# 1 Choosing a Templating System

## 1.1 Description

Everything you wanted to know about templating systems and didn't dare to ask. Well, not everything....

## 1.2 Introduction

Go on, admit it: you've written a templating system. It's okay, nearly everyone has at some point. You start out with something beautifully simple like `$HTML =~ s/\$(\w+)/${$1}/g` and end up adding conditionals and loops and includes until you've created your very own unmaintainable monster.

Luckily for you, you are not the first to think it might be nice to get the HTML out of your code. Many have come before, and more than a few have put their contributions up on CPAN. At this time, there are so many templating modules on CPAN that it's almost certain you can find one that meets your needs. This document aims to be your guide to those modules, leading you down the path to the templating system of your dreams.

And, if you just went straight to CPAN in the first place and never bothered to write your own, congratulations: you're one step ahead of the rest of us.

### *1.2.1 On A Personal Note*

Nothing can start an argument faster on the mod_perl mailing list than a claim that one approach to templating is better than another. People get very attached to the tools they've chosen. Therefore, let me say up front that I am biased. I've been at this for a while and I have opinions about what works best. I've tried to present a balanced appraisal of the features of various systems in this document, but it probably won't take you long to figure out what I like. Besides, attempts to be completely unbiased lead to useless documents that don't contain any real information. So take it all with a pound of salt and if you think I've been unfair to a particular tool through a factual error or omission, let me know.

## 1.3 Why Use Templates?

Why bother using templates at all? Print statements and CGI.pm were good enough for Grandpa, so why should you bother learning a new way to do things?

### *1.3.1 Consistency of Appearance*

It doesn't take a genius to see that making one navigation bar template and using it in all of your pages is easier to manage than hard-coding it every where. If you build your whole site like this, it's much easier to make site-wide changes in the look and feel.

### *1.3.2 Reusability*

Along the same lines, building a set of commonly used components makes it easier to create new pages.

### *1.3.3 Better Isolation from Changes*

Which one changes more often, the logic of your application or the HTML used to display it? It actually doesn’t matter which you answered, as long as it’s one of them. Templates can be a great abstraction layer between the application logic and the display logic, allowing one to be updated without touching the other.

### *1.3.4 Division of Labor*

Separating your Perl code from your HTML means that when your marketing department decides everything should be green instead of blue, you don’t have to lift a finger. Just send them to the HTML coder down the hall. It’s a beautiful thing, getting out of the HTML business.

Even if the same people in your organization write the Perl code and the HTML, you at last have the opportunity for more people to be working on the project in parallel.

## 1.4 What Are the Differences?

Before we look at the available options, let’s go through an explanation of some of the things that make them different.

### *1.4.1 Execution Models*

Although some try to be flexible about it, most templating systems expect you to use some variation of the two basic execution models, which I will refer to as "pipeline" and "callback." In the callback style, you let the template take over and it has the application’s control flow coded into it. It uses callbacks to modules or snippets of in-line Perl code to retrieve data for display or perform actions like user authentication. Some popular examples of systems using this model include Mason, Embperl, and Apache::ASP.

The pipeline style does all the work up front in a standard CGI or mod_perl handler, then decides which template to run and passes some data to it. The template has no control flow logic in it, just presentation logic, e.g. show this graphic if this item is on sale. Popular systems supporting this approach include HTML::Template and Template Toolkit.

The callback model works very well for publishing-oriented sites where the pages are essentially mix and match sets of articles and lists. Ideally, a site can be broken down into visual "components" or pieces of pages which are general enough for an HTML coder to re-combine them into entirely new kinds of pages without any help from a programmer.

The callback model can get a bit hairy when you have to code logic that can result in totally different content being returned. For example, if you have a system that processes some form input and takes the user to different pages depending on the data submitted. In these situations, it’s easy to end up coding a spaghetti of includes and redirects, or putting what are really multiple pages in the same file.

On the other hand, a callback approach can result in fewer files (if the Perl code is in the HTML file), and feels easier and more intuitive to many developers. It’s a simple step from static files to static files with a few in-line snippets of code in them. This is part of why PHP is so popular with new developers.

The pipeline model is more like a traditional model-view-controller design. Working this way can provide additional performance tuning opportunities over an approach where you don’t know what data will be needed at the beginning of the request. You can aggregate database queries, make smarter choices about caching, etc. It can also promote a cleaner separation of application logic and presentation. However, this approach takes longer to get started with since it’s a bigger conceptual hurdle and always involves at least two files: one for the Perl code and one for the template.

Keep in mind, many systems offer significant flexibility for customizing their execution models. For example, Mason users could write separate components for application logic and display, letting the logic components choose which display component to run after fetching their data. This allows it to be used in a pipeline style. A Template Toolkit application could be written to use a simple generic handler (like the Apache::Template module included in the distribution) with all the application logic placed in the template using object calls or in-line Perl. This would be using it in a callback style.

HTML::Template is fairly rigid about insisting on a pipeline approach. It doesn’t provide methods for calling back into Perl code during the HTML formatting stage; you have to do the work before running the template. The author of the module consider this a feature since it prevents developers from cheating on the separation of application code and presentation.

### *1.4.2 Languages*

Here’s the big issue with templating systems. This is the one that always cranks up the flame on web development mailing lists.

Some systems use in-line Perl statements. They may provide some extra semantics, like Embperl’s operators for specifying whether the code’s output should be displayed or Mason’s `<%init>` sections for specifying when the code gets run, but at the end of the day your templates are written in Perl.

Other systems provide a specialized mini-language instead of (or in addition to) in-line Perl. These will typically have just enough syntax to handle variable substitution, conditionals, and looping. HTML::Template and Template Toolkit are popular systems using this approach.

Here’s how a typical discussion of the merits of these approaches might go:

**IN-LINE:** Mini-languages are stupid. I already know Perl and it’s easy enough. Why would you want to use something different?

**MINI-LANG:** Because my HTML coder doesn’t know Perl, and this is easier for him.

**IN-LINE:** Maybe he should learn some Perl. He’d get paid more.

**MINI-LANG:** Whatever. You just want to use in-line Perl so you can handle change requests by putting little hacks in the template instead of changing your modules. That’s sloppy coding.

**IN-LINE:** That’s efficient coding. I can knock out data editing screens in half the time it takes you, and then I can go back through, putting all the in-line code into modules and just have the templates call them.

**MINI-LANG:** You could, but you won’t.

**IN-LINE:** Is it chilly up there in that ivory tower?

**MINI-LANG:** Go write some VBScript, weenie.

etc.

Most people pick a side in this war and stay there. If you are one of the few who hasn’t fully decided yet, you should take a moment to think about who will be building and maintaining your templates, what skills those people have, and what will allow them to work most efficiently.

Here’s an example of a simple chunk of template using first an in-line style (Apache::ASP in this case) and then a mini-language style (Template Toolkit). This code fetches an object and displays some properties of it. The data structures used are identical in both examples. First Apache::ASP:

```
<% my $product = Product->load('sku' => 'bar1234'); %>

<% if ($product->isbn) { %>
  It's a book!
<% } else { %>
  It's NOT a book!
<% } %>

<% foreach my $item (@{$product->related}) { %>
  You might also enjoy <% $item->name %>.
<% } %>
```

And now Template Toolkit:

```
[% USE product(sku=bar1234) %]

[% IF product.isbn %]
  It's a book!
[% ELSE %]
  It's NOT a book!
[% END %]

[% FOREACH item = product.related %]
  You might also enjoy [% item.name %].
[% END %]
```

There is a third approach, based on parsing an HTML document into a DOM tree and then manipulating the contents of the nodes. The only module using this approach is HTML_Tree. The idea is similar to using a mini-language, but it doesn’t require any non-standard HTML tags and it doesn’t embed any logic about loops or conditionals in the template itself. This is nice because it means your templates are valid HTML documents that can be viewed in a browser and worked with in most standard HTML tools. It also means people working with the templates can put placeholder data in them for testing and it will simply be replaced when the template is used. This preview ability only breaks down when you need an if/else type construct in the template. In that situation, both the "if" and "else" chunks of HTML would show up when previewing.

## *1.4.3 Parsers and Caching*

The parsers for these templating systems are implemented in one of three ways: they parse the template every time ("repeated parse"), they parse it and cache the resulting parse tree ("cached parse tree"), or they parse it, convert it to Perl code, and compile it ("compiled").

Systems that compile templates to Perl take advantage of Perl’s powerful runtime code evaluation capabilities. They examine the template, generate a chunk of Perl code from it, and `eval` the generated code. After that, subsequent requests for the template can be handled by running the compiled bytecode in memory. The complexity of the parsing and code generation steps varies based on the number of bells and whistles the system provides beyond straight in-line Perl statements.

Compiling to Perl and then to Perl bytecode is slow on the first hit but provides excellent performance once the template has been compiled, since the template becomes a Perl subroutine call. This is the same approach used by systems like JSP (Java ServerPages). It is most effective in environments with a long-running Perl interpreter, like mod_perl.

HTML::Template, HTML_Tree, and the 2.0 beta release of Embperl all use a cached parse tree approach. They parse templates into their respective internal data structures and then keep the parsed structure for each processed template in memory. This is similar to the compiled Perl approach in terms of performance and memory requirements, but does not actually involve Perl code generation and thus doesn’t require an `eval` step. Which way is faster, caching the parse tree or compiling? It’s hard to objectively measure, but anecdotal evidence seems to support compilation. Template Toolkit used a cached parse tree approach for version 1, but switched to a compilation approach for version 2 after tests showed it to offer a significant speed increase. However, as will be discussed later, either approach is more than fast enough.

In contrast to this, a repeated parse approach may sound very slow. However, it can be pretty fast if the tokens being parsed for are simple enough. Systems using this approach generally use very simple tokens, which allows them to use fast and simple parsers.

Why would you ever use a system with this approach if compilation has better performance? Well, in an environment without a persistent Perl interpreter like vanilla CGI this can actually be faster than a compiled approach since the startup cost is lower. The caching of Perl bytecode done by compilation systems is useless when the Perl interpreter doesn’t stick around for more than one request.

There are other reasons too. Compiled Perl code takes up a lot of memory. If you have many unique templates, they can add up fast. Imagine how much RAM it would take up if every page that used server-side includes (SSI) had to stay in memory after it had been accessed. (Don’t worry, the `Apache::SSI` module doesn’t use compilation so it doesn’t have this problem.)

## *1.4.4 Application Frameworks vs. Just Templates*

Some of the templating tools try to offer a comprehensive solution to the problems of web development. Others offer just a templating solution and assume you will fit this together with other modules to build a complete system.

Some common features offered in the frameworks include:

### 1.4.4.1 URL Mapping

All of the frameworks offer a way to map a URL to a template file. In addition to simple mappings similar to the handling of static documents, some offer ways to intercept all requests within a certain directory for pre-processing, or create an object inheritance scheme out of the directory structure of a site.

### 1.4.4.2 Session Tracking

Most interactive sites need to use some kind of session tracking to associate application state data with a user. Some tools make this very easy by handling all the cookies or URL-munging for you and letting you simply read and write from an object or hash that contains the current user’s session data. A common approach is to use the Apache::Session module for storage.

### 1.4.4.3 Output Caching

While caching of output is outside the scope of most templating systems, Mason includes it as a standard feature. In addition to page-level caching, Mason also offers fine-grained caching of output from sections within a page.

### 1.4.4.4 Form Handling

How will you live without CGI.pm to parse incoming form data? Many of these tools will do it for you, making it available in a convenient data structure. Some also validate form input, and even provide "sticky" form widgets that keep their selected values when re-displayed or set up default values based on data you provide.

### 1.4.4.5 Debugging

Everyone knows how painful it can be to debug a CGI script. Templating systems can make it worse, by screwing up Perl’s line numbers with generated code. To help fix the problem they’ve created, some offer built-in debugging support, including extra logging, or integration with the Perl debugger.

If you want to use a system that just does templates but you need some of these other features and don’t feel like implementing them yourself, there are some tools on CPAN which provide a framework you can build on. The libservlet distribution, which provides an interface similar to the Java servlet API, is independent of any particular templating system. Apache::PageKit and CGI::Application are other options in this vein, but both of these are currently tied to HTML::Template. OpenInteract is another framework, this time tied to Template Toolkit. All of these could be adapted for the "just templates" module of your choice with fairly minimal effort.

## 1.5 The Contenders

Okay, now that you know something about what separates these tools from each other, let's take a look at the top choices for Perl templating systems. This is not an exhaustive list: I've only included systems that are currently maintained, well-documented, and have managed to build up a significant user community. In short, I've left out a dozen or so less popular systems. At the end of this section, I'll mention a few systems that aren't as commonly used but may be worth a look.

### *1.5.1 SSI*

SSI is the granddaddy of templating systems, and the first one that many people used since it comes as a standard part of most web servers. With mod_perl installed, mod_include gains some additional power. Specifically, it is able to take a new `#perl` directive (though only if mod_perl is statically built) which allows for in-line subroutine calls. It can also efficiently include the output of Apache::Registry scripts by using the Apache::Include module.

The Apache::SSI module implements the functionality of mod_include entirely in Perl, including the additional `#perl` directive. The main reasons to use it are to post-process the output of another handler (with Apache::Filter) or to add your own directives. Adding directives is easy through subclassing. You might be tempted to implement a complete template processor in this way, by adding loops and other constructs, but it's probably not worth the trouble with so many other tools out there.

SSI follows the callback model and is mostly a mini-language, although you can sneak in bits of Perl code as anonymous subs in `#perl` directives. Because SSI uses a repeated parse implementation, it is safe to use it on large numbers of files without worrying about memory bloat.

SSI is a great choice for sites with fairly simple templating needs, especially ones that just want to share some standard headers and footers between pages. However, you should consider whether or not your site will eventually need to grow into something with more flexibility and power before settling on this simple approach.

### *1.5.2 HTML::Mason*

Mason has been around for a few years now, and has built up a loyal following. It was originally created as a Perl clone of some of the most interesting features from Vignette StoryServer, but has since become it's own unique animal. It comes from a publishing background, and includes features oriented towards splitting up pages into re-useable chunks, or "components."

Mason uses in-line Perl with a compilation approach, but has a feature to help keep the perl code out of the HTML coder's way. Components (templates) can include a section of Perl at the end of the file which is wrapped inside a special tag indicating that it should be run first, before the rest of the template. This allows programmers to put all the logic for a component down at the bottom away from the HTML, and then use short in-line Perl snippets in the HTML to insert values, loop through lists, etc.

Mason is a site development framework, not just a templating tool. It includes a very handy caching feature that can be used for capturing the output of components or simply storing data that is expensive to compute. It is currently the only tool that offers this sort of caching as a built-in. It also implements an argument parsing scheme which allows a component to specify the names, types, and default values that it expects to be passed, either from another component or from the values passed in the URI query string.

While the documentation mostly demonstrates a callback execution model, it is possible to use Mason in a pipeline style. This can be accomplished in various ways, including designating components as "autohandlers" which run before anything else for requests within a certain directory structure. An autohandler could do some processing and set up data for a display template which only includes minimal in-line Perl. There is also support for an object-oriented site approach, applying concepts like inheritance to the site directory structure. For example, the component at /store/book/ might inherit a standard layout from the component at /store/, but override the background color and navigation bar. Then /store/music/ can do the same, with a different color. This can be a very powerful paradigm for developing large sites.

Mason's approach to debugging is to create "debug files" which run Mason outside of a web server environment, providing a fake web request and activating the debugger. This can be helpful if you're having trouble getting Apache::DB to behave under mod_perl, or using an execution environment that doesn't provide built-in debugger support.

Another unique feature is the ability to leave the static text parts of a large template on disk, and pull them in with a file seek when needed rather than keeping them in RAM. This exchanges some speed for a significant savings in memory when dealing with templates that are mostly static text.

There are many other features in this package, including filtering of HTML output and a page previewing utility. Session support is not built-in, but a simple example showing how to integrate with Apache::Session is included. Mason's feature set can be a bit overwhelming for newbies, but the high-quality documentation and helpful user community go a long way.

## *1.5.3 HTML::Embperl*

Embperl makes its language choice known up front: embedded perl. It is one of the most popular in-line Perl templating tools and has been around longer than most of the others. It has a solid reputation for speed and ease of use.

It is commonly used in a callback style, with Embperl intercepting URIs and processing the requested file. However, it can optionally be invoked through a subroutine call from another program, allowing it to be used in a pipeline style. Templates are compiled to Perl bytecode and cached.

Embperl has been around long enough to build up an impressive list of features. It has the ability to run code inside a Safe compartment, support for automatically cleaning up globals to make mod_perl coding easier, and extensive debugging tools including the ability to e-mail errors to an administrator.

The main thing that sets Embperl apart from other in-line Perl systems is its tight HTML integration. It can recognize `TABLE` tags and automatically iterate over them for the length of an array. It automatically provides sticky form widgets. An array or hash reference placed at the end of a query string in an `HREF` or `SRC` attribute will be automatically expanded into query string "name=value" format. `META`

`HTTP-EQUIV` tags are turned into true HTTP headers.

Another reason people like Embperl is that it makes some of the common tasks of web application coding so simple. For example, all form data is always available just by reading the magic variable %fdat. Sessions are supported just as easily, by reading and writing to the magic %udat hash. There is also a hash for storing persistent application state. HTML-escaping is automatic (though it can be toggled on and off).

Embperl includes something called EmbperlObject, which allows you to apply OO concepts to your site hierarchy in a similar way to the inheritance features mentioned for Mason, above. This is a very convenient way to code sites with styles that vary by area, and is worth checking out.

One drawback of older versions of Embperl was the necessity to use built-in replacements for most of Perl's control structures like "if" and "foreach" when they are being wrapped around non-Perl sections. For example:

```
[$ if ($foo) $]
  Looks like a foo!
[$ else $]
  Nope, it's a bar.
[$ endif $]
```

These may seem out of place in a system based around in-line Perl. As of version 1.2b2, it is possible to use Perl's standard syntax instead:

```
[$ if ($foo) { $]
  Looks like a foo!
[$ } else { $]
  Nope, it's a bar.
[$ } $]
```

At the time of this writing, a new 2.x branch of Embperl is in beta testing. This includes some interesting features like a more flexible parsing scheme which can be modified to users' tastes. it also supports direct use of the Perl debugger on Embperl templates, and provides performance improvements.

### *1.5.4 Apache::ASP*

Apache::ASP started out as a port of Microsoft's Active Server Pages technology, and its basic design still follows that model. It uses in-line Perl with a compilation approach, and provides a set of simple objects for accessing the request information and formulating a response. Scripts written for Microsoft's ASP using Perl (via ActiveState's PerlScript) can usually be run on this system without changes. (Pages written in VBScript are not supported.)

Like the original ASP, it has hooks for calling specified code when certain events are triggered, such as the start of a new user session. It also provides the same easy-to-use state and session management. Storing and retrieving state data for a whole application or a specific user is as simple as a single method call. It can even support user sessions without cookies by munging URLs -- a unique feature among these systems.

A significant addition that did not come from Microsoft ASP is the XML and XSLT support. There are two options provided: XMLSubs and XSLT transforms. XMLSubs is a way of adding custom tags to your pages. It maps XML tags to your subroutines, so that you can add something like `<site:header page="Page Title" />` to your pages and have it translate into a subroutine call like `&site::header({title => "Page Title"})`. It can handle processing XML tags with body text as well.

The XSLT support allows the output of ASP scripts to be filtered through XSLT for presentation. This allows your ASP scripts to generate XML data and then format that data with a separate XSL stylesheet. This support is provided through integration with the XML::XSLT module.

Apache::ASP provides sticky widgets for forms through the use of the HTML::FillInForm module. It also has built-in support for removing extra whitespace from generated output, gzip compressing output (for browsers that support it), tracking performance using Time::HiRes, automatically mailing error messages to an administrator, and many other conveniences and tuning options. This is a mature package which has evolved to handle real-world problems.

One thing to note about the session and state management in this system is that it currently only supports clusters through the use of network filesystems like NFS or SMB. (Joshua Chamas, the module’s author, has reported much better results from Samba file-sharing than from NFS.) This may be an issue for large-scale server clusters, which usually rely on a relational database for network storage of sessions. Support database storage of sessions is planned for a future release.

## *1.5.5 Text::Template*

This module has become the de facto standard general purpose templating module on CPAN. It has an easy interface and thorough documentation. The examples in the docs show a pipeline execution style, but it’s easy to write a mod_perl handler that directly invokes templates, allowing a callback style. The module uses in-line Perl. It has the ability to run the in-line code in a Safe compartment, in case you are concerned about mistakes in the code crashing your server.

The module relies on creative uses of in-line code to provide things that people usually expect from templating tools, like includes. This can be good or bad. For example, to include a file you could just call Text::Template::fill_in_file(filename). However, you’ll have to specify the complete file path and nothing will stop you from using /etc/passwd as the file to be included. Most of the fancier templating tools have concepts like include paths, which allow you to specify a list of directories to search for included files. You could write a subroutine that works this way, and make it available in your template’s namespace, but it’s not built in.

Each template is loaded as a separate object. Templates are compiled to Perl and only parsed the first time they are used. However, to take full advantage of this caching in a persistent environment like mod_perl, your program will have to keep track of which templates have been used, since Text::Template does not have a way of globally tracking this and returning cached templates when possible.

Text::Template is not tied to HTML, and is just a templating module, not a web application framework. It is perfectly at home generating e-mails, PDFs, etc.

## *1.5.6 Template Toolkit*

One of the more recent additions to the templating scene, Template Toolkit is a very flexible mini-language system. It has a complete set of directives for working with data, including loops and conditionals, and it can be extended in a number of ways. In-line Perl code can be enabled with a configuration option, but is generally discouraged. It uses compilation, caching the compiled bytecode in memory and optionally caching the generated Perl code for templates on disk. Although it is commonly used in a pipeline style, the included Apache::Template module allows templates to be invoked directly from URLs.

Template Toolkit has a large feature set, so we’ll only be able cover some of the highlights here. The TT distribution sets a gold standard for documentation thoroughness and quality, so it’s easy to learn more if you choose to.

One major difference between TT and other systems is that it provides simple access to complex data structures through the concept of a dot operator. This allows people who don’t know Perl to access nested lists and hashes or call object methods. For example, we could pass in this Perl data structure:

```
$vars = {
         customer => {
                      name    => ’Bubbles’,
                      address => {
                                  city => ’Townsville’,
                                 }
                     }
        };
```

Then we can refer to the nested data in the template:

```
Hi there, [% customer.name %]!
How are things in [% customer.address.city %]?
```

This is simpler and more uniform than the equivalent syntax in Perl. If we pass in an object as part of the data structure, we can use the same notation to call methods within that object. If you’ve modeled your system’s data as a set of objects, this can be very convenient.

Templates can define macros and include other templates, and parameters can be passed to either. Included templates can optionally localize their variables so that changes made while the included template is executing do not affect the values of variables in the larger scope.

There is a filter directive, which can be used for post-processing output. Uses for this range from simple HTML entity conversion to automatic truncation (useful for pulldown menus when you want to limit the size of entries) and printing to STDERR.

TT supports a plugin API, which can be used to add extra capabilities to your templates. The provided plugins can be broadly organized into data access and formatting. Standard data access plugins include modules for accessing XML data or a DBI data source and using that data within your template. There’s a plugin for access to CGI.pm as well.

Formatting plugins allow you to display things like dates and prices in a localized style. There’s also a table plugin for use in displaying lists in a multi-column format. These formatting plugins do a good job of covering the final 5% of data display problems that often cause people who are using an in-house system to embed a little bit of HTML in their Perl modules.

In a similar vein, TT includes some nice convenience features for template writers like eliminating white space around tags and the ability to change the tag delimiters -- things that may sound a little esoteric, but can sometimes make templates significantly easier to work with.

The TT distribution also includes a script called ttree which allows for processing an entire directory tree of templates. This is useful for sites that pre-publish their templated pages and serve them statically. The script checks modification times and only updates pages that require it, providing a make-like functionality. The distribution also includes a sample set of template-driven HTML widgets which can be used to give a consistent look and feel to a collection of documents.

## *1.5.7 HTML::Template*

HTML::Template is a popular module among those looking to use a mini-language rather than in-line Perl. It uses a simple set of tags which allow looping (even on nested data structures) and conditionals in addition to basic value insertion. The tags are intentionally styled to look like HTML tags, which may be useful for some situations.

As the documentation says, it "does just one thing and it does quickly and carefully" -- there is no attempt to add application features like form-handling or session tracking. The module follows a pipeline execution style. Parsed templates are stored in a Perl data structure which can be cached in any combination of memory, shared memory (using IPC::SharedCache), and disk. The documentation is complete and well-written, with plenty of examples.

You may be wondering how this module is different from Template Toolkit, the other popular mini-language system. Beyond the obvious differences in syntax, HTML::Template is faster and simpler, while Template Toolkit has more advanced features, like plugins and dot notation. Here’s a simple example comparing the syntax:

HTML::Template:

```
<TMPL_LOOP list>
    <a href="<TMPL_VAR url>"><b><TMPL_VAR name></b></a>
</TMPL_LOOP>
```

Template Toolkit:

```
[% FOREACH list %]
    <a href="[% url %]"><b>[% name %]</b></a>
[% END %]
```

And now, a few honorable mentions:

### 1.5.8 AxKit2

Previous versions of this document covered Apache::AxKit. However, the coverage was not very good and that project has since been replaced by AxKit2.

Like its predecessor, AxKit2 is all about XML. It is more of a framework than a templating system per se, but includes support for multiple templating systems within it. The systems supported in the core release include XSLT, TAL (see the notes on Petal and Template::TAL below), XSP, an XML-based mini-language which can be extended with your own tags, and facilities for writing custom tag libraries.

Rather than short-change AxKit2 here with an inadequate description, I would encourage people who are intrigued by what they've heard so far to go and check out AxKit2 on CPAN. If you like working with XML, this may be a very good fit for you.

### 1.5.9 HTML_Tree

As mentioned earlier, HTML Tree uses a fairly unique method of templating: it loads in an HTML page, parses it to a DOM, and then programmatically modifies the contents of nodes. This allows it to use genuine valid HTML documents as templates, something which none of these other modules can do. The learning curve is a little steeper than average, but this may be just the thing if you are concerned about keeping things simple for your HTML coders. Unfortunately, HTML_Tree seems to be a dead project at this point and has not had an update in years. (Note that the name is "HTML_Tree", not "HTML::Tree".)

### 1.5.10 Petal and Template::TAL

Both of these modules are based on the TAL templating language created by the developers of the (Python) Zope CMS. (Petal offers some additions, while Template::TAL tries to be a strict implementation of the TAL spec.) The basic idea is to make your templates XML documents and use attributes in a TAL namepace to specify data, loops, and conditionals. This means it is essentially using a mini-language, but the templates end up being valid XML documents, which allows them to be edited with XML tools.

There are a couple of downsides to TAL. One is the verbosity. Compared to most of the other tools listed here, TAL is very verbose. This is a consequence of using XML attributes for everything. Here's an example from the Template::TAL docs:

```
<li tal:repeat="user users">
  <a href="?" tal:attributes="href user/url"><span tal:replace="user/name"/></a>
</li>
```

The other issue is the need for your templates to be valid XML (most likely XHTML). Petal attempts to be more forgiving by implementing a custom parser and allowing XHTML and HTML. This should allow the use of WYSIWYG tools to edit templates. In practice though, the custom parser is easily confused by HTML that browsers would handle without a problem. A custom parser is also problematic from a maintenance perspective since it doesn't benefit from improvements in the commonly used XML parsers. Template::TAL uses XML::LibXML.

Both Petal and Template::TAL have seen relatively recent maintenance releases, which makes them a much safer bet than HTML_Tree at this point.

### *1.5.11 ePerl*

Possibly the first module to embed Perl code in a text or HTML file, ePerl is getting a bit long in the tooth. The mod_perl-aware version, Apache::ePerl, caches compiled bytecode in memory to achieve solid performance, and some people find it refreshingly simple to use. However, it lacks many of the features that the other more modern systems have, and may be difficult to compile on recent versions of Perl.

### *1.5.12 CGI::FastTemplate*

This module takes a minimalistic approach to templating, which makes it unusually well suited to use in CGI programs. It parses templates with a single regular expression and does not support anything in templates beyond simple variable interpolation. Loops are handled by including the output of other templates. Unfortunately, this leads to a Perl coding style that is more confusing than most, and a proliferation of template files. However, some people swear by this dirt-simple approach.

## 1.6 Performance

People always seem to worry about the performance of templating systems. If you’ve ever built a large-scale application, you should have enough perspective on the relative costs of different actions to know that your templating system is not the first place to look for performance gains. All of the systems mentioned here have excellent performance characteristics in persistent execution environments like mod_perl. Compared to such glacially slow operations as fetching data from a database or file, the time added by the templating system is almost negligible.

If you think your templating system is slowing you down, get the facts: pull out Devel::DProf and see. If one of the tools mentioned here is at the top of the list for wall clock time used, you should pat yourself on the back -- you’ve done a great job tuning your system and removing bottlenecks! Personally, I have only seen this happen when I had managed to successfully cache nearly every part of the work to handle a request except running a template.

However, if you really are in a situation where you need to squeeze a few extra microseconds out of your page generation time, there are performance differences between systems. They’re pretty much what you would expect: systems that do the least run the fastest. Using in-line print() statements is faster than using templates. Using simple substitution is faster than using in-line Perl code. Using in-line Perl code is faster than using a mini-language.

The only templating benchmark available at this time is one developed by Joshua Chamas, author of Apache::ASP. It includes a "hello world" test, which simply checks how fast each system can spit back those famous words, and a "hello 2000" test, which exercises the basic functions used in most dynamic pages. It is available from the following URL:

http://www.chamas.com/bench/hello.tar.gz

Results from this benchmark currently show SSI, Apache::ASP, and HTML::Embperl having the best performance of the lot. Not all of the systems mentioned here are currently included in the test. If your favorite was missed, you might want to download the benchmark code and add it. As you can well imagine, benchmarking people's pet projects is largely a thankless task and Joshua deserves some recognition and support for this contribution to the community.

### *1.6.1 CGI Performance Concerns*

If you're running under CGI, you have bigger fish to fry than worrying about the performance of your templating system. Nevertheless, some people are stuck with CGI but still want to use a templating system with reasonable performance. CGI is a tricky situation, since you have to worry about how much time it will take for Perl to compile the code for a large templating system on each request. CGI also breaks the in-memory caching of templates used by most of these systems, although the slower disk-based caching provided by Mason, HTML::Template, and Template Toolkit will still work. (HTML::Template does provide a shared memory cache for templates, which may improve performance, although shared memory on my Linux system is usually slower than using the filesystem. Benchmarks and additional information are welcome.)

Your best performance bet with CGI is to use one of the simpler tools, like CGI::FastTemplate or Text::Template. They are small and compile quickly, and CGI::FastTemplate gets an extra boost since it relies on simple regex parsing and doesn't need to eval any in-line Perl code. Almost everything else mentioned here will add tenths of seconds to each page in compilation time alone.

## 1.7 Updates

These modules are moving targets, and a document like this is bound to contain some mistakes. Send your corrections to <perrin (at) elem.com>. Future versions of this document will be announced on the mod_perl mailing list, and possibly other popular Perl locations as well.

by Perrin Harkins

## 1.8 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Perrin Harkins <perrin (at) elem.com>.

## 1.9 Authors

- Perrin Harkins <perrin (at) elem.com>.

Only the major authors are listed above. For contributors see the Changes file.

# Table of Contents: