

2nd Edition

Tools & Techniques for Perl Developers



Advanced Perl Programming

O'REILLY®

Simon Cozens

Templating Tools

A recent thread on `comp.lang.perl.moderated` enumerated the Perl rites of passage—the perfectly good wheels that every journeyman Perl programmer reinvents. These were found to be a templating system, a database abstraction layer, an HTML parser, a processor for command-line arguments, and a time/date handling module.

See if you recognize yourself in the following story: you need to produce a form letter of some description. You’ve got a certain amount of fixed content, and a certain amount that changes. So you set up a template a little like this:

```
my $template = q{
    Dear $name,

    We have received your request for a quote for $product, and have
    calculated that it can be delivered to you by $date at a cost of
    approximately $cost.

    Thank you for your interest,

    Acme Integrated FooCorp.
};
```

Then you struggle with some disgusting regular expression along the lines of `s/(\$w+)/$1/eeg`, and eventually you get something that more or less does the job.

As with all projects, the specifications change two days after it goes live, so you suddenly need to extend your simple template to handle looping over arrays, conditionals, and eventually executing Perl code in the middle of the template itself. Before you realize what’s happened, you’ve created your own templating language.

Don’t worry if that’s you. Nearly everyone’s done it at least once. That’s why there’s a wide selection of modules on CPAN for templating text and HTML output, ranging from being only slightly more complex than `s/(\$w+)/$1/eeg` to complete independent templating languages.

Before we start looking at these modules, though, let’s consider the built-in solution—the humble Perl format.


```

use Text::Template;
my $template = Text::Template->new(TYPE => "FILE",
                                  SOURCE => "email.tpl");

my $output = $template->fill_in();

```

So, let's say we've got the following template:

```

Dear { $who },

  Thank you for the { $modulename } Perl module, which has saved me
  { $hours } hours of work this year. This would have left me free to play
  { int($hours*2.4) } games of go, which I would have greatly appreciated
  had I not spent the time goofing off on IRC instead.

Love,
Simon

```

We set up our template object and our variables, and then we process the template:

```

use Text::Template;
my $template = Text::Template->new(TYPE => "FILE",
                                  SOURCE => "email.tpl");

$who = "Mark";
$modulename = "Text::Template";
$hours = 15;
print $template->fill_in();

```

And the output would look like:

```

Dear Mark,

  Thank you for the Text::Template Perl module, which has saved me
  15 hours of work this year. This would have left me free to play
  36 games of go, which I would have greatly appreciated
  had I not spent the time goofing off on IRC instead.

Love,
Simon

```

Notice that the fill-in variables—`$who`, `$modulename`, and so on—are not my variables. When you think about it, this ought to be obvious—the my variables are not in `Text::Template`'s scope, and therefore it wouldn't be able to see them. This is a bit unpleasant: `Text::Template` has access to your package variables, and you have to do a bit more work if you want to avoid giving `use strict` a fit.

`Text::Template` has two solutions to this. The first is pretty simple—just move the fill-in variables into a completely different package:

```

use Text::Template;
my $template = Text::Template->new(TYPE => "FILE",
                                  SOURCE => "email.tpl");

$Temp::who = "Mark";
$Temp::modulename = "Text::Template";
$Temp::hours = 15;
print $template->fill_in(PACKAGE => "Temp");

```

That's slightly better, but it still doesn't please people for whom global variables are pure evil. If that's you, you can get around the problem by passing in a portable symbol table—that is, a hash:

```
use Text::Template;
my $template = Text::Template->new(TYPE => "FILE",
                                  SOURCE => "email.tpl");

print $template->fill_in(HASH => {
    who => "Mark",
    modulename => "Text::Template",
    hours => 15
});
```

Loops, Arrays, and Hashes

So much for simple templates. Because `Text::Template` evaluates the code in braces as honest-to-goodness Perl code, we can do a whole lot more with templates. Let's suppose we're invoicing for some design work:

```
$client = "Acme Motorhomes and Eugenics Ltd.";
%jobs =
    ("Designing the new logo" => 450.00,
     "Letterheads" => 300.00,
     "Web site redesign"      => 900.00,
     "Miscellaneous Expenses" => 33.75
    );
```

We can create a template to do the work for us—the invoicing work, that is, not the design work:

```
{my $total=0; ''}
To {$client}:

Thank you for consulting the services of Fungly Foobar Design
Associates. Here is our invoice in accordance with the work we have
carried out for you:

{
    while (my ($work, $price) = each %jobs) {
        $OUT .= $work . (" " x (50 - length $work)). sprintf("£%6.2f", $price)."\n";
        $total += $price;
    }
}

Total                                     {sprintf "£%6.2f",$total}

Payment terms 30 days.

Many thanks,
Fungly Foobar
```

What's going on here? First, we set up a private variable, `$total`, in the template and set it to zero. However, since we don't want a 0 appearing at the top of our template, we make sure our code snippet returns '' so it adds nothing to the output. This is a handy trick.

Next we want to loop over the jobs hash. Adding each price to the total is simple enough, but we also want to add a line to the template for each job. What we'd like to say is something like this:

```
{
  while (my ($work, $price) = each %jobs) {
    {
      $total += $price;
    }
  }
}

{$work}                                f{$price}
```

However, `Text::Template` doesn't work like that: each snippet of code must be an independent, syntactically correct piece of Perl. So how do we write multiple lines to the template? This is where the magical `$OUT` variable comes in. If you use `$OUT` in your template, that's taken as the output from the code snippet. We can append to this variable each time we go through the loop, and it'll all be filled into the template at the end.

Security and Error Checking

One of the advantages of templating is that you can delegate the non-programming bits of your application—design of HTML pages, wording of form letters, and so on—to people who aren't necessarily programmers. One of the disadvantages with powerful templating systems like `Text::Template` is that it only takes one joker to discover `{ system("rm -rf /") }` and one or both of you is out of a job. Clearly there needs to be a way to secure your templates against this sort of abuse.

`Text::Template` offers two ways to protect yourself from this kind of coworker, um, I mean abuse. The first is through Perl's ordinary tainting mechanism. In taint mode, Perl will refuse to run templates from external files. This protects you from people meddling with the template files, but only because you can't use template files at all any more; you must specify templates as strings instead.

If you can actually trust the files in the filesystem, then you'll need to tell `Text::Template` to untaint the file data; this is done with the `UNTAINT` option:

```
my $template = new Text::Template (TYPE => "FILE",
                                  UNTAINT => 1,
                                  SOURCE => $filename);
```

Now you will be able to use the template in `$filename`, if `$filename` itself has passed taint checks.

The second mechanism is much more fine-grained; the `SAFE` option allows you to specify a Safe compartment in which to run the code snippets:

```
my $compartment = new Safe; # Default set of operations is pretty safe
$text = $template->fill_in(SAFE => $compartment);
```

If you're really concerned about security, you'll want to do more tweaking than just using the default set of restricted operations.

What if things go wrong in other ways? You don't want your application to die if the code snippets contain invalid Perl, or throw a divide-by-zero error. While `Text::Template` traps eval errors by default, you may find yourself wanting more control of error handling. This is where the `BROKEN` option comes in.

The `BROKEN` option allows you to supply a subroutine reference to execute when a code snippet produces a syntax error or fails in any other way. Without `BROKEN`, you get a default error message inserted into your output:

```
Dear Program fragment delivered error ``syntax error at template line 1'',
```

By specifying a `BROKEN` subroutine, you get more control over what is inserted into the output. In many cases, the only sensible thing to do if your template is broken would be to abort processing of the template altogether. You can do this by returning `undef` from your `BROKEN` routine, and `Text::Template` will return as much output as it was able to build up.

Of course, you now need to be able to tell whether the template completed successfully or whether it was aborted by a `BROKEN` routine. The way to do this is to use the callback argument `BROKEN_ARG`. If you pass a `BROKEN_ARG` to your template constructor, it will be passed into your `BROKEN` callback.* This allows us to do something like this:

```
my $succeeded = 1;

$template->fill_in(BROKEN => \&broken_sub, BROKEN_ARG => \$succeeded);

if (!$succeeded) {
    die "Template failed to fill in...";
}

sub broken_sub {
    my %params = @_;
    ${$params{arg}} = 0;
    undef;
}
```

* Allowing a user-defined argument is a great way to make a callback extremely extensible.

As you can see, the callback is called with a hash; the argument specified by `BROKEN_ARG` is the `arg` element of the hash. In this case, that's a reference to the `$succeeded` flag; we dereference the reference and set the flag to zero, indicating an error, before returning `undef` to abort processing.

In case you feel you can make use of the broken template, `Text::Template` supplies the code snippet as the `text` element of the hash; I haven't been able to think of anything sensible to do with this yet. To assist with error reporting, the other entries in the hash are `line`, the line number in the template where the error occurred, and `error`, the value of `$@` indicating the error.

Text::Template Tricks

Using `{` and `}` to delimit code is fine for most uses of `Text::Template`—when you're generating form letters or emails, for instance. But what if you're generating text that makes heavy use of `{` and `}`—HTML pages including JavaScript, for example, or TEX code for typesetting?

One solution is to escape the braces that you don't want to be processed as Perl snippets with backslashes:

```
if (browser == "Opera") \{
    ...
\}
```

However, as one user pointed out, if you're generating TeX, which attaches meaning to backslashes and braces, you're entering a world of pain:

```
\textit{\{ {title} \} \dotfill \textbf{\{ \${cost} \}}
```

A much nicer solution would be to specify alternate delimiters, and get rid of the backslash escaping:

```
\textit{ [[[ $title ]]] } \dotfill \textbf{ [[[ $cost ]]] }
```

Much clearer!

To do this with `Text::Template`, use the `DELIMITERS` option on either the constructor or the `fill_in` method:

```
print $template->fill_in(DELIMITERS => [ '[[[', ']]]' ]);
```

This actually runs faster than the default because it doesn't do any special backslash processing, but needless to say, you have to ensure that your delimiters do not appear in the literal text of your template.

Mark suggests a different trick if this isn't appropriate: use Perl's built-in quoting operators to escape the braces. If we have a program fragment `{ q{ Hello } }`, this returns the string "Hello" and inserts it into the template output. So another way to get literal text without escaping the braces is simply to add more braces!

```

{ q{
    if (browser == "Opera") { ... }
} }

```

Another problem is that your fingers fall off from typing:

```

my $template = new Text::Template(...);
$template->fill_in();

```

all the time. The object-oriented style is perfect when you have a template that you need to fill in hundreds of times—a form letter, for instance—but not so great if you're just filling it in once. For these cases, `Text::Template` can export a subroutine, `fill_in_file`. This does the preparation and filling in all in one go:

```

use Text::Template qw(fill_in_file);

print fill_in_file("email.tmpl", PACKAGE => "Q", ...);

```

Note that you do have to import this function specifically.

HTML::Template

HTML formatting is slightly different from plaintext formatting—there are essentially two main schools of thought. The first, used by `HTML::Template`, is similar to the method we saw in `Text::Template`; the template is stored somewhere, and a Perl program grabs it and fills it in. The other school of thought is represented by `HTML::Mason`, which we'll look at next; this is inside-out—instead of running a Perl program that prints out a load of HTML, you create an HTML file that contains embedded snippets of Perl and run that.

To compare these two approaches, we're going to build the same application in `HTML::Template`, `HTML::Mason`, and `Template Toolkit`, an aggregator of RSS (Remote Site Summary) feeds to grab headlines from various web sites and push them onto a single page. (Similar to `Amphetadesk`, <http://www.disobey.com/amphetadesk/>, and O'Reilly's `Meerkat`, <http://www.oreillynet.com/meerkat/>.) RSS is an XML-based format for providing details of individual items on a site; it's generally used for providing a feed of stories from news sites.

Variables and Conditions

First, though, we'll take a brief look at how `HTML::Template` does its stuff, how to get values into it, and how to get HTML out.

As with `Text::Template`, templates are specified in separate files. `HTML::Template`'s templates are ordinary HTML files, but with a few special tags. The most important of these is `<TMPL_VAR>`, which is replaced by the contents of a Perl variable. For instance, here's a very simple page:

```

<html>
  <head><title>Product details for <TMPL_VAR NAME=PRODUCT></title></head>
  <body>
    <h1> <TMPL_VAR NAME=PRODUCT> </h1>
    <div class="desc">
      <TMPL_VAR NAME=DESCRIPTION>
    </div>
    <p class="price">Price: $<TMPL_VAR NAME=PRICE></p>
    <hr />
    <p>Price correct as at <TMP_VAR NAME=DATE></p>
  </body>
</html>

```

When filled in with the appropriate details, this should output something like:

```

<html>
  <head><title>Product details for World's Biggest Enchilada</title></head>
  <body>
    <h1> World's Biggest Enchilada </h1>
    <div class="desc">
      Recently discovered in the Mexican rain forests....
    </div>
    <p class="price">Price: $1504.39</p>
    <hr />
    <p>Price correct as at 15:18 PST, 7 Mar 2005</p>
  </body>
</html>

```

In order to fill in those values, we write a little CGI program similar to the following one:

```

use strict;
use HTML::Template;

my $template = HTML::Template->new(filename => "catalogue.tpl");

$template->param( PRODUCT    => "World's Biggest Enchilada" );
$template->param( DESCRIPTION => $description );
$template->param( PRICE      => 1504.39 );
$template->param( DATE       => format_date(localtime) );

print "Content-Type: text/html\n\n", $template->output;

```

Again, as with `Text::Template`, our driver program is very simple—load up the template, fill in the values, produce it. However, there are a few other things we can do with our templating language, and hence there are a few other tags that allow us a little more flexibility.

For instance, suppose we happen to have a picture of the world’s biggest enchilada—that would be something worth putting on our web page. However, we don’t have pictures for everything in the database; we want to output a pictures section only if we actually do have an image file kicking about. So, we could add something like this to our template:

```

<TMPL_IF NAME=PICTURE_URL>
  <div class="photo">

```

```

    
</div>
</TMPL_IF>

```

This means that if `PICTURE_URL` happens to have a true value—that is, if we’ve given it something like a real URL—then we include the photo `<DIV>`. As these `<TMPL_...>` tags are not real HTML tags, only things processed by `HTML::Template`, it’s not a problem to stick one in the middle of another HTML tag, as we have here with ``.

Of course, if we don’t have a picture, we might want to stick another one in its place, which we can do with the `<TMPL_ELSE>` pseudotag:

```

<div class="photo">
<TMPL_IF NAME=PICTURE_URL>
    
<TMPL_ELSE>
    
</TMPL_IF>
</div>

```

Notice that although our `<TMPL_IF>` must be matched by a `</TMPL_IF>`, `<TMPL_ELSE>` is not matched.

But perhaps we’re being unduly complex; all we need in this example is a default value for our `PICTURE_URL`, and we can do this directly with a `DEFAULT` attribute to `<TMPL_VALUE>`:

```

<div class="photo">
    
    "/>
</div>

```

Validation

Some people worry, quite rightly, about the effect that this sort of indiscriminate SGML abuse has on checking templates for validity. (Although, sadly many more people *don't* worry about HTML validity.) Further, those who use DTD-aware validating editors might wonder how to get these pseudotags into their documents in a nice way.

`HTML::Template` has a way around this; instead of writing the tags as though they were ordinary HTML tags, you can also write them as though they were comments, like so:

```

<!-- TMPL_IF NAME=PICTURE_URL -->
<div class="photo">
    
</div>
<!-- /TMPL_IF -->

```

Loops

If we're going to get anywhere with our RSS example, we'll need to loop over a series of items—the stories in our newsreel. Thankfully, `HTML::Template` provides the `<TMPL_LOOP>` pseudotag for treating a variable as an array. For instance, the following code:

```
<ul>
  <TMPL_LOOP NAME=STORIES>
    <li> From <TMPL_VAR NAME=FEED_NAME>: <TMPL_VAR NAME=STORY_NAME> </li>
  </TMPL_LOOP>
</ul>
```

when provided the appropriate data structure, loops over the items in the `STORIES` array reference and produces output like so:

```
<ul>

  <li> From Slashdot: NASA Finds Monkeys on Mars </li>

  <li> From use.perl: Perl 6 Release Predicted for 2013 </li>

</ul>
```

The trick is that the array reference needs to contain an array of hashes, and each hash provides the appropriate variable names:

```
$template->param(STORIES => [
  { FEED_NAME => "Slashdot", STORY_NAME => "NASA Finds Monkeys on Mars" },
  { FEED_NAME => "use.perl", STORY_NAME => "Perl 6 Release Predicted for 2013" }
]);
```

RSS Aggregation

With this knowledge, putting together our RSS aggregator is pretty trivial; first, we grab all the feeds we're interested in, then sort out their stories and put them into a data structure suitable for feeding to a `<TMPL_LOOP>`.

We'll use `LWP` and `XML::RSS` to obtain and parse the RSS feeds. In our example, we're going to pretend that we're behind a pretty impressive web cache, so we have no problems fetching the RSS feeds repeatedly; in real life, you may want to save the XML to files with fixed names and check how old the files on disk are before fetching them from the web again.

We'll start our RSS aggregator by writing a little Perl program to grab and organize the feeds:

```
#!/usr/bin/perl

use LWP::Simple;
use XML::RSS;
my @stories;
```

```

while (<DATA>) {
  chomp;
  my $xml = get($_) or next;
  my $rss = XML::RSS->new;
  eval { $rss->parse($xml) }; next if $@;
  for my $item (@{$rss->'items'}) {
    push @stories, {
      FEED_NAME => $rss->channel->'title',
      FEED_URL  => $rss->channel->'link',

      STORY_NAME => $item->'title',
      STORY_URL  => $item->'link',
      STORY_DESC => $item->'description',
      STORY_DATE => $item->'dc'->'date'
    }
  }
}

@stories = sort { $b->{STORY_DATE} cmp $a->{STORY_DATE} } @stories;

__DATA__
http://slashdot.org/slashdot.rss
http://use.perl.org/perl-news-short.rdf
http://www.theregister.co.uk/tonys/slashdot.rdf
http://blog.simon-cozens.org/blosxom.cgi/xml
http://www.oreillynet.com/~rael/index.rss

```

Next we need to design a template to receive this list of feeds. Now, I'm an abysmal HTML designer, which is why I like templates so much. I can create something rough that does the job and hand it to someone with imagination to do the presentation bits. So here's a rough-and-ready template:

```

<html>
  <head> <title> Today's News </title> </head>
  <body>
    <h1> News Stories Collected at <TMPL_VAR TIME> </h1>

    <TMPL_LOOP STORIES>
      <table border="1">
        <tr>
          <td>
            <h2>
              <a href="<TMPL_VAR STORY_URL>"> <TMPL_VAR STORY_NAME> </a>
            </h2>
            <p> <TMPL_VAR STORY_DESC> </p>
            <hr>
            <p> <i> From
              <a href="<TMPL_VAR FEED_URL>"> <TMPL_VAR FEED_NAME> </a>
            </i> </p>
          </td>
        </tr>
      </table>
    </TMPL_LOOP>
  </body>
</html>

```

(Notice that we're using short forms of the pseudotags: it's OK to say `SOME_VARIABLE` instead of `NAME=SOME_VARIABLE` where it's unambiguous.)

Finally, we put the finishing touches on our driver program, which merely takes the array we generated and feeds it to `HTML::Template`:

```
#!/usr/bin/perl

use LWP::Simple;
use XML::RSS;
use HTML::Template;

my @stories;

while (<DATA>) {
    chomp;
    my $xml = get($_) or next;
    my $rss = XML::RSS->new;
    eval { $rss->parse($xml) }; next if $@;
    for my $item (@{$rss->{'items'}}) {
        push @stories, {
            FEED_NAME => $rss->channel->{'title'},
            FEED_URL  => $rss->channel->{'link'},

            STORY_NAME => $item->{'title'},
            STORY_URL  => $item->{'link'},
            STORY_DESC => $item->{'description'},
            STORY_DATE => $item->{'dc'}->{'date'}
        }
    }
}

my $template = HTML::Template->new(filename => "aggregator.tpl");

$template->param( STORIES => [
    sort {$b->{STORY_DATE} cmp $a->{STORY_DATE}} @stories
    ] );
$template->param( TIME => scalar localtime );

delete $_->{STORY_DATE} for @stories;

print "Content-Type: text/html\n\n", $template->output;

__DATA__
http://blog.simon-cozens.org/bloxsom.cgi/xml
http://slashdot.org/slashdot.rss
http://use.perl.org/perl-news-short.rdf
http://www.theregister.co.uk/tonys/slashdot.rdf
http://www.oreillynet.com/~rael/index.rss
```

We need to delete the `STORY_DATE` once we've used it for ordering, as `HTML::Template` gets irate if we have loop variables that we don't use in our template.

Plug this into a CGI-enabled web server, and, lo and behold, we have a cheap and cheerful Amphetadesk clone.

HTML::Mason

One of the big drawbacks of `HTML::Template` is that it forces us, to some degree, to mix program logic and presentation, something that we sought to avoid by using templates. For instance, that last template got a little difficult to follow, with variable and HTML tags crowding up the template and obscuring what was actually going on. What we would prefer, then, is a system that allows us to further abstract out the individual elements of what we expect our templates to do, and this is where `HTML::Mason` comes in.

As we've mentioned, `HTML::Mason` is an inside-out templating system. As well as templating, it could also be described as a component abstraction system for building HTML web pages out of smaller, reusable pieces of logic. Here's a brief overview of how to use it, before we go on to implement the same RSS aggregator application.

Basic Components

In Mason, everything is a component. Here's a simple example of using components. Suppose we have three files: *test.html* in Example 3-1, *Header* in Example 3-2, and *Footer* in Example 3-3.

Example 3-1. test.html

```
<& /Header &>
<p>
  Hello World
</p>
<& /Footer &>
```

Example 3-2. Header

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Some Web Application</title>
    <link rel=stylesheet type="text/css" href="nt.css">
  </head>
<body>
```

Example 3-3. Footer

```
<hr>
<div class="footer">
  <address>
    <a href="mailto:webmaster@yourcompany.com">webmaster@yourcompany.com</a>
```

Example 3-3. Footer (continued)

```
</address>
</div>
</body>
</html>
```

HTML::Mason builds up the page by including the components specified inside `<& &>` tags. When creating *test.html*, Mason first includes the *Header* component found at the document root, then the rest of the HTML, then the *Footer* component.

Components may call other components. So far, we've done nothing outside the scope of server-side includes.

Basic Dynamism

So where does the templating come in? There are three basic ways of adding templates to Mason pages. Here's the first, a simple modification to our *Footer* component.

```
<hr>
<div class="footer">
  <address>
    <a href="mailto:webmaster@yourcompany.com">webmaster@yourcompany.com</a>
  </address>
  Generated: <% scalar localtime %>
</div>
</body>
</html>
```

If you wrap some Perl code in `<% ... %>` tags, the result of the Perl expression is inserted into the resulting HTML.

That's all very well for simple expressions, but what about actual Perl logic? For this, Mason has an ugly hack: a single `%` at the beginning of a line is interpreted as Perl code. This lets you do things like Example 3-4, to dump out the contents of a hash.

Example 3-4. Hashdump

```
<table>
  <tr>
    <th> key </th>
    <th>value</th>
  </tr>

  % for (keys %hash) {
    <tr>
      <td> <% $_ %> </td>
      <td> <% $hash{$_} %> </td>
    </tr>
  % }
</table>
```

Example 3-4. Hashdump (continued)

```
<%ARGS>
%hash => undef
</%ARGS>
```

There's a few things to notice in this example. First, see how we intersperse ordinary HTML with logic, using `% ...`, and evaluated Perl expressions, using `<% ... %>`. The only places `%` is special are at the start of a line and as part of the `<% ... %>` tag; the `%` of `%hash` is plain Perl.

The second thing to notice in the example is how we get the hash into the component in the first place. That's the purpose of the `<%ARGS>` section—it declares arguments to pass to the component. And how do we pass in those arguments? Here's something that might call *Hashdump*:

```
% my %foo = ( one => 1, two => 2 );

<& /Hashdump, hash => %foo &>
```

So altogether, we have an example of declaring `my` variables inside a component, passing a named parameter to another component, and having that component receive the parameter and make use of it. Mason will try to do something sensible if you pass parameters of different types than the types you've declared in the `<%ARGS>` section of the receiving component (here we passed a hash to fill in the `%hash` parameter, for instance), but life is easier if you stick to the same types.

Perl Blocks

There's a final way of adding Perl logic to your components, but it's not used much in the form we're about to describe. If you've got long Perl sections, you won't want to put a `%` at the beginning of every line. Instead, you can wrap the whole thing up in a `<%PERL>...</%PERL>` block.

However, something you will see quite often in real-life components is the `<%INIT>...</%INIT>` block. This can be placed anywhere in the component, although typically it's placed at the end to keep it away from all the HTML. No matter where it's placed, it always runs first, before anything else in the component. It's a good place to declare and initialize any variables you're going to use (by the way—Mason forces use `strict...`) and do any heavy computation that needs to happen before you do the displaying.

Another vaguely useful thing to know about is the `<%ONCE>...</%ONCE>` block, which is executed only at startup—think of it as the Mason equivalent of a Perl `BEGIN` block.

Our RSS Aggregator

We're now in a position where we can start putting together our RSS aggregator. The example in this section is taken from some code I wrote for a portal site. It's worth

noting that I threw it together in a matter of around two or three hours. The intention was to support logins, personalized lists of feeds, personalized ordering, and so on. Although I didn't get that far, what I had after those two or three hours is worth looking at.*

Let's start by thinking of what we want on the front page. I opted for a two-column design, shown in Figure 3-1, with the left column containing an invitation to log in to the portal and a list of the feeds available. As an additional flourish, the list of feeds are categorized into folders, represented by directories in the filesystem. The right column contains the logged-in user's favorite feeds, the feeds from a given folder if a folder has been clicked, or a default set of feeds in all other cases.



Figure 3-1. The RSS aggregator

Let's begin to build the site. First, we'll want a header and a footer to take away most of the boring parts of the HTML generation, as in Examples 3-5 and 3-6.

Example 3-5. Header

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html lang="en">

<head>
<title> My Portal </title>
<link rel="stylesheet" type="text/css" href="/stylesheets/portal.css">
</head>
<body class="pagetable">
```

* Feel free, of course, to implement all these things as an exercise in HTML::Mason programming.

Example 3-5. Header (continued)

```

<h1>My Portal</h1>
```

Example 3-6. Footer

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

Now we're going to use a slight Mason trick: instead of wrapping every page in the header and footer manually, we use an *autohandler*, a component that is applied to all pages, as in Example 3-7.

Example 3-7. Autohandler

```
<& /header &>
<% $m->call_next %>
<& /footer &>
```

Behind the scenes, Mason pages are processed by one or more *handlers*, reminiscent of Apache `mod_perl` handlers. Indeed, `$m` in our code is the Mason request object, which is similar to the Apache request object.*

In the lineup of Mason handlers, first come the autohandlers, which handle every request; then come dhandlers, which handle particular URIs; and finally comes the ordinary Mason handler for the page you're trying to process. Our example shows the simplest but most common autohandler: call a header component, then pass this request on to the next handler in the Mason handler chain, and finally call a footer component. This ensures that every page has its header and footer.

Next, we'll think about what the index has to be. As we've said, we're going for a two-column design, something like Example 3-8.

Example 3-8. index.html

```
<table>

<tr>
<td valign="top">
<& /LoginBox &>
<& /Directories &>

<%INIT>
$open = ($open =~ /(\\w+)/) ? $1 : '';
</%INIT>

</td>
<td width=4>&nbsp;&nbsp;&nbsp;&nbsp;</td>
```

* If you need the actual Apache request object in Mason, it's available as `$r`.

Example 3-8. *index.html* (continued)

```
<td width='100%'>

%# Am I logged in ?
% if (0) {
<& /LoggedInPane &>
%} elseif ($open) {
<& /DirectoryPane, open => $open &>
%} else {
<& /StandardPane &>
%}
</td>

</table>

<%ARGS>
$open => undef
</%ARGS>
```

As promised, the column on the left contains a login box and the directory of feeds. The right-hand side has three states: one pane for those who are logged in (which is ifdef'ed out since user control is left for future expansion), one if a particular directory has been opened, and one if the user has just come to the site's front page.*

What about the value of `$open`? Mason allows components to take arguments, either via CGI or by being passed in from other components. In this case, *index.html* is a top-level component and will receive its arguments via CGI—that is, if we request the URL <http://www.oursite.com/rss/index.html?open=News>, then `$open` will be set to `News`. The directory pane component receives its arguments from *index.html*, and so we pass it the value of `$open` we received.

Because `$open` later names a directory on the web server, we sanitize its value to avoid directory-perusal attacks such as passing in a query of `open=../../../../`. We do this in the `<%INIT%>` phase by replacing the parameter passed in with the first word in the string. If the parameter has no word characters, we set it to an empty string so the remainder of the code acts as if no directory was selected.

Now, our site is going to be made up of a load of boxes of various titles and different colors, so let's have a couple of helper components to draw boxes for us. We're going to allow the box to have a user-defined color, title, and optional title link. Experience has shown that the best way to do this is to create components for the start of the box and the end of the box. The start of the box, shown in Example 3-9, creates a table inside a table.

* Therefore, as it happens, all requests will go through *index.html*, and we could have put our header and footer code in there, but using an autohandler is cleaner and actually more conventional.

Example 3-9. BoxTop

```
<table bgcolor="#777777" cellspacing=0 border=0 cellpadding=0>
<tr><td rowspan=2></td>
<td valign=middle align=left bgcolor="<%=color%">
&nbsp;
<font size=-1 color="#ffffff">
<b>
<% $title_href && "<a href=\"$title_href\">"|n %>
<%=title |n %>
<% $title_href && "</a>" |n %>
</b></font></td></tr>
<td rowspan=2>&nbsp;</td></tr>
<tr><td colspan=2 bgcolor="#eeeeee" valign=top align=left width=100%>
<table cellpadding=2 width=100%><tr><td>

<%ARGS>
$title_href => undef
$title => undef

$color => "#000099"
</ARGS>
```

One thing to notice from this is the `|n` directive that appears at the end of some of the interpolated Perl sections. The reason for these is to turn off Mason's default HTML entity escaping code. For instance, if we had passed in a value for `$title_href`, then this line:

```
<% $title_href && "</a>" %>
```

would want to output ``. However, as Mason tries to escape HTML entities for you, this would become ``—so we need to turn that off.

The box ending code, shown in Example 3-10, is much simpler and merely ends the two tables we opened.

Example 3-10. BoxEnd

```
</td></tr></table>

</td></tr>
<tr><td colspan=4>&nbsp;</td></tr>
</table>
```

As an example of these box drawing components, let's first dispatch the dummy login box for completeness, as in Example 3-11.

Example 3-11. LoginBox

```
<& BoxTop, title="Login" &>
<small>Log in to Your Portal:</small><br/>
<form>
<ul>
<li> Barcode: <input name="barcode">
```

Example 3-11. LoginBox (continued)

```
<li> Password: <input name="password">
</ul>
</form>
<& /BoxEnd &>
```

When Mason processes that component, it produces HTML that looks like this:

```
<table bgcolor="#777777" cellspacing=0 border=0 cellpadding=0>
<tr><td rowspan=2></td>
<td valign=middle align=left bgcolor="#000099">
  &nbsp;
  <font size=-1 color="#ffffff">
  <b> Login </b></font></td>
<td rowspan=2>&nbsp;</td></tr>
<tr><td colspan=2 bgcolor="#eeeeee" valign=top align=left width=100%>
<table cellpadding=2 width=100%><tr><td>
<small>Log in to Your Portal:</small><br/>
<form>
<ul>
<li> Barcode: <input name="barcode">
<li> Password: <input name="password">
</ul>
</form>
</td></tr>
<tr><td colspan=4>&nbsp;</td></tr>
</table>
```

Now we need to make some decisions about our site's layout. As we've mentioned, we're going to put our feeds in the filesystem, categorized by directory. We'll actually have each individual feed be a Mason component, drawing on a library component we'll call *RSSBox*. Our *Directories* component is a box containing a list of categories; clicking on a category displays all the feeds in that category. As each category is a directory, we can create the list, as in Example 3-12.

Example 3-12. Directories

```
<& /BoxTop, title=> "Resources" &>
<ul>
<%%$Portal::dirs%>
</ul>
<& /BoxEnd &>
<%ONCE>
  my $root = "/var/portal/";
  for my $top (grep { -d $_ } glob("$root*")) {
```

Example 3-12. Directories (continued)

```
$top =~ s/$root//;
$Portal::dirs .= qq{
    <li><a href="/?open=$top">$top</a>
} unless $top =~ /\W/;
}
</%ONCE>
```

What's happening here is that when the server starts up, it looks at all the subdirectories of our portal directory and strips them of their root (in this instance, */var/portal/*) to turn them into a link for the purposes of our application. For instance, a directory called */var/portal/News* would turn into a link */?open=News* with the heading *News*. This link redirects back to our home page, where the open parameter causes the *DirectoryPane* to be presented and opens the feeds in the selected directory. The code skips any directories with non-word characters in the name, so it only generates links that will pass the parameter check on open.

Let's think about how that pane is implemented. We know that we open a directory and find it full of Mason component files. We want to then dynamically include each of those component files in turn, to build up our directory of feeds.

The trick to dynamically calling a component is the `comp` method on the Mason request object `$m`; this is the Perl-side version of the `<& comp &` component include tag. Hence, our directory pane ends up looking like Example 3-13.

Example 3-13. *DirectoryPane*

```
<%ARGS>
$open
</%ARGS>

% for (grep {-f $_} glob( "/var/portal/$open/*" ) ) {
% s|var/portal/||;

<% $m->comp($_) %>
% }
```

We first receive the name of the directory we're trying to open. Next we look at each file in that directory, strip off the name of the root directory (ideally this would all be provided by a configuration file), and then call the component with that name. This means that if we have a directory called *Technology* containing the following files:

```
01-Register
02-Slashdot
03-MacNews
04-LinuxToday
05-PerlDotCom
```

then calling `<& /DirectoryPane, open => "Technology" &` would have the effect of saying:

```

<& /Technology/01-Register &>
<& /Technology/02-Slashdot &>
<& /Technology/03-MacNews &>
<& /Technology/04-LinuxToday &>
<& /Technology/05-PerlDotCom &>

```

The standard pane, shown in Example 3-14, appears when no directory is open. It consists of whatever feeds we choose to make default.

Example 3-14. StandardPane

```

<& /BoxTop, title=> "Hello!", color => "dd2222"&>
Welcome to your portal! From here you can subscribe to a wide range of
news and alerting services; if you log in, you can customize this home
page.
<& /BoxEnd &>

```

```

<& /Weather/01-Oxford &>
<& /Technology/02-Slashdot &>
<& /News/01-BBC &>
<& /People/03-Rael &>
...

```

So what’s in the individual files? As we’ve mentioned, they make use of an *RSSBox* component, and they simply pass in the URL for the feed and optionally a color, a maximum number of items, and a name for the feed. They also pass in a parameter to say whether we want to display just the titles and links for each RSS item, or the description as well. For instance, */News/01-BBC* looks like this:

```

<& /RSSBox, URL =>"http://www.newsisfree.com/HPE/xml/feeds/60/60.xml",
Color =>"#dd0000" &>

```

whereas Rael Dornfest’s blog looks like this:

```

<& /RSSBox, URL => "http://www.oreillynet.com/~rael/index.rss",
Color=> "#cccc00", Title => "Rael Dornfest", Full => 0 &>

```

As we’ll see in a moment, the beauty of this modular system is that we can have components that do things other than fire off RSS feeds if we want.

But first, let’s complete our portal by writing the *RSSBox* library that all these sources use. First, we want a *ONCE* block to load up the modules we need:

```

<%ONCE>
use XML::RSS;
use LWP::Simple;
</%ONCE>

```

Next we take our arguments, setting appropriate defaults:

```

<%ARGS>
$URL
$Color => "#0000aa"
$Max => 5
$Full => 1

```

```
$Title => undef
</%ARGS>
```

Before we start outputting any content, we load up the feed in question and parse it with the `XML::RSS` module. We call Mason's `cache_self` method to have this component handle caching its output; if the same URL is accessed within 10 minutes, the cached copy will be presented instead:

```
<%INIT>
return if $m->cache_self(key => $URL, expires_in => '10 minutes');
my $rss = new XML::RSS;
eval { $rss->parse(get($URL));};
my $title = $Title || $rss->channel('title');
</%INIT>
```

And now we are ready to go. So let's look at this altogether in Example 3-15.

Example 3-15. RSSBox

```
<%ONCE>
use XML::RSS;
use LWP::Simple;
</%ONCE>

<%ARGS>
$URL
$Color => "#0000aa"
$Max => 5
$Full => 1
$Title => undef
</%ARGS>

<%INIT>
my $rss = new XML::RSS;
eval { $rss->parse(get($URL));};
my $title = $Title || $rss->channel('title');
my $site = $rss->channel('link');
</%INIT>

<BR>
<& BoxTop, color => $Color, title => $title, title_href => $site &>

    <dl class="rss">
% my $count = 0;
% for (@{$rss->{items}}) {
    <dt class="rss">
        <a href="<% $_->{link} %>"> <% $_->{title} %> </a>
    </dt>
% if ($Full) {
        <dd> <% $_->{description} %> </dd>
% }

% last if ++$count >= $Max;
% }
```

Example 3-15. *RSSBox* (continued)

```
</dl>
<& /BoxEnd &>
```

There isn't much to it; for each item in the feed, we want to provide a link, the item's title, and, optionally, the description. We stop if we have more items than we want.

This demonstrates how powerful Mason can be; as I said, the total development time for this site was a couple of hours at most. The entire site takes considerably fewer than 200 lines of code. And, as we mentioned, we have the flexibility to include components that are not RSS. For instance, we don't actually have an RSS feed of the Oxford weather. However, there is a web page that spits out a weather report in a well-known format. This means that *Weather/01-Oxford* does not call *RSSBox* at all, but is in fact the following:

```
<%INIT>
use LWP::Simple;
my @lines = grep /Temperature|Pressure|humidity|^Sun|Rain/,
              split /\n/,
              get('http://www-atm.physics.ox.ac.uk/user/cfinlay/now.htm');
</%INIT>

<br>
<& /BoxTop, title => "Oxford Weather", color => "#dd00dd" &>

<ul>
% for (@lines) {
  <li> <% $_ %> </li>
% }
</ul>

<& /BoxEnd &>
```

And that sums up Mason—simple, extensible, and highly powerful.



Of course, there are many other Mason tricks for you to learn—too many to cover here. Dave Rolsky and Ken Williams's fantastic book *Embedding Perl in HTML with Mason* (<http://www.masonbook.com/>) covers many of them, including more details about getting Mason up and running in your web server. Also check out the Mason home page (<http://www.masonhq.com>).

Template Toolkit

While the solutions we've seen so far have been primarily for Perl programmers—embedding Perl code in some other medium—Andy Wardley's Template Toolkit (<http://www.template-toolkit.org/>) is slightly different. It uses its own templating language to express components, loops, method calls, data structure elements, and more; it's therefore useful for teaching to designers who have no knowledge of the

Perl side of your application* but who need to work on the presentation. As the documentation puts it, you should think of the Template Toolkit language as a set of layout directives for displaying data, not calculating it.

Like Mason, it seamlessly handles compiling, caching, and delivering your templates. However, unlike Mason, it's designed to provide general-purpose display and formatting capabilities in a very extensible way. As an example, you can use Template Toolkit to dynamically serve up PDF documents containing graphs based on data from a database—and all this using nothing other than the standard plugins and filters and all within the Template Toolkit mini language.

But before we look at the clever stuff, let's look at the very simple uses of Template Toolkit. In the simplest cases, it behaves a lot like `Text::Template`. We take a template object, feed it some values, and give it a template to process:

```
use Template;
my $template = Template->new();
my $variables = {
    who      => "Andy Wardley",
    modulename => "Template Toolkit",
    hours    => 30,
    games    => int(30*2.4)
};
$template->process("thankyou.txt", $variables);
```

This time, our template looks like the following:

```
Dear [% who %],
    Thank you for the [% modulename %] Perl module, which has saved me
[% hours %] hours of work this year. This would have left me free to play
[% games %] games of go, which I would have greatly appreciated
had I not spent the time goofing off on IRC instead.

Love,
Simon
```

Lo and behold, the templated text appears on standard output. Notice, however, that our variables inside the [% and %] delimiters aren't Perl variables with the usual type sign in front of them; instead, they're now Template Toolkit variables. Template Toolkit variables can be more than just simple scalars, though; complex data structures and even Perl objects are available to Template Toolkit through a simple, consistent syntax. Let's go back to our design work invoices, but with a slightly different data structure:

```
my $invoice = {
    client => "Acme Motorhomes and Eugenics Ltd.",
    jobs => [
        { cost => 450.00, description => "Designing the new logo" },
        { cost => 300.00, description => "Letterheads and complements slips" },
    ],
};
```

* And probably no desire to find out!

```

    { cost => 900.00, description => "Web site redesign" },
    { cost => 33.75, description => "Miscellaneous Expenses" }
  ],
  total => 0
};

$invoice->{total} += $_->{cost} for @{$invoice->{jobs}};

```

How would we design a template to fit that data? Obviously, we're going to need to loop over the jobs in the anonymous array and extract various hash values. Here's how it's done:

```

To [% client %]:

Thank you for consulting the services of Fungly Foobar Design
Associates. Here is our invoice in accordance with the work we have
carried out for you:

[% FOREACH job = jobs %]
    [% job.description %] : [% job.cost %]
[% END %]

Total                                     $[% total %]

Payment terms 30 days.

Many thanks,
Fungly Foobar

```

As you can see, the syntax is inspired by Perl—we can `foreach` over a list and use a local variable `job` to represent each element of the iterator. The dot operator is equivalent to Perl's `->`—it dereferences array and hash reference elements and can also call methods on objects.

However, there's something slightly wrong with this example; since we can expect our descriptions to be of variable width, our costs aren't going to line up nicely at the end.* What can we do about this? This is where a nice, extensible feature of the Template Toolkit called *filters* comes in.

Filters

Template Toolkit filters are a little like Unix filters—they're little routines that take an input, transform it, and spit it back out again. And just like Unix filters, they're connected to our template output with a pipe symbol (`|`).

In this case, the filter we want is the oddly named `format` filter, which performs `printf`-like formatting on its input:

```

[% job.description | format("%60s") %] : [% job.cost %]

```

* We completely glossed over this in the `Text::Template` example; did you notice?

This fixes the case where the data is being produced by our template processor—`job.description` is turned into a real description, and then filtered. But we can also filter whole blocks of template content. For example, if we wanted to format the output as HTML, we could apply the `html_entity` filter to replace entities with their HTML encoding:

```
[% FILTER html_entity %]
Payment terms: < 30 days.
[% END %]
```

This turns into: `Payment terms: < 30 days.`

This is another example of a Template Toolkit block; we've seen `FOREACH` blocks and `FILTER` blocks. There's also the `IF/ELSIF/ELSE` block:

```
[% IF delinquent %]
    Our records indicate that this is the second issuing of this
    invoice. Please pay IMMEDIATELY.
[% ELSE %]
    Payment terms: <30 days.
[% END %]
```

Other interesting filters include the `upper`, `lower`, `ucfirst`, and `lcfirst` filters to change the casing of the text; `uri` to URI-escape any special characters; `eval` to treat the text to another level of template processing, and `perl_eval` to treat the output as Perl, eval it, and then add the output to the template. For a more complete list of filters with examples, see the `Template::Manual::Filters` documentation.

Plugins

While filters are an interface to simple Perl functionality—built-in functions like `eval`, `uc`, and `sprintf`, or simple text substitutions—plugins are used to interface to more complex functions. Typically, they're used to expose the functionality of a Perl module to the format language.

For instance, the `Template::Plugin::Autoformat` plugin allows one to use `Text::Autoformat`'s autoformatting functionality. Just as with the Perl module, use the `USE` directive to tell the format processor to load the plugin. This then exports the `autoformat` subroutine and a corresponding `autoformat` filter:

```
[% USE autoformat(right=78) %]
[% address | autoformat %]
```

This assures that the address is printed in a nice block on the right-hand side of the page.

A particularly neat plugin is the `Template::Plugin::XML::Simple` module, which allows you to parse an XML data file using `XML::Simple` and manipulate the resulting data structure from inside a template. Here we use `USE` to return a value:

```
[% USE document = XML.Simple("app2ed.xml") %]
```

And now we have a data structure created from the structure and text of an XML document. We can explore this data structure by entering the elements, just as we did in “XML Parsing” in Chapter 2:

```
The author of this book is
[% document.bookinfo.authorgroup.author.firstname # 'Simon' %]
[% document.bookinfo.authorgroup.author.surname # 'Cozens' %]
```

Actually writing a plugin module like this is surprisingly easy—and, in fact, something we’re going to need to do for our RSS example. First, we create a new module called `Template::Plugin::Whatever`, where `Whatever` is what we want our plugin to be known as inside the template language. This module will load up whatever module we want to interface to. We’ll also need it to inherit from `Template::Plugin`. Let’s go ahead and write an interface to Tony Bowden’s `Data::BT::PhoneBill`, a module for querying UK telephone bills.

```
package Template::Plugin::PhoneBill;
use base 'Template::Plugin';
use Data::BT::PhoneBill;
```

Now we want to receive a filename when the plugin is USED and turn that into the appropriate object. Therefore we write a new method to do just that:

```
sub new {
    my ($class, $context, $filename) = @_;
    return Data::BT::PhoneBill->new($filename);
}
```

`$context` is an object passed by Template Toolkit to represent the context we’re being evaluated in. And that’s basically it—you can add error checking to make sure the filename exists and that the module can parse the phone bill properly, but the guts of a plugin are as we’ve shown.

Now that we’ve created the plugin, we can access the phone bill just like we did with the `XML::Simple` data structure:

```
[% USE bill = PhoneBill("mybill.txt") %]

[% WHILE call = bill.next_call %]
Call made on [% call.date %] to [% call.number %]...
[% END %]
```

An interesting thing to notice is that when we were using the `XML::Simple` plugin, we accessed elements of the data structure with the dot operator: `document.bookinfo` and so on. In that case, we were navigating hash references; the Perl code would have looked like `$document->{bookinfo}->{authorgroup}->{author}...`. In this example, we’re using precisely the same dot operator syntax, but, instead of navigating hash references, we’re calling methods: `call.date` would translate to `$call->date`. However, it all looks the same to the template writer. This abstraction of the underlying data structure is one of the big strengths of Template Toolkit.

Components and Macros

When we looked at `HTML::Mason`, one of the things we praised was the ability to split template functionality up into multiple components, then include those components with particular parameters. It shouldn't be a surprise that we can do precisely the same in Template Toolkit.

The mechanism through which we pull in components is the `INCLUDE` directive. For instance, we can specify our box drawing library in a way very similar to the `HTML::Mason` method, as in Example 3-16.

Example 3-16. BoxTop

```
<table bgcolor="#777777" cellspacing=0 border=0 cellpadding=0>
<tr>
  <td rowspan=2></td>
  <td valign=middle align=left bgcolor="[% color %]">
    &nbsp;
    <font size=-1 color="#ffffff">
      <b>
        [% IF title_href %]
          <a href="[% title_href %]"> [% title %] </a>
        [% ELSE %]
          [% title %]
        [% END %]
      </b>
    </font>
  </td>
  <td rowspan=2>&nbsp;</td>
</tr>
<tr>
  <td colspan=2 bgcolor="#e0e0e0" valign=top align=left width=100%>
    <table cellpadding=2 width=100%>
      <tr><td>
```

And in the same way as `HTML::Mason`, we can use local parameters when we include these components:

```
[% INCLUDE boxtop
      title = "Login"
      ...
    %]
```

However, Template Toolkit provides another method of abstracting out common components, the `MACRO` directive. We can define a `MACRO` to expand to any Template Toolkit code; let's start by defining it to simply `INCLUDE` the drawing component:

```
[% MACRO boxtop INCLUDE boxtop %]
[% MACRO boxend INCLUDE boxend %]
```

With this, we can draw boxes with a little less syntax:

```
[% boxtop(title="My Box") %]
```

```
<P> Hello, people! </P>
[% boxend %]
```

Instead of using a component file and `INCLUDE`, we can also associate a block of Template Toolkit directives with a macro name.

```
[% MACRO boxtop BLOCK %]
<table bgcolor="#777777" cellspacing=0 border=0 cellpadding=0>
<tr>
...
[% END %]

[% MACRO boxend BLOCK %]
</td></tr></table>

</td></tr>
<tr><td colspan=4>&nbsp;</td></tr>
</table>
[% END %]
```

Eventually, we can build up a library of useful macros and then `INCLUDE` that, instead of having a bunch of component files hanging around.

Let's assume we've created such a library and it contains these two box-drawing macros, and now we'll move on to putting together our RSS aggregator.

The RSS Aggregator

When it comes to writing the aggregator, we first look at the list of Template Toolkit plugins and notice with some delight that there's already a `Template::Plugin::XML::RSS`, which talks to `XML::RSS`. Unfortunately, our delight is short-lived, as we soon discover that this expects to get a filename rather than a URL or a string of XML data. We don't really want to be writing out files and then parsing them in again.

So let's create our own subclass of `Template::Plugin::XML::RSS` that fetches URLs and parses those instead:

```
package Template::Plugin::XML::RSS::URL;
use base 'Template::Plugin::XML::RSS';
use LWP::Simple;

sub new {
    my ($class, $context, $url) = @_;

    return $class->fail('No URL specified') unless $url;

    my $url_data = get($url)
        or return $class->fail("Couldn't fetch $url");

    my $rss = XML::RSS->new
        or return $class->fail('failed to create XML::RSS');

    eval { $rss->parse($url_data) } and not $@
```

```

    or return $class->fail("failed to parse $url: @$");

    return $rss;
}

1;

```

Now we can build up the equivalent of the `RSSBox` component we made in Mason:

```

[% MACRO RSSBox(url) USE rss = XML.RSS.URL(url) %]
[% box_top(title = rss.channel.title, title_href = rss.channel.link) %]

<dl class="rss">
  [% FOREACH item = news.items %]
    <dt class="rss">
      <a href="[% item.link %]"> [% item.title %] </a>
      [% IF full %]
        <dd> [% item.description %] </dd>
      [% END %]
    </dt>
  [% END %]
</dl>
[% box_end %]
[% END %]

```

The important difference between this and the Mason example is that this piece of code handles everything itself—the whole process of obtaining and parsing the RSS feed is available to the template designer. There’s no Perl code here to be seen at all. It’s also considerably more concise and easier to read and understand. Now that we have this macro, we can produce an HTML box full of RSS stories with a simple call to it:

```

[% RSSBox("http://slashdot.org/slashdot.rss") %]

```

From here on, constructing an RSS aggregator is a simple matter of templating; all of the Perl work has been abstracted away.

AxKit

Although we include it in our list of templating systems, AxKit (<http://www.axkit.org>) is a slightly different kettle of fish from the modules we’ve seen so far; this is no mere templating system, it’s a fully fledged XML application server for Apache. The most common use of AxKit is to transform XML to HTML on-the-fly for delivery over the web.

However, thanks to XSP (Extensible Server Pages), developed by the Apache Cocoon project, AxKit can be used as an extraordinarily extensible templating system. The basic idea behind XSP is that certain XML tags trigger the execution of given Perl routines. At a very basic level, you can use tags to delimit raw Perl code:

```

<p>
Good
<xsp:logic>
if ((localtime)[2] >= 12) {

```

```

    <i>Afternoon</i>
  }
  else {
    <i>Morning</i>
  }
</xsp:logic>
</p>

```

Notice that AxKit is quite happy for you to intersperse XML marked-up data with your Perl code. Because AxKit parses the XML, it knows that `<i>Afternoon</i>` is data, not Perl code, and treats it appropriately. This also means that if you have an XML guru handy, he can find a way of validating your HTML-with-embedded-XSP. In fact, since AxKit parses everything as XML, your HTML must be well-formed and valid or you won't get anything out of AxKit at all.

However, AxKit does not stop at this basic level; XSP allows you to create tag libraries with frontend Perl code. For instance, the `AxKit::XSP::ESQL` taglib provides a wrapper around the DBI libraries. These tag libraries define their own XML namespaces and place tags inside them. So your XML would use a namespace declaration to import the tag library:

```

<xsp:page
  language="perl"
  xmlns:xsp="http://apache.org/xsp/core/v1"
  xmlns:esql="http://apache.org/xsp/SQL/v2"
>

```

and this would allow you to use `<esql: . . . >` tags in your page:

```

<esql:connection>
<esql:driver>Pg</esql:driver>
<esql:dburl>dbname=rss</esql:dburl>
<esql:username>www</esql:username>
<esql:password></esql:password>
<esql:execute-query>
  <esql:query>
    select description, url, title from feeds
  </esql:query>
<esql:results>
  <ul>
    <esql:row-results>
      <li>
        <a>
          <xsp:attribute name="href">
            <esql:get-string column="url"/>
          </xsp:attribute>
          <esql:get-string column="name"/>
        </a> - <esql-get-string column="description"/>
      </li>
    </esql:row-results>
  </ul>
</esql:results>

```

```
<esql:no-results> <p> Couldn't get any results! </p> </esql:no-results>
</esql:execute-query>
</esql:connection>
```

This executes the SQL query near the top of the XML and turns it into an HTML list. The only potentially non-obvious part is where we use `<xsp:attribute>`. The key to understanding this is that a document processed by AxKit has to be 100% valid, well-formed XML. On the other hand, with `HTML::Template` and `HTML::Mason` we could get away with things like `<a href="<TMPL_VAR URL>">` or `<a href="<% $url |n%>">`—in a sense, putting tags inside tags.

But with AxKit, the whole document is parsed as XML, and then transformations are applied. With the above examples, AxKit would parse the tag as having the perfectly valid (but nonsensical) attribute values `<TMPL_VAR URL>` and `<% $url|n>` and do no more processing on them. Worse still, we can't get away with anything like `<a href=<esql:get-string column="url"/>>` as that's not even well-formed XML.

So we play a slight trick. We ask the XSP layer to rewrite the `<a>` tag, after everything has been parsed, with the appropriate `href` attribute. This keeps everything well-formed and parsable.

There are many other tag libraries that perform the same function as Template Toolkit's plugins and give the XML author access to high-level Perl functionality; my own `AxKit::XSP::ObjectTaglib` allows the programmer to easily wrap any object-oriented module into a tag library.

We're not going to implement our RSS aggregator in AxKit, as it turns out, because AxKit is a fully featured XML processor. All of the heavy lifting can be done in XSLT stylesheets, and there's almost no Perl content involved.

Instead, for more on AxKit, we'll refer you to *Perl and XML* (O'Reilly) and <http://www.axkit.org>, the AxKit home page.

Conclusion

In this chapter, we've looked at a few of the available templating tools that are commonly used in Perl; from simple formats—`sprintf`, and the like—on through `Text::Template` and `HTML::Template`, and then up to the more sophisticated solutions of `HTML::Mason` and Template Toolkit.

But we've missed out on one quite important question: which one should you use? As usual, the answer depends partly on what you need and partly on your tastes.

First, consider the distinction between Perl-based systems like `Text::Template` and `Text::Autoformat`, and inside-out modules like `HTML::Mason`. If the main purpose of your program is to provide some templated output, as in the case of a web-based application, then you probably want to gravitate toward the `HTML::Mason` and Template Toolkit end of the spectrum.

You also need to consider who's going to be writing the templates and whether you want to expose them to Perl code. Template Toolkit, AxKit, and HTML::Template all tend to keep the templater away from Perl, whereas HTML::Mason forces the templater to get down and dirty with it.

Second, there's the element of personal taste. I'm not a great fan of HTML::Template, preferring the way Mason does things; I find AxKit very powerful but at times very frustrating because of its insistence on clean XML; and I'm beginning to like Template Toolkit the more I use it, but prefer Mason basically because I'm more used to it.

Your tastes may differ. It's just as well, that as with so many things in Perl, there's more than one way to do it.