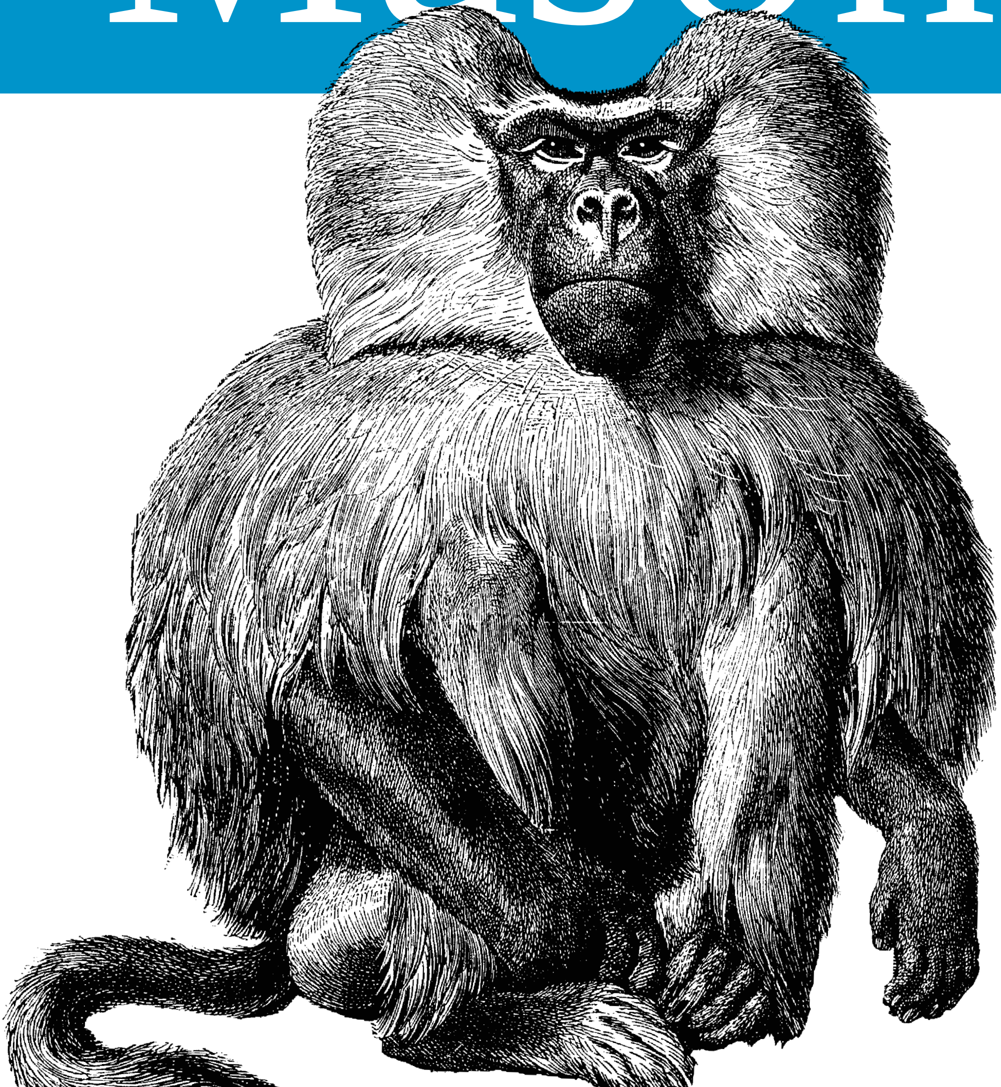
Component-based Templating System

Embedding Perl in HTML with Mason



O'REILLY®

Dave Rolsky & Ken Williams

Embedding Perl in HTML with Mason

Dave Rolsky and Ken Williams

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 5

Advanced Features

In the previous chapters you have been introduced to the basic features of Mason, and you should have a fairly good idea by now of how you might actually go about constructing a dynamic web site from Mason components. You have seen a few of Mason's unique features, such as the autohandler mechanism, the dhandler mechanism, and the ability to pass arbitrary data between components.

In this chapter we'll go beyond the basics and learn more about advanced ways to use Mason components to design large dynamic sites. You'll learn how to define multiple components in the same text file, how to create components on the fly from Perl strings, how to manage multiple component root directories, and (finally!) how to use all of Mason's object-oriented features.

Subcomponents

Although we often imagine a one-to-one correspondence between text files and Mason components, it is actually possible to define multiple components in a single text file. This is achieved by using a `<%def></%def>` block, a special Mason directive that defines one component from within another. The component embedded within the `<%def>` block is called a *subcomponent*, and it is visible only to the component within which it resides: component A may not access component B's subcomponents directly.

The subcomponent may use any of the standard Mason component directives, such as `<%args>`, `<%init>`, `%`-lines, and so on. The only exceptions are that you may not use `<%def>` or `<%method>` blocks within subcomponents nor may you use "global" blocks like `<%once>` or `<%shared>`.

Subcomponents are most useful when you have some piece of processing to repeat several times that is used only in a certain specific situation and doesn't merit its own separate component file.

Here is an example of defining and calling a subcomponent. Note that the component is assigned a name inside the <%def> tag (the name often starts with a period, purely by convention) and that you use the regular component-calling mechanisms ($m->comp() or a <& &> tag) to invoke it.

```
<h2>Information about certain Minnesota cities:</h2>

% my @cities = ("Young America", "Sleepy Eye", "Nisswa", "Embarrass",
%               "Saint Cloud", "Little Canada", "Burnsville", "Luverne");
% foreach my $name (@cities) {
 <hr>
 <& .city_info, city => $name, state => 'MN' &>
% }

<%def .city_info>
<%args>
 $city
 $state
</%args>
 <table border="2">
  <tr> <th colspan="2"><% $city %></th> </tr>
  <tr> <td>Population:</td>  <td><% $population %></td>             </tr>
  <tr> <td>Coordinates:</td> <td><% "$latitude, $longitude" %></td> </tr>
  <tr> <td>Mayor:</td>       <td><% $mayor %></td>                  </tr>
 </table>
<%init>
 my ($population, $latitude, $longitude, $mayor) =
   $dbh->selectrow_array("SELECT population, latitude, longitude, mayor
                          FROM cities
                          WHERE city=? and state=?",
                          undef, $city, $state);
</%init>
</%def>
```

Since a subcomponent is visible only to the component that defines, and because it has all the capabilities that regular components have, you may think of subcomponents as roughly analogous to privately scoped anonymous subroutine references in Perl.

Creating Components on the Fly

You may encounter situations in which you want to use Mason's templating features and data management tools, but you don't want to create a full-blown component root hierarchy on disk to house your components. Perhaps you want to create a component from an isolated file or directly from a string containing the component text.

For these situations, the Mason interpreter provides the make_component() method. It accepts a comp_file or comp_source parameter (letting you create a component from a file or a string, respectively) and returns a Component object.

```
# Creating a component from scratch
#!/usr/bin/perl -w

use strict;
use HTML::Mason;

my $source = <<'EOF';
<%args>
 $planet
</%args>
Hello, <% $planet %>!
EOF

my $interp = HTML::Mason::Interp->new();
my $comp = $interp->make_component(comp_source => $source);
$interp->exec($comp, planet => 'Neptune');
```

And here is a component that creates another component at runtime:

```
<& $comp &>

<%init>
 my $comp = $m->interp->make_component(
   comp_file => '/home/slappy/my_comps/foo',
 );
</%init>
```

Of course, creating components at runtime is slower than creating them ahead of time, so if you need to squeeze out all the performance you possibly can, you might need to think of a speedier method to achieve your goals. And as always, benchmark everything so you really know what the effects are.

If the compiler encounters syntax errors when attempting to compile the component, a fatal exception will be thrown inside the `make_component()` method. If you want to trap these errors, you may wrap the `make_component()` method in Perl's `eval {}` block, and check `$@` after the method call.

Sharing Data Among Component Sections

By default, the scope of variables created within an `<%init>` block, a Perl line, or any other Mason markup sections is the entire component. This is tremendously convenient, because it lets you initialize variables in the `<%init>` block, then use their values across the rest of the component. So most of the time, the techniques discussed in this section won't be needed.

There is one limitation to variables created within the `<%init>` section, however: their values won't be seen by any subcomponents you might define. This is true for two reasons. First, the subcomponents may themselves contain an `<%init>` section, so the relevance of the main component's `<%init>` section isn't necessarily clear. Second, a subcomponent may actually be a *method* (more on this later), in which case it is accessible to the outside world without first calling the main component, so the <%init> section never has a chance to run.

Sometimes you need to share data between a component and its subcomponents, however, and for these situations Mason provides the <%shared> and <%once> blocks. A <%shared> block runs before the main component or any of its methods or subcomponents and may run initialization code. Any variables created here will be visible to the entire main component and any of its subcomponents, including the main component's <%init> section, if any. The <%once> block is similar—the only difference is that code in the <%once> block won't run every time the component is called. It will run only when the component itself is loaded. The initialized values will remain intact for the lifetime of the component object, which may be until you make changes to the component source file and Mason reloads it or until the web server child expires and gets replaced by a new one.

A <%shared> section is great when a component and its subcomponents have a tight relationship and may make complicated use of shared data. In contrast, <%once> sections are useful for caching values that change infrequently but may take a long time to compute. See Example 5-1.

Example 5-1. sharing_example.mas

```
<%def .subcomponent>
 visible $color in .subcomponent is <% $color %>
</%def>

visible $color in main component is <% $color %>
<& .subcomponent &>

<%shared>
 my $color = 'bone';
</%shared>
```

A similar example, but using a <%once> section, is shown in Example 5-2.

Example 5-2. once_example.mas

```
<%def .subcomponent>
 visible $flavor in .subcomponent is <% $flavor %>
</%def>

visible $flavor in main component is <% $flavor %>
<& .subcomponent &>

<%once>
 my $flavor = 'gamey';
</%once>
```

A cautionary note about the <%shared> and <%once> sections: they do *not* let you transparently share data among Apache children (this would require actual shared memory segments and can be done with modules like IPC::Shareable), or among multiple components (this can easily be done with global variables). It is also unwise to use variables created in a <%once> section for saving state information that you intend to change, since the next time the component is loaded your changes will be lost.

You should also remember that variables defined via an <%args> block are not visible in a <%shared> block, meaning that the only access to arguments inside a shared block is via the %ARGS hash or one of the request object methods such as request_args.

Methods and Attributes

The ability to use Mason's component-level object-oriented methods and attributes can give you powerful techniques for managing your site.

As explained in Chapter 3, one of the major benefits of object-oriented techniques is that they help you reduce redundancy in your site. Site redundancy is a much bigger problem than most people realize. How many times have you forgone a site revision because performing the revision would be "too intrusive," and you can't afford the downtime? How many Internet web sites have you seen that look promising at first, but fail to fix problems and don't adapt to usage patterns over the long run? Nobody likes to be stuck with an unmaintainable site, and the only way to avoid it is to design the site to be adaptable and extensible in the first place. Eliminating redundancy goes a long way toward this goal.

Methods

Methods in Mason are actually quite simple. A method is just like a subcomponent, but instead of defining it with a <%def> section, you use a <%method> section:

```
<%method .my_method>
 Any regular component syntax here...
</%method>
```

The difference between subcomponents and methods is primarily in how they can be invoked from other components. A method can only be invoked using special method syntax. We present three ways of doing this here:

```
# Fetch the bottommost child of the current component
my $self = $m->base_comp;
$self->call_method('.my_method');

# Shortcut for the above two lines
$m->comp('SELF:.my_method');

# Same thing, using <& &> syntax
<& SELF:.my_method &>
```

Let's think about what happens when you invoke a method. Suppose there is a component called */staff/flintoff.mas*, whose parent is */staff/autohandler*, whose parent is in turn */autohandler*. While *any* of these components are executing (which might be when a top-level request comes in for */staff/flintoff.mas* or when */staff/flintoff.mas* is called from another component), calling `$m->base_comp` from within any of these three components will return a component object representing */staff/flintoff.mas*. In the example, that component object is stored in `$self`. Invoking `call_method('.my_method')` will search `$self` and its hierarchy of parents for a method called `.my_method`, starting the search at `$self` and proceeding upward. If such a method is found, it gets executed. If no such method is found, a fatal error occurs. You may want to call `$self->method_exists('.my_method')` first if you're not sure whether the method exists.

Remember that methods are full-blown subcomponents, so you may also pass them arguments when you invoke them. Examples 5-3 and 5-4 demonstrate a more sophisticated example of method invocation.

Example 5-3. /autohandler

```
<html>
<& $m->call_next &>
</html>
<%method .body_tag>
 <%args>
  $bgcolor => 'white'
  $textcolor => 'black'
 </%args>
 <body onLoad="prepare_images()" bgcolor="<% $bgcolor %>" text="<% $textcolor %>">
</%method>
```

Example 5-4. /important_advice.mas

```
<head><title>A Blue Page With Red Text</title></head>

<& SELF:.body_tag, bgcolor=>'blue', textcolor=>'red' &>
 Never put anything bigger than your elbow into your ear.
</body>
```

The central thing to note about this example is the way the main component and the autohandler cooperate to produce the `<body>` tag. The designer of this site has chosen to make the `bgcolor` and `textcolor` page attributes configurable by each page, and the autohandler will generate the rest, including the call to the JavaScript function `prepare_images()`.

Incidentally, note that the autohandler took responsibility for the `<html>` and `</html>` tags, while the main component generated everything in the `<head>` and `<body>` sections. This is not necessarily good design—you must determine the right factorization for each site you create—but it made the example straightforward.

Now that you know what methods are and how they work, we can explore some ways that you can use them to design your site to be flexible and maintainable.

Using Methods for Titles and Headers

The most familiar example of commonality within a site's structure is probably the overall design of pages. Most web sites want to have a common design structure across multiple pages, including common colors and fonts, common navigational elements and headers, common keywords in `<META>` tags, and so on. In this section, we explore how you can use methods for the specific problem of generating commonly styled headers and titles for your pages.

Generating titles and headers was the major motivation behind developing Mason's method capabilities in the first place. Consider for a moment the "title and header problem": it is often desirable to control the top and bottom of an HTML page centrally, for all the reasons we've tried to drum into your skull throughout this chapter. However, while large portions of the top and bottom of the page may be the same for all pages on your site, certain small pieces may be different on every page—titles and headers often fall into this category. So, you would like a way to generate the large common portions of your pages centrally but insert the noncommon titles and headers where they belong.

Mason's methods provide a perfect answer. Each title and header can be specified using a method. Then an autohandler can generate the common headers and footers, calling the base component's title and header methods to insert the page-specific information in its proper place (Examples 5-5 and 5-6).

Example 5-5. autohandler

```
<html>
<head><title><& SELF:title &></title></head>
<body>
 <center><h3><& SELF:header &></h3></center>
% $m->call_next;
 <center><a href="/">-home-</a></center>
</body>
</html>
<%method title>
 www.Example.com
</%method>
<%method header>
 Welcome to Example.com
</%method>
```

Example 5-6. fancy_page.html

```
<p>This page isn't all <i>that</i> fancy, but it might be the
fanciest one we've seen yet.</p>
<%method title>
 Fancy Page
</%method>

<%method header>
```

Example 5-6. fancy_page.html (continued)

```
 A Very Fancy Page
</%method>
```

The autohandler provides a default title and header, so if the base component `fancy_page.html` didn't provide a title or header method, the autohandler would use its default values. If none of the components in the parent hierarchy (*autohandler* and *fancy_page.html* in this case) defines a certain method and that method is invoked, a fatal exception will be thrown. If you don't want to have a default title and header, ensuring that each page sets its own, you can simply omit the default methods in the autohandler. If a page fails to set its title or header, you will know it pretty quickly in the development cycle.

Remember that methods are Mason components, so they can contain more than just static text. You might compute a page's title or header based on information determined at runtime, for example.

Methods with Dynamic Content

As you know, methods and inheritance may be used to let a page and its autohandler share the responsibility for generating page elements like headers and titles. Since these elements may often depend on user input or other environmental conditions (e.g., "Welcome, Jon Swartz!" or "Information about your 9/13/2001 order"), you'll need a way to set these properties (like "Jon Swartz" or "9/13/2001") at runtime. Why is this an issue? Well, the following *won't* work:

```
<%method title>
 <!-- this method is invoked in the autohandler -->
 Information about your <% $order_date %> order
</%method>

Your order included the following items:
 ...generate item listing here...

<%init>
 my $order_date = $session{user}->last_order_date;
</%init>
```

The reason that won't work is that variables set in the `<%init>` block won't be visible inside the `<%method title>` block. Even if the scope of `$order_date` included the `<%method title>` block (it doesn't), the sequence of events at runtime wouldn't allow its value to be seen:

1. A request for */your_order.html* is received. Mason constructs the runtime inheritance hierarchy, assigning */autohandler* as */your_order.html*'s parent.
2. Mason executes the */autohandler* component, which invokes its `SELF:title` method. The `title` method invoked is the one contained in */your_order.html*.

3. The /your_order.html:title method runs, and the value of the $order_date is still unset—in fact, the variable is undeclared, so Perl will complain that the Global symbol "$order_date" requires explicit package name. Let's suppose you trapped this error with eval {}, so that we can continue tracing the sequence of events.
4. Control returns to */autohandler*, which eventually calls $m->call_next and passes control to */your_order.html*.
5. */your_order.html* runs its <%init> section and then its main body. Note that it would set $order_date much too late to affect the title method back in step 3.
6. */your_order.html* finishes and passes control back to */autohandler*, and the request ends.

What's a Mason designer to do? The solution is simple: use a <%shared> block instead of an <%init> block to set the $order_date variable. This way, the variable can be shared among all the methods of the */your_order.html* component, and it will be set at the proper time (right before step 2 in the previous listing) for it to be useful when the methods are invoked.

The proper code is remarkably similar to the improper code; the only difference is the name of the block in which the $order_date variable is set:

```
<%method title>
 <!-- this method is invoked in the autohandler -->
 Information about your <% $order_date %> order
</%method>

Your order included the following items:
 ...generate item listing here...

<%shared>
 my $order_date = $session{user}->last_order_date;
</%shared>
```

<%shared> blocks are executed only once per request, whenever the first component sharing the block needs it. Its scope lasts only to the end of the request. Because of this, <%shared> blocks are ideal for sharing scoped variables or performing component-specific initialization code that needs to happen only once per request.

Now imagine another scenario, one in which the method needs to examine the incoming arguments in order to generate its output. For instance, suppose you request /view_user.html?id=2982, and you want the title of the page to display some information about user 2982. You'll have to make sure that the user ID is available to the method, because under normal conditions it isn't. The two most common ways to get this information in the method are either for the method to call $m->request_args() or for the autohandler to pass its %ARGS to the method when calling it. The method could then either declare $id in an <%args> block or examine the incoming %ARGS hash directly. An example using request_args() follows:

```
<%method title>
 <!-- this method is invoked in the autohandler -->
 User page for <% $user->name %>
</%method>

 ... display information about $user ...

<%shared>
 my $user = MyApp::User->new(id => $m->request_args->{id});
</%shared>
```

Note that we cached the $user object with a shared variable so that we didn't have to create a new user object twice.

Attributes

Sometimes you want to take advantage of Mason's inheritance system, but you don't necessarily need to inherit the full components. For instance, in our first title and header example, the title and header methods contained just plain text and didn't use any of the dynamic capabilities of components. You might therefore consider it wasteful in this case to bring the full component-processing system to bear on the generation of headers and footers.

If you find yourself in this situation, Mason's component *attributes* may be of interest. An attribute is like a method in the way its inheritance works, but the value of an attribute is a Perl scalar variable, not a Mason component.

Examples 5-7 and 5-8 rewrite our previous autohandler example using attributes instead of methods.

Example 5-7. autohandler

```
<html>
<head><title><% $m->base_comp->attr('title') %></title></head>
<body>
 <center><h3><% $m->base_comp->attr('header') %></h3></center>
% $m->call_next;
 <center><a href="/">-home-</a></center>
</body>
</html>

<%attr>
 title  => "FancyMasonSite.Example.com"
 header => "Welcome to FancyMasonSite.Example.com"
</%attr>
```

Example 5-8. fancy_page.html

```
<p>This page isn't all <i>that</i> fancy, but it might be the
fanciest one we've seen yet.</p>

<%attr>
 title  => "Fancy Page"
 header => "A Very Fancy Page"
</%attr>
```

Attributes can be used like this, for small bits of text that become part of every page, but whose values vary from page to page. They can also be used to set properties of the component, such as whether to display a certain navigation bar or to omit it, whether the user must have certain characteristics in order to view this page,* and so on.

In the current version of Mason, each attribute in an `<%attr>` block must be on a single line. This means that you cannot use multiple lines for clarity or to specify multiline values. Future versions of Mason may provide additional syntax options for multiline attributes. If you run up against this limitation, you may want to use a method instead of an attribute anyway, since methods can more easily deal with more complex definitions.

Another limitation of attributes in the current version of Mason is that their values are completely static properties of the component and can't change from one request to the next. This may or may not be addressed by a future version of Mason. In any case, if you think you need dynamic attributes, you probably actually need to use methods instead.

Top-Down Versus Bottom-Up Inheritance

The attentive reader will have noticed that there are two distinct facets to Mason's inheritance—the first is the "content wrapping" behavior, by which a parent component can output some content, then call its child component, then output some more content. The typical example of this is an autohandler that generates headers and footers, wrapped around the output of its base component. The second facet of inheritance is the method and attribute system, by which designers can define general behavior and properties in the parent components and specific behavior in the children.

A major difference between these two facets is the direction of inheritance. Mason will begin the search for methods and attributes by starting with the bottommost

* In a `mod_perl` setting, authentication and authorization often happen before the content generation phase (i.e., before Mason even steps into the picture). However, you may wish to bypass the `auth` control phases and do your own authorization in the autohandler, just so that you can use Mason's attributes to control the behavior. For instance, you might give an unauthenticated user a different view of a certain page, rather than denying access outright.

child and working its way toward the parent, but it will begin the component's content generation by starting at the topmost parent and working its way toward the bottommost child (assuming each parent calls `$m->call_next`).

Although this behavior may seem odd if this is the first time you've encountered it, the inheritance system will seem like second nature after you've worked with it for a while. To help form an intuitive notion of what's happening, simply remember that autohandlers (or other parent components) specify general behavior, whereas top-level (i.e., child; i.e., "regular") components dictate specific behavior, overriding the parents.

Calling Components with Content Blocks

As you saw earlier, `<%filter>` blocks can be quite handy. The example we showed in Chapter 3 altered the `src` attribute of `<img>` tags in order to point them to a different server.

In Chapter 8 we will show an example that filters a link menu of `<a href>` tags to find the link for the current page and changes it to a `<b>` tag instead, in order to highlight the current page.

Both of these examples work just fine as long as we are willing to filter the output of an entire component, but sometimes we'd like to limit the filtering to just one section of the component. Consider the following autohandler:

```
<html>
<head>
<title><& SELF:title &></title>
</head>
<body>

<& SELF:top_menu, %ARGS &>

% $m->call_next;

</body>
</html>
```

This component calls the `top_menu` method, expecting it to produce some sort of menu of links. We'd like to use the menu-filtering trick just mentioned, but using a regular `<%filter>` block in this component would filter not only the menu of links but also the entire page. That's a waste of processing, not to mention a potential source of bugs—and we hate bugs.

Another option would be to insert a `<%filter>` block directly into the source of the `top_menu` method. However, the method may be defined in many different places; the whole point of using a method instead of a regular component call is that any component may redefine the method as it chooses. So we'd end up adding the *same* filter block to every definition of the `top_menu` method. That's a pretty poor solution.

What we really want is a solution that allows us to write the code once but apply it to only the portion of the output that we choose. Of course, there is such a thing called a "component call with content," introduced in Mason Version 1.10. It looks just like a regular `<& &>` component call, except that there's an extra pipe (`|`) character to distinguish it and a corresponding end tag, `</&>`. Using a component call with content, we can apply the desired filter to just the menu of links:

```
<html>
<head>
<title><& SELF:title &></title>
</head>
<body>

<&| .top_menu_filter &>
 <& SELF:top_menu, %ARGS &>
</&>

% $m->call_next;

</body>
</html>
```

So the `.top_menu_filter` component—presumably a subcomponent defined in the same file—is somehow being passed the output from the call to `<& SELF:top_menu, %ARGS &>`. The `.top_menu_filter` component would look something like this:

```
<%def .top_menu_filter>
% my $text = $m->content;
% my $uri = $r->uri;
% $text =~ s,<a href="\Q$uri\E[^"]*">([^<]+)</a>,<b>$1</b>,;
<% $text %>
</%def>
```

This looks more or less like any other `<%filter>` block, but with two main differences. First, the body of a `<%filter>` block contains plain Perl code, but since `.top_menu_filter` is a subcomponent, it contains Mason code. Second, we access the text to filter via a call to `$m->content` instead of in the `$_` variable. The `$m->content()` method returns the evaluated output of the content block, which in this case is the output of the `SELF:top_menu` component.

Mason goes through some contortions in order to trick the wrapped portion of the component into thinking that it is still in the original component. If we had a component named *bob.html*, as shown in the example below:

```
<&| .uc &>
I am in <% $m->current_comp->name %>
</&>

<%def .uc>
<% uc $m->content %>
</%def>
```

we would expect the output to be:

```
I AM IN BOB.HTML
```

And indeed, that is what will happen. You can also nest these sorts of calls:

```
<&| .ucfirst &>
  <&| .reverse &>
I am in <% $m->current_comp->name %>
  </&>
</&>

<%def .reverse>
<% scalar reverse $m->content %>
</%def>
<%def .ucfirst>
<% join ' ', map {ucfirst} split / /, $m->content %>
</%def>
```

This produces:

```
Lmth.bob Ni Ma I
```

As you can see, the filtering components are called from innermost to outermost.

It may have already occurred to you, but this can actually be used to implement something in Mason that looks a lot like Java Server Page taglibs. Without commenting on whether the taglib concept is conducive to effective site management or not, we'll show you how to create a similar effect in Mason. Here's a simple SQL select expressed in something like a taglib style:

```
<table>
 <tr>
  <th>Name</th>
  <th>Age</th>
 </tr>
<&| /sql/select, query => 'SELECT name, age FROM User' &>
 <tr>
  <td>%name</td>
  <td>%age</td>
 </tr>
</&>
</table>
```

The idea is that the `query` argument specifies the SQL query to run, and the content block dictates how each row returned should be displayed. Fields are indicated here by a `%` and then the name of the field.

Now let's write the */sql/select* component.

```
<%args>
 $query
</%args>
<%init>
 my $sth = $dbh->prepare($query);
 while ( my $row = $sth->fetchrow_hashref ) {
   my $content = $m->content;
   $content =~ s/%(\w+)/$row->{$1}/g;
   $m->print($content);
 }
</%init>
```

Obviously, this example is grossly simplified (it doesn't handle things like bound SQL variables, and it doesn't handle extra embedded `%` characters very well), but it demonstrates the basic technique.

Seeing all this, you may wonder if you can somehow use this feature to implement control structures, again a taglib-esque idea. The answer is yes, with some caveats. We say "with caveats" because due to the way this feature is implemented, with closures, you have to jump through a few hoops. Here is something that will *not* work:

```
<&| /loop, items => ['one', 'two', 'three'] &>
<% $item %>
</&>
```

And in */loop*:

```
<%args>
 @items
</%args>
% foreach my $item (@items) {
<% $m->content %>
% }
```

Remember, the previous example will not work. The reason should be obvious. At no time is the variable `$item` declared in the calling component, either as a global or lexical variable, so a syntax error will occur when the component is compiled.

So how can this idea be made to work? Here is one way. Rewrite the calling component first:

```
% my $item;
<&| /loop, items => ['one', 'two', 'three'], item => \$item &>
<% $item %>
</&>
```

Then rewrite */loop*:

```
<%args>
 $item
 @items
</%args>
% foreach (@items) {
%   $$item = $_;
<% $m->content %>
% }
```

This takes advantages of how Perl treats lexical variables inside closures, but explaining this in detail is *way* beyond the scope of this book.

You can also achieve this same thing with a global variable. This next version assumes that `$item` has been declared using `allow_globals`:

```
<&| /loop, items => ['one', 'two', 'three'] &>
<% $item %>
</&>
```

And */loop* becomes this:

```
<%args>
 @items
</%args>
% foreach $item (@items) {
<% $m->content %>
% }
```

This version is perhaps a little less funky, but it could lead to having more globals than you'd really like.

An in-between solution using Perl's special `$_` variable can solve many of these problems. This variable is a global but is automatically localized by loop controls like `foreach` or `while`. So we can now write:

```
<&| /loop, items => ['one', 'two', 'three'] &>
<% $_ %>
</&>
```

And for */loop*:

```
<%args>
 @items
</%args>
% foreach (@items) {
<% $m->content %>
% }
```

Magic. It isn't perfect, but it looks kind of neat.

In any case, Mason was designed to use Perl's built-in control structures, so we don't feel too bad that it's awkward to build your own.

Advanced Inheritance

In Chapter 3 we introduced you to the concept of component inheritance, and in this chapter we have discussed some of the ways you can use inheritance to create flexible, maintainable Mason sites. Now we show how inheritance interacts with other Mason features, such as multiple component roots and multiple autohandlers.

Inheritance and Multiple Component Roots

It is possible to tell Mason to search for components in more than one directory—in other words, to specify more than one component root. This is analogous to telling

Perl to look for modules in the various @INC directories or to telling Unix or Windows to look for executable programs in your PATH. In Chapters 6 and 7 you will learn more about how to configure Mason; for now, we will just show by example:

```
my $ah = HTML::Mason::ApacheHandler->new(
  comp_root => [
                [main => '/usr/http/docs'],
                [util => '/usr/http/mason-util'],
               ]
);
```

or, in an Apache configuration file:

```
PerlSetVar MasonCompRoot 'main => /usr/http/docs'
PerlAddVar MasonCompRoot 'util => /usr/http/mason-util'
```

This brings up some interesting inheritance questions. How do components from the two component roots relate to each other? For instance, does a component in */usr/http/docs* inherit from a top-level autohandler in */usr/http/mason-util*? With this setup, under what conditions will a component call from one directory find a component in the other directory? The answers to these questions are not obvious unless you know the rules.

The basic rule is that Mason always searches for components based on their component paths, not on their source file paths. It will be perfectly happy to have a component in one component root inherit from a component in another component root. When calling one component from another, you always specify only the path, not the particular component root to search in, so Mason will search all roots.

If it helps you conceptually, you might think of the multiple component roots as getting merged together into one big über-root that contains all the files from all the multiple roots, with conflicts resolved in favor of the earliest-listed root.

Let's think about some specific cases. Using the two component roots given previously, suppose you have a component named */dir/top_level.mas* in the main component root and a component named */dir/autohandler* in the util component root. */dir/top_level.mas* will inherit from */dir/autohandler* by default. Likewise, if */dir/top_level.mas* calls a component called *other.mas*, Mason will search for *other.mas* first in the main component root, then in the utils root. It makes no difference whether the component call is done by using the component path *other.mas* or */dir/other.mas*; the former gets transformed immediately into the latter by prepending the */dir/top_level.mas*'s dir_path.

If there are two components with the same path in the main and util roots, you won't be able to call the one in the util root by path no matter how hard you try, because the one in main overrides it.

This behavior is actually quite handy in certain situations. Suppose you're creating lots of sites that function similarly, but each individual site needs to have some small tweaks. There might be small differences in the functional requirements, or you might need to put a different "look and feel" on each site. One simple way to do this is to use multiple component roots, with each site having its own private root and a shared root:

```
my $interp = HTML::Mason::Interp->new(
  comp_root => [
                 [mine   => '/etc/httpd/sites/bobs-own-site'],
                 [shared => '/usr/local/lib/mason/common'],
                ]
);
```

The `shared` root can provide a top-level autohandler that establishes a certain generic look and feel to the site, and the `mine` root can create its own top-level autohandler to override the one in `shared`.

Using this setup, any component call—no matter whether it occurs in a component located in the `mine` or `shared` component root—will look for the indicated component, first in `mine`, then in `shared` if none is found in `mine`.

An Advanced Inheritance Example

An example can help showcase several of the topics we've discussed in this chapter. The component in this section was originally written by John Williams, though we've removed a few features for the pedagogical purposes of this book. It implements an autohandler that allows you to run predefined SQL queries via a Mason component interface. For example, you might use the component as in Example 5-9.

Example 5-9. A calling component

```
<table>
<tr><th>Part Number</th><th>Quantity</th><th>Price</th><th>Total</th></tr>
<&| /query/order_items:exec, bind => [$OrderID] &>
<tr>
   <td><% $_->{PARTNUM} %></td>
   <td><% $_->{QUANTITY} %></td>
   <td><% $_->{PRICE} %></td>
   <td><% $_->{QUANTITY} * $_->{PRICE} %></td>
</tr>
</&>
</table>
```

Note that we're passing a content block to the `/query/order_items:exec` method call. The idea is that the method will repeat the content block for every database row returned by an SQL query, and the `$_` variable will hold the data for each row, as returned by the `DBI` method `fetchrow_hashref()`. The query itself is specified in the */query/order_items* file, which could look like Example 5-10.

Example 5-10. /query/order_items

```
SELECT * FROM items WHERE order_id = ?
```

Yes, it's just one line. Where is the exec method we called earlier? It's in the parent component, which (since we didn't specify otherwise with an `inherit` flag) is *query/autohandler*. This autohandler is the component that does all the work; see Example 5-11.

Example 5-11. /query/autohandler

```
<%flags>
 inherit => undef
</%flags>

<%method exec>
 <%args>
  @bind => ()
 </%args>

 <%init>
  local $dbh->{RaiseError} = 1;

  # Get the SQL from the base component
  my $sql = $m->scomp($m->base_comp, %ARGS);
  my $q = $dbh->prepare($sql);
  $q->execute(@bind);

  # Return now if called without content
  # (useful for insert/update/delete statements).
  return $dbh->rows unless defined $m->content;

  # Call the content block once per row
  local $_;
  while ($_ = $q->fetchrow_hashref('NAME_uc')) {
    $m->print( $m->content );
  }

  # Don't print any of the whitespace in this method
  return;
 </%init>
</%method>
```

Let's step our way through the autohandler. The only code outside the exec method ensures that this component is parentless. It's not strictly necessary, but we include it to make sure this example is isolated from any other interaction.

We access the database inside the exec method. Since we haven't declared the `$dbh` variable, it's assumed that it's already set up for us as a global variable, probably initialized in the site's top-level autohandler. The first thing we do is make sure that the code will throw an exception if anything goes wrong during the query, so we locally set `$dbh->{RaiseError}` to `1`. Any exceptions thrown will be the responsibility of someone higher up the calling chain.

Next, we get the text of the SQL query. It's contained in our base component, which in our example was */query/order_items*. We call this component to get its output. Note that we also pass `%ARGS` to our base component, which lets us do additional substitutions into the SQL statement. For example, we could have a query that sorts by one of several different fields, using `ORDER BY <% $ARGS{sort} %>` inside the SQL statement.

After we fetch the SQL and prepare the query, we execute the query, passing any bound variables to the `$q->execute()` method. If there was no content block passed to us, then we're done—this allows the component to be used for INSERT/ UPDATE/DELETE statements in addition to SELECT statements.

Finally, we iterate through the rows returned by the query, storing the data for each row in `$_`. Note that we localize `$_` before using it, since blowing away any value that's already in there would be extremely impolite.

Be sure to notice that the `$m->content` body is reexecuted for each row. Because of this, its execution happens within the scope of the current `$_` variable. This is really the only way this all could work, but it's subtle enough that we had to point it out.

The advantage of using inheritance this way is that you capture the complicated parts of a task in a single component, and then all the rest of the components become very simple. If you get familiar with Mason's inheritance model, you can create very sophisticated applications with a minimum of redundancy and hassle.

Subrequests

Once in a while you may want to call one component from another as if it were a top-level component, having it go through the full content-wrapping and dhandler-checking process. This is what *subrequests* are for, and the manner in which they work is similar to how subrequests work in Apache. Subrequests were introduced in Mason Version 1.10.

When executing a subrequest, you may simply execute it via the request object's `subexec()` method, or you may first create the object via `make_subrequest()` and then execute it via `exec()`. The `subexec()` method takes the same arguments as the `comp()` method:

```
Calling /some/comp:
% $m->subexec( '/some/comp', foo => 1 );
```

The output of the subrequest goes to the same place as normal component output, but can be captured in a variable fairly easily by using `make_subrequest()` to provide explicit arguments when creating the request:

```
<%perl>
 my $output;
 my $req = $m->make_subrequest
     ( comp => '/some/comp', args => [ foo => 1 ], out_method => \$output );
 $req->exec;
 $output =~ s/something/something else/g;
</%perl>
/some/comp produced:
<% $output %>
```

As this illustrates, one of the interesting things you can do with a subrequest is override some of the parameters provided by its parent request. So we can do:

```
Calling /some/comp:
<%perl>
 my $req = $m->make_subrequest
     ( comp => '/some/comp', autoflush => 1 );
 $req->exec;
</%perl>
```

In both cases, the request object created by the `make_subrequest` method inherits its parameters from the parent request, except for those that are explicitly overridden.

A Caution About Autohandler Inheritance

Remember that Mason determines the default parent of every component but that you can always specify a different parent using the special `inherit` flag in the `<%flags>` section. This applies not just to regular components, but to autohandlers too: any component, including the top-level autohandler, can inherit from a component of your choosing.

However, specifying an autohandler's inheritance explicitly can easily lead to an infinite inheritance chain if you're not careful. Suppose you set the parent of the top-level autohandler to a component called */syshandler*. In setting up the inheritance chain at runtime, Mason will attempt to find the parent of */syshandler*, and the default is (you guessed it) `autohandler` in the same directory. So unless you've overridden */syshandler*'s parent, you'll have */autohandler* inheriting from */syshandler*, and */syshandler* inheriting from */autohandler*. Not a good situation: it will cause a fatal runtime error due to the inheritance loop. The solution is to set the `inherit` flag for */syshandler* to `undef`, terminating the inheritance chain. Yes, this sounds a bit like SCSI; perhaps someday someone will invent the equivalent of USB in the Mason world. Until then, just make sure you don't create any loops.