

*Scalable Templating for the Web*

# Perl Template Toolkit



**O'REILLY**<sup>®</sup>

*Darren Chamberlain,  
Dave Cross & Andy Wardley*

---

# Building a Complete Web Site Using the Template Toolkit

This chapter puts the Template Toolkit into context. We show several different ways of using the Template Toolkit to simplify the process of building and managing web site content. We start with some simple examples showing the use of template variables and template components that allow web content to be constructed in a modular fashion. As we progress further into the chapter, we look at more advanced techniques that address the issues of managing the site structure, generating menus and other navigation components, and defining and using complex data.

Although the focus of this chapter is on generating web content, it also serves as a general introduction to the Template Toolkit. It demonstrates techniques that can be adapted to different application areas. This chapter will quickly get you up to speed using the Template Toolkit, but without bogging you down in too much gory detail (we're saving that for the rest of the book). We come back to the Web to look at more advanced examples of static and dynamic web content in Chapter 11 and Chapter 12.

Although we may touch briefly on some more advanced issues, we try not to bore you with too much detail, except where it is absolutely necessary to illustrate a key point or explain an important concept. Chapter 3 discusses the syntax and structure of templates and the use of variables, while Chapter 4 covers the various template directives. More information relating to filters and plugins can be found in Chapter 5 and Chapter 6, respectively. More advanced topics concerning the use of the Template Toolkit for generating web content and interfacing to web applications can be found in Chapter 11 and Chapter 12.

We assume a Unix system in the examples in this chapter, but the principles apply equally well to other operating systems. On a Microsoft Windows machine, for example, the File Explorer can be used to create folders (directories) and shortcuts (symbolic links) using the familiar point-and-click interface. Another option we can highly recommend is to install *Cygwin*. *Cygwin* is freely available from <http://www.cygwin.com> and provides you with a Unix-like environment on Win32.

# Getting Started

Every big web site is made up of individual pages. Let's start with a small and simple page, showing how to eliminate basic repetition using templates. In later sections, we can build on this to generate more pages and add more complex elements.

## A Single Page

Example 2-1 shows the HTML markup of a page that displays the customary "Hello World" message, complete with a title, footer, and various other bits of HTML paraphernalia.

*Example 2-1. hello.html*

```
<html>
  <head>
    <title>Arthur Dent: Greet the Planet</title>
  </head>

  <body bgcolor="#FF6600">
    <h1>Greet the Planet</h1>

    <p>
      Hello World!
    </p>

    <hr />

    <div align="middle">
      &copy; Copyright 2003 Arthur Dent
    </div>
  </body>
</html>
```

HTML is relatively straightforward in terms of syntax and semantics. We'll assume that you've got at least a passing acquaintance with the basics of HTML. If you don't, *HTML & XML* by Chuck Musciano and Bill Kennedy (O'Reilly) provides a definitive guide to the subject.

Although HTML is simple, it does tend to be rather verbose. It's all too easy for the core content of the page to be obscured by the extra markup required around it. There's also some repetition that we would like to avoid. The page title and author's name both appear twice in the same page, for example. We can also assume that other pages in the site will be using similar pieces of data, repeated over and over again in numerous different places.

The author's name, background color, and copyright message are a few examples of items that we would really rather define in just one place in case we ever decide to change them. We don't want to have to edit every page in the site when we need to

change the copyright message (at the start of a new year, for example), or decide that blue is the new orange and want to use it as the background color for every page.

## A “Hello World” HTML Template

We can address these issues by applying the basic principles of template processing. Rather than creating the HTML page directly, we write a template for generating the HTML page. In this document, we use template variables to store these values instead of hardcoding them.

Example 2-2 shows a source template for the HTML page in Example 2-1. The author’s name, page title, background color, and year have been replaced by the variables `author`, `title`, `bgcol`, and `year`, respectively.

*Example 2-2. hello.tt*

```
<html>
  <head>
    <title>[% author %]: [% title %]</title>
  </head>

  <body bgcolor="[% bgcol %]">
    <h1>[% title %]</h1>

    <p>
      Hello World!
    </p>

    <hr />

    <div align="middle">
      &copy; Copyright [% year %] [% author %]
    </div>
  </body>
</html>
```

## Processing Templates with `tpage`

Of course, a template isn’t something a browser can make sense of. We need to process the template to generate HTML to send to the browser. Let’s use the `tpage` command we met in Chapter 1:

```
$ tpage --define author="Arthur Dent" \
> --define title="Greet the Planet" \
> --define bgcol="#FF6600" \
> --define year=2003 \
> hello.tt > hello.html
```

The `hello.html` now contains the same HTML that we saw in Example 2-1. This time, however, it has been generated from a template. The benefit of this approach is that

we easily change any of these variable values and generate a new HTML page, simply by invoking *tpage* with a different set of parameters.

## Template Components

Example 2-2 shows a template for generating a complete HTML page. We refer to this kind of template as a *page template* to distinguish it from the other kind of template that we're now going to introduce: the *template component*.

We use the term “template component” to help us identify those smaller templates that contain a reusable chunk of text, markup, or other content, but don't constitute complete pages in their own right. Template components are no different from page templates as far as the Template Toolkit is concerned—they're all just text files with embedded directives that need processing and get treated equally. Examples of typical template components include headers, footers, menus, and other user interface elements that you will typically want to use and reuse in different page templates across the site.

When we start using *ttree* a little later in this chapter, we will need to be more careful about storing our page templates separately from any template components. For now, however, we can keep them all in the same directory, simplifying matters for the purpose of our examples. As a general naming convention, we use a *.tt* or *.html* file extension for page templates (e.g., *hello.tt*), and no extension for component templates (e.g., *header*), but this is entirely arbitrary. If you want to give them an extension (e.g., *header.ttc*), that's fine.

## Headers and Footers

Our first components can be created easily. Extract the header and footer blocks from Example 2-2 and save them in their own *header* and *footer* template files, as in Examples 2-3 and 2-4.

*Example 2-3. header*

```
<html>
  <head>
    <title>[% author %]: [% title %]</title>
  </head>

  <body bgcolor="[% bgcolor %]">
    <h1>[% title %]</h1>
```

*Example 2-4. footer*

```
<hr />

<div align="middle">
  &copy; Copyright [% year %] [% author %]
```

Example 2-4. footer (continued)

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

## The PROCESS directive

We can now load these template components into a page template using the PROCESS directive. Example 2-5 shows this in action.

Example 2-5. *goodbye.tt*

```
[% PROCESS header %]

<p>
  Goodbye World.
</p>

[% PROCESS footer %]
```

When the Template Toolkit encounters a PROCESS directive, it loads the template from the file named immediately after the PROCESS keyword (*header* and *footer* are the two templates in this example), processes it to resolve any embedded directives, and then inserts the generated output into the calling template in place of the original directive.

We can use *tpage* to process the *goodbye.tt* template and save the generated output to *goodbye.html*:

```
$ tpage --define author="Arthur Dent" \
> --define title="We'll Meet Again" \
> --define bgcolor="#FF6600" \
> --define year=2003 \
> goodbye.tt > goodbye.html
```

The output generated, shown in Example 2-6, shows how the header and footer have been processed into place and the variable references within them correctly resolved.

Example 2-6. *goodbye.html*

```
<html>
  <head>
    <title>Arthur Dent: We'll Meet Again</title>
  </head>

  <body bgcolor="#FF6600">
    <h1>We'll Meet Again</h1>

    <p>
      Goodbye World.
    </p>
```

Example 2-6. *goodbye.html* (continued)

```
<hr />

<div align="middle">
  &copy; Copyright 2003 Arthur Dent
</div>
</body>
</html>
```

## The INSERT directive

The Template Toolkit provides a number of different directives for loading external template components. The `INSERT` directive, for example, inserts the contents of a template, but *without* processing any directives that may be embedded in it:

```
[% INSERT footer %]
```

`INSERT` is faster than `PROCESS` because there's much less work involved in inserting a file than there is in processing it as a template. It's not going to work for us in our current example because of the year and author variables in the footer that need resolving. If we `INSERT` the footer as it is, we'll see the `[% year %]` and `[% author %]` directives passed through as literal text.

However, we can hardcode the variables in the footer to make it a fixed block of text that we can then load using `INSERT`. For example:

```
<hr />

<div align="middle">
  &copy; Copyright 2003 Arthur Dent
</div>
</body>
</html>
```

Although we've no longer got the benefit of using variables or other template directives, we are still defining the footer in one place where we can easily make changes, should we ever need to.

In most day-to-day applications, the difference in speed between `INSERT` and `PROCESS` isn't going to be noticeable unless you really go looking for it. You're generally better off using whatever is most convenient for you, the template author. Worry about performance only if and when it ever becomes an issue. With this in mind, we'll leave our variables in the footer and continue to use `PROCESS`.

The other directives for loading templates are `INCLUDE` and `WRAPPER`, which we'll be looking at shortly.

## Benefits of Modularity

Separating commonly used blocks of markup into reusable template component files in this way allows you to take a modular approach to building your web content. This brings a number of important benefits.

The first is that the page templates become easier to write, edit, and maintain. You can quickly and easily add new pages by reusing existing template components to do the repetitive work, leaving the template author to concentrate on adding the core content. When it comes to updating the content, it becomes a lot easier to find what you're looking for because you don't have to pore through great chunks of HTML markup that define header, footers, menus, and other user interface elements.

In other words, we're achieving a *clear separation of concerns* between the core content of the pages and the parts that deal mainly with presentation. Content authors can concentrate on writing content without worrying about what kind of fancy user interface the web designers have dreamt up to fit around it

The second benefit is that the headers, footers, and other template components can easily be updated at any time, and need to be modified only in one place. Changing the copyright messages, the background color, or perhaps the layout of the footer, for *every* page on the site, becomes as easy as editing the one template component file and then processing the page templates to rebuild the site content.

So the clear separation of concerns also works the other way around. Web designers can concentrate on building a nice user interface for the entire site without having to worry too much about the content of individual pages.

Even if you're the all-in-one web designer, content author, and webmaster for your site, it is still useful to maintain a clear separation between these different aspects. You may have many hats to wear, but you'll be most comfortable wearing just one at a time.

## Defining Variables

Our current use of *tpage* for processing templates is hardly streamlined. We're spending a lot of time typing variable values on the command line, something that can only get worse as we add more pages that require processing to the site.

It would be easy to mistype the value for a variable, for example, or perhaps supply the wrong value altogether. You wouldn't see any complaint from the Template Toolkit. It would just go right ahead and process the template with whatever values you supplied, possibly leading to an error on an HTML page that could go unnoticed.

## Configuration Template

A better approach is to create a template component that defines any commonly used variables in one place. Example 2-7 shows our *config* template.

*Example 2-7. config*

```
[% author = 'Arthur Dent'  
   bgcolor = '#FF6600' # orange  
   year = 2003  
   copyr = "Copyright $year $author"  
-%]
```

You can define any number of variables in a single directive, as Example 2-7 illustrates. The Template Toolkit is very flexible in terms of the syntax it supports inside its tags, allowing you to spread your directives over several lines, adding as little or as much whitespace as you like for formatting purposes. You don't need to put each on a separate line as we have here—they can all go on the same line as long as some kind of whitespace is separating them. In the end, it's your choice. The Template Toolkit isn't fussy about how you lay out your directives, as long as you follow the basic rules of syntax, which we'll be introducing throughout this chapter and describing in greater detail in Chapter 3.

### Comments

You can add comments to annotate your code, as shown in the second line of Example 2-7: `# orange`. A comment starts with the `#` character and continues to the end of the current line. The comment is ignored by the Template Toolkit, and processing continues as normal on the next line.

If `#` is used as the first character immediately following the opening `[%` tag, the Template Toolkit ignores the entire directive up to the closing `[%`:

```
[%# this is a comment  
   this line is also part of the comment  
%]
```

### Variable values

In Example 2-7, the four variables set are `author`, `bgcolor`, `year`, and `copyr`. The first two are defined as the literal strings `'Arthur Dent'` and `'#FF6600'`. The `'` single quotation marks surrounding the values indicate that the contents should be used as provided. This makes it clear to the Template Toolkit that the `#` character in the definition for `bgcolor`, for example, is part of the value and not the start of a comment. The third variable, `year`, is defined as the integer value `2003`. Numbers such as these (and also floating-point numbers such as `2.718`) don't need to be quoted, but can be if you prefer.

The last variable, `copyr`, shows an example of a double-quoted string, in which the value is enclosed by " characters. Here the Template Toolkit looks for any references to variables embedded in the string, denoted by the \$ character, and replaces (*interpolates*) them for the corresponding values. In this example, the values for year and author will be interpolated into the string, resulting in the `copyr` variable being set to "Copyright 2003 Arthur Dent".

## Loading the Configuration Template

The `config` template can now be loaded using the `PROCESS` directive to gain access to these variable definitions. This is shown in Example 2-8, which also defines the `title` variable specific to this page. This is really no different from the way you might define a constant or global variable at the start of a program in Perl or some other programming language. It's good practice to do this at the top of the file, where any future changes can easily be made.

*Example 2-8. earth.tt*

```
[% title = 'Earth' -%]
[% PROCESS config -%]
[% PROCESS header %]

<p>
  Mostly Harmless.
</p>

[% PROCESS footer %]
```

Notice the - character placed immediately before the closing %] tags at the end of the directives on the first two lines. This tells the Template Toolkit to remove, or *chomp*, the newline and any other whitespace following the directive. Some older web browsers don't like to see whitespace appearing before the opening <html> element, so this ensures that the *header* file is inserted right at the top of the output. In effect, it is as if we had written the template like so:

```
[% title = 'Earth' %][% PROCESS config %][% PROCESS header %]
...
```

Now the template can be processed using `tpage` without the need to provide variable values as command-line arguments:

```
$ tpage earth.tt > earth.html
```

## Merging directives

The start of each page template can be simplified by defining the `title` variable and the `PROCESS` directives within a single directive tag. Each command is separated from the next by a ; (semicolon) character.

For example, we can write:

```
[% title = 'Earth';  
    PROCESS config;  
    PROCESS header  
%]
```

instead of the more verbose:

```
[% title = 'Earth' -%]  
[% PROCESS config -%]  
[% PROCESS header %]
```

There's no need for a semicolon at the end of the last directive, but the Template Toolkit won't complain if it finds one there. As we saw earlier, semicolons aren't required between variable definitions that appear one after another. However, a semicolon is required if you switch from setting variables (which is technically the SET directive, although the explicit keyword is rarely used) to another kind of directive (e.g., PROCESS) in the same tag:

```
[% pi = 3.142      # semicolon optional  
    e = 2.718      # " " " "  
    i = 1.414;     # semicolon mandatory  
    PROCESS config; # " " " "  
    phi = 1.618    # semicolon optional  
%]
```

The distinction becomes a little more obvious when we use the SET keyword explicitly and add some whitespace to format the directives more clearly:

```
[% SET pi = 3.142  
    e = 2.718  
    i = 1.414;  
  
    PROCESS config;  
  
    SET phi = 1.618  
%]
```

There's one final improvement we can make to the block at the start of our page templates. The two PROCESS directives can be merged into one, with the names of the templates separated by a + character:

```
[% title = 'Earth';  
  
    PROCESS config  
    + header  
%]
```

The general rule of whitespace being insignificant inside directives applies equally well to the PROCESS directive, allowing us to list all the files on the same line, or across a number of lines, as we've done here. This flexibility allows us to lay out this header block in such a way that it's clear from a glance what's going on, and with the bare minimum of extra syntax cluttering up this high-level view.

Example 2-9 shows this in the context of a complete page template.

*Example 2-9. magrethea.tt*

```
[% title = 'Magrethea';  
  
    PROCESS config  
      + header  
-%]  
  
<p>  
  Home of the custom-made  
  luxury-planet building industry.  
</p>  
  
[% PROCESS footer %]
```

## Generating Many Pages

The *tpage* program is fine for processing single templates, but isn't really designed to handle the many pages that comprise a typical web site. For this, *ttree* is much more appropriate. It works by drilling down through a source directory of your choosing, looking for templates to process. The output generated is saved in a corresponding file in a separate destination directory.

In addition to working well with a large number of template files, *ttree* also provides a much greater range of configuration options that allow you to modify the behavior of the Template Toolkit when processing templates. This allows you to further simplify the process of generating and maintaining web content in a number of interesting ways that we'll explore throughout this section.

Our templates will need to be organized a little more carefully when using *ttree*. In particular, we need to separate those page templates that represent complete HTML pages (*hello.tt*, *goodbye.tt*, *earth.tt*, and *magrethea.tt* in our previous examples) from those that are reusable template components (*config*, *header*, and *footer*).

## Creating a Project Directory

We'll start by creating a directory for our web site, complete with subdirectories for the source templates for HTML pages (*src*), a library of reusable template components (*lib*), and the generated HTML pages (*html*). We'll also create a directory for miscellaneous files (*etc*), including a configuration file for *ttree*, and another (*bin*) for any scripts we accrue to assist in building the site and performing maintenance tasks.

```
$ cd /home/dent  
$ mkdir web  
$ cd web  
$ mkdir src lib html etc bin
```

## ttree Configuration File

Now we need to define a configuration file for *ttree*. Example 2-10 shows an example of a typical *etc/ttree.cfg* file.

*Example 2-10. etc/ttree.cfg*

```
# directories
src = /home/dent/web/src
lib = /home/dent/web/lib
dest = /home/dent/web/html

# copy images and other binary files
copy = \.(png|gif|jpg)$

# ignore CVS, RCS, and Emacs temporary files
ignore = \b(CVS|RCS)\b
ignore = ^#

# misc options
verbose
recurse
```

Options can appear in any order in the configuration file. In certain cases (such as `lib`, `copy`, and `ignore`), an option can be repeated any number of times.

The first section defines the three important template directories:

```
# directories
src = /home/dent/web/src
lib = /home/dent/web/lib
dest = /home/dent/web/html
```

The `src` option tells *ttree* where to look for HTML page templates. The `lib` option (of which there can be many) tells it where the library of additional template components can be found. Finally, the `dest` option specifies the destination directory for the generated HTML pages.

The next two sections provide regular expressions that *ttree* uses to identify files that should be copied rather than processed through the Template Toolkit (`copy`), and to identify files that should be ignored altogether (`ignore`):

```
# copy images and other binary files
copy = \.(png|gif|jpg)$

# ignore CVS, RCS, and Emacs temporary files
ignore = \b(CVS|RCS)\b
ignore = ^#
```

In this example, we're setting the options so that any images with `png`, `gif`, or `jpg` file extensions are copied, and any CVS or temporary files left lying around by our favorite text editor are ignored.

The next section sets two *tree* flags:

```
# misc options
verbose
recurse
```

The verbose flag causes *tree* to print additional information to STDERR about what it's doing, while it's doing it. The recurse flag tells it to recurse down into any sub-directories under the src directory.

## Running *tree* for the First Time

When you run *tree* for the first time, it will display the following prompt, which asks if you'd like it to create a default *.ttreerc* file:

```
Do you want me to create a sample '.ttreerc' file for you?
(file: /home/dent/.ttreerc) [y/n]:
```

Answer y to have it create the file in your home directory.

This file is used to provide a default configuration for *tree*. If you've got only one web site to maintain, you can copy the contents of the *etc/ttree.cfg* file into it and run *tree* without any command-line options:

```
$ ttree
```

If you've got more than one site to maintain, you'll probably want to keep separate configuration files for each. In that case, you can use the *-f* command-line option to provide the name of the configuration file when you invoke *tree*:

```
$ ttree -f /home/dent/web/etc/ttree.cfg
```

## Using a Build Script

Rather than providing a command-line configuration option for *tree* each time you use it, you may prefer to write a simple build script that does it for you (as in Example 2-11).

*Example 2-11. bin/build*

```
ttree -f /home/dent/web/etc/ttree.cfg $@
```

The *\$@* at the end of the line passes any command-line arguments on to the *tree* program, in addition to the *-f* option that is provided explicitly.

## *tree* Configuration Directory

Another alternative is to set the *cfg* option in the *.ttreerc* file to denote a default directory for *tree* configuration files. You could set this to point to the project directory:

```
cfg = /home/dent/web/etc
```

and then invoke *ttree* with the short name of the configuration file:

```
$ tpage -f ttree.cfg
```

If you have many different web sites to maintain, another option is to create one general directory for *ttree* configuration files and use symbolic links from this directory to the project-specific files. The *.ttree* directory in your home directory is a common choice. In the *.ttreerc* file, we specify it like so:

```
cfg = /home/dent/.ttree
```

Then we prepare the directory, creating a symbolic link to our project-specific configuration file. We give it a memorable name (e.g., *dentweb*) to distinguish it from the various other *ttree.cfg* files that we may create links to from this directory:

```
$ cd /home/dent
$ mkdir .ttree
$ cd .ttree
$ ln -s /home/dent/web/etc/ttree.cfg dentweb
```

With these changes in place, *ttree* can then be invoked using the *-f* option to specify the *dentweb* configuration file:

```
$ tpage -f dentweb
```

The settings in the *.ttreerc* file and the magic of symbolic links result in *ttree* ending up with the right configuration file without us having to specify the full path to it every time. The other benefit of this approach is that *ttree* can be invoked from any directory and the correct configuration file will still be located.

## Calling *ttree* Through the Build Script

From now on we'll assume that the *bin/build* script invokes *ttree* with the appropriate option to locate the configuration file. For the sake of clarity, we'll use it in the examples that follow whenever we want to build the site content, rather than calling *ttree* directly. Any other commands that you want performed when the site is built (e.g., copying files, restarting the web server or database) can also be added here.

As we saw in Example 2-11, any command-line options that we provide to the script are forwarded to *ttree*. One particularly useful option is *-h*, which provides a helpful summary of all the different *ttree* options:

```
$ bin/build -h
ttree 2.63 (Template Toolkit version 2.10)

usage: ttree [options] [files]

Options:
  -a      (--all)           Process all files, regardless of modification
  -r      (--recurse)      Recurse into sub-directories
  -p      (--preserve)     Preserve file ownership and permission
  -n      (--nothing)      Do nothing, just print summary (enables -v)
  -v      (--verbose)      Verbose mode
```

```

-h      (--help)      This help
-dbg    (--debug)     Debug mode
-s DIR  (--src=DIR)   Source directory
-d DIR  (--dest=DIR)  Destination directory
-c DIR  (--cfg=DIR)   Location of configuration files
-l DIR  (--lib=DIR)   Library directory (INCLUDE_PATH) (multiple)
-f FILE (--file=FILE) Read named configuration file (multiple)

```

File search specifications (all may appear multiple times):

```

--ignore=REGEX  Ignore files matching REGEX
--copy=REGEX    Copy files matching REGEX
--accept=REGEX  Process only files matching REGEX

```

Additional options to set Template Toolkit configuration items:

```

--define var=value  Define template variable
--interpolate       Interpolate '$var' references in text
--anycase           Accept directive keywords in any case.
--pre_chomp         Chomp leading whitespace
--post_chomp        Chomp trailing whitespace
--trim              Trim blank lines around template blocks
--eval_perl         Evaluate [% PERL %] ... [% END %] code blocks
--load_perl         Load regular Perl modules via USE directive
--pre_process=TEMPLATE Process TEMPLATE before each main template
--post_process=TEMPLATE Process TEMPLATE after each main template
--process=TEMPLATE  Process TEMPLATE instead of main template
--wrapper=TEMPLATE  Process TEMPLATE wrapper around main template
--default=TEMPLATE  Use TEMPLATE as default
--error=TEMPLATE    Use TEMPLATE to handle errors
--start_tag=STRING  STRING defines start of directive tag
--end_tag=STRING    STRING defined end of directive tag
--tag_style=STYLE   Use pre-defined tag STYLE
--plugin_base=PACKAGE Base PACKAGE for plugins
--compile_ext=STRING File extension for compiled template files
--compile_dir=DIR   Directory for compiled template files
--perl5lib=DIR      Specify additional Perl library directories

```

## A Place for Everything, and Everything in Its Place

Before we can run the build script to generate the site content, we will need to move our page and library template files into place.

The source templates for the HTML pages should now be moved into the *src* directory where *ttree* can find them. The HTML files that *ttree* generates in the *html* output directory will be given the same filename as the *src* template from which they are generated. For this reason, we'll be using a *.html* file extension on our page templates from now on.

Also, move the template components *config*, *header*, and *footer* into the *lib* directory. These are (for now) also identical to those shown in the earlier examples.

## Running the Build Script

Now we can run the *bin/build* script to invoke *ttree* to build the site content:

```
$ bin/build
ttree 2.63 (Template Toolkit version 2.10)

    Source: /home/dent/web/src
    Destination: /home/dent/web/html
    Include Path: [ /home/dent/web/lib ]
    Ignore: [ \b(CVS|RCS)\b, ^# ]
    Copy: [ \.(png|gif|jpg)$ ]
    Accept: [ * ]

+ earth.html
+ magrethea.html
```

The sample output from *ttree* shown here indicates that two page templates, *earth.html* and *magrethea.html*, were found in the *src* directory. The + character to the left of the filenames indicates that the templates were processed successfully. Corresponding *earth.html* and *magrethea.html* files will have been created in the *html* directory containing the output generated by processing the templates.

Now that we've set up *ttree* and told it where our page templates are located, we can add new pages to the site by simply adding them to the *src* directory. When you next run the build script, *ttree* will locate the new page templates, even if they're located deep in a subdirectory (thanks to the recurse option), and process them into the corresponding place in the *html* directory.

You can now build all the static web pages in your site using a single, simple command.

## Skipping Unmodified Templates

When *ttree* is run it tries to be smart in working out which templates need to be processed and which don't. It does this by comparing the file modification time of the page template with the corresponding output file (if any) that it previously generated.

Run the *bin/build* script again, and the + characters to the left of the filename change to the - character:

```
$ bin/build
ttree 2.63 (Template Toolkit version 2.10)

    Source: /home/dent/web/src
    Destination: /home/dent/web/html
    Include Path: [ /home/dent/web/lib ]
    Ignore: [ \b(CVS|RCS)\b, ^# ]
    Copy: [ \.(png|gif|jpg)$ ]
    Accept: [ * ]

- earth.html (not modified)
- magrethea.html (not modified)
```

This indicates that the templates weren't processed the second time around, with the message to the right of the filenames explaining why. In this case, *ttree* has recognized that the source templates, *src/earth.html* and *src/magrethea.html*, haven't been modified since the corresponding output files, *html/earth.html* and *html/magrethea.html*, were created. Given that nothing has changed, there's no need to reprocess the templates.

There may be times when you want to force *ttree* to build a particular page or even all the pages on the site, regardless of any file modification times. You can process one or more pages by naming them explicitly on the command line:

```
$ bin/build earth.html magrethea.html
```

One time that you might want to force all pages to be rebuilt is when you modify a header, footer, or some other template component that is used by all the pages. Unfortunately, *ttree* isn't smart enough to figure out which library templates are used by which page templates.\* The `-a` option tells *ttree* to ignore file modification times and process all page templates, regardless:

```
$ bin/build -a
```

## Adding Headers and Footers Automatically

In addition to the fact that *ttree* works well with large collections of page templates, it also has the benefit of providing a large number of configuration options that allow you to change the way it works and how it uses the underlying Template Toolkit processor. Two of the most convenient and frequently used options are `pre_process` and `post_process`. These allow you to specify one or more templates that should be *automatically* added to the top or bottom of each page template, respectively. This can be used to add standard headers and footers to a generated page, but pre- and postprocessed templates may not generate any visible output at all. For example, we can use a preprocessed template to configure some variables that we might want defined for use in the page template or other template components.

The following can be added to the bottom of the *etc/ttree.cfg* file to have the *config* and *header* templates preprocessed (in that order so that we can use variables defined in *config* in the *header*) and the *footer* template postprocessed:

```
pre_process = config
pre_process = header
post_process = footer
```

Now the page templates can be made even simpler, as Example 2-12 shows.

\* This occurs not because *ttree* is being lazy. It's actually very difficult, if not impossible, to do it accurately without processing the templates in their entirety. By this time, the Template Toolkit has already done the hard work, so there's nothing to be gained by discovering that the template didn't need processing after all.

Example 2-12. *src/magrethea.html*

```
[% title = 'Magrethea' -%]  
  
<p>  
  Home of the custom-made  
  luxury-planet building industry.  
</p>
```

Remember that you'll need to use the `-a` option to force *ttree* to rebuild all pages in the site to have the changes take effect:

```
$ bin/build -a
```

## Defining META Tags

There is one problem with this approach. The *header* template is processed in its entirety before the main page template gets a look in. This means that the `title` variable isn't set to any value when the *header* is processed. It doesn't get set until the page template is processed, by which time it's too late for the *header* to use it.

The Template Toolkit won't complain if it encounters a variable for which it doesn't have a value defined. Instead, it will quietly use an empty string (i.e., nothing at all) for the value of the variable and continue to process the remainder of the template. The `DEBUG` option (described in the Appendix) can be set to have it raise an error in these cases, and can be useful to help track down mistyped variable names and those that have somehow eluded definition.

We can use the `META` directive to solve our immediate problem. It works by allowing us to define values within the page template that *are* accessible for use in the *header* and any other preprocessed templates, *before* the main page template is itself processed.

Example 2-13 shows how this is done. Instead of defining the `title` in a `SET` directive (which technically we were, even if we had omitted the `SET` keyword for convenience), we use the `META` directive, but otherwise leave the definition of the variable unmodified.

Example 2-13. *src/milliways.html*

```
[% META title = 'Milliways' %]  
  
<p>  
  The Restaurant at the  
  End of the Universe.  
</p>
```

Variables defined like this are made available as soon as the template is loaded. This happens *before* any of the preprocessed templates are processed so that these `META` variables are defined and ready for use.

There are some subtle differences between META variables and normal SET variables. The first is that you can't use double-quoted strings to interpolate other variables into the values for META variables. You *can* use double-quoted strings, but you can't embed variables in them and expect them to get resolved. The simple reason for this is that META variables are defined before the template is processed with any live data. At this time, there aren't any variables defined, so there's no point trying to use them.

The second difference is that the variables must be accessed using the `template.` prefix:

```
[% template.title %] not [% title %]
```

The `template` variable is a special variable provided by the Template Toolkit containing information about the current page template being processed. It defines a number of items, including the name of the template file (`template.name`) and the modification time (`template.modtime`), as well as any META variables defined in the template (`template.title`).

The dot operator, `.`, is the Template Toolkit's standard notation for accessing a variable such as `title` that is one small part of a larger, more complex data structure such as `template`. It doesn't matter for now (or generally at all) how this is implemented behind the scenes because the dot operator hides or *abstracts* that detail from you so that you don't need to worry about it.

We'll be coming back to the dot operator later on in this chapter when we look at defining and using complex data structures. For now, it is sufficient to know that `template.title` is how we access the `title` META variable defined in the main page template.

We can easily modify our *header* template to accommodate these requirements and restore the page title to the generated header (see Example 2-14).

*Example 2-14. lib/header*

```
<html>
  <head>
    <title>[% author %]: [% template.title %]</title>
  </head>

  <body bgcolor="[% bgcol %]">
    <h1>[% template.title %]</h1>
```

## More Template Components

You can create any number of different reusable template components to help you generate the content for your web site. Whenever you find yourself repeating the same, or a similar, block of markup in more than one place, you might want to consider moving it into a separate template file that you can then use and reuse when-

ever you need it. This not only saves you a lot of typing, but also ensures that the HTML generated in each place you use it is identical, or as near to identical as you would like it to be, accounting for any variables that might change from one use to the next.

Example 2-15 shows a template component for displaying an entry from Arthur's favorite reference book.

*Example 2-15. lib/entry*

```
<p>
  The Hitch Hiker's Guide to the Galaxy
  has this to say on the subject of
  "[% title %]".
</p>

<table border="0">
  <tr valign="top">
    <td>
      <b>[% title %]:</b>
    </td>
    <td>
      [% content %]
    </td>
  </tr>
</table>
```

The template uses two variables, title and content. The value for title can in this case be copied from `template.title`, thereby providing the title set in the META directive for the page. A value for content will be set explicitly for the sake of simplicity. These variables can be set either before the PROCESS directive:

```
[% title = template.title
  content = 'Mostly harmless'
%]
```

```
[% PROCESS entry %]
```

or as part of the PROCESS directive, following the template name as additional arguments:

```
[% PROCESS entry
  title = template.title
  content = 'Mostly harmless'
%]
```

The end result is the same. The Template Toolkit treats all variables as global by default so that you can define a variable in one template and use it later in another without having to explicitly pass it as an argument every time. In both of the preceding examples, the title and content variables are defined globally and can subsequently be used in both the called template (*entry*) and the calling template (*earth.tt*) after the point of definition.

In the following fragment, for example, the reference to the content variable at the end of the template will generate the value “Mostly harmless” as set in the earlier PROCESS directive:

```
[% PROCESS entry
  title = template.title
  content = 'Mostly harmless'
%]

[% content %] # Mostly harmless
```

## The INCLUDE Directive

There may be times when you would rather keep the definition of certain variables local to a particular template. The INCLUDE directive provides a way of doing this. In terms of syntax, it is used in exactly the same way as the PROCESS directive in all except the keyword.

The key difference between INCLUDE and PROCESS is that INCLUDE *localizes* any variables that are passed to the template as arguments in the directive. The variables passed have local values for the template component being processed by INCLUDE, but then revert to their previous values or undefined states.

In the following fragment, we define two variables at the start of the template whose values we would like to preserve to be used in the sentence at the end:

```
[% name = 'Zaphod Beeblebrox'
  title = 'President of the Galaxy'
%]

[% INCLUDE entry
  title = 'Earth'
  content = 'Mostly harmless'
%]

Hi! I'm [% name %], [% title %].
```

The INCLUDE directive provides local definitions for the title and content variables for the *entry* template to display. However, the original value for the title variable will be left untouched, and there will be no trace of the content variable outside of the *entry* template.

The final line of the template generates the output that we’re expecting:

```
Hi! I'm Zaphod Beeblebrox, President of the Galaxy.
```

Had we used PROCESS instead of INCLUDE, the value for title would have been overwritten and the output generated by the final line would incorrectly read:

```
Hi! I'm Zaphod Beeblebrox, Earth.
```

There is one important caveat to be aware of. The `INCLUDE` directive only localizes simple variables. Any complex variables containing dot operators are effectively global regardless of whether you use `INCLUDE`, `PROCESS`, or any other directive.

Dotted variables are a little like Perl's package variables. In Perl, you can refer to a variable as, for example, `$My::Dog::Spot`. This tells Perl the precise location for the variable `$Spot` in the `My::Dog` package. In the Template Toolkit, the equivalent variable would be something like `my.dog.spot`.

On the other hand, a Perl variable written as just `$Spot` could be either a “global” (for these purposes) variable defined in the current package, or a lexically scoped variable in the current subroutine, for example. Similarly, in the Template Toolkit, the equivalent variable `spot` could also be a global variable or a local copy created by invoking a template using `INCLUDE`.

The explanation isn't important as long as you remember the simple rule: the `INCLUDE` localizes only simple variables that don't contain any “.” dots.

## Setting Default Values

When you define a reusable template component, you may want to provide default values for any variables used in the template. For example, the following template component might want to ensure that sensible values are provided for the `<title>` element and `bgcolor` attribute in the `<body>`, even if the respective `title` and `bgcol` variables aren't set:

```
<html>
  <head>
    <title>[% title %]</title>
  </head>
  <body bgcolor="[% bgcol %]">
    ...
```

### The `DEFAULT` directive

One way to achieve this is by using the `DEFAULT` directive. The syntax is the same as `SET` in everything but the keyword, allowing you to provide default values for one or more variables:

```
[% DEFAULT
  title = "Arthur Dent's Web Site"
  bgcolor = '#FF6600'
-%]
<html>
  <head>
    <title>[% title %]</title>
  </head>
  <body bgcolor="[% bgcolor %]">
    ...
```

The key difference between `DEFAULT` and `SET` is that `DEFAULT` will set the variable to the value prescribed only if it is currently undefined, if it is set to an empty string, or if it contains the number zero. (Perl programmers will recognize the similarity with Perl's idea about what is *true* and *false* when it comes to the value of a variable.) The component will use any existing values for `title` and `bgcol`, either defined globally or passed as explicit arguments when the template is used. Otherwise, it will use the values provided in the `DEFAULT` directive.

## Expressions

Another approach is to use Template Toolkit *expressions* instead of just variables. Expressions allow you to make logical statements including the `and` and `or` operators, both of which can be written in either upper- or lowercase. For example, we can write:

```
[% bgcol or '#FF6600' %]
```

instead of just:

```
[% bgcol %]
```

The tertiary `?:` operator is another option. It provides the equivalent of an `IF...THEN...ELSE` construct, in which the expression to the left of the `?` is evaluated to determine whether it is true or false. If true, whatever comes after the `?` and before the `:` is used. Otherwise, it returns whatever follows the `:`.

Here's an example showing how the `?:` operator can be used to generate an appropriate title for the page:

```
[% title ? "Arthur Dent: $title"
      : "Arthur Dent's Web Site"
%]
```

If the `title` variable is set, the string `"Arthur Dent: $title"` is used. This uses variable interpolation to insert the current value for the `title` variable into the string, following Arthur's name. If `title` isn't set to anything that the Template Toolkit considers meaningfully true, the string `"Arthur Dent's Web Site"` is instead used. The expression doesn't need to be split across two lines as we've shown here, but in this case it helps to make the code clearer and easier to read.

So if `title` is set to `Earth`, the directive will generate the following output:

```
Arthur Dent: Earth
```

If the `title` isn't set, it will instead generate this output:

```
Arthur Dent's Web Site
```

Expressions can also contain comparison operators, as shown in the following example. These are discussed in detail in Chapter 3.

```
[% age > 18 ? 'Welcome to my site...'
      : "Sorry, but you're not old enough..."
%]
```

**= versus ==.** One important distinction worth mentioning now is the difference between = and ==. The first performs an assignment, setting the variable named on the left to the value (or expression) on the right:

```
[% foo = bar %]
```

The second is the equality comparison operator, which tests to see whether the string values of the items on either side are identical:

```
[% foo == bar ? 'equal' : 'not equal' %]
```

**Setting variables using expressions.** Expressions can also be used to set the value of a variable. For example, the `pagetitle` variable can be set to either of the values previously shown, depending on the setting of `title`, using the following code:

```
[% pagetitle = title ? "Arthur Dent: $title"  
      : "Arthur Dent's Web Site"  
%]
```

It's perfectly valid to use a variable in an expression to update the same variable. Everything to the right of the = is evaluated first, and the resulting value is then used to set the variable specified to the left of the =:

```
[% title = title ? "Arthur Dent: $title"  
      : "Arthur Dent's Web Site"  
%]
```

**Setting variables using directives.** You can also assign the output of a directive to a variable. In the following example, the `header` template is processed using the `PROCESS` directive and the generated output is stored in the `headtext` variable:

```
[% headtext = PROCESS header %]
```

## The IF Directive

The IF directive can be used to encode more complex conditional logic in templates. It evaluates the expression following the IF keyword, which in these examples will be a simple variable. If the expression is true, the following block, up to the matching END directive, is processed. Otherwise, it is ignored.

Here's a simple example:

```
<body  
[%- IF bgcolor -%]  
  bgcolor="[% bgcolor %]"  
[%- END -%]  
>
```

This example uses an IF block to add the `bgcolor` attribute to the HTML `<body>` element, but only if the `bgcolor` variable is defined and contains a true value. By careful placement of - characters at the start and end of the IF and END directives, we're enabling the Template Toolkit's prechomping and postchomping facility. This

removes the newline characters before the [% tags and after the %] tags so that the output lines up in the correct place in the <body> element.

So, for a bgcolor value of #FF6600, the following output would be generated:

```
<body bgcolor="#FF6600">
```

For an undefined bgcolor, we would instead see the following:

```
<body>
```

Like many of the Template Toolkit directives that expect a block to follow, the IF directive can be used in *side-effect* notation.

For example, you can write:

```
[% INCLUDE header IF title %]
```

instead of the more laborious:

```
[% IF title; INCLUDE header; END %]
```

This works only when you've got a single directive or variable as the content for the block—in this example, it's the INCLUDE header directive. Our earlier example, which constructed the <body> tag, included both text and a reference to the bgcolor variable in the block. However, we can write this using a double-quoted string to interpolate the value for bgcolor:

```
<body [%- " bgcolor=\ "$bgcolor\ "" IF bgcolor %]>
```

Matters are complicated a little by the need to escape the double quotes inside the double quotes. The \ character tells the Template Toolkit that the following " is part of the string, and not the quote that terminates it. Overall it's an improvement over the more explicit IF...END form and illustrates a useful principle.

You can add an ELSE block after the IF block, which will be processed if the variable (or more generally, the expression) is false. For example:

```
[% IF bgcolor -%]  
<body bgcolor="[% bgcolor %]">  
[%- ELSE -%]  
<body>  
[%- END -%]
```

There is also the ELSIF directive, which allows you to define different blocks for different conditions:

```
[% IF name == 'Arthur Dent'  
  OR name == 'Ford Prefect' %]  
Hello [% name %]!  
[% ELSIF name.match('(?:vogon)') %]  
I'm sorry, but there's no one at home.  
Please don't bother calling again.  
[% ELSE %]  
Hello World!  
[% END %]
```

In this example, the ELSIF expression uses the `match` virtual method to test whether the name contains anything looking remotely Vogon. The argument passed to the `match` method is a Perl regular expression, allowing us to use the `(?i:...)` grouping to construct a case-insensitive match. An ELSE block is also provided in case neither the IF nor ELSIF conditions match.

The SWITCH directive, described in detail in Chapter 4, provides an alternative for more complicated multiway matching.

## Wrapper and Layout Templates

Now it's time to bring out some of the bigger guns of the Template Toolkit. The WRAPPER directive and layout templates let you define a common look for web pages in a single file, rather than scattering the components over *header* and *footer* files.

### The WRAPPER Directive

The *entry* template from Example 2-15 works well when the content to be displayed is relatively simple. However, it quickly becomes cumbersome for longer entries such as the one shown here:

```
[% INCLUDE entry
  title = 'Vogon Poetry'
  content = 'Vogon poetry is of course the
            third worst in the Universe.
            The second worst is that of...

            ...etc...

            ...in the destruction of the
            planet Earth'
%]
```

Special care must be taken when quoting content that contains quote characters. Consider the following extract that illustrates this problem:

```
Grunthos is reported to have been "disappointed"
by the poem's reception.
```

If this is enclosed in single-quote characters, the apostrophe in “poem's” must be escaped by preceding it with a backslash `\` character (the apostrophe and single-quote characters are one and the same for these purposes):

```
[% INCLUDE entry
  title = 'Grunthos the Flatulent'
  content = 'Grunthos is reported to have
            been "disappointed" by the
            poem\'s reception.'
%]
```

Another alternative is to use double quotes to define the variable, allowing single quotes to remain as they are. But in this case, any occurrences of double quotes will then need to be escaped:

```
[% INCLUDE entry
    title = 'Grunthos the Flatulent'
    content = "Grunthos is reported to have
                been \"disappointed\" by the
                poem's reception."
%]
```

A better solution is to use the WRAPPER directive. It works in a similar way to INCLUDE, but uses an additional END directive to enclose a block of template content. The WRAPPER directive uses this block as the value for the content variable:

```
[% WRAPPER entry
    title = 'Grunthos the Flatulent'
%]
Grunthos is reported to have
been "disappointed" by the
poem's reception.
[% END %]
```

The immediate benefit in this example is that the extract is now a block of plain text rather than a quoted string. There is no longer any need to escape the quote characters within it.

The WRAPPER block can contain any combination of text and template directives, even including other nested WRAPPER blocks. The following fragment shows a simple example in which the reaction variable is used to report Grunthos' reaction:

```
[% reaction = 'disappointed' %]

[% WRAPPER entry
    title = 'Grunthos the Flatulent'
%]
Grunthos is reported to have
been "[% reaction %]" by the
poem's reception.
[% END %]
```

The WRAPPER block is processed first to resolve any directives within it. Then the complete block, including any output generated dynamically by embedded directives, is passed to the *entry* template as the value for the content variable.

It's no coincidence that we chose content as a variable name in the *entry* template in Example 2-15, knowing full well that we would later use it in this example for WRAPPER. The WRAPPER directive always assigns the block content to the content variable, and in that sense it's one of the Template Toolkit's "special" variables, like the template variable that we used earlier. However, there's nothing to stop you from using it as a regular variable, and indeed it makes a good choice in any template for a variable that you might one day want to define as a block in a WRAPPER directive.

The end result is that the *entry* template works as expected, whether we call it using `INCLUDE` and pass the content explicitly as a variable, or call it using `WRAPPER` and define the content implicitly in the enclosed block.

## Using an Automatic Wrapper Template

In Examples 2-4 and 2-14, we created separate *header* and *footer* files to add to the start and end of each HTML page generated. One problem with this approach is that neither file contains valid HTML markup. The *header* provides the opening tag of the `html` element, for example, but the corresponding closing tag is located at the end of the *footer* file.

Having HTML elements split across separate files makes them harder to maintain, and increases the likelihood of them being accidentally mismatched or incorrectly nested. It is also likely to confuse or infuriate any HTML-aware text editors or validation tools that you may be using.

A better approach is to use a *wrapper* template to combine the *header* and *footer* into one template. The content variable is used to denote the position for the page content. This is shown in Example 2-16.

*Example 2-16. lib/wrapper*

```
<html>
  <head>
    <title>[% author %]: [% template.title %]</title>
  </head>

  <body bgcolor="[% bgcolor %]">
    <h1>[% template.title %]</h1>

    [% content %]

    <hr />

    <div align="middle">
      &copy; [% copyr %]
    </div>
  </body>
</html>
```

We need to modify the *etc/ttree.cfg* file to specify the new *wrapper* template using the *wrapper* option. The fact that our *wrapper* template happens to be called *wrapper* is entirely coincidental (but intentional). We could have named the file *tom*, *dick*, *larry*, or something else if we wanted to, but it wouldn't be as succinct or descriptive as *wrapper*.

We're still using the `pre_process` option to load the `config` template, but we can now remove the references to the `header` and `footer` (or comment them out as shown here), replacing them with a single `wrapper` option:

```
pre_process = config
wrapper     = wrapper
# pre_process = header
# post_process = footer
```

With the `wrapper` option in place, the Template Toolkit processes the main page template (after preprocessing the `config` template) and then calls the `wrapper` template, passing the generated page content as the content variable. It has the same effect as if there were an explicit `WRAPPER` directive around the entire page content:

```
[% WRAPPER wrapper %]
  The entire page content goes here...
[% END %]
```

Of course, the benefit of having the Template Toolkit apply a `wrapper` automatically is that you don't need to edit any of your page templates to add it explicitly. You can switch from using `pre_process` and `post_process` to `wrapper`, or you can change the name of any of the `header`, `footer`, or `wrapper` templates, without having to make any changes to your core content.

To put the change into effect, run the `bin/build` script with the `-a` option to have it rebuild all pages in the site:

```
$ bin/build -a
```

## Using Layout Templates

Most real web sites will require far more complex layout templates than the simple `wrapper` we saw in Example 2-16. A common practice is to use HTML tables to place different elements such as headers, footers, and menus in a consistent position and formatting style. These elements may themselves be built using tables and other HTML elements, perhaps nested several times over. This can quickly lead to confusing markup that is hard to read and even harder to update.

Consider the following example, which illustrates how difficult nested tables can be to write and maintain:

```
<table border="0" cellpadding="0" cellspacing="0">
  <tr valign="top">
    <td>
      <table border="0">
        <tr>
          <td>
            Oh Dear!
          </td>
          <td>
            This is not a good example
            of a layout template...
          </td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```

```

        <td>
            <table>
                ...etc...
            </table>
        </tr>
    </table>
</td>
<td>
    <table>
        ...etc...
    </table>
</td>
.
.
.

```

The sensible formatting helps to make the structure clearer through use of indenting. However, it is still difficult to match rows and cells with their corresponding tables, and there is little indication of what the different tables contribute to the overall layout.

A better approach is to build the layout using several different templates. For example, we can simplify the preceding template by moving the inner tables to separate templates:

```

<table border="0" cellpadding="0" cellspacing="0">
  <tr valign="top">
    <td>
      [% PROCESS sidebar %]
    </td>
    <td>
      [% PROCESS topmenu %]
    </td>
    .
    .
    .

```

Now we can easily see the high-level structure without getting bogged down in the detail of the nested tables. Furthermore, by giving our templates names that reflect their purpose (e.g., `sidebar` and `topmenu`), we effectively have a self-documenting template that shows at a glance what it does. Another benefit is that the individual elements, the `sidebar` and `topmenu` in this example, will themselves be much easier to write and maintain in isolation. They also become reusable, allowing you to incorporate them into another part of the site (or perhaps another site) with a `PROCESS` or similar directive.

## Layout Example

Let's work through a complete example now, applying this principle to the presentation framework for our web site. Example 2-17 shows an alternate version of the *wrapper* template that delegates the task to two further templates, *html* and *layout*.

*Example 2-17. lib/wrapper2*

```
[% WRAPPER html + layout;  
    content;  
    END  
-%]
```

The two wrapper templates, *html* and *layout*, are both specified in the one WRAPPER directive, separated using the + character in the same way that we used it with the PROCESS directive in Example 2-9. In this case, the page content will be processed first, then the *layout* template, and finally the *html* template. Remember that the WRAPPER directive works “inside out” by processing the wrapped content first, and then the wrapping templates.

If we unwrap the preceding directive into two separate WRAPPER calls, it should become more obvious why the WRAPPER directive processes the templates in the *reverse* order to how they’re specified:

```
[% WRAPPER html;  
    WRAPPER layout;  
    content;  
    END;  
    END  
%]
```

The end result is that it does what you would expect, regardless of the slightly counterintuitive order in which it does it. The *html* template ends up wrapping the *layout* template, which in turn wraps the value of the content variable, which in this case is the output from processing the main page template.

### Side-effect wrappers

The WRAPPER directive can also be used in side-effect notation. Consider the following fragment:

```
[% WRAPPER layout;  
    content;  
    END  
%]
```

You can simplify this by writing it as follows:

```
[% content WRAPPER layout %]
```

The *wrapper* template shown in Example 2-17 can be rewritten in the same way, as shown in Example 2-18.

*Example 2-18. lib/wrapper3*

```
[% content WRAPPER html + layout -%]
```

## Separating layout concerns

Using two separate layout templates, *html* and *layout*, allows us to make a clear separation between the different kinds of markup that we're adding to each page. The *html* template adds the `<head>` and `<body>` elements required to make each page valid HTML. The *layout* template deals with the overall presentation of the visible page content, adding a header, footer, menu, and other user interface components.

Example 2-19 shows the *html* template.

*Example 2-19. lib/html*

```
<html>
  <head>
    <title>[% author %]: [% template.title %]</title>
  </head>

  <body bgcolor="[% bgcol %]">
    [% content %]
  </body>
</html>
```

Example 2-20 shows the *layout* template.

*Example 2-20. lib/layout*

```
<table border="0" width="100%">
  <tr>
    <td colspan="2">
      [% PROCESS pagehead %]
    </td>
  </tr>
  <tr valign="top">
    <td width="150">
      [% PROCESS menu %]
    </td>
    <td>
      [% content %]
    </td>
  </tr>
  <tr>
    <td colspan="2" align="center">
      [% PROCESS pageinfo %]
    </td>
  </tr>
</table>
```

We've created a new header template, *pagehead*, shown in Example 2-21, which generates a headline for the page. It's simple for now, but we can easily change it to something more complicated at a later date.

*Example 2-21. lib/pagehead*

```
<h1>[% template.title %]</h1>
```

We're also using another template, *menu*, to handle the generation of a menu for the site. We'll be looking at this shortly.

Example 2-22 shows the final template used in the layout, *pageinfo*. This incorporates the copyright message and some information about the page template being processed.

*Example 2-22. lib/pageinfo*

```
[% USE Date %]

&copy; [% copyr %]

<br />

[% template.name -%]
last modified
[%- Date.format(template.modtime) %]
```

Notice how we're using the `template.name` and `template.modtime` variables to access the filename and modification time of the current page template. The `template.modtime` value is returned as a large number that means something to computers\* but not a great deal to humans. To turn this into something more meaningful, we're using the `Date` plugin to format the number as a human-readable string.

## Plugins and the USE directive

Plugins are a powerful feature of the Template Toolkit that allow you to load and use complex functionality in your templates, but without having to worry about any of the underlying implementation detail. Plugins are covered in detail in Chapter 6, but there's not much you need to know to start using them.

In Example 2-22, we first load the `Date` plugin with the `USE` directive:

```
[% USE Date %]
```

This creates a `Date` template variable that contains a reference to a plugin object (of the `Template::Plugin::Date` class, but you don't need to know that). We can then call the `format` method against the `Date` object using the dot operator, passing the value for `template.modtime` as an argument:

```
[%- Date.format(template.modtime) %]
```

The output generated would look something like this:

```
17:43:35 14-Jul-2003
```

\* It's the number of seconds that have elapsed since January 1, 1970, known as the the *Unix epoch*.

That's all we need to do to load and use the Date plugin. Dozens of plugins are available for doing all kinds of different tasks, described in detail in Chapter 6.

## Menu Components

In the *layout* template in Example 2-20, we delegate the task of generating a menu for the web site to the *menu* template. Before we look at how the template does this, let's see an example of the kind of HTML that we would like it to generate.

```
<table border="0">
  <tr>
    <td>
      
    </td>
    <td>
      <a href="earth.html">Earth</a>
    </td>
  </tr>
  <tr>
    <td>
      
    </td>
    <td>
      <a href="magrethea.html">Magrethea</a>
    </td>
  </tr>
</table>
```

The entire menu is defined as a `<table>` element, containing one `<tr>` row for each item, each of which holds two `<td>` cells, one to display an icon, the other a link to a particular page. Only two items are in this simple example, but already we can see how it gets repetitive very quickly. This suggests that we can modularize the markup into separate template components.

## Simple Menu Template

Example 2-23 shows a *menu* template that defines the outer `<table>` elements and uses a second template, *menuitem*, to generate each item.

*Example 2-23. lib/menu*

```
<table border="0">
[%
PROCESS menuitem
text = 'Earth'
link = 'earth.html';

PROCESS menuitem
text = 'Magrethea'
link = 'magrethea.html';
```

Example 2-23. *lib/menu* (continued)

```
    %]  
  </table>  
  
  [% BLOCK menuitem %]  
  <tr>  
    <td>  
        
    </td>  
    <td>  
      <a href "[% link %]">[% text %]</a>  
    </td>  
  </tr>  
  [% END %]
```

## The BLOCK directive

We could easily define the *menuitem* template in a separate file as we have with other components, but it would require us to split the HTML `<table>` markup into different files. This would make it harder to maintain and possibly lead to tag mismatch or other formatting errors.

Instead, we define the *menuitem* template inside the *menu* template using the BLOCK directive. The argument following the BLOCK keyword is a name for the template component, which can then be used in any PROCESS, INCLUDE, or WRAPPER directives. The content of the component follows, and can contain any kind of Template Toolkit directives up to the corresponding END directive.

```
  [% BLOCK menuitem %]  
    <tr>  
      <td>  
          
      </td>  
      <td>  
        <a href "[% link %]">[% text %]</a>  
      </td>  
    </tr>  
  [% END %]
```

The *menuitem* template block is defined at the bottom of the *menu* template, but that doesn't stop us from using it earlier in the same template, before it is defined.

The *menuitem* block will remain defined while the *menu* template is being processed. Any other templates that are called from within the *menu* template (e.g., by a PROCESS or INCLUDE directive) will also be able to use the *menuitem* block.

## Component Libraries

When a template is loaded using the PROCESS directive, any BLOCK definitions within it will be imported and available for use in the calling template. Templates loaded using the INCLUDE directive keep to themselves and don't export their BLOCK defini-

tions (or any of their local variables, as described in the earlier discussion of the INCLUDE directive).

This feature allows you to create single template files that contain libraries of smaller template components, defined using the BLOCK directive. This is illustrated in Example 2-24.

*Example 2-24. lib/mylib*

```
[% BLOCK image -%]
  
[%- END %]

[% BLOCK link -%]
  <a href="[% link %]">[% text %]</a>
[%- END %]

[% BLOCK icon;
  INCLUDE image
    src   = '/images/icon.png'
    alt   = 'dot icon'
    width = 4
    height = 4 ;
END
-%]
```

Notice how the *icon* BLOCK definition is defined within a single directive, and consists of nothing more than a call to the *image* template component, defined earlier in the same file. This illustrates how easy it is to reuse existing components to quickly adapt them for more specific, or alternate purposes.

The BLOCK definitions can be loaded from the *mylib* template with a PROCESS directive. Then they can be used just like any other template component. Example 2-25 shows a variation of the *menu* template from Example 2-23 in which the *icon* and *link* components are used to generate the menu items.

*Example 2-25. lib/menu2*

```
[% PROCESS mylib %]

<table border="0">
[%
  PROCESS menuitem
    text = 'Earth'
    link = 'earth.html';

  PROCESS menuitem
    text = 'Magrethea'
    link = 'magrethea.html';
%]
</table>
```

Example 2-25. *lib/menu2* (continued)

```
[% BLOCK menuitem %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link %]
  </td>
</tr>
[% END %]
```

## The EXPOSE\_BLOCKS option

You can also set an option that allows you to use BLOCK directives without having to first PROCESS the template in which they're defined. The `expose_blocks` option for *ttree* and the corresponding EXPOSE\_BLOCKS option for the Template module can be set to make this possible.

For example, by adding the following to the *etc/ttree.cfg* file:

```
expose_blocks
```

we can then access a BLOCK in the *mylib* template like so:

```
[% PROCESS mylib/icon %]
```

The template name, *mylib*, is followed by the BLOCK name, *icon*, separated by a / (slash) character. The notation is intentionally identical to how you would specify the *icon* file in the *mylib* directory. This is another example of how the Template Toolkit abstracts certain underlying implementation details so that you don't tie yourself down to one particular way of doing something.

At a later date, for example, you might decide to split the *mylib* template into separate files, stored in the *mylib* directory. The same directive will continue to work because the syntax is exactly the same for blocks in files as it is for files in directories:

```
[% PROCESS mylib/icon %]
```

This gives you more flexibility in allowing you to change the way you organize your template components, without having to worry about how that might affect the templates that use them.

## The FOREACH Directive

The menu component from Example 2-25 can be simplified further by first defining a list of menu items and then iterating over them using the FOREACH directive. Example 2-26 demonstrates this.

*Example 2-26. lib/menu3*

```
[% PROCESS mylib %]

[% menu = [
  { text = 'Earth'
    link = 'earth.html' }
  { text = 'Magrethea'
    link = 'magrethea.html' }
]
%]

<table border="0">
[% FOREACH item IN menu %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link
      text = item.text
      link = item.link
    %]
  </td>
</tr>
[% END %]
</table>
```

The menu variable is defined as a list of hash arrays, each containing a text and link item:

```
[% menu = [
  { text = 'Earth'
    link = 'earth.html' }
  { text = 'Magrethea'
    link = 'magrethea.html' }
]
%]
```

The main body of the template defines an HTML `<table>` element. Within the table, the `FOREACH` directive iterates through the menu list, setting the `item` variable to each element in turn.

```
<table border="0">
[% FOREACH item IN menu %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link
      text = item.text
      link = item.link
    %]
  </td>
```

```

</tr>
[% END %]
</table>

```

The block following the FOREACH directive, up to the corresponding END, can contain text and other directives, even including nested FOREACH blocks. To make the code easier to read, we might prefer to define the *menuitem* BLOCK, as shown in Example 2-25. This allows us to simplify the FOREACH directive, merging it into a single tag.

```

<table border="0">
[% FOREACH item IN menu;
    PROCESS menuitem
        text = item.text
        link = item.link;
    END
%]
</table>

```

The FOREACH block now contains just one directive to PROCESS the *menuitem* component. The text and link variables are set to the *item.text* and *item.link* values, respectively.

When the items in a FOREACH list are hash arrays, as they are in Example 2-26, you can omit the name of the *item* variable:

```

<table border="0">
[% FOREACH menu;
    PROCESS menuitem;
    END
%]
</table>

```

In this case, the values in each hash array will be made available as local variables inside the FOREACH block. So *item.text* becomes the *text* variable, and *item.link* becomes *link*, but only within the scope of the FOREACH block. This conveniently allows us to process the *menuitem* template without needing to explicitly dereference the *item* variables.

There's one more improvement we can make by taking advantage of the Template Toolkit's side-effect notation. Instead of writing the PROCESS *menuitem* directive in the FOREACH block all by itself, we can put it *before* the FOREACH and do away with the semicolons and END keyword:

```

<table border="0">
[% PROCESS menuitem FOREACH menu %]
</table>

```

All these enhancements to the *menu* template are shown in Example 2-27.

*Example 2-27. lib/menu4*

```

[% PROCESS mylib %]

[% menu = [

```

Example 2-27. *lib/menu4* (continued)

```
{ text = 'Earth'
  link = 'earth.html' }
{ text = 'Magrethea'
  link = 'magrethea.html' }
]
%]

<table border="0">
[% PROCESS menuitem FOREACH menu %]
</table>

[% BLOCK menuitem %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link %]
  </td>
</tr>
[% END %]
```

## Defining and Using Complex Data

The variables that we have used so far have mostly been simple *scalar* variables that contain just one value. The few exceptions include the tantalizing glimpses of the template variable, and the Date plugin in Example 2-22. As we saw in Chapter 1, the Template Toolkit also supports lists and hash arrays for complex data, and allows you to access Perl subroutines and objects.

In this section, we will look more closely at defining and using complex data structures, and describe the different Template Toolkit directives for inspecting, presenting, and manipulating them.

## Structured Configuration Templates

Larger sites will typically use dozens of different global site variables to represent colors, titles, URLs, copyright messages, and various other parameters. The Template Toolkit places no restriction on the number of different variables you use, but you and your template authors may soon lose track of them if you have too many.

Another problem with having lots of global variables lying around is that you might accidentally overwrite one of them. We saw in Example 2-7 how the *author* variable was used to store the name of the site author, Arthur Dent, for use in the *header* and *footer* templates. At some later date, we might decide to add a *quote* template component that also uses the *author* variable. This is shown in Example 2-28.

Example 2-28. *lib/quote*

```
<blockquote>
  [% quote %]
</blockquote>

-- [% author %]
```

There’s no problem if we use `INCLUDE` to load the template, providing a local variable value for `author`:

```
[% INCLUDE quote
  author = 'Douglas Adams'
  quote = 'I love deadlines. I like the
          whooshing sound they make as
          they fly by.'
%]
```

The value for `author` supplied as a parameter to the `INCLUDE` directive (Douglas Adams) remains set as a local variable within the *quote* template. It doesn’t affect the global `author` variable that is defined in the `config` (Arthur Dent).

However, it is all too easy to forget that the `author` variable is “reserved”—especially if it’s just one of a large number of such variables—and to use `PROCESS` instead of `INCLUDE`:

```
[% PROCESS quote
  author = 'Douglas Adams'
  quote = 'I love deadlines. I like the
          whooshing sound they make as
          they fly by.'
%]
```

The `PROCESS` directive doesn’t localize any variables. As a result, our global `author` variable now is incorrectly set to Douglas Adams instead of Arthur Dent. One solution is to religiously use `INCLUDE` instead of `PROCESS` at every opportunity. However, that’s just working around the problem rather than addressing the real issue. Furthermore, the `INCLUDE` directive is quite a bit slower than `PROCESS`, and if performance is a concern for you, you should be looking to use `PROCESS` wherever possible.

Variables are localized for the `INCLUDE` directive in a part of the Template Toolkit called the *Stash*. It saves a copy of all the current variables in use before the template is processed, and then restores them to these original values when processing is complete. Understandably, this process takes a certain amount of time (not much in human terms, but still a finite amount), and the more variables you have, the longer it takes.

It is worth stressing that for most users of the Template Toolkit, these performance issues will be of no concern whatsoever. If you’re using the Template Toolkit to generate static web content offline, it makes little difference if a template takes a few hundredths or thousandths of a second longer to process. Even for generating dynamic content online, performance issues such as these probably aren’t going to

concern you unless you have particularly complicated templates or your site is heavily loaded and continually generating lots of dynamic content.

The more important issue is one of human efficiency. We would like to make it easier for template authors to keep track of the variables in use, make it harder for them to accidentally trample on them in a template component, and ideally, allow them to use `PROCESS` or `INCLUDE`, whichever is most appropriate to the task at hand.

The answer is to use a nested data structure to define all the sitewide variables under one global variable. Example 2-29 shows how numerous configuration variables can be defined as part of the site data structure, in this case implemented using a hash array.

*Example 2-29. lib/site*

```
[% site = {
  author = 'Arthur Dent'
  bgcolor = '#FF6600' # orange
  year = 2003
}

site.copyr = "Copyright $site.year $site.author"
%]
```

To interpolate the values for the year and author to generate the copyright string, we must now give them their full names, `site.year` and `site.author`. We need to set the `site.copyr` variable *after* the initial site data structure is defined so that we can use these variables. In effect, the `site` variable doesn't exist until the closing brace, so any references to it before that point will return empty values (unless the site has previously been set to contain these items at some earlier point).

```
[% site = {
  author = 'Arthur Dent'
  bgcolor = '#FF6600' # orange
  year = 2003

  # this doesn't work because site.year
  # and site.author are undefined at
  # this point
  copyr = "Copyright $site.year $site.author"
}
%]
```

Sitewide values can now be accessed through the `site` hash in all templates, leaving `author`, `bgcolor`, `year`, and all the other variables (except `site`, of course) free to be used, modified, and updated as “temporary” variables by page templates and template components. Now there's just one variable to keep track of, so there's much less chance of accidentally overwriting an important piece of data because you forgot it was there. It also means that the `INCLUDE` directive works faster because it has only one variable to localize instead of many. The Stash copies only the top-level variables in the process of localizing them and doesn't drill down through any of the nested data structures it finds.

## Layered Configuration Templates

As your site data structure becomes more complicated, you might find it easier to build it in layers using several templates. Example 2-30 shows a preprocessed configuration template that loads the *site*, *col*, and *url* templates using `PROCESS`.

*Example 2-30. lib/configs*

```
[% PROCESS site
    + col
    + url
-%]
```

We have already seen the *site* template in Example 2-29. Example 2-31 shows the *col* and *url* configuration templates.

*Example 2-31. lib/col*

```
[% site.rgb = {
    white = '#FFFFFF'
    black = '#000000'
    orange = '#FF6600'
}

site.col = {
    back = site.rgb.orange
    text = site.rgb.white
}
-%]
```

Example 2-31 shows the definition of a `site.rgb` hash and then another, `site.col`, which references values in the first. Template authors can use explicit colors, by referencing `site.rgb.orange`, for example, to fetch the correct RGB value, `#FF6600`. Or they can code their templates to use colors defined in the `site.col` structure—for example, referencing `site.col.back` in the *html* template to set the `bgcolor` attribute of the HTML `<body>` element. Either way, the colors are defined in one place, and the symbolic names allow us to see at a glance that the background color for the pages in the site is currently orange.

The *url* template is a little simpler, but also illustrates how variables can be built in stages (see Example 2-32).

*Example 2-32. lib/url*

```
[% url = 'http://tt2.org/ttbook'

site.url = {
    root = url
    home = "$url/index.html"
    help = "$url/help.html"
}
```

Example 2-32. *lib/url* (continued)

```
    images = "$url/images"
  }
-%]
```

The benefits of this approach are twofold. The first is that you can save yourself a great deal of typing by replacing a long-winded URL with a shorter variable name. The second benefit is that you can easily change all the URL values in a single stroke by changing the root `url` from which they are constructed.

One advantage of building a complex data structure from several templates is that you can easily replace one of the templates without affecting the others. For example, you might want to use a different set of URL values at some point. Rather than edit the `url` template, you can copy the contents to a new file (e.g., `url2`), make the changes there, and then update the `configs` template accordingly:

```
[% PROCESS site
    + col
    + url2
-%]
```

If you must revert to the old URLs at a later date, you need to change only the `configs` template to load `url` instead of `url2`. You can also use this approach to load different configuration templates based on a conditional expression. For example:

```
[% PROCESS site
    + col;

    IF developing;
      PROCESS url2;
    ELSE;
      PROCESS url;
    END
-%]
```

## Choosing Global Variables Wisely

Fewer global variables are better, but don't try to cram everything into the one site variable if more would do the job better. Try and separate your variables into structures according to their general purpose and relevance to different aspects of the site. For example, you can define one structure containing everything related to the site as a whole (e.g., `site`), and another related to the individual page being processed (e.g., `page`):

```
[% site = {
    title = "Arthur Dent's Web Site"
    author = 'Arthur Dent'
    # ...etc...
}
```

```

    page = {
        title = template.title
        author = template.author or site.author
    }
%]

```

You may also want to define others to represent a user, server, application, or request depending on how you're using the Template Toolkit and what you're using it for.

The Template Toolkit allows you to use upper- or lowercase, or some combination of the two, to specify variable names. It's not recommended that you use all uppercase variable names, as they might clash with current (or future) Template Toolkit directives. However, you might like to capitalize your global variables to help you remember that they're special in some way (e.g., Site versus site):

```

[% Site = {
    # ...etc...
}
Page = {
    # ...etc...
}
User = {
    # ...etc...
}
%]

```

## Passing Around Data Structures

You can pass a complex data structure around the Template Toolkit as easily as you would a scalar variable. Example 2-33 shows a configuration template that defines the `site.menu` data structure to contain the menu items that we used earlier in Example 2-26.

*Example 2-33. lib/menudef*

```

[% site.menu = [
    { text = 'Earth'
      link = 'earth.html' }
    { text = 'Magrethea'
      link = 'magrethea.html' }
]
%]

```

We've moved the definition of the sitewide menu into a central configuration file and will need to add it to the list of templates loaded by the `PROCESS` directive in the pre-processed *configs* template shown in Example 2-30:

```

[% PROCESS site
    + col
    + url
    + menudef
-%]

```

Now we can remove the definition of the menu structure from the component (or components) that generate the menu in a particular style, as shown in Example 2-34.

*Example 2-34. lib/menu5*

```
[% PROCESS mylib %]

<table border="0">
[%- FOREACH item IN menu;
    PROCESS menuitem
        text = item.text
        link = item.link;
END
-%]
</table>

[% BLOCK menuitem %]
<tr>
    <td>
        [% PROCESS icon %]
    </td>
    <td>
        [% PROCESS link %]
    </td>
</tr>
[% END %]
```

The value for menu (site.menu in this case) is passed to the *menu5* template as an argument in an INCLUDE directive:

```
[% INCLUDE menu5
    menu = site.menu
%]
```

The benefit of this approach is that the component that generates the menu is now generic, and will work with any menu data you care to define. Wherever you need a menu in the same style, simply call the component and pass in a different definition of menu data:

```
[% INCLUDE menu5
    menu = [
        { text = 'Milliways'
          link = 'milliways.html' }
        { text = 'Hotblack Desiato'
          link = 'desiato.html' }
    ]
%]
```

Separating the definition of a menu from its presentation also makes it easier to change the menu style at a later date. There's only one generic menu component to update or replace, regardless of how many times it is used in various places around the site. If you want two or more different menu styles, simply create additional menu components with different names or in different locations. For example, you

may have *site\_menu* and *page\_menu*, or *site/menu* and *page/menu*, or perhaps something such as *slick/graphical/menu* and *plain/text/menu*.

## Assessment

This brings us nicely back to where we started, looking at the basic principle of template processing: separating your *data* from the way it is *presented*. It's not always clear where your data belongs: in a configuration template; defined in a Perl script; or perhaps stored in a SQL database or XML file. Sometimes you'll want to begin by defining some simple variables in a configuration template so that you can start designing the layout and look and feel of the site. Later on, you might choose to define that data somewhere else, passing it in from a Perl script or making it available through a plugin.

The beauty of the Template Toolkit is that it really doesn't matter. It abstracts the details of the underlying implementation behind the uniform dotted notation for accessing data so that your templates keep working when your storage requirements change, as they inevitably will for many web sites.

It also makes it easy to include things such as loops, conditional statements, and other templates as easy as possible so that you can concentrate on presentation, rather than getting bogged down in the more precise details of full-blown programming language syntax. This is what we mean when we describe the Template Toolkit as a *presentation language* rather than a *programming language*.

It is an example of a *domain-specific language* that in many ways is similar to SQL, which is a domain-specific language for formulating database queries. As such, it should generally be used for what it is good at, rather than being contorted into doing something that might be a lot easier in another language. That doesn't mean that you can't use the Template Toolkit to do CGI programming, embed Perl, or even write Vagon poetry, if that's your thing, but that's not necessarily where its particular strengths lie.\*

And that's where Perl comes in. The Template Toolkit is designed to integrate with Perl code as cleanly and as easily as possible. When you want to do something more than the Template Toolkit provides, it is easy to append your own additions using a real programming language such as Perl. The plugin mechanism makes it easy to load external Perl code into templates so that you're not always writing Perl wrapper scripts just to add something of your own.

However, this total separation is not something that the Template Toolkit enforces, although the default settings for various configuration options such as `EVAL_PERL` do

---

\* Although the jury is still grooping hooptiously at the implorations of generating Vagon Poetry using the Template Toolkit.

tend to encourage it. Sometimes you just want to define a simple Perl subroutine in a template, for example, and don't want to bother with a separate Perl script or plugin module. The Template Toolkit gives you the freedom to do things such as this when you really want to.

For example, by enabling the `EVAL_PERL` option (see Chapter 4 and the Appendix for details), we can quickly define a Perl subroutine and bind it to a template variable, using a `PERL` block such as the following:

```
[% PERL %]
$stash->set( help => sub {
    my $entry = shift;
    return "$entry: mostly harmless";
} );
[% END %]
```

The `$stash->set( var => $value )` code, shown here binding the `help` variable to the Perl subroutine, is the Perl equivalent of writing `[% var = value %]` in a template—except, of course, that you can't usually define a subroutine directly in a template, only by using Perl code with `EVAL_PERL` set (which we think is a sensible restriction). This block can easily be defined in a preprocessed configuration template to keep it out of harm's way, leaving the template authors to use the simple variable:

```
[% help('Earth') %]
```

The important thing is to achieve an *appropriate* separation of concerns, rather than a *total* separation of concerns. Sometimes it's easier to define everything in one template or Perl program and to use a clear layout to separate the different parts. Splitting a small and self-contained document into several different pieces, each comprising just one part of the jigsaw puzzle, can make it hard to see the big picture. On the other hand, a more complex web site may have bigger pieces that absolutely need to be maintained in isolation from the other parts. Remember, there is no golden rule, so the Template Toolkit doesn't try and enforce one on you.

The techniques that we've taught you in this chapter will allow you to address most, if not all, of the simple but common problems that you'll typically face when building and maintaining a web site. We'll be coming back to the Web in Chapter 11 to look at some further ways in which the Template Toolkit can be used to enhance your site and make your life easier. In Chapter 12, we'll be showing how it can be used to handle the presentation layer to simplify the process of building and customizing web applications.