   </bean>

  <!-- Put definitions for your own beans and services here -->
   <!--
      Note that we're using a special bean called 'factory-bean' to
      retrieve the DSN which was defined above, in the
      reactorConfiguration bean. It was defined as a property of
      that bean, so we can use what's referred to as a factory-method
      (built by ColdSpring) to retrieve the value of any of its
      properties.
   -->
   <bean id="contact" class="ows.54.MGcontactMgr.model.contact">
      <constructor-arg name="dsn">
         <bean factory-bean="reactorConfiguration" factory-method="getDSN" />
      </constructor-arg>
   </bean>
   <bean id="contactGateway"
        class="ows.54.MGcontactMgr.model.contactGateway">
      <constructor-arg name="dsn">
         <bean factory-bean="reactorConfiguration" factory-method="getDSN" />
      </constructor-arg>
   </bean>
</beans>
```

Note that the `dsn` is being passed to our app's two beans (`contact` and `contactGateway`), near the bottom of Listing 54.23. This will be relevant later on. But for right now, know that by defining objects here (with the ColdSpring `<bean>` tags), ColdSpring will make these objects accessible to other parts of our application, such as the controller.

Listing 54.24 `model-glue.xml`

```
<modelglue>
   <controllers>
      <controller name="MyController"
type="ows.54.MGcontactMgr.controller.Controller">
         <message-listener message="GetContact" function="getOneContact" />
         <message-listener message="GetContacts" function="getContactList" />
         <message-listener message="OnRequestStart" function="OnRequestStart" />
         <message-listener message="OnQueueComplete" function="OnQueueComplete" />
         <message-listener message="OnRequestEnd" function="OnRequestEnd" />
      </controller>
   </controllers>

   <event-handlers>
      <event-handler name="page.index">
         <broadcasts>
            <message name="GetContacts" />
         </broadcasts>
         <views>
            <include name="body" template="dspBody.cfm" />
            <include name="main" template="dspTemplate.cfm" />
         </views>
         <results />
      </event-handler>
```

Listing 54.24 (continued)

```
          <event-handler name="ShowContact">
             <broadcasts>
                <message name="GetContact" />
             </broadcasts>
             <views>
                <include name="body" template="dspContact.cfm" />
                <include name="main" template="dspTemplate.cfm" />
             </views>
             <results>
                <result name="BadContact" do="HomePage" redirect="true"/>
             </results>
          </event-handler>

          <event-handler name="view.template">
             <broadcasts />
             <results />
             <views>
                <include name="template" template="dspTemplate.cfm" />
             </views>
          </event-handler>

          <event-handler name="exception">
             <broadcasts />
             <views>
                <include name="body" template="dspException.cfm" />
             </views>
          </event-handler>
       </event-handlers>
    </modelglue>
```

Now take a look at `model-glue.xml` in Listing 54.24. Running the application generates the default event, `page.index`. The framework creates an `event` object making it available to the various ColdFusion templates and CFCs involved in the process. The `page.index` event handler broadcasts a message named `GetContacts` for one of the `message-listeners` in the `<controllers>` section to pick-up.

Looking in the `<controllers>` section, you can see that we have one controller defined and it has several *listeners* defined. The `<message-listerner>` tags point to methods in the controller—a CFC identified in the `<controller>` tag as `/ows/54/MGContactMgr/controller/Controller.cfc`.

> In MG apps, you start with a generic `controller.cfc` that is part of the framework. It extends the model-glue controller and includes `onRequestStart()` and `onRequestEnd()` methods which you can customize.

So the `page.index` event invokes the `getContacts` listener. That listener invokes a function in `controller.cfc` named `getContactList`, which you'll find in Listing 54.25.

Listing 54.25 Portion of `controller.cfc`

```
    <cfcomponent displayname="Controller"
    extends="ModelGlue.unity.controller.Controller" output="false">
       <!---
          Calls model to retrieve list of all contacts. It will place
          a query object named contactList in the event.
```

Listing 54.25 (continued)

```
--->
<cffunction name="getContactList" access="Public"
returntype="void" output="false" hint="Produces list of contacts">
  <cfargument name="event" type="ModelGlue.Core.Event" required="true">

  <cfset variables.contactGateway =
        GetModelGlue().GetBean("contactGateway")>
  <cfset arguments.event.setValue("contactList",
        variables.contactGateway.fetchContactList())>
</cffunction>

<!---
   Calls model to retrieve one contact, based on ContactID
   stored in the event. It will place a query object named
   contact in the event if it's successful and will put
   a result named "BadContact" in the event if it fails.
--->
<cffunction name="getOneContact" access="Public"
returntype="void" output="false" hint="Gets one contact">
   <cfargument name="event" type="ModelGlue.Core.Event" required="true">
   <cfset var ID = arguments.event.getValue("ContactID")>

  <cfset variables.contact = GetModelGlue().GetBean("contact")>

   <cftry>
      <cfset arguments.event.setValue("contact",
         variables.contact.fetchContact(ID))>
      <cfcatch>
         <cfset arguments.event.addResult("BadContact")>
      </cfcatch>
   </cftry>
</cffunction>
…
</cfcomponent>
```

Take a look at the function `getContactList()`, which is invoked by the listener. You'll see that it takes one argument, an event object, which is part of the MG framework. The event object encapsulates any form or URL values that are part of the event that began when the application was loaded and the default event was fired off. You don't need to refer to them by their URL or Form scope, they're now just part of the `event` object and you can use the MG `event.getValue()` function to retrieve them and `event.setValue()` to inject data into the event object. When you're creating controller functions that will be invoked from listeners, you'll almost always take the `event` as an argument.

Note that the `onRequestStart` and `onRequestEnd` listeners (and their related controller functions) are part of the generic `controller.cfc` that you'll start virtually all MG applications from. These don't appear in Listing 24.5 for the sake of brevity.

You'll see that `getContactList()` simply gets some data by calling the function `fetchContactList()` in an object in the model, `variables.contactGateway`. Where did `variables.contactGateway` come from? It was instantiated by ColdSpring based on our `contactGateway` bean back in the `coldspring.xml` file! We use two MG functions—`getModelGlue()` and `getBean()`—to retrieve an instance of our bean, meaning we don't have to instantiate it here. Nice!

Let's follow the event down into `fetchContactList()` in the model. See the code for the `contactGateway.cfc` in Listing 54.26

Listing 54.26 `contactGateway.cfc`

```
<cfcomponent displayname="contactGateway" hint="Gets lists of contacts">
<!---
   contactGateway
   Leon Chalnick, 7/2007
   Produces groups of contacts from database.
--->
<!---
   Init method is passed DSN. It came from bean definition in
  coldspring.xml and is stored in variables scope so that
   is accessible to fucntion in this module.
--->
<cffunction name="init" returntype="any">
   <cfargument name="dsn" type="string" required="Yes">
   <cfset variables.dsn = arguments.dsn>
</cffunction>

<!---
   Produces a query containing first and last
   names of all contacts.
--->
<cffunction name="fetchContactList"
 returntype="query" access="public" hint="Lists contactID, first and last names">

<cfquery name="qry" datasource="#variables.dsn#">
   select contactID, FirstName, LastName
   from contacts
</cfquery>
<cfreturn qry>
</cffunction>

</cfcomponent>
```

`contactGateway.cfc` contains an initialization function—which is invoked by ColdSpring when the bean is instantiated—and the `fetchContactLists()` function, which is invoked in the controller by the `getContactList()` function. It returns a query result containing any records in the database. Note that the `dsn` was passed into this object, which is stored in the variables scope. (`contactGateway` was instantiated when it was loaded from `coldspring.xml` by MGU.) That's why it's accessible in the functions within the component.

> Note that this is a relatively simplistic way of retrieving the data. This actually helps you see one of the benefits of the separation of code and data. We could use this simplistic approach while we're prototyping the app and then refine it to a more OO approach later, with very little, if any, changes to the user interface or controller code.

The query result is returned to the controller and the controller—as you saw in `getContactList()` in Listing 54.26—actually stuffs the result into the `event` object in a variables named `contactList`. What's next? The processing returns to the framework in the event-handler. Two views are defined:

```
<include name="body" template="dspBody.cfm" />
<include name="main" template="dspTemplate.cfm" />
```

Looking at `dspBody.cfm` (in the `/views` folder), seen in Listing 54.27, we start by retrieving `contactList` from something called the `viewState`. The name *contactList* came from the code in the controller, where we stuffed the query result returned by the model into the event. In MG, the `event` object is accessible inside another MG object called a *viewstate*.

Listing 54.27 `dspBody.cfm`

```
<!---
   dspBody.cfm
   Leon Chalnick 7/2007
   Produces content for body (center) of page
   used in response to default event. If there
   were any contact in the db, then they will
   have been stored in a query result in the
   event named contactList. If there weren't
   any, then we create an empty query and
   store use it instead
--->
<cfset allContacts = viewState.getValue("contactList", queryNew("empty"))>

<h2>Contact List</h2>

<cfif isDefined("allContacts.ContactID")>
   <cfoutput query="allContacts">
   <a
href="#viewState.getValue("myself")#ShowContact&ContactId=#ContactId#">#FirstName#
#LastName#</a><br>
   </cfoutput>
<cfelse>
   <p>No contacts at this time.</p>
</cfif>
```

The query result from the event is stored in a local query variable which we then loop over, outputting the results. Note how a special variable called `myself` is retrieved from the `viewstate` too, and used to build a URL linking to the individual contacts being displayed. `myself` is a MG variable that contains the value of the application's URL right up to "`…event=`", so in this case it will produce something like this:

```
http://localhost:8500/ows/54/MGcontactMgr/index.cfm?event=
```

You can see that we're adding the event named `ShowContact` and outputting the contact's ID to the end of the URL too. And by now, I imagine you have a pretty good idea of what happens when you click on this link.

NOTE

Note that in MG, the output of one view can be retrieved inside subsequent views. Review the event-handler for `HomePage` in `model-glue.xml`. You'll see two views. The output from the first is stored in a variable called body. The next view uses `dspTemplate.cfm`. The following code from `dspTemplate.cfm` enables us to grab the output from the preceding view (`body`) and employ it in `dspTemplate.cfm`, as if it had been saved in `<cfsavecontent>`.

```
<cfif viewCollection.exists("body")>
  <cfoutput>#viewCollection.getView("body")#</cfoutput>
</cfif>
```

Solving Common OOP Problems with Model-Glue: Unity

Model-Glue: Unity (MGU) is a newer version of MG that incorporates the Reactor and Cold-Spring frameworks with MG in order to address some common development problems for OOP developers. Crack open the `contact.cfc` in the Mach-II example app (it's in the `ContactManagerDB\model` subdirectory). You'll find a whole lot of repetitive "getter and setter" code, common to OO programming.

Now imagine a real contact manager app with maybe 10 tables and 150 fields…and think about writing all that getter and setter code for that! OO programmers spend a great deal of time writing and tweaking this code.

Reactor is a code generator that enables you to very rapidly provide basic data access capabilities when you're using SQL Server, mySQL, Oracle or PostGereSQL databases. With the inclusion of Reactor in MGU, a new `<scaffold>` tag has been added to the XML dialect used in `model-glue.xml`. The `<scaffold>` tag is like a meta tag in that it provides the functionality of an entire event handler section! It will produce your DAO CFCs, writing all the OO getter and setter functions needed for basic Create, Read, Update and DELETE (CRUD) functions integral to most database applications. Further, *it will generate simple views for lists and your basic CRUD functions*. The code it generates won't necessarily be the stuff you want to use, but it can all be easily tweaked and customized. So Reactor *significantly mitigates* the time you need to spend on these mind-numbing programming tasks.

Unfortunately, we don't really have the space to include a full blown example with Reactor (nor does Reactor support our sample Derby database) but there are a number of excellent resources on the web, including Joe Rinehart's MGU presentation in Adobe Breeze format, located here:
`https://admin.adobe.acrobat.com/_a200985228/p45971854/`

ColdSpring manages object dependencies, freeing you from having to concern yourself with where and how objects get used by injecting them into your application and making them available when needed. In the preceding example, you saw ColdSpring in action, instantiating beans defined in `coldspring.xml` for us, making them easy to access elsewhere. ColdSpring also simplifies the addition of complex business services to your CF apps. MGU also uses ColdSpring internally to configure itself.

Model-Glue: Unity Benefits

Model-Glue: Unity really ties OOP with the MVC design pattern in a very clear and cohesive way. Like Mach-II and Fusebox, it makes it easy to separate data, business logic and the presentation layer. With the incorporation of Reactor and ColdSpring, it becomes possible to prototype apps very quickly.

Model-Glue: Unity Drawbacks

As with all of these frameworks, increased functionality leads to increased complexity. There is a lot more to understand. And in the case of MGU, everything you need is not available in one, nice package (you must download, install and configure ColdSpring and Reactor separately, which opens the possibility of configuration issues).

Issues to Consider with All Development Frameworks

Recognize that adopting a methodology and/or framework isn't a decision to take lightly. It can be a significant commitment. Varying learning curves are associated with the frameworks you've looked at in this chapter. Unless you're currently using MVC type designs, these will require that you structure your applications very differently from what you or the people you work with are used to. For example, a commitment to an object-oriented methodology by a team of less-experienced programmers can prove to be challenging.

Another important point is that different frameworks exploit different features of the CFML language, and this language isn't static. It has evolved significantly over the years. For example, the introduction of ColdFusion components when CFMX was released had a profound effect on a number of frameworks (most notably, `cfObjects` and `smartObjects`, perhaps); most popular frameworks changed quite a bit afterwards. The point is, you should be concerned with the effect that future changes might have on the approach taken by the methodology.

The frameworks discussed in this chapter have been updated many times over the months and years. What were thought to be best practices 6 months ago may be significantly altered within the next 6 months. In other words, the folks who develop these frameworks are always uncovering better ways of doing things. You may adopt a framework at a certain point in time, go off and write a lot of code based on the framework at that point in time only to see a newer version of the framework released which points out shortcomings in the preceding version, which you adopted in a big way.

A final note: Most application-development tools provide some degree of a programming framework themselves. They often include features that support certain types of coding. There is some inherent danger in coding approaches that attempt to circumvent the way the developer (Adobe in this case) intended for the product to be used.

Conclusions

ColdFusion provides a rich development environment in which you have a great deal of freedom. If you're in the business of developing ColdFusion or other applications, however, it might make sense to adopt a methodology or framework. The most significant benefit from taking this approach is that you and others you work with will be able to more easily maintain the applications you create. Other benefits will vary a bit from one approach to the next. However, all of the frameworks discussed in this chapter are intended to help you build more modular reusable code. Along with good coding practices, these frameworks also help you write code more effectively.

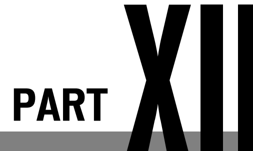
It's important to note that you don't need to use any of these frameworks to produce good ColdFusion applications. You should, however, adopt some sort of methodical approach to developing applications—perhaps something you and your associates develop on your own—so that you can still reap the benefits associated with using a methodology.

NOTE

For more information on Fusebox, see `http://www.fusebox.org`.

For more information on Mach-II, see `http://www.mach-ii.com`

For more information on Model-Glue: Unity see `http:www.Model-Glue.com`.



Appendices

C ICU4J-Supported Locales E545

D Locale Differences E555

APPENDIX C

ColdFusion MX 8 and ICU4J-Supported Locales

Introduction

This appendix provides a list of ColdFusion MX 8 and ICU4J/CLDR supported locales. The ColdFusion code used to produce this data can be found in Listings CH.1 and HC.2. Note that Listing CH.1 uses core Java to retrieve the ColdFusion locales to eliminate the "old" style locale identifiers such as English (United States) in keeping with "best" practice recommendation to use Java style locale identifiers.

Listing C.1 `cfLocales.cfm`—ColdFusion MX 8—Supported Locales

```
<!---
Listing H.1: cfLocales.cfm
retrieves list of core java/coldfusion supported locales.
 --->

<cfscript>
//get coldfusion/core java locales
locale=createObject("java","java.util.Locale");
locales=locale.getAvailableLocales();
// prettify the locale sorting
l="";
for (i=1; i <= arrayLen(locales);i++) {
      l=listAppend(l,locales[i]);
}
l=listToArray(listSort(l,'textNoCase'));
// now get the actual locales, verbose for illustration
for (i=1; i <= arrayLen(l);i++) {
      if (listLen(l[i],"_")==1)
              thisLocale=locale.init(l[i]);
      if (listLen(l[i],"_")==2) {
              language=listFirst(l[i],"_");
              country=listLast(l[i],"_");
              thisLocale=locale.init(language,country);
      }
```

Listing C.1 (continued)

```
        if (listLen(l[i],"_")==3) {
                language=listFirst(l[i],"_");
                country=listgetAt(l[i],2,"_");
                variant=listLast(l[i],"_");
                thisLocale=locale.init(language,country,variant);
        }

        name=thisLocale.getDisplayName();
        writeoutput("#l[i]# #chr(9)# #name#<br>");
  }
  </cfscript>
```

Table C.1 lists ColdFusion-supported locales.

Table C.1 ColdFusion MX 8 Supported Locales

COLDFUSION MX 8 LOCALE	LOCALE NAME
ar	Arabic
ar_AE	Arabic (United Arab Emirates)
ar_BH	Arabic (Bahrain)
ar_DZ	Arabic (Algeria)
ar_EG	Arabic (Egypt)
ar_IQ	Arabic (Iraq)
ar_JO	Arabic (Jordan)
ar_KW	Arabic (Kuwait)
ar_LB	Arabic (Lebanon)
ar_LY	Arabic (Libya)
ar_MA	Arabic (Morocco)
ar_OM	Arabic (Oman)
ar_QA	Arabic (Qatar)
ar_SA	Arabic (Saudi Arabia)
ar_SD	Arabic (Sudan)
ar_SY	Arabic (Syria)
ar_TN	Arabic (Tunisia)
ar_YE	Arabic (Yemen)
be	Belarusian
be_BY	Belarusian (Belarus)
bg	Bulgarian
bg_BG	Bulgarian (Bulgaria)
ca	Catalan
ca_ES	Catalan (Spain)
cs	Czech
cs_CZ	Czech (Czech Republic)
da	Danish
da_DK	Danish (Denmark)
de	German
de_AT	German (Austria)
de_CH	German (Switzerland)
de_DE	German (Germany)
de_LU	German (Luxembourg)
el	Greek
el_CY	Greek (Cyprus)
el_GR	Greek (Greece)
en	English
en_AU	English (Australia)
en_CA	English (Canada)
en_GB	English (United Kingdom)
en_IE	English (Ireland)
en_IN	English (India)
en_MT	English (Malta)
en_NZ	English (New Zealand)
en_PH	English (Philippines)
en_SG	English (Singapore)
en_US	English (United States)
en_ZA	English (South Africa)
es	Spanish
es_AR	Spanish (Argentina)
es_BO	Spanish (Bolivia)
es_CL	Spanish (Chile)
es_CO	Spanish (Colombia)
es_CR	Spanish (Costa Rica)
es_DO	Spanish (Dominican Republic)
es_EC	Spanish (Ecuador)
es_ES	Spanish (Spain)
es_GT	Spanish (Guatemala)
es_HN	Spanish (Honduras)
es_MX	Spanish (Mexico)
es_NI	Spanish (Nicaragua)
es_PA	Spanish (Panama)
es_PE	Spanish (Peru)
es_PR	Spanish (Puerto Rico)

Table C.1 (continued)

COLDFUSION MX 8 LOCALE	LOCALE NAME	COLDFUSION MX 8 LOCALE	LOCALE NAME
es_PY	Spanish (Paraguay)	mt_MT	Maltese (Malta)
es_SV	Spanish (El Salvador)	nl	Dutch
es_US	Spanish (United States)	nl_BE	Dutch (Belgium)
es_UY	Spanish (Uruguay)	nl_NL	Dutch (Netherlands)
es_VE	Spanish (Venezuela)	no	Norwegian
et	Estonian	no_NO	Norwegian (Norway)
et_EE	Estonian (Estonia)	no_NO_NY	Norwegian (Norway,Nynorsk)
fi	Finnish		
fi_FI	Finnish (Finland)	pl	Polish
fr	French	pl_PL	Polish (Poland)
fr_BE	French (Belgium)	pt	Portuguese
fr_CA	French (Canada)	pt_BR	Portuguese (Brazil)
fr_CH	French (Switzerland)	pt_PT	Portuguese (Portugal)
fr_FR	French (France)	ro	Romanian
fr_LU	French (Luxembourg)	ro_RO	Romanian (Romania)
ga	Irish	ru	Russian
ga_IE	Irish (Ireland)	ru_RU	Russian (Russia)
hi_IN	Hindi (India)	sk	Slovak
hr	Croatian	sk_SK	Slovak (Slovakia)
hr_HR	Croatian (Croatia)	sl	Slovenian
hu	Hungarian	sl_SI	Slovenian (Slovenia)
hu_HU	Hungarian (Hungary)	sq	Albanian
in	Indonesian	sq_AL	Albanian (Albania)
in_ID	Indonesian (Indonesia)	sr	Serbian
is	Icelandic	sr_BA	Serbian (Bosnia and Herzegovina)
is_IS	Icelandic (Iceland)		
it	Italian	sr_CS	Serbian (Serbia and Montenegro)
it_CH	Italian (Switzerland)		
it_IT	Italian (Italy)	sv	Swedish
iw	Hebrew	sv_SE	Swedish (Sweden)
iw_IL	Hebrew (Israel)	th	Thai
ja	Japanese	th_TH	Thai (Thailand)
ja_JP	Japanese (Japan)	th_TH_TH	Thai (Thailand,TH)
ja_JP_JP	Japanese (Japan,JP)	tr	Turkish
ko	Korean	tr_TR	Turkish (Turkey)
ko_KR	Korean (South Korea)	uk	Ukrainian
lt	Lithuanian	uk_UA	Ukrainian (Ukraine)
lt_LT	Lithuanian (Lithuania)	vi	Vietnamese
lv	Latvian	vi_VN	Vietnamese (Vietnam)
lv_LV	Latvian (Latvia)	zh	Chinese
mk	Macedonian	zh_CN	Chinese (China)
mk_MK	Macedonian (Macedonia)	zh_HK	Chinese (Hong Kong)
ms	Malay	zh_SG	Chinese (Singapore)
ms_MY	Malay (Malaysia)	zh_TW	Chinese (Taiwan)
mt	Maltese		

Listing C.2 `icu4jLocales.cfm`—ICU4Jj—Supported Locales

```
!---
Listing CH2: icu4jLocales.cfm
retrieves list of icu4j supported locales.
 --->
<cfscript>
// get available ULocales
locales=
createObject("java","com.ibm.icu.util.ULocale").getAvailableLocales();
// no need to "sort" the ULocales
for (i=1; i LTE arrayLen(locales); i=i+1) {
  writeoutput("#locales[i]# #locales[i].getDisplayName()#<br>");
}
</cfscript>
```

Table C.2 lists ICU4J and CLDR locales.

Table C.2 ICU4J/CLDR Locales

ICU4J LOCALE	LOCALE NAME
af	Afrikaans
af_ZA	Afrikaans (South Africa)
am	Amharic
am_ET	Amharic (Ethiopia)
ar	Arabic
ar_AE	Arabic (United Arab Emirates)
ar_BH	Arabic (Bahrain)
ar_DZ	Arabic (Algeria)
ar_EG	Arabic (Egypt)
ar_IQ	Arabic (Iraq)
ar_JO	Arabic (Jordan)
ar_KW	Arabic (Kuwait)
ar_LB	Arabic (Lebanon)
ar_LY	Arabic (Libya)
ar_MA	Arabic (Morocco)
ar_OM	Arabic (Oman)
ar_QA	Arabic (Qatar)
ar_SA	Arabic (Saudi Arabia)
ar_SD	Arabic (Sudan)
ar_SY	Arabic (Syria)
ar_TN	Arabic (Tunisia)
ar_YE	Arabic (Yemen)
as	Assamese
as_IN	Assamese (India)
az	Azerbaijani
az_Cyrl	Azerbaijani (Cyrillic)
az_Cyrl_AZ	Azerbaijani (Cyrillic, Azerbaijan)
az_Latn	Azerbaijani (Latin)
az_Latn_AZ	Azerbaijani (Latin, Azerbaijan)
be	Belarusian
be_BY	Belarusian (Belarus)
bg	Bulgarian
bg_BG	Bulgarian (Bulgaria)
bn	Bengali
bn_IN	Bengali (India)
ca	Catalan
ca_ES	Catalan (Spain)
cs	Czech
cs_CZ	Czech (Czech Republic)
cy	Welsh
cy_GB	Welsh (United Kingdom)
da	Danish
da_DK	Danish (Denmark)
de	German
de_AT	German (Austria)
de_BE	German (Belgium)
de_CH	German (Switzerland)
de_DE	German (Germany)
de_LU	German (Luxembourg)
el	Greek
el_GR	Greek (Greece)
en	English
en_AU	English (Australia)
en_BE	English (Belgium)
en_BW	English (Botswana)
en_CA	English (Canada)
en_GB	English (United Kingdom)
en_HK	English (Hong Kong SAR China)
en_IE	English (Ireland)
en_IN	English (India)
en_MT	English (Malta)
en_NZ	English (New Zealand)
en_PH	English (Philippines)
en_PK	English (Pakistan)

Table C.2 (continued)

ICU4J LOCALE	LOCALE NAME
en_SG	English (Singapore)
en_US	English (United States)
en_US_POSIX	English (United States, Computer)
en_VI	English (U.S. Virgin Islands)
en_ZA	English (South Africa)
en_ZW	English (Zimbabwe)
eo	Esperanto
es	Spanish
es_AR	Spanish (Argentina)
es_BO	Spanish (Bolivia)
es_CL	Spanish (Chile)
es_CO	Spanish (Colombia)
es_CR	Spanish (Costa Rica)
es_DO	Spanish (Dominican Republic)
es_EC	Spanish (Ecuador)
es_ES	Spanish (Spain)
es_GT	Spanish (Guatemala)
es_HN	Spanish (Honduras)
es_MX	Spanish (Mexico)
es_NI	Spanish (Nicaragua)
es_PA	Spanish (Panama)
es_PE	Spanish (Peru)
es_PR	Spanish (Puerto Rico)
es_PY	Spanish (Paraguay)
es_SV	Spanish (El Salvador)
es_US	Spanish (United States)
es_UY	Spanish (Uruguay)
es_VE	Spanish (Venezuela)
et	Estonian
et_EE	Estonian (Estonia)
eu	Basque
eu_ES	Basque (Spain)
fa	Persian
fa_AF	Persian (Afghanistan)
fa_IR	Persian (Iran)
fi	Finnish
fi_FI	Finnish (Finland)
fo	Faroese
fo_FO	Faroese (Faroe Islands)
fr	French
fr_BE	French (Belgium)
fr_CA	French (Canada)
fr_CH	French (Switzerland)
fr_FR	French (France)
fr_LU	French (Luxembourg)
ga	Irish
ga_IE	Irish (Ireland)
gl	Galician
gl_ES	Galician (Spain)
gu	Gujarati
gu_IN	Gujarati (India)
gv	Manx
gv_GB	Manx (United Kingdom)
haw	Hawaiian
haw_US	Hawaiian (United States)
he	Hebrew
he_IL	Hebrew (Israel)
hi	Hindi
hi_IN	Hindi (India)
hr	Croatian
hr_HR	Croatian (Croatia)
hu	Hungarian
hu_HU	Hungarian (Hungary)
hy	Armenian
hy_AM	Armenian (Armenia)
hy_AM_REVISED	Armenian (Armenia, Revised Orthography)
id	Indonesian
id_ID	Indonesian (Indonesia)
is	Icelandic
is_IS	Icelandic (Iceland)
it	Italian
it_CH	Italian (Switzerland)
it_IT	Italian (Italy)
ja	Japanese
ja_JP	Japanese (Japan)
kk	Kazakh
kk_KZ	Kazakh (Kazakhstan)
kl	Kalaallisut
kl_GL	Kalaallisut (Greenland)
kn	Kannada
kn_IN	Kannada (India)
ko	Korean
ko_KR	Korean (South Korea)
kok	Konkani
kok_IN	Konkani (India)
kw	Cornish
kw_GB	Cornish (United Kingdom)
lt	Lithuanian
lt_LT	Lithuanian (Lithuania)
lv	Latvian
lv_LV	Latvian (Latvia)
mk	Macedonian
mk_MK	Macedonian (Macedonia)
ml	Malayalam
ml_IN	Malayalam (India)
mr	Marathi
mr_IN	Marathi (India)
ms	Malay

Table C.2 (continued)

ICU4J LOCALE	LOCALE NAME	ICU4J LOCALE	LOCALE NAME
ms_BN	Malay (Brunei)	sr_Latn	Serbian (Latin)
ms_MY	Malay (Malaysia)	sr_Latn_CS	Serbian (Latin, Serbia And Montenegro)
mt	Maltese	sv	Swedish
mt_MT	Maltese (Malta)	sv_FI	Swedish (Finland)
nb	Norwegian Bokmål	sv_SE	Swedish (Sweden)
nb_NO	Norwegian Bokmål (Norway)	sw	Swahili
nl	Dutch	sw_KE	Swahili (Kenya)
nl_BE	Dutch (Belgium)	sw_TZ	Swahili (Tanzania)
nl_NL	Dutch (Netherlands)	ta	Tamil
nn	Norwegian Nynorsk	ta_IN	Tamil (India)
nn_NO	Norwegian Nynorsk (Norway)	te	Telugu
om	Oromo	te_IN	Telugu (India)
om_ET	Oromo (Ethiopia)	th	Thai
om_KE	Oromo (Kenya)	th_TH	Thai (Thailand)
or	Oriya	ti	Tigrinya
or_IN	Oriya (India)	ti_ER	Tigrinya (Eritrea)
pa	Punjabi	ti_ET	Tigrinya (Ethiopia)
pa_IN	Punjabi (India)	tr	Turkish
pl	Polish	tr_TR	Turkish (Turkey)
pl_PL	Polish (Poland)	uk	Ukrainian
ps	Pashto	uk_UA	Ukrainian (Ukraine)
ps_AF	Pashto (Afghanistan)	ur	Urdu
pt	Portuguese	ur_IN	Urdu (India)
pt_BR	Portuguese (Brazil)	ur_PK	Urdu (Pakistan)
pt_PT	Portuguese (Portugal)	uz	Uzbek
ro	Romanian	uz_Cyrl	Uzbek (Cyrillic)
ro_RO	Romanian (Romania)	uz_Cyrl_UZ	Uzbek (Cyrillic, Uzbekistan)
ru	Russian	uz_Latn	Uzbek (Latin)
ru_RU	Russian (Russia)	uz_Latn_UZ	Uzbek (Latin, Uzbekistan)
ru_UA	Russian (Ukraine)	vi	Vietnamese
sk	Slovak	vi_VN	Vietnamese (Vietnam)
sk_SK	Slovak (Slovakia)	zh	Chinese
sl	Slovenian	zh_Hans	Chinese (Simplified Han)
sl_SI	Slovenian (Slovenia)	zh_Hans_CN	Chinese (Simplified Han, China)
so	Somali	zh_Hans_SG	Chinese (Simplified Han, Singapore)
so_DJ	Somali (Djibouti)	zh_Hant	Chinese (Traditional Han)
so_ET	Somali (Ethiopia)	zh_Hant_HK	Chinese (Traditional Han, Hong Kong SAR China)
so_KE	Somali (Kenya)	zh_Hant_MO	Chinese (Traditional Han, Macao SAR China)
so_SO	Somali (Somalia)	zh_Hant_TW	Chinese (Traditional Han, Taiwan)
sq	Albanian		
sq_AL	Albanian (Albania)		
sr	Serbian		
sr_Cyrl	Serbian (Cyrillic)		
sr_Cyrl_CS	Serbian (Cyrillic, Serbia And Montenegro)		

Appendix D

Locale Differences

Introduction

This appendix provides a comparison of ColdFusion 8 and ICU4J/CLDR supported locales for date, time, numeric and currency formatting. The ColdFusion code used to produce this data can be found in Listing D.1. Note that Listing D.1 uses core Java to retrieve the ColdFusion locales to eliminate the “old” style locale identifiers such as English (United States) in keeping with “best” practice recommendation to use Java style locale identifiers.

Listing D.1 `compareLocales.cfm`—Compare ColdFusion 8 and ICU4J Supported Locales

```
<!---
Listing D.1: compareLocales.cfm
compares core java/coldfusion supported locales against icu4j locales for date, time,
numeric and currency formatting.
 --->
<cfscript>
// test data
now=now();
// lets try one of the new javacast options
aNumber=javacast("bigdecimal",2550);
aCurrency=javacast("bigdecimal",2550.25);
//get coldfusion/core java locales
locale=createObject("java","java.util.Locale");
locales=locale.getAvailableLocales();
// prettify the locale sorting
l="";
for (i=1; i <= arrayLen(locales);i++) {
  l=listAppend(l,locales[i]);
}
l=listToArray(listSort(l,'textNoCase'));
// ULocale, icu4j locale, dateFormat, timeFormat, etc.
icu4jLocale=createObject("java","com.ibm.icu.util.ULocale");
icu4jDateFormat=createObject("java","com.ibm.icu.text.DateFormat");
icu4jNumberFormat=createObject("java","com.ibm.icu.text.NumberFormat");
```

Listing D.1 (CONTINUED)

```
// table start & header
writeoutput("<table width='75%'><tr align='left'><th>locale</th>
<th>date</th><th>time</th><th>number</th>
<th>currency</th></tr>");
// loop over the locales
for (i=1; i <= arrayLen(l);i++) {
  //get the actual locales, verbose for illustration
  if (listLen(l[i],"_")==1)
    thisLocale=locale.init(l[i]);
  if (listLen(l[i],"_")==2) {
    language=listFirst(l[i],"_");
    country=listLast(l[i],"_");
    thisLocale=locale.init(language,country);
  }
  if (listLen(l[i],"_")==3) {
    language=listFirst(l[i],"_");
    country=listgetAt(l[i],2,"_");
    variant=listLast(l[i],"_");
    thisLocale=locale.init(language,country,variant);
  }
  // get the comparable icu4j locale
  thisICU4JLocale=icu4jLocale.forLocale(thisLocale);
  // compare dateformats
  // love how easy this is in cf
  cfDF=lsDateFormat(now,"FULL",thisLocale.toString());
  icu4jDF=icu4jDateFormat.getDateInstance(icu4jDateFormat.FULL,
  thisICU4JLocale);
  df=icu4jDF.format(now());
  if (cfDF==df)
    dfResult="=";
  else
    dfResult=chr(8800);
  // compare time formats
  cfTF=lsTimeFormat(now,"MEDIUM",thisLocale.toString());
icu4jTF=icu4jDateFormat.getTimeInstance(icu4jDateFormat.MEDIUM,
  thisICU4JLocale);
  tf=icu4jTF.format(now());
  if (cfTF==tf)
    tfResult="=";
  else
    tfResult=chr(8800);
  // compare number formats
  // yup still loving how easy this is in cf
  cfNF=lsNumberFormat(aNumber,"_,___",thisLocale.toString());
  icu4jNF=icu4jNumberFormat.getInstance(thisICU4JLocale);
  nf=icu4jNF.format(aNumber);
  if (cfNF==nf)
    nfResult="=";
  else
    nfResult=chr(8800);
  // compare currency formats
  // nope, still not bored yet w/how easy this
  cfCF=lsCurrencyFormat(aCurrency,"local",thisLocale.toString());
  icu4jCF=icu4jNumberFormat.getCurrencyInstance(thisICU4JLocale);
  cf=icu4jCF.format(aCurrency);
  if (cfCF==cf)
```

Listing D.1 (CONTINUED)

```
      cfResult="=";
    else
      cfResult=chr(8800);
    writeoutput("<tr><td>#thisLocale.toString()#
  #thisICU4JLocale.getDisplayname()#</td><td>#dfResult#</td>
    <td>#tfResult#</td><td>#nfResult#</td>
    <td>#cfResult#</td></tr>");
  } //for
  </cfscript>
```

Table D.1 summarizes locale formatting for ColdFusion 8 and ICU4J supported locales.

Table D.1 Locale Formatting Between ColdFusion 8 and ICU4J Supported Locales

LOCALE	DATE	TIME	NUMBER	CURRENCY
ar Arabic	≠	≠	≠	≠
ar_AE Arabic (United Arab Emirates)	≠	≠	≠	≠
ar_BH Arabic (Bahrain)	≠	≠	≠	≠
ar_DZ Arabic (Algeria)	≠	≠	≠	≠
ar_EG Arabic (Egypt)	≠	≠	≠	≠
ar_IQ Arabic (Iraq)	≠	≠	≠	≠
ar_JO Arabic (Jordan)	≠	≠	≠	≠
ar_KW Arabic (Kuwait)	≠	≠	≠	≠
ar_LB Arabic (Lebanon)	≠	≠	≠	≠
ar_LY Arabic (Libya)	≠	≠	≠	≠
ar_MA Arabic (Morocco)	≠	≠	≠	≠
ar_OM Arabic (Oman)	≠	≠	≠	≠
ar_QA Arabic (Qatar)	≠	≠	≠	≠
ar_SA Arabic (Saudi Arabia)	≠	≠	≠	≠
ar_SD Arabic (Sudan)	≠	≠	≠	≠
ar_SY Arabic (Syria)	≠	≠	≠	≠
ar_TN Arabic (Tunisia)	≠	≠	≠	≠
ar_YE Arabic (Yemen)	≠	≠	≠	≠
be Belarusian	≠	=	=	=
be_BY Belarusian (Belarus)	≠	=	=	≠
bg Bulgarian	≠	=	=	=
bg_BG Bulgarian (Bulgaria)	≠	=	=	≠
ca Catalan	≠	=	=	=
ca_ES Catalan (Spain)	≠	=	=	≠
cs Czech	≠	=	=	=
cs_CZ Czech (Czech Republic)	≠	=	=	=
da Danish	≠	≠	=	=
da_DK Danish (Denmark)	≠	≠	=	=
de German	=	=	=	=
de_AT German (Austria)	=	=	=	=
de_CH German (Switzerland)	=	=	=	=
de_DE German (Germany)	=	=	=	=
de_LU German (Luxembourg)	=	=	=	=
el Greek	≠	=	=	=
el_CY Greek (Cyprus)	≠	=	=	≠
el_GR Greek (Greece)	≠	=	=	≠
en English	=	=	=	=

Table D.1 (continued)

en_AU English (Australia)	=	=	=	=
en_CA English (Canada)	=	=	=	=
en_GB English (United Kingdom)	=	=	=	=
en_IE English (Ireland)	≠	=	=	=
en_IN English (India)	≠	=	=	≠
en_MT English (Malta)	=	=	=	=
en_NZ English (New Zealand)	=	=	=	=
en_PH English (Philippines)	=	=	=	=
en_SG English (Singapore)	≠	≠	=	=
en_US English (United States)	=	=	=	=
en_ZA English (South Africa)	≠	≠	=	≠
es Spanish	=	=	=	=
es_AR Spanish (Argentina)	=	=	=	=
es_BO Spanish (Bolivia)	=	≠	=	≠
es_CL Spanish (Chile)	=	=	=	≠
es_CO Spanish (Colombia)	=	=	=	=
es_CR Spanish (Costa Rica)	=	≠	≠	≠
es_DO Spanish (Dominican Republic)	=	≠	=	=
es_EC Spanish (Ecuador)	=	=	=	=
es_ES Spanish (Spain)	=	=	=	=
es_GT Spanish (Guatemala)	=	≠	=	=
es_HN Spanish (Honduras)	=	≠	=	=
es_MX Spanish (Mexico)	=	≠	=	=
es_NI Spanish (Nicaragua)	≠	≠	=	≠
es_PA Spanish (Panama)	=	≠	=	≠
es_PE Spanish (Peru)	=	=	=	≠
es_PR Spanish (Puerto Rico)	=	≠	=	=
es_PY Spanish (Paraguay)	=	≠	=	≠
es_SV Spanish (El Salvador)	=	≠	=	≠
es_US Spanish (United States)	=	=	=	=
es_UY Spanish (Uruguay)	=	≠	=	≠
es_VE Spanish (Venezuela)	=	≠	=	≠
et Estonian	≠	=	=	=
et_EE Estonian (Estonia)	≠	=	=	≠
fi Finnish	≠	≠	=	≠
fi_FI Finnish (Finland)	≠	≠	=	=
fr French	=	=	=	=
fr_BE French (Belgium)	=	=	=	=
fr_CA French (Canada)	=	=	=	=
fr_CH French (Switzerland)	≠	=	=	=
fr_FR French (France)	=	=	=	=
fr_LU French (Luxembourg)	=	=	≠	≠
ga Irish	=	=	=	=
ga_IE Irish (Ireland)	=	=	=	=
hi_IN Hindi (India)	≠	=	≠	≠
hr Croatian	≠	=	=	=
hr_HR Croatian (Croatia)	≠	=	=	=
hu Hungarian	=	=	=	=
hu_HU Hungarian (Hungary)	=	=	=	≠
in Indonesian	=	=	=	=
in_ID Indonesian (Indonesia)	=	=	=	=
is Icelandic	≠	=	=	=

Table D.1 (continued)

is_IS Icelandic (Iceland)	≠	=	=	=
it Italian	=	≠	=	=
it_CH Italian (Switzerland)	≠	=	=	=
it_IT Italian (Italy)	=	≠	=	=
iw Hebrew	=	=	=	=
iw_IL Hebrew (Israel)	=	=	=	≠
ja Japanese	≠	=	=	=
ja_JP Japanese (Japan)	≠	=	=	≠
ja_JP_JP Japanese (Japan, JP)	≠	=	=	≠
ko Korean	=	≠	=	=
ko_KR Korean (South Korea)	=	=	=	≠
lt Lithuanian	≠	≠	=	=
lt_LT Lithuanian (Lithuania)	≠	≠	=	=
lv Latvian	≠	=	=	=
lv_LV Latvian (Latvia)	≠	=	=	=
mk Macedonian	≠	≠	=	=
mk_MK Macedonian (Macedonia)	≠	≠	=	≠
ms Malay	=	=	=	=
ms_MY Malay (Malaysia)	=	=	=	=
mt Maltese	=	=	=	=
mt_MT Maltese (Malta)	=	=	=	=
nl Dutch	=	=	=	=
nl_BE Dutch (Belgium)	=	=	=	=
nl_NL Dutch (Netherlands)	=	=	=	=
no Norwegian	≠	≠	=	=
no_NO Norwegian (Norway)	≠	≠	=	≠
no_NO_NY Norwegian (Norway, NY)	≠	≠	=	≠
pl Polish	≠	=	=	=
pl_PL Polish (Poland)	≠	=	=	=
pt Portuguese	=	=	=	=
pt_BR Portuguese (Brazil)	=	=	=	=
pt_PT Portuguese (Portugal)	=	=	=	=
ro Romanian	=	=	=	=
ro_RO Romanian (Romania)	=	=	=	≠
ru Russian	≠	=	=	≠
ru_RU Russian (Russia)	≠	=	=	≠
sk Slovak	≠	=	=	=
sk_SK Slovak (Slovakia)	≠	=	=	=
sl Slovenian	≠	=	=	=
sl_SI Slovenian (Slovenia)	≠	=	=	≠
sq Albanian	≠	=	=	=
sq_AL Albanian (Albania)	≠	=	=	=
sr Serbian	=	=	=	=
sr_BA Serbian (Bosnia and Herzegovina)	=	=	=	≠
sr_CS Serbian (Serbia And Montenegro)	=	=	=	≠
sv Swedish	≠	≠	=	=
sv_SE Swedish (Sweden)	≠	≠	=	≠
th Thai	**a**	≠	=	=
th_TH Thai (Thailand)	≠	=	=	=
th_TH_TH Thai (Thailand, TH)	**b**	**b**	=	**b**
tr Turkish	=	=	=	=
tr_TR Turkish (Turkey)	=	=	=	≠

Table D.1 (CONTINUED)

uk Ukrainian	=	=	≠	≠
uk_UA Ukrainian (Ukraine)	=	=	≠	≠
vi Vietnamese	=	=	=	≠
vi_VN Vietnamese (Vietnam)	=	=	=	=
zh Chinese	≠	=	=	=
zh_CN Chinese (China)	≠	≠	=	=
zh_HK Chinese (Hong Kong SAR China)	≠	=	=	=
zh_SG Chinese (Singapore)	=	=	=	=
zh_TW Chinese (Taiwan)	≠	≠	=	=

Key:
= ColdFusion 8 and ICU4J formats are equal.
≠ ColdFusion 8 and ICU4J formats are not equal.
a ColdFusion 8/core Java “helpfully” converts Thai locale dates to Buddhist calendar; ICU4J does not.
b ICU4J does not format using Thai digits, which is incorrect for this locale variant.