Programming, Administration, Performance Tips

Practical mod_perl

O'REILLY®

Stas Bekman & Eric Cholet

Practical mod_perl

Practical mod_perl

Stas Bekman and Eric Cholet

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 6

Coding with mod_perl in Mind

This is the most important chapter of this book. In this chapter, we cover all the nuances the programmer should know when porting an existing CGI script to work under mod_perl, or when writing one from scratch.

This chapter's main goal is to teach the reader how to think in mod_perl. It involves showing most of the mod_perl peculiarities and possible traps the programmer might fall into. It also shows you some of the things that are impossible with vanilla CGI but easily done with mod_perl.

Before You Start to Code

There are three important things you need to know before you start your journey in a mod_perl world: how to access mod_perl and related documentation, and how to develop your Perl code when the `strict` and `warnings` modes are enabled.

Accessing Documentation

mod_perl doesn't tolerate sloppy programming. Although we're confident that you're a talented, meticulously careful programmer whose programs run perfectly every time, you still might want to tighten up some of your Perl programming practices.

In this chapter, we include discussions that rely on prior knowledge of some areas of Perl, and we provide short refreshers where necessary. We assume that you can already program in Perl and that you are comfortable with finding Perl-related information in books and Perl documentation. There are many Perl books that you may find helpful. We list some of these in the reference sections at the end of each chapter.

If you prefer the documentation that comes with Perl, you can use either its online version (start at *http://www.perldoc.com/* or *http://theoryx5.uwinnipeg.ca/CPAN/perl/*) or the *perldoc* utility, which provides access to the documentation installed on your system.

To find out what Perl manpages are available, execute:

```
panic% perldoc perl
```

For example, to find what functions Perl has and to learn about their usage, execute:

```
panic% perldoc perlfunc
```

To learn the syntax and to find examples of a specific function, use the *-f* flag and the name of the function. For example, to learn more about `open( )`, execute:

```
panic% perldoc -f open
```

The *perldoc* supplied with Perl versions prior to 5.6.0 presents the information in POD (Plain Old Documentation) format. From 5.6.0 onwards, the documentation is shown in manpage format.

You may find the *perlfaq* manpages very useful, too. To find all the FAQs (Frequently Asked Questions) about a function, use the *-q* flag. For example, to search through the FAQs for the `open( )` function, execute:

```
panic% perldoc -q open
```

This will show you all the relevant *question* and *answer* sections.

Finally, to learn about *perldoc* itself, refer to the *perldoc* manpage:

```
panic% perldoc perldoc
```

The documentation available through *perldoc* provides good information and examples, and should be able to answer most Perl questions that arise.

Chapter 23 provides more information about mod_perl and related documentation.

The strict Pragma

We're sure you already do this, but it's absolutely essential to start all your scripts and modules with:

```
use strict;
```

It's especially important to have the `strict` pragma enabled under mod_perl. While it's not required by the language, its use cannot be too strongly recommended. It will save you a great deal of time. And, of course, clean scripts will still run under mod_cgi!

In the rare cases where it is necessary, you can turn off the `strict` pragma, or a part of it, inside a block. For example, if you want to use symbolic references (see the *perlref* manpage) inside a particular block, you can use `no strict 'refs';`, as follows:

```
use strict;
{
    no strict 'refs';
    my $var_ref = 'foo';
    $$var_ref = 1;
}
```

Starting the block with `no strict 'refs';` allows you to use symbolic references in the rest of the block. Outside this block, the use of symbolic references will trigger a runtime error.

Enabling Warnings

It's also important to develop your code with Perl reporting every possible relevant warning. Under mod_perl, you can turn this mode on globally, just like you would by using the *-w* command-line switch to Perl. Add this directive to *httpd.conf*:

```
PerlWarn On
```

In Perl 5.6.0 and later, you can also enable warnings only for the scope of a file, by adding:

```
use warnings;
```

at the top of your code. You can turn them off in the same way as `strict` for certain blocks. See the *warnings* manpage for more information.

We will talk extensively about warnings in many sections of the book. Perl code written for mod_perl should run without generating any warnings with both the `strict` and `warnings` pragmas in effect (that is, with `use strict` and `PerlWarn On` or `use warnings`).

Warnings are almost always caused by errors in your code, but on some occasions you may get warnings for totally legitimate code. That's part of why they're warnings and not errors. In the unlikely event that your code really does reveal a spurious warning, it is possible to switch off the warning.

Exposing Apache::Registry Secrets

Let's start with some simple code and see what can go wrong with it. This simple CGI script initializes a variable `$counter` to `0` and prints its value to the browser while incrementing it:

```
#!/usr/bin/perl -w
use strict;

print "Content-type: text/plain\n\n";

my $counter = 0;

for (1..5) {
    increment_counter( );
}

sub increment_counter {
    $counter++;
    print "Counter is equal to $counter !\n";
}
```

When issuing a request to */perl/counter.pl* or a similar script, we would expect to see the following output:

```
Counter is equal to 1 !
Counter is equal to 2 !
Counter is equal to 3 !
Counter is equal to 4 !
Counter is equal to 5 !
```

And in fact that's what we see when we execute this script for the first time. But let's reload it a few times.... After a few reloads, the counter suddenly stops counting from 1. As we continue to reload, we see that it keeps on growing, but not steadily, starting almost randomly at 10, 10, 10, 15, 20..., which makes no sense at all!

```
Counter is equal to 6 !
Counter is equal to 7 !
Counter is equal to 8 !
Counter is equal to 9 !
Counter is equal to 10 !
```

We saw two anomalies in this very simple script:

- Unexpected increment of our counter over 5
- Inconsistent growth over reloads

The reason for this strange behavior is that although `$counter` is incremented with each request, it is never reset to 0, even though we have this line:

```
my $counter = 0;
```

Doesn't this work under mod_perl?

The First Mystery: Why Does the Script Go Beyond 5?

If we look at the *error_log* file (we *did* enable warnings), we'll see something like this:

```
Variable "$counter" will not stay shared
at /home/httpd/perl/counter.pl line 13.
```

This warning is generated when a script contains a named (as opposed to an anonymous) nested subroutine that refers to a lexically scoped (with `my()`) variable defined outside this nested subroutine.

Do you see a nested named subroutine in our script? We don't! What's going on? Maybe it's a bug in Perl? But wait, maybe the Perl interpreter sees the script in a different way! Maybe the code goes through some changes before it actually gets executed? The easiest way to check what's actually happening is to run the script with a debugger.

Since we must debug the script when it's being executed by the web server, a normal debugger won't help, because the debugger has to be invoked from within the web server. Fortunately, we can use Doug MacEachern's `Apache::DB` module to debug our

script. While Apache::DB allows us to debug the code interactively (as we will show in Chapter 21), we will use it noninteractively in this example.

To enable the debugger, modify the *httpd.conf* file in the following way:

```
PerlSetEnv PERLDB_OPTS "NonStop=1 LineInfo=/tmp/db.out AutoTrace=1 frame=2"
PerlModule Apache::DB
<Location /perl>
    PerlFixupHandler Apache::DB
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
    PerlSendHeader On
</Location>
```

We have added a debugger configuration setting using the PERLDB_OPTS environment variable, which has the same effect as calling the debugger from the command line. We have also loaded and enabled Apache::DB as a PerlFixupHandler.

In addition, we'll load the Carp module, using <Perl> sections (this could also be done in the *startup.pl* file):

```
<Perl>
    use Carp;
</Perl>
```

After applying the changes, we restart the server and issue a request to */perl/counter.pl*, as before. On the surface, nothing has changed; we still see the same output as before. But two things have happened in the background:

- The file */tmp/db.out* was written, with a complete trace of the code that was executed.
- Since we have loaded the Carp module, the *error_log* file now contains the real code that was actually executed. This is produced as a side effect of reporting the "Variable "$counter" will not stay shared at..." warning that we saw earlier.

Here is the code that was actually executed:

```
package Apache::ROOT::perl::counter_2epl;
use Apache qw(exit);
sub handler {
    BEGIN {
        $^W = 1;
    };
    $^W = 1;

    use strict;

    print "Content-type: text/plain\n\n";

    my $counter = 0;
```

```
        for (1..5) {
            increment_counter( );
        }

        sub increment_counter {
            $counter++;
            print "Counter is equal to $counter !\n";
        }
    }
```

Note that the code in *error_log* wasn't indented—we've indented it to make it obvious that the code was wrapped inside the `handler( )` subroutine.

From looking at this code, we learn that every `Apache::Registry` script is cached under a package whose name is formed from the `Apache::ROOT::` prefix and the script's URI (*/perl/counter.pl*) by replacing all occurrences of `/` with `::` and `.` with `_2e`. That's how mod_perl knows which script should be fetched from the cache on each request—each script is transformed into a package with a unique name and with a single subroutine named `handler( )`, which includes all the code that was originally in the script.

Essentially, what's happened is that because `increment_counter( )` is a subroutine that refers to a lexical variable defined outside of its scope, it has become a *closure*. Closures don't normally trigger warnings, but in this case we have a nested subroutine. That means that the first time the enclosing subroutine `handler( )` is called, both subroutines are referring to the same variable, but after that, `increment_counter( )` will keep its own copy of `$counter` (which is why `$counter` is not *shared*) and increment its own copy. Because of this, the value of `$counter` keeps increasing and is never reset to 0.

If we were to use the `diagnostics` pragma in the script, which by default turns terse warnings into verbose warnings, we would see a reference to an inner (nested) subroutine in the text of the warning. By observing the code that gets executed, it is clear that `increment_counter( )` is a named nested subroutine since it gets defined inside the `handler( )` subroutine.

Any subroutine defined in the body of the script executed under `Apache::Registry` becomes a nested subroutine. If the code is placed into a library or a module that the script `require( )`s or `use( )`s, this effect doesn't occur.

For example, if we move the code from the script into the subroutine `run()`, place the subroutines in the *mylib.pl* file, save it in the same directory as the script itself, and `require( )` it, there will be no problem at all.* Examples 6-1 and 6-2 show how we spread the code across the two files.

Example 6-1. mylib.pl

```
my $counter;
sub run {
    $counter = 0;
```

* Don't forget the `1;` at the end of the library, or the `require( )` call might fail.

Example 6-1. mylib.pl (continued)

```
    for (1..5) {
        increment_counter( );
    }
}
sub increment_counter {
    $counter++;
    print "Counter is equal to $counter !\n";
}
1;
```

Example 6-2. counter.pl

```
use strict;
require "./mylib.pl";
print "Content-type: text/plain\n\n";
run( );
```

This solution is the easiest and fastest way to solve the nested subroutine problem. All you have to do is to move the code into a separate file, by first wrapping the initial code into some function that you later call from the script, and keeping the lexically scoped variables that could cause the problem out of this function.

As a general rule, it's best to put all the code in external libraries (unless the script is very short) and have only a few lines of code in the main script. Usually the main script simply calls the main function in the library, which is often called `init( )` or `run( )`. This way, you don't have to worry about the effects of named nested subroutines.

As we will show later in this chapter, however, this quick solution might be problematic on a different front. If you have many scripts, you might try to move more than one script's code into a file with a similar filename, like *mylib.pl*. A much cleaner solution would be to spend a little bit more time on the porting process and use a fully qualified package, as in Examples 6-3 and 6-4.

Example 6-3. Book/Counter.pm

```
package Book::Counter;

my $counter = 0;

sub run {
    $counter = 0;
    for (1..5) {
        increment_counter( );
    }
}

sub increment_counter {
    $counter++;
    print "Counter is equal to $counter !<BR>\n";
}
```

Example 6-3. Book/Counter.pm (continued)

```
1;
__END__
```

Example 6-4. counter-clean.pl

```
use strict;
use Book::Counter;

print "Content-type: text/plain\n\n";
Book::Counter::run();
```

As you can see, the only difference is in the package declaration. As long as the package name is unique, you won't encounter any collisions with other scripts running on the same server.

Another solution to this problem is to change the lexical variables to global variables. There are two ways global variables can be used:

- Using the `vars` pragma. With the `use strict 'vars'` setting, global variables can be used after being declared with `vars`. For example, this code:

  ```
  use strict;
  use vars qw($counter $result);
  # later in the code
  $counter = 0;
  $result  = 1;
  ```

  is similar to this code if `use strict` is not used:

  ```
  $counter = 0;
  $result  = 1;
  ```

  However, the former style of coding is much cleaner, because it allows you to use global variables by declaring them, while avoiding the problem of misspelled variables being treated as undeclared globals.

  The only drawback to using `vars` is that each global declared with it consumes more memory than the undeclared but fully qualified globals, as we will see in the next item.

- Using fully qualified variables. Instead of using `$counter`, we can use `$Foo::counter`, which will place the global variable `$counter` into the package `Foo`. Note that we don't know which package name `Apache::Registry` will assign to the script, since it depends on the location from which the script will be called. Remember that globals must always be initialized before they can be used.

Perl 5.6.x also introduces a third way, with the `our()` declaration. `our()` can be used in different scopes, similar to `my()`, but it creates global variables.

Finally, it's possible to avoid this problem altogether by always passing the variables as arguments to the functions (see Example 6-5).

Example 6-5. counter2.pl

```
#!/usr/bin/perl -w
use strict;

print "Content-type: text/plain\n\n";

my $counter = 0;

for (1..5) {
    $counter = increment_counter($counter);
}

sub increment_counter {
    my $counter = shift;

    $counter++;
    print "Counter is equal to $counter !\n";

    return $counter;
}
```

In this case, there is no variable-sharing problem. The drawback is that this approach adds the overhead of passing and returning the variable from the function. But on the other hand, it ensures that your code is doing the right thing and is not dependent on whether the functions are wrapped in other blocks, which is the case with the `Apache::Registry` handlers family.

When Stas (one of the authors of this book) had just started using mod_perl and wasn't aware of the nested subroutine problem, he happened to write a pretty complicated registration program that was run under mod_perl. We will reproduce here only the interesting part of that script:

```
use CGI;
$q = CGI->new;
my $name = $q->param('name');
print_response( );

sub print_response {
    print "Content-type: text/plain\n\n";
    print "Thank you, $name!";
}
```

Stas and his boss checked the program on the development server and it worked fine, so they decided to put it in production. Everything seemed to be normal, but the boss decided to keep on checking the program by submitting variations of his profile using *The Boss* as his username. Imagine his surprise when, after a few successful submissions, he saw the response "*Thank you, Stas!*" instead of "*Thank you, The Boss!*"

After investigating the problem, they learned that they had been hit by the nested subroutine problem. Why didn't they notice this when they were trying the software on their development server? We'll explain shortly.

To conclude this first mystery, remember to keep the `warnings` mode `On` on the development server and to watch the *error_log* file for warnings.

The Second Mystery—Inconsistent Growth over Reloads

Let's return to our original example and proceed with the second mystery we noticed. Why have we seen inconsistent results over numerous reloads?

What happens is that each time the parent process gets a request for the page, it hands the request over to a child process. Each child process runs its own copy of the script. This means that each child process has its own copy of `$counter`, which will increment independently of all the others. So not only does the value of each `$counter` increase independently with each invocation, but because different children handle the requests at different times, the increment seems to grow inconsistently. For example, if there are 10 *httpd* children, the first 10 reloads might be correct (if each request went to a different child). But once reloads start reinvoking the script from the child processes, strange results will appear.

Moreover, requests can appear at random since child processes don't always run the same requests. At any given moment, one of the children could have served the same script more times than any other, while another child may never have run it.

Stas and his boss didn't discover the aforementioned problem with the user registration system before going into production because the *error_log* file was too crowded with warnings continuously logged by multiple child processes.

To immediately recognize the problem visually (so you can see incorrect results), you need to run the server as a single process. You can do this by invoking the server with the *-X* option:

```
panic% httpd -X
```

Since there are no other servers (children) running, you will get the problem report on the second reload.

Enabling the `warnings` mode (as explained earlier in this chapter) and monitoring the *error_log* file will help you detect most of the possible errors. Some warnings can become errors, as we have just seen. You should check every reported warning and eliminate it, so it won't appear in *error_log* again. If your *error_log* file is filled up with hundreds of lines on every script invocation, you will have difficulty noticing and locating real problems, and on a production server you'll soon run out of disk space if your site is popular.

Namespace Issues

If your service consists of a single script, you will probably have no namespace problems. But web services usually are built from many scripts and handlers. In the

following sections, we will investigate possible namespace problems and their solutions. But first we will refresh our understanding of two special Perl variables, @INC and %INC.

The @INC Array

Perl's @INC array is like the PATH environment variable for the shell program. Whereas PATH contains a list of directories to search for executable programs, @INC contains a list of directories from which Perl modules and libraries can be loaded.

When you use( ), require( ), or do( ) a filename or a module, Perl gets a list of directories from the @INC variable and searches them for the file it was requested to load. If the file that you want to load is not located in one of the listed directories, you must tell Perl where to find the file. You can either provide a path relative to one of the directories in @INC or provide the absolute path to the file.

The %INC Hash

Perl's %INC hash is used to cache the names of the files and modules that were loaded and compiled by use( ), require( ), or do( ) statements. Every time a file or module is successfully loaded, a new key-value pair is added to %INC. The key is the name of the file or module as it was passed to one of the three functions we have just mentioned. If the file or module was found in any of the @INC directories (except "."), the filenames include the full path. Each Perl interpreter, and hence each process under mod_perl, has its own private %INC hash, which is used to store information about its compiled modules.

Before attempting to load a file or a module with use( ) or require( ), Perl checks whether it's already in the %INC hash. If it's there, the loading and compiling are not performed. Otherwise, the file is loaded into memory and an attempt is made to compile it. Note that do( ) loads the file or module unconditionally—it does not check the %INC hash. We'll look at how this works in practice in the following examples.

First, let's examine the contents of @INC on our system:

```
panic% perl -le 'print join "\n", @INC'
/usr/lib/perl5/5.6.1/i386-linux
/usr/lib/perl5/5.6.1
/usr/lib/perl5/site_perl/5.6.1/i386-linux
/usr/lib/perl5/site_perl/5.6.1
/usr/lib/perl5/site_perl
.
```

Notice . (the current directory) as the last directory in the list.

Let's load the module strict.pm and see the contents of %INC:

```
panic% perl -le 'use strict; print map {"$_ => $INC{$_}"} keys %INC'
strict.pm => /usr/lib/perl5/5.6.1/strict.pm
```

Since `strict.pm` was found in the */usr/lib/perl5/5.6.1/* directory and */usr/lib/perl5/5.6.1/* is a part of `@INC`, `%INC` includes the full path as the value for the key `strict.pm`.

Let's create the simplest possible module in */tmp/test.pm*:

```
1;
```

This does absolutely nothing, but it returns a true value when loaded, which is enough to satisfy Perl that it loaded correctly. Let's load it in different ways:

```
panic% cd /tmp
panic% perl -e 'use test; \
       print map { "$_ => $INC{$_}\n" } keys %INC'
test.pm => test.pm
```

Since the file was found in . (the directory the code was executed from), the relative path is used as the value. Now let's alter `@INC` by appending */tmp*:

```
panic% cd /tmp
panic% perl -e 'BEGIN { push @INC, "/tmp" } use test; \
       print map { "$_ => $INC{$_}\n" } keys %INC'
test.pm => test.pm
```

Here we still get the relative path, since the module was found first relative to ".". The directory */tmp* was placed after . in the list. If we execute the same code from a different directory, the "." directory won't match:

```
panic% cd /
panic% perl -e 'BEGIN { push @INC, "/tmp" } use test; \
       print map { "$_ => $INC{$_}\n" } keys %INC'
test.pm => /tmp/test.pm
```

so we get the full path. We can also prepend the path with `unshift( )`, so that it will be used for matching before ".". We will get the full path here as well:

```
panic% cd /tmp
panic% perl -e 'BEGIN { unshift @INC, "/tmp" } use test; \
       print map { "$_ => $INC{$_}\n" } keys %INC'
test.pm => /tmp/test.pm
```

The code:

```
BEGIN { unshift @INC, "/tmp" }
```

can be replaced with the more elegant:

```
use lib "/tmp";
```

This is almost equivalent to our `BEGIN` block and is the recommended approach.

These approaches to modifying `@INC` can be labor intensive: moving the script around in the filesystem might require modifying the path.

Name Collisions with Modules and Libraries

In this section, we'll look at two scenarios with failures related to namespaces. For the following discussion, we will always look at a single child process.

A first faulty scenario

It is impossible to use two modules with identical names on the same server. Only the first one found in a `use()` or a `require()` statement will be loaded and compiled. All subsequent requests to load a module with the same name will be skipped, because Perl will find that there is already an entry for the requested module in the `%INC` hash.

Let's examine a scenario in which two independent projects in separate directories, *projectA* and *projectB*, both need to run on the same server. Both projects use a module with the name `MyConfig.pm`, but each project has completely different code in its `MyConfig.pm` module. This is how the projects reside on the filesystem (all located under the directory */home/httpd/perl*):

```
projectA/MyConfig.pm
projectA/run.pl
projectB/MyConfig.pm
projectB/run.pl
```

Examples 6-6, 6-7, 6-8, and 6-9 show some sample code.

Example 6-6. projectA/run.pl

```
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Inside project: ", project_name();
```

Example 6-7. projectA/MyConfig.pm

```
sub project_name { return 'A'; }
1;
```

Example 6-8. projectB/run.pl

```
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Inside project: ", project_name();
```

Example 6-9. projectB/MyConfig.pm

```
sub project_name { return 'B'; }
1;
```

Both projects contain a script, *run.pl*, which loads the module `MyConfig.pm` and prints an *indentification* message based on the `project_name()` function in the `MyConfig.pm` module. When a request to */perl/projectA/run.pl* is issued, it is supposed to print:

```
Inside project: A
```

Similarly, */perl/projectB/run.pl* is expected to respond with:

```
Inside project: B
```

When tested using single-server mode, only the first one to run will load the MyConfig.pm module, although both *run.pl* scripts call use MyConfig. When the second script is run, Perl will skip the use MyConfig; statement, because MyConfig.pm is already located in %INC. Perl reports this problem in the *error_log*:

```
Undefined subroutine
&Apache::ROOT::perl::projectB::run_2epl::project_name called at
/home/httpd/perl/projectB/run.pl line 4.
```

This is because the modules didn't declare a package name, so the project_name( ) subroutine was inserted into *projectA/run.pl*'s namespace, Apache::ROOT::perl::projectB::run_2epl. Project B doesn't get to load the module, so it doesn't get the subroutine either!

Note that if a library were used instead of a module (for example, config.pl instead of MyConfig.pm), the behavior would be the same. For both libraries and modules, a file is loaded and its filename is inserted into %INC.

A second faulty scenario

Now consider the following scenario:

```
project/MyConfig.pm
project/runA.pl
project/runB.pl
```

Now there is a single project with two scripts, *runA.pl* and *runB.pl*, both trying to load the same module, MyConfig.pm, as shown in Examples 6-10, 6-11, and 6-12.

Example 6-10. project/MyConfig.pm

```
sub project_name { return 'Super Project'; }
1;
```

Example 6-11. project/runA.pl

```
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Script A\n";
print "Inside project: ", project_name( );
```

Example 6-12. project/runB.pl

```
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Script B\n";
print "Inside project: ", project_name( );
```

This scenario suffers from the same problem as the previous two-project scenario: only the first script to run will work correctly, and the second will fail. The problem occurs because there is no package declaration here.

We'll now explore some of the ways we can solve these problems.

A quick but ineffective hackish solution

The following solution should be used only as a short term bandage. You can force reloading of the modules either by fiddling with `%INC` or by replacing `use()` and `require()` calls with `do()`.

If you delete the module entry from the `%INC` hash before calling `require()` or `use()`, the module will be loaded and compiled again. See Example 6-13.

Example 6-13. project/runA.pl

```
BEGIN {
    delete $INC{"MyConfig.pm"};
}
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Script A\n";
print "Inside project: ", project_name();
```

Apply the same fix to *runB.pl*.

Another alternative is to force module reload via `do()`, as seen in Example 6-14.

Example 6-14. project/runA.pl forcing module reload by using do() instead of use()

```
use lib qw(.);
do "MyConfig.pm";
print "Content-type: text/plain\n\n";
print "Script B\n";
print "Inside project: ", project_name();
```

Apply the same fix to *runB.pl*.

If you needed to `import()` something from the loaded module, call the `import()` method explicitly. For example, if you had:

```
use MyConfig qw(foo bar);
```

now the code will look like:

```
do "MyConfig.pm";
MyConfig->import(qw(foo bar));
```

Both presented solutions are ultimately ineffective, since the modules in question will be reloaded on each request, slowing down the response times. Therefore, use these only when a very quick fix is needed, and make sure to replace the hack with one of the more robust solutions discussed in the following sections.

A first solution

The first faulty scenario can be solved by placing library modules in a subdirectory structure so that they have different path prefixes. The new filesystem layout will be:

```
projectA/ProjectA/MyConfig.pm
projectA/run.pl
projectB/ProjectB/MyConfig.pm
projectB/run.pl
```

The *run.pl* scripts will need to be modified accordingly:

```
use ProjectA::MyConfig;
```

and:

```
use ProjectB::MyConfig;
```

However, if later on we want to add a new script to either of these projects, we will hit the problem described by the second problematic scenario, so this is only half a solution.

A second solution

Another approach is to use a full path to the script, so the latter will be used as a key in `%INC`:

```
require "/home/httpd/perl/project/MyConfig.pm";
```

With this solution, we solve both problems but lose some portability. Every time a project moves in the filesystem, the path must be adjusted. This makes it impossible to use such code under version control in multiple-developer environments, since each developer might want to place the code in a different absolute directory.

A third solution

This solution makes use of package-name declaration in the `require()`d modules. For example:

```
package ProjectA::Config;
```

Similarly, for *ProjectB*, the package name would be `ProjectB::Config`.

Each package name should be unique in relation to the other packages used on the same *httpd* server. `%INC` will then use the unique package name for the key instead of the filename of the module. It's a good idea to use at least two-part package names for your private modules (e.g., `MyProject::Carp` instead of just `Carp`), since the latter will collide with an existing standard package. Even though a package with the same name may not exist in the standard distribution now, in a later distribution one may come along that collides with a name you've chosen.

What are the implications of package declarations? Without package declarations in the modules, it is very convenient to `use()` and `require()`, since all variables and subroutines from the loaded modules will reside in the same package as the script

itself. Any of them can be used as if it was defined in the same scope as the script itself. The downside of this approach is that a variable in a module might conflict with a variable in the main script; this can lead to hard-to-find bugs.

With package declarations in the modules, things are a bit more complicated. Given that the package name is `PackageA`, the syntax `PackageA::project_name()` should be used to call a subroutine `project_name()` from the code using this package. Before the package declaration was added, we could just call `project_name()`. Similarly, a global variable `$foo` must now be referred to as `$PackageA::foo`, rather than simply as `$foo`. Lexically defined variables (declared with `my()`) inside the file containing `PackageA` will be inaccessible from outside the package.

You can still use the unqualified names of global variables and subroutines if these are imported into the namespace of the code that needs them. For example:

```
use MyPackage qw(:mysubs sub_b $var1 :myvars);
```

Modules can export any global symbols, but usually only subroutines and global variables are exported. Note that this method has the disadvantage of consuming more memory. See the `perldoc Exporter` manpage for information about exporting other variables and symbols.

Let's rewrite the second scenario in a truly clean way. This is how the files reside on the filesystem, relative to the directory */home/httpd/perl*:

```
project/MyProject/Config.pm
project/runA.pl
project/runB.pl
```

Examples 6-15, 6-16, and 6-17 show how the code will look.

Example 6-15. project/MyProject/Config.pm

```
package MyProject::Config
sub project_name { return 'Super Project'; }
1;
```

Example 6-16. project/runB.pl

```
use lib qw(.);
use MyProject::Config;
print "Content-type: text/plain\n\n";
print "Script B\n";
print "Inside project: ", MyProject::Config::project_name();
```

Example 6-17. project/runA.pl

```
use lib qw(.);
use MyProject::Config;
print "Content-type: text/plain\n\n";
print "Script A\n";
print "Inside project: ", MyProject::Config::project_name();
```

As you can see, we have created the *MyProject/Config.pm* file and added a package declaration at the top of it:

```
package MyProject::Config
```

Now both scripts load this module and access the module's subroutine, `project_name( )`, with a fully qualified name, `MyProject::Config::project_name( )`.

See also the *perlmodlib* and *perlmod* manpages.

From the above discussion, it also should be clear that you cannot run development and production versions of the tools using the same Apache server. You have to run a dedicated server for each environment. If you need to run more than one development environment on the same server, you can use `Apache::PerlVINC`, as explained in Appendix B.

Perl Specifics in the mod_perl Environment

In the following sections, we discuss the specifics of Perl's behavior under mod_perl.

exit( )

Perl's core `exit( )` function shouldn't be used in mod_perl code. Calling it causes the mod_perl process to exit, which defeats the purpose of using mod_perl. The `Apache::exit( )` function should be used instead. Starting with Perl Version 5.6.0, mod_perl overrides `exit( )` behind the scenes using `CORE::GLOBAL::`, a new *magical* package.

The CORE:: Package

`CORE::` is a special package that provides access to Perl's built-in functions. You may need to use this package to override some of the built-in functions. For example, if you want to override the `exit( )` built-in function, you can do so with:

```
use subs qw(exit);
exit( ) if $DEBUG;
sub exit { warn "exit( ) was called"; }
```

Now when you call `exit( )` in the same scope in which it was overridden, the program won't exit, but instead will just print a warning "exit( ) was called". If you want to use the original built-in function, you can still do so with:

```
# the 'real' exit
CORE::exit( );
```

`Apache::Registry` and `Apache::PerlRun` override `exit( )` with `Apache::exit( )` behind the scenes; therefore, scripts running under these modules don't need to be modified to use `Apache::exit( )`.

If `CORE::exit()` is used in scripts running under mod_perl, the child will exit, but the current request won't be logged. More importantly, a proper exit won't be performed. For example, if there are some database handles, they will remain open, causing costly memory and (even worse) database connection leaks.

If the child process needs to be killed, `Apache::exit(Apache::Constants::DONE)` should be used instead. This will cause the server to exit gracefully, completing the logging functions and protocol requirements.

If the child process needs to be killed cleanly after the request has completed, use the `$r->child_terminate` method. This method can be called anywhere in the code, not just at the end. This method sets the value of the `MaxRequestsPerChild` configuration directive to `1` and clears the `keepalive` flag. After the request is serviced, the current connection is broken because of the `keepalive` flag, which is set to false, and the parent tells the child to cleanly quit because `MaxRequestsPerChild` is smaller than or equal to the number of requests served.

In an `Apache::Registry` script you would write:

```
Apache->request->child_terminate;
```

and in *httpd.conf*:

```
PerlFixupHandler "sub { shift->child_terminate }"
```

You would want to use the latter example only if you wanted the child to terminate every time the registered handler was called. This is probably not what you want.

You can also use a post-processing handler to trigger child termination. You might do this if you wanted to execute your own cleanup code before the process exits:

```
my $r = shift;
$r->post_connection(\&exit_child);

sub exit_child {
    # some logic here if needed
    $r->child_terminate;
}
```

This is the code that is used by the `Apache::SizeLimit` module, which terminates processes that grow bigger than a preset quota.

die( )

`die()` is usually used to abort the flow of the program if something goes wrong. For example, this common idiom is used when opening files:

```
open FILE, "foo" or die "Cannot open 'foo' for reading: $!";
```

If the file cannot be opened, the script will `die()`: script execution is aborted, the reason for death is printed, and the Perl interpreter is terminated.

You will hardly find any properly written Perl scripts that don't have at least one `die()` statement in them.

CGI scripts running under mod_cgi exit on completion, and the Perl interpreter exits as well. Therefore, it doesn't matter whether the interpreter exits because the script died by natural death (when the last statement in the code flow was executed) or was aborted by a `die()` statement.

Under mod_perl, we don't want the process to quit. Therefore, mod_perl takes care of it behind the scenes, and `die()` calls don't abort the process. When `die()` is called, mod_perl logs the error message and calls `Apache::exit()` instead of `CORE::die()`. Thus, the script stops, but the process doesn't quit. Of course, we are talking about the cases where the code calling `die()` is not wrapped inside an exception handler (e.g., an `eval {}` block) that traps `die()` calls, or the `$SIG{__DIE__}` sighandler, which allows you to override the behavior of `die()` (see Chapter 21). The reference section at the end of this chapter mentions a few exception-handling modules available from CPAN.

Global Variable Persistence

Under mod_perl a child process doesn't exit after serving a single request. Thus, global variables persist inside the same process from request to request. This means that you should be careful not to rely on the value of a global variable if it isn't initialized at the beginning of each request. For example:

```
# the very beginning of the script
use strict;
use vars qw($counter);
$counter++;
```

relies on the fact that Perl interprets an undefined value of `$counter` as a zero value, because of the increment operator, and therefore sets the value to `1`. However, when the same code is executed a second time in the same process, the value of `$counter` is not undefined any more; instead, it holds the value it had at the end of the previous execution in the same process. Therefore, a cleaner way to code this snippet would be:

```
use strict;
use vars qw($counter);
$counter = 0;
$counter++;
```

In practice, you should avoid using global variables unless there really is no alternative. Most of the problems with global variables arise from the fact that they keep their values across functions, and it's easy to lose track of which function modifies the variable and where. This problem is solved by localizing these variables with `local()`. But if you are already doing this, using lexical scoping (with `my()`) is even better because its scope is clearly defined, whereas localized variables are seen and

can be modified from anywhere in the code. Refer to the *perlsub* manpage for more details. Our example will now be written as:

```
use strict;
my $counter = 0;
$counter++;
```

Note that it is a good practice to both declare and initialize variables, since doing so will clearly convey your intention to the code's maintainer.

You should be especially careful with Perl special variables, which cannot be lexically scoped. With special variables, `local()` must be used. For example, if you want to read in a whole file at once, you need to `undef()` the input record separator. The following code reads the contents of an entire file in one go:

```
open IN, $file or die $!;
$/ = undef;
$content = <IN>; # slurp the whole file in
close IN;
```

Since you have modified the special Perl variable `$/` globally, it'll affect any other code running under the same process. If somewhere in the code (or any other code running on the same server) there is a snippet reading a file's content line by line, relying on the default value of `$/` (`\n`), this code will work incorrectly. Localizing the modification of this special variable solves this potential problem:

```
{
  local $/; # $/ is undef now
  $content = <IN>; # slurp the whole file in
}
```

Note that the localization is enclosed in a block. When control passes out of the block, the previous value of `$/` will be restored automatically.

STDIN, STDOUT, and STDERR Streams

Under mod_perl, both `STDIN` and `STDOUT` are tied to the socket from which the request originated. If, for example, you use a third-party module that prints some output to `STDOUT` when it shouldn't (for example, control messages) and you want to avoid this, you must temporarily redirect `STDOUT` to */dev/null*. You will then have to restore `STDOUT` to the original handle when you want to send a response to the client. The following code demonstrates a possible implementation of this workaround:

```
{
    my $nullfh = Apache::gensym( );
    open $nullfh, '>/dev/null' or die "Can't open /dev/null: $!";
    local *STDOUT = $nullfh;
    call_something_thats_way_too_verbose( );
    close $nullfh;
}
```

The code defines a block in which the STDOUT stream is localized to print to */dev/null*. When control passes out of this block, STDOUT gets restored to the previous value.

STDERR is tied to a file defined by the ErrorLog directive. When native *syslog* support is enabled, the STDERR stream will be redirected to */dev/null*.

Redirecting STDOUT into a Scalar Variable

Sometimes you encounter a black-box function that prints its output to the default file handle (usually STDOUT) when you would rather put the output into a scalar. This is very relevant under mod_perl, where STDOUT is tied to the Apache request object. In this situation, the IO::String package is especially useful. You can re-tie() STDOUT (or any other file handle) to a string by doing a simple select() on the IO::String object. Call select() again at the end on the original file handle to re-tie() STDOUT back to its original stream:

```
my $str;
my $str_fh = IO::String->new($str);

my $old_fh = select($str_fh);
black_box_print();
select($old_fh) if defined $old_fh;
```

In this example, a new IO::String object is created. The object is then selected, the black_box_print() function is called, and its output goes into the string object. Finally, we restore the original file handle, by re-select()ing the originally selected file handle. The $str variable contains all the output produced by the black_box_print() function.

print()

Under mod_perl, CORE::print() (using either STDOUT as a filehandle argument or no filehandle at all) will redirect output to Apache::print(), since the STDOUT file handle is tied to Apache. That is, these two are functionally equivalent:

```
print "Hello";
$r->print("Hello");
```

Apache::print() will return immediately without printing anything if $r->connection->aborted returns true. This happens if the connection has been aborted by the client (e.g., by pressing the Stop button).

There is also an optimization built into Apache::print(): if any of the arguments to this function are scalar references to strings, they are automatically dereferenced. This avoids needless copying of large strings when passing them to subroutines. For example, the following code will print the actual value of $long_string:

```
my $long_string = "A" x 10000000;
$r->print(\$long_string);
```

To print the reference value itself, use a double reference:

```
$r->print(\\$long_string);
```

When `Apache::print()` sees that the passed value is a reference, it dereferences it once and prints the real reference value:

```
SCALAR(0x8576e0c)
```

Formats

The interface to file handles that are linked to variables with Perl's `tie()` function is not yet complete. The `format()` and `write()` functions are missing. If you configure Perl with `sfio`, `write()` and `format()` should work just fine.

Instead of `format()`, you can use `printf()`. For example, the following formats are equivalent:

```
format   printf
---------------
##.##    %2.2f
####.##  %4.2f
```

To print a string with fixed-length elements, use the `printf()` format `%n.ms` where *n* is the length of the field allocated for the string and *m* is the maximum number of characters to take from the string. For example:

```
printf "[%5.3s][%10.10s][%30.30s]\n",
       12345, "John Doe", "1234 Abbey Road"
```

prints:

```
[  123][  John Doe][                1234 Abbey Road]
```

Notice that the first string was allocated five characters in the output, but only three were used because *m*=5 and *n*=3 (`%5.3s`). If you want to ensure that the text will always be correctly aligned without being truncated, *n* should always be greater than or equal to *m*.

You can change the alignment to the left by adding a minus sign (`-`) after the `%`. For example:

```
printf "[%-5.5s][%-10.10s][%-30.30s]\n",
       123, "John Doe", "1234 Abbey Road"
```

prints:

```
[123  ][John Doe  ][1234 Abbey Road                ]
```

You can also use a plus sign (`+`) for the right-side alignment. For example:

```
printf "[%+5s][%+10s][%+30s]\n",
       123, "John Doe", "1234 Abbey Road"
```

prints:

```
[  123][  John Doe][                1234 Abbey Road]
```

Another alternative to `format()` and `printf()` is to use the `Text::Reform` module from CPAN.

In the examples above we've printed the number *123* as a string (because we used the `%s` format specifier), but numbers can also be printed using numeric formats. See *perldoc -f sprintf* for full details.

Output from System Calls

The output of `system()`, `exec()`, and `open(PIPE,"|program")` calls will not be sent to the browser unless Perl was configured with `sfio`. To learn if your version of Perl is `sfio`-enabled, look at the output of the *perl -V* command for the *useperlio* and *d_sfio* strings.

You can use backticks as a possible workaround:

```
print `command here`;
```

But this technique has very poor performance, since it forks a new process. See the discussion about forking in Chapter 10.

BEGIN blocks

Perl executes `BEGIN` blocks as soon as possible, when it's compiling the code. The same is true under mod_perl. However, since mod_perl normally compiles scripts and modules only once, either in the parent process or just once per child, `BEGIN` blocks are run only once. As the *perlmod* manpage explains, once a `BEGIN` block has run, it is immediately undefined. In the mod_perl environment, this means that `BEGIN` blocks will not be run during the response to an incoming request unless that request happens to be the one that causes the compilation of the code. However, there are cases when `BEGIN` blocks will be rerun for each request.

`BEGIN` blocks in modules and files pulled in via `require()` or `use()` will be executed:

- Only once, if pulled in by the parent process.
- Once per child process, if not pulled in by the parent process.
- One additional time per child process, if the module is reloaded from disk by `Apache::StatINC`.
- One additional time in the parent process on each restart, if `PerlFreshRestart` is `On`.
- On every request, if the module with the `BEGIN` block is deleted from `%INC`, before the module's compilation is needed. The same thing happens when `do()` is used, which loads the module even if it's already loaded.

BEGIN blocks in Apache::Registry scripts will be executed:

- Only once, if pulled in by the parent process via Apache::RegistryLoader.
- Once per child process, if not pulled in by the parent process.
- One additional time per child process, each time the script file changes on disk.
- One additional time in the parent process on each restart, if pulled in by the parent process via Apache::RegistryLoader and PerlFreshRestart is On.

Note that this second list is applicable only to the scripts themselves. For the modules used by the scripts, the previous list applies.

END Blocks

As the *perlmod* manpage explains, an END subroutine is executed when the Perl interpreter exits. In the mod_perl environment, the Perl interpreter exits only when the child process exits. Usually a single process serves many requests before it exits, so END blocks cannot be used if they are expected to do something at the end of each request's processing.

If there is a need to run some code after a request has been processed, the $r->register_cleanup( ) function should be used. This function accepts a reference to a function to be called during the PerlCleanupHandler phase, which behaves just like the END block in the normal Perl environment. For example:

```
$r->register_cleanup(sub { warn "$$ does cleanup\n" });
```

or:

```
sub cleanup { warn "$$ does cleanup\n" };
$r->register_cleanup(\&cleanup);
```

will run the registered code at the end of each request, similar to END blocks under mod_cgi.

As you already know by now, Apache::Registry handles things differently. It does execute all END blocks encountered during compilation of Apache::Registry scripts at the end of each request, like mod_cgi does. That includes any END blocks defined in the packages use( )d by the scripts.

If you want something to run only once in the parent process on shutdown and restart, you can use register_cleanup( ) in *startup.pl*:

```
warn "parent pid is $$\n";
Apache->server->register_cleanup(
    sub { warn "server cleanup in $$\n" });
```

This is useful when some server-wide cleanup should be performed when the server is stopped or restarted.

CHECK and INIT Blocks

The CHECK and INIT blocks run when compilation is complete, but before the program starts. CHECK can mean "checkpoint," "double-check," or even just "stop." INIT stands for "initialization." The difference is subtle: CHECK blocks are run just after the compilation ends, whereas INIT blocks are run just before the runtime begins (hence, the *-c* command-line flag to Perl runs up to CHECK blocks but not INIT blocks).

Perl calls these blocks only during perl_parse( ), which mod_perl calls once at startup time. Therefore, CHECK and INIT blocks don't work in mod_perl, for the same reason these don't:

```
panic% perl -e 'eval qq(CHECK { print "ok\n" })'
panic% perl -e 'eval qq(INIT  { print "ok\n" })'
```

$^T and time( )

Under mod_perl, processes don't quit after serving a single request. Thus, $^T gets initialized to the server startup time and retains this value throughout the process's life. Even if you don't use this variable directly, it's important to know that Perl refers to the value of $^T internally.

For example, Perl uses $^T with the -M, -C, or -A file test operators. As a result, files created after the child server's startup are reported as having a negative age when using those operators. -M returns the age of the script file relative to the value of the $^T special variable.

If you want to have -M report the file's age relative to the current request, reset $^T, just as in any other Perl script. Add the following line at the beginning of your scripts:

```
local $^T = time;
```

You can also do:

```
local $^T = $r->request_time;
```

The second technique is better performance-wise, as it skips the time( ) system call and uses the timestamp of the request's start time, available via the $r->request_time method.

If this correction needs to be applied to a lot of handlers, a more scalable solution is to specify a fixup handler, which will be executed during the fixup stage:

```
sub Apache::PerlBaseTime::handler {
    $^T = shift->request_time;
    return Apache::Constants::DECLINED;
}
```

and then add the following line to *httpd.conf*:

```
PerlFixupHandler Apache::PerlBaseTime
```

Now no modifications to the content-handler code and scripts need to be performed.

Command-Line Switches

When a Perl script is run from the command line, the shell invokes the Perl interpreter via the `#!/bin/perl` directive, which is the first line of the script (sometimes referred to as the *shebang line*). In scripts running under mod_cgi, you may use Perl switches as described in the *perlrun* manpage, such as *-w*, *-T*, or *-d*. Under the `Apache::Registry` handlers family, all switches except *-w* are ignored (and use of the *-T* switch triggers a warning). The support for *-w* was added for backward compatibility with mod_cgi.

Most command-line switches have special Perl variable equivalents that allow them to be set/unset in code. Consult the *perlvar* manpage for more details.

mod_perl provides its own equivalents to *-w* and *-T* in the form of configuration directives, as we'll discuss presently.

Finally, if you still need to set additional Perl startup flags, such as *-d* and *-D*, you can use the `PERL5OPT` environment variable. Switches in this variable are treated as if they were on every Perl command line. According to the *perlrun* manpage, only the *-[DIMUdmw]* switches are allowed.

Warnings

There are three ways to enable warnings:

Globally to all processes
: In *httpd.conf*, set:

```
PerlWarn On
```

You can then fine-tune your code, turning warnings off and on by setting the `$^W` variable in your scripts.

Locally to a script
: Including the following line:

```
#!/usr/bin/perl -w
```

will turn warnings on for the scope of the script. You can turn them off and on in the script by setting the `$^W` variable, as noted above.

Locally to a block
: This code turns warnings on for the scope of the block:

```
{
    local $^W = 1;
    # some code
}
# $^W assumes its previous value here
```

This turns warnings off:

```
{
    local $^W = 0;
    # some code
}
# $^W assumes its previous value here
```

If `$^W` isn't properly localized, this code will affect the current request and all subsequent requests processed by this child. Thus:

```
$^W = 0;
```

will turn the warnings off, no matter what.

If you want to turn warnings on for the scope of the whole file, as in the previous item, you can do this by adding:

```
local $^W = 1;
```

at the beginning of the file. Since a file is effectively a block, file scope behaves like a block's curly braces ({ }), and `local $^W` at the start of the file will be effective for the whole file.

While having warnings mode turned on is essential for a development server, you should turn it globally off on a production server. Having warnings enabled introduces a non-negligible performance penalty. Also, if every request served generates one warning, and your server processes millions of requests per day, the *error_log* file will eat up all your disk space and the system won't be able to function normally anymore.

Perl 5.6.x introduced the `warnings` pragma, which allows very flexible control over warnings. This pragma allows you to enable and disable groups of warnings. For example, to enable only the syntax warnings, you can use:

```
use warnings 'syntax';
```

Later in the code, if you want to disable syntax warnings and enable signal-related warnings, you can use:

```
no  warnings 'syntax';
use warnings 'signal';
```

But usually you just want to use:

```
use warnings;
```

which is the equivalent of:

```
use warnings 'all';
```

If you want your code to be really clean and consider all warnings as errors, Perl will help you to do that. With the following code, any warning in the lexical scope of the definition will trigger a fatal error:

```
use warnings FATAL => 'all';
```

Of course, you can fine-tune the groups of warnings and make only certain groups of warnings fatal. For example, to make only closure problems fatal, you can use:

```
use warnings FATAL => 'closure';
```

Using the warnings pragma, you can also disable warnings locally:

```
{
  no warnings;
  # some code that would normally emit warnings
}
```

In this way, you can avoid some warnings that you are aware of but can't do anything about.

For more information about the warnings pragma, refer to the *perllexwarn* manpage.

Taint mode

Perl's *-T* switch enables *taint mode*. In taint mode, Perl performs some checks on how your program is using the data passed to it. For example, taint checks prevent your program from passing some external data to a system call without this data being explicitly checked for nastiness, thus avoiding a fairly large number of common security holes. If you don't force all your scripts and handlers to run under taint mode, it's more likely that you'll leave some holes to be exploited by malicious users. (See Chapter 23 and the *perlsec* manpage for more information. Also read the re pragma's manpage.)

Since the *-T* switch can't be turned on from within Perl (this is because when Perl is running, it's already too late to mark *all* external data as tainted), mod_perl provides the PerlTaintCheck directive to turn on taint checks globally. Enable this mode with:

```
PerlTaintCheck On
```

anywhere in *httpd.conf* (though it's better to place it as early as possible for clarity).

For more information on taint checks and how to untaint data, refer to the *perlsec* manpage.

Compiled Regular Expressions

When using a regular expression containing an interpolated Perl variable that you are confident will not change during the execution of the program, a standard speed-optimization technique is to add the /o modifier to the regex pattern. This compiles the regular expression once, for the entire lifetime of the script, rather than every time the pattern is executed. Consider:

```
my $pattern = '^\d+$'; # likely to be input from an HTML form field
foreach (@list) {
    print if /$pattern/o;
}
```

This is usually a big win in loops over lists, or when using the grep() or map() operators.

In long-lived mod_perl scripts and handlers, however, the variable may change with each invocation. In that case, this memorization can pose a problem. The first request processed by a fresh mod_perl child process will compile the regex and perform the search correctly. However, all subsequent requests running the same code in the same process will use the memorized pattern and not the fresh one supplied by users. The code will appear to be broken.

Imagine that you run a search engine service, and one person enters a search keyword of her choice and finds what she's looking for. Then another person who happens to be served by the same process searches for a different keyword, but unexpectedly receives the same search results as the previous person.

There are two solutions to this problem.

The first solution is to use the `eval q//` construct to force the code to be evaluated each time it's run. It's important that the `eval` block covers the entire processing loop, not just the pattern match itself.

The original code fragment would be rewritten as:

```
my $pattern = '^\d+$';
eval q{
    foreach (@list) {
        print if /$pattern/o;
    }
}
```

If we were to write this:

```
foreach (@list) {
    eval q{ print if /$pattern/o; };
}
```

the regex would be compiled for every element in the list, instead of just once for the entire loop over the list (and the `/o` modifier would essentially be useless).

However, watch out for using strings coming from an untrusted origin inside `eval`—they might contain Perl code dangerous to your system, so make sure to sanity-check them first.

This approach can be used if there is more than one pattern-match operator in a given section of code. If the section contains only one regex operator (be it `m//` or `s///`), you can rely on the property of the *null pattern*, which reuses the last pattern seen. This leads to the second solution, which also eliminates the use of `eval`.

The above code fragment becomes:

```
my $pattern = '^\d+$';
"0" =~ /$pattern/; # dummy match that must not fail!
foreach (@list) {
    print if //;
}
```

The only caveat is that the dummy match that boots the regular expression engine *must* succeed—otherwise the pattern will not be cached, and the `//` will match everything. If you can't count on fixed text to ensure the match succeeds, you have two options.

If you can guarantee that the pattern variable contains no metacharacters (such as *, +, ^, $, \d, etc.), you can use the dummy match of the pattern itself:

```
$pattern =~ /\Q$pattern\E/; # guaranteed if no metacharacters present
```

The `\Q` modifier ensures that any special regex characters will be escaped.

If there is a possibility that the pattern contains metacharacters, you should match the pattern itself, or the nonsearchable `\377` character, as follows:

```
"\377" =~ /$pattern|^\377$/; # guaranteed if metacharacters present
```

Matching patterns repeatedly

Another technique may also be used, depending on the complexity of the regex to which it is applied. One common situation in which a compiled regex is usually more efficient is when you are matching any one of a group of patterns over and over again.

To make this approach easier to use, we'll use a slightly modified helper routine from Jeffrey Friedl's book *Mastering Regular Expressions* (O'Reilly):

```
sub build_match_many_function {
    my @list = @_;
    my $expr = join '||',
        map { "\$_[0] =~ m/\$list[$_]/o" } (0..$#list);
    my $matchsub = eval "sub { $expr }";
    die "Failed in building regex @list: $@" if $@;
    return $matchsub;
}
```

This function accepts a list of patterns as an argument, builds a match regex for each item in the list against `$_[0]`, and uses the logical `||` (OR) operator to stop the matching when the first match succeeds. The chain of pattern matches is then placed into a string and compiled within an anonymous subroutine using `eval`. If `eval` fails, the code aborts with `die( )`; otherwise, a reference to this subroutine is returned to the caller.

Here is how it can be used:

```
my @agents = qw(Mozilla Lynx MSIE AmigaVoyager lwp libwww);
my $known_agent_sub = build_match_many_function(@agents);

while (<ACCESS_LOG>) {
    my $agent = get_agent_field($_);
    warn "Unknown Agent: $agent\n"
        unless $known_agent_sub->($agent);
}
```

This code takes lines of log entries from the *access_log* file already opened on the ACCESS_LOG file handle, extracts the agent field from each entry in the log file, and tries to match it against the list of known agents. Every time the match fails, it prints a warning with the name of the unknown agent.

An alternative approach is to use the qr// operator, which is used to compile a regex. The previous example can be rewritten as:

```
my @agents = qw(Mozilla Lynx MSIE AmigaVoyager lwp libwww);
my @compiled_re = map qr/$_/, @agents;

while (<ACCESS_LOG>) {
    my $agent = get_agent_field($_);
    my $ok = 0;
    for my $re (@compiled_re) {
        $ok = 1, last if /$re/;
    }
    warn "Unknown Agent: $agent\n"
        unless $ok;
}
```

In this code, we compile the patterns once before we use them, similar to build_match_many_function( ) from the previous example, but now we save an extra call to a subroutine. A simple benchmark shows that this example is about 2.5 times faster than the previous one.

Apache::Registry Specifics

The following coding issues are relevant only for scripts running under the Apache::Registry content handler and similar handlers, such as Apache::PerlRun. Of course, all of the mod_perl specifics described earlier apply as well.

__END__ and __DATA__ Tokens

An Apache::Registry script cannot contain __END__ or __DATA__ tokens, because Apache::Registry wraps the original script's code into a subroutine called handler( ), which is then called. Consider the following script, accessed as */perl/test.pl*:

```
print "Content-type: text/plain\n\n";
print "Hi";
```

When this script is executed under Apache::Registry, it becomes wrapped in a handler( ) subroutine, like this:

```
package Apache::ROOT::perl::test_2epl;
use Apache qw(exit);
sub handler {
    print "Content-type: text/plain\n\n";
    print "Hi";
}
```

If we happen to put an __END__ tag in the code, like this:

```
print "Content-type: text/plain\n\n";
print "Hi";
__END__
Some text that wouldn't be normally executed
```

it will be turned into:

```
package Apache::ROOT::perl::test_2epl;
use Apache qw(exit);
sub handler {
    print "Content-type: text/plain\n\n";
    print "Hi";
    __END__
    Some text that wouldn't be normally executed
}
```

When issuing a request to */perl/test.pl*, the following error will then be reported:

```
Missing right bracket at .... line 4, at end of line
```

Perl cuts everything after the __END__ tag. Therefore, the subroutine handler( )'s closing curly bracket is not seen by Perl. The same applies to the __DATA__ tag.

Symbolic Links

Apache::Registry caches the script in the package whose name is constructed from the URI from which the script is accessed. If the same script can be reached by different URIs, which is possible if you have used symbolic links or aliases, the same script will be stored in memory more than once, which is a waste.

For example, assuming that you already have the script at */home/httpd/perl/news/news.pl*, you can create a symbolic link:

```
panic% ln -s /home/httpd/perl/news/news.pl /home/httpd/perl/news.pl
```

Now the script can be reached through both URIs, */news/news.pl* and */news.pl*. This doesn't really matter until the two URIs get advertised and users reach the same script from the two of them.

Now start the server in single-server mode and issue a request to both URIs:

```
http://localhost/perl/news/news.pl
http://localhost/perl/news.pl
```

To reveal the duplication, you should use the Apache::Status module. Among other things, it shows all the compiled Apache::Registry scripts (using their respective packages). If you are using the default configuration directives, you should either use this URI:

```
http://localhost/perl-status?rgysubs
```

or just go to the main menu at:

```
http://localhost/perl-status
```

and click on the "Compiled Registry Scripts" menu item.

If the script was accessed through the two URIs, you will see the output shown in Figure 6-1.

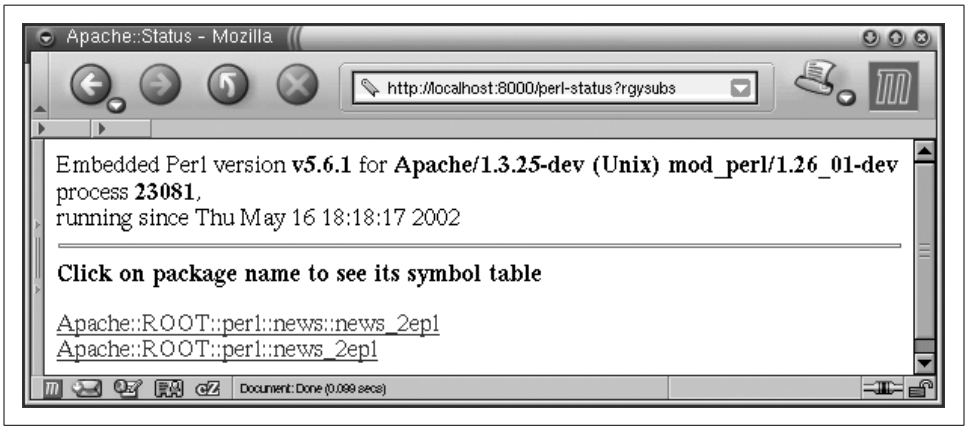

Figure 6-1. Compiled Registry Scripts output

You can usually spot this kind of problem by running a link checker that goes recursively through all the pages of the service by following all links, and then using `Apache::Status` to find the symlink duplicates (without restarting the server, of course). To make it easier to figure out what to look for, first find all symbolic links. For example, in our case, the following command shows that we have only one symlink:

```
panic% find /home/httpd/perl -type l
/home/httpd/perl/news.pl
```

So now we can look for that symlink in the output of the Compiled Registry Scripts section.

Notice that if you perform the testing in multi-server mode, some child processes might show only one entry or none at all, since they might not serve the same requests as the others.

Return Codes

`Apache::Registry` normally assumes a return code of *OK (200)* and sends it for you. If a different return code needs to be sent, `$r->status()` can be used. For example, to send the return code 404 (*Not Found*), you can use the following code:

```
use Apache::Constants qw(NOT_FOUND);
$r->status(NOT_FOUND);
```

If this method is used, there is no need to call `$r->send_http_header()` (assuming that the `PerlSendHeader Off` setting is in effect).

Transition from mod_cgi Scripts to Apache Handlers

If you don't need to preserve backward compatibility with mod_cgi, you can port mod_cgi scripts to use mod_perl-specific APIs. This allows you to benefit from features not available under mod_cgi and gives you better performance for the features available under both. We have already seen how easily `Apache::Registry` turns scripts into handlers before they get executed. The transition to handlers is straightforward in most cases.

Let's see a transition example. We will start with a mod_cgi-compatible script running under `Apache::Registry`, transpose it into a Perl content handler without using any mod_perl-specific modules, and then convert it to use the `Apache::Request` and `Apache::Cookie` modules that are available only in the mod_perl environment.

Starting with a mod_cgi-Compatible Script

Example 6-18 shows the original script's code.

Example 6-18. cookie_script.pl

```
use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);

init( );
print_header( );
print_status( );

sub init {
    $q = new CGI;
    $switch = $q->param("switch") ? 1 : 0;
    my %cookies = CGI::Cookie->fetch;
    $sessionID = exists $cookies{'sessionID'}
        ? $cookies{'sessionID'}->value
        : '';

    # 0 = not running, 1 = running
    $status = $sessionID ? 1 : 0;
    # switch status if asked to
    $status = !$status if $switch;

    if ($status) {
        # preserve sessionID if it exists or create a new one
        $sessionID ||= generate_sessionID( ) if $status;
    } else {
        # delete the sessionID
        $sessionID = '';
    }
```

Example 6-18. cookie_script.pl (continued)

```
}

sub print_header {
    my $c = CGI::Cookie->new(
         -name    => 'sessionID',
         -value   => $sessionID,
         -expires => '+1h'
    );

    print $q->header(
         -type   => 'text/html',
         -cookie => $c
    );
}

# print the current Session status and a form to toggle the status
sub print_status {

    print qq{<html><head><title>Cookie</title></head><body>};

    print "<B>Status:</B> ",
        $status
          ? "Session is running with ID: $sessionID"
          : "No session is running";


    # change status form
    my $button_label = $status ? "Stop" : "Start";
    print qq{<hr>
       <form>
         <input type=submit name=switch value=" $button_label ">
       </form>
            };

    print qq{</body></html>};

}

# A dummy ID generator
# Replace with a real session ID generator
########################
sub generate_sessionID {
    return scalar localtime;
}
```

The code is very simple. It creates a session when you press the Start button and deletes it when you pressed the Stop button. The session is stored and retrieved using cookies.

We have split the code into three subroutines. `init()` initializes global variables and parses incoming data. `print_header()` prints the HTTP headers, including the cookie

header. Finally, `print_status()` generates the output. Later, we will see that this logical separation will allow an easy conversion to Perl content-handler code.

We have used a few global variables, since we didn't want to pass them from function to function. In a big project, you should be very restrictive about what variables are allowed to be global, if any. In any case, the `init()` subroutine makes sure all these variables are reinitialized for each code reinvocation.

We have used a very simple `generate_sessionID()` function that returns a current date-time string (e.g., Wed Apr 12 15:02:23 2000) as a session ID. You'll want to replace this with code that generates a unique and unpredictable session ID each time it is called.

Converting into a Perl Content Handler

Let's now convert this script into a content handler. There are two parts to this task: first configure Apache to run the new code as a Perl handler, then modify the code itself.

First we add the following snippet to *httpd.conf*:

```
PerlModule Book::Cookie
<Location /test/cookie>
    SetHandler perl-script
    PerlHandler Book::Cookie
</Location>
```

and restart the server.

When a request whose URI starts with */test/cookie* is received, Apache will execute the `Book::Cookie::handler()` subroutine (which we will look at presently) as a content handler. We made sure we preloaded the `Book::Cookie` module at server startup with the `PerlModule` directive.

Now we modify the script itself. We copy its contents to the file *Cookie.pm* and place it into one of the directories listed in `@INC`. In this example, we'll use */home/httpd/perl*, which we added to `@INC`. Since we want to call this package `Book::Cookie`, we'll put *Cookie.pm* into the */home/httpd/perl/Book/* directory.

The changed code is in Example 6-19. As the subroutines were left unmodified from the original script, they aren't reproduced here (so you'll see the differences more clearly.)

Example 6-19. Book/Cookie.pm

```
package Book::Cookie;
use Apache::Constants qw(:common);

use strict;
use CGI;
use CGI::Cookie;
```

Example 6-19. Book/Cookie.pm

```
use vars qw($q $switch $status $sessionID);

sub handler {
    my $r = shift;

    init( );
    print_header( );
    print_status( );

    return OK;
}

# all subroutines unchanged

1;
```

Two lines have been added to the beginning of the code:

```
package Book::Cookie;
use Apache::Constants qw(:common);
```

The first line declares the package name, and the second line imports constants commonly used in mod_perl handlers to return status codes. In our case, we use the `OK` constant only when returning from the `handler( )` subroutine.

The following code is left unchanged:

```
use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);
```

We add some new code around the subroutine calls:

```
sub handler {
    my $r = shift;

    init( );
    print_header( );
    print_status( );

    return OK;
}
```

Each content handler (and any other handler) should begin with a subroutine called `handler( )`. This subroutine is called when a request's URI starts with */test/cookie*, as per our configuration. You can choose a different subroutine name—for example, `execute( )`—but then you must explicitly specify that name in the configuration directives in the following way:

```
PerlModule Book::Cookie
<Location /test/cookie>
    SetHandler perl-script
    PerlHandler Book::Cookie::execute
</Location>
```

We will use the default name, `handler()`.

The `handler()` subroutine is just like any other subroutine, but generally it has the following structure:

```
sub handler {
    my $r = shift;

    # the code

    # status (OK, DECLINED or else)
    return OK;
}
```

First, we retrieve a reference to the request object by shifting it from `@_` and assigning it to the `$r` variable. We'll need this a bit later.

Second, we write the code that processes the request.

Third, we return the status of the execution. There are many possible statuses; the most commonly used are `OK` and `DECLINED`. `OK` tells the server that the handler has completed the request phase to which it was assigned. `DECLINED` means the opposite, in which case another handler will process this request. `Apache::Constants` exports these and other commonly used status codes.

In our example, all we had to do was to wrap the three calls:

```
init();
print_header();
print_status();
```

inside the `handler()` skeleton:

```
sub handler {
    my $r = shift;

    return OK;
}
```

Last, we need to add `1;` at the end of the module, as we do with any Perl module. This ensures that `PerlModule` doesn't fail when it tries to load `Book::Cookie`.

To summarize, we took the original script's code and added the following seven lines:

```
package Book::Cookie;
use Apache::Constants qw(:common);

sub handler {
    my $r = shift;

    return OK;
}
1;
```

and we now have a fully-fledged Perl content handler.

Converting to use the mod_perl API and mod_perl-Specific Modules

Now that we have a complete PerlHandler, let's convert it to use the mod_perl API and mod_perl-specific modules. First, this may give us better performance where the internals of the API are implemented in C. Second, this unleashes the full power of Apache provided by the mod_perl API, which is only partially available in the mod_cgi-compatible modules.

We are going to replace CGI.pm and CGI::Cookie with their mod_perl-specific equivalents: Apache::Request and Apache::Cookie, respectively. These two modules are written in C with the XS interface to Perl, so code that uses these modules heavily runs much faster.

Apache::Request has an API similar to CGI's, and Apache::Cookie has an API similar to CGI::Cookie's. This makes porting straightforward. Essentially, we just replace:

```
use CGI;
$q = new CGI;
```

with:

```
use Apache::Request ();
$q = Apache::Request->new($r);
```

And we replace:

```
use CGI::Cookie ();
my $cookie = CGI::Cookie->new(...)
```

with:

```
use Apache::Cookie ();
my $cookie = Apache::Cookie->new($r, ...);
```

Example 6-20 is the new code for Book::Cookie2.

Example 6-20. Book/Cookie2.pm

```
package Book::Cookie2;
use Apache::Constants qw(:common);

use strict;
use Apache::Request ();
use Apache::Cookie ();
use vars qw($r $q $switch $status $sessionID);

sub handler {
    $r = shift;

    init();
    print_header();
    print_status();

    return OK;
```

Example 6-20. Book/Cookie2.pm (continued)

```
}

sub init {

    $q = Apache::Request->new($r);
    $switch = $q->param("switch") ? 1 : 0;

    my %cookies = Apache::Cookie->fetch;
    $sessionID = exists $cookies{'sessionID'}
        ? $cookies{'sessionID'}->value : '';

    # 0 = not running, 1 = running
    $status = $sessionID ? 1 : 0;
    # switch status if asked to
    $status = !$status if $switch;

    if ($status) {
        # preserve sessionID if it exists or create a new one
        $sessionID ||= generate_sessionID( ) if $status;
    } else {
        # delete the sessionID
        $sessionID = '';
    }
}

sub print_header {
    my $c = Apache::Cookie->new(
         $r,
         -name    => 'sessionID',
         -value   => $sessionID,
         -expires => '+1h');

    # Add a Set-Cookie header to the outgoing headers table
    $c->bake;

    $r->send_http_header('text/html');
}

# print the current Session status and a form to toggle the status
sub print_status {

    print qq{<html><head><title>Cookie</title></head><body>};

    print "<B>Status:</B> ",
        $status
            ? "Session is running with ID: $sessionID"
            : "No session is running";

    # change status form
    my $button_label = $status ? "Stop" : "Start";
    print qq{<hr>
       <form>
```

Example 6-20. Book/Cookie2.pm (continued)

```
        <input type=submit name=switch value=" $button_label ">
      </form>
          };

    print qq{</body></html>};

}

# replace with a real session ID generator
sub generate_sessionID {
    return scalar localtime;
}

1;
```

The only other changes are in the `print_header( )` function. Instead of passing the cookie code to `CGI`'s `header( )` function to return a proper HTTP header, like this:

```
print $q->header(
   -type   => 'text/html',
   -cookie => $c);
```

we do it in two stages. First, the following line adds a `Set-Cookie` header to the outgoing headers table:

```
$c->bake;
```

Then this line sets the `Content-Type` header to *text/html* and sends out the whole HTTP header:

```
$r->send_http_header('text/html');
```

The rest of the code is unchanged.

The last thing we need to do is add the following snippet to *httpd.conf*:

```
PerlModule Book::Cookie2
<Location /test/cookie2>
    SetHandler perl-script
    PerlHandler Book::Cookie2
</Location>
```

Now the magic URI that will trigger the above code execution will be one starting with */test/cookie2*. We save the code in the file */home/httpd/perl/Book/Cookie2.pm*, since we have called this package `Book::Cookie2`.

As you've seen, converting well-written CGI code into mod_perl handler code is straightforward. Taking advantage of mod_perl-specific features and modules is also generally simple. Very little code needs to be changed to convert a script.

Note that to make the demonstration simple to follow, we haven't changed the style of the original package. But by all means consider doing that when porting real code: use lexicals instead of globals, apply mod_perl API functions where applicable, etc.

Loading and Reloading Modules

You often need to reload modules in development and production environments. mod_perl tries hard to avoid unnecessary module reloading, but sometimes (especially during the development process) we want some modules to be reloaded when modified. The following sections discuss issues related to module loading and reloading.

The @INC Array Under mod_perl

Under mod_perl, `@INC` can be modified only during server startup. After each request, mod_perl resets `@INC`'s value to the one it had before the request.

If mod_perl encounters a statement like the following:

```
use lib qw(foo/bar);
```

it modifies `@INC` only for the period during which the code is being parsed and compiled. Afterward, `@INC` is reset to its original value. Therefore, the only way to change `@INC` permanently is to modify it at server startup.

There are two ways to alter `@INC` at server startup:

- In the configuration file, with:

  ```
  PerlSetEnv PERL5LIB /home/httpd/perl
  ```

  or:

  ```
  PerlSetEnv PERL5LIB /home/httpd/perl:/home/httpd/mymodules
  ```

- In the *startup.pl* file:

  ```
  use lib qw(/home/httpd/perl /home/httpd/mymodules);
  1;
  ```

  As always, the startup file needs to be loaded from *httpd.conf*:

  ```
  PerlRequire /path/to/startup.pl
  ```

To make sure that you have set `@INC` correctly, configure `perl-status` into your server, as explained in Chapter 21. Follow the "Loaded Modules" item in the menu and look at the bottom of the generated page, where the contents of `@INC` are shown:

```
@INC =
/home/httpd/mymodules
/home/httpd/perl
/usr/lib/perl5/5.6.1/i386-linux
/usr/lib/perl5/5.6.1
/usr/lib/perl5/site_perl/5.6.1/i386-linux
/usr/lib/perl5/site_perl/5.6.1
/usr/lib/perl5/site_perl
.
/home/httpd/httpd_perl/
/home/httpd/httpd_perl/lib/perl
```

As you can see in our setup, we have two custom directories prepended at the beginning of the list. The rest of the list contains standard directories from the Perl distribution, plus the *$ServerRoot* and *$ServerRoot/lib/perl* directories appended at the end (which mod_perl adds automatically).

Reloading Modules and Required Files

When working with mod_cgi, you can change the code and rerun the CGI script from your browser to see the changes. Since the script isn't cached in memory, the server starts up a new Perl interpreter for each request, which loads and recompiles the script from scratch. The effects of any changes are immediate.

The situation is different with mod_perl, since the whole idea is to get maximum performance from the server. By default, the server won't spend time checking whether any included library modules have been changed. It assumes that they weren't, thus saving the time it takes to `stat()` the source files from any modules and libraries you `use()` and `require()` in your script.

If the scripts are running under `Apache::Registry`, the only check that is performed is to see whether your main script has been changed. If your scripts do not `use()` or `require()` any other Perl modules or packages, there is nothing to worry about. If, however, you are developing a script that includes other modules, the files you `use()` or `require()` aren't checked for modification, and you need to do something about that.

There are a couple of techniques to make a mod_perl-enabled server recognize changes in library modules. They are discussed in the following sections.

Restarting the server

The simplest approach is to restart the server each time you apply some change to your code. Restarting techniques are covered in Chapter 5. After restarting the server about 50 times, you will tire of it and look for other solutions.

Using Apache::StatINC

Help comes from the `Apache::StatINC` module. When Perl pulls in a file with `require()`, it stores the full pathname as a value in the global hash `%INC` with the filename as the key. `Apache::StatINC` looks through `%INC` and immediately reloads any file that has been updated on the disk.

To enable this module, add these two lines to *httpd.conf*:

```
PerlModule Apache::StatINC
PerlInitHandler Apache::StatINC
```

To be sure it really works, turn on debug mode on your development system by adding `PerlSetVar StatINCDebug On` to your configuration file. You end up with something like this:

```
PerlModule Apache::StatINC
PerlInitHandler Apache::StatINC
<Location /perl>
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
    PerlSendHeader On
    PerlSetVar StatINCDebug On
</Location>
```

Be aware that only the modules located in `@INC` are reloaded on change, and you can change `@INC` only before the server has been started (in the startup file).

Note the following trap: because "`.`", the current directory, is in `@INC`, Perl knows how to `require( )` files with pathnames relative to the current script's directory. After the code has been parsed, however, the server doesn't remember the path. So if the code loads a module `MyModule` located in the directory of the script and this directory is not in `@INC`, you end up with the following entry in `%INC`:

```
'MyModule.pm' => 'MyModule.pm'
```

When `Apache::StatINC` tries to check whether the file has been modified, it won't be able to find the file, since *MyModule.pm* is not in any of the paths in `@INC`. To correct this problem, add the module's location path to `@INC` at server startup.

Using Apache::Reload

`Apache::Reload` is a newer module that comes as a drop-in replacement for `Apache::StatINC`. It provides extra functionality and is more flexible.

To make `Apache::Reload` check all the loaded modules on each request, just add the following line to *httpd.conf*:

```
PerlInitHandler Apache::Reload
```

To reload only specific modules when these get changed, three alternatives are provided: registering the module implicitly, registering the module explicitly, and setting up a dummy file to *touch* whenever you want the modules reloaded.

To use implicit module registration, turn off the `ReloadAll` variable, which is on by default:

```
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
```

and add the following line to every module that you want to be reloaded on change:

```
use Apache::Reload;
```

Alternatively, you can explicitly specify modules to be reloaded in *httpd.conf*:

```
PerlInitHandler Apache::Reload
PerlSetVar ReloadModules "Book::Foo Book::Bar Foo::Bar::Test"
```

Note that these are split on whitespace, but the module list *must* be in quotes, or Apache will try to parse the parameter list itself.

You can register groups of modules using the metacharacter *:

```
PerlSetVar ReloadModules "Foo::* Bar::*"
```

In the above example, all modules starting with `Foo::` and `Bar::` will become registered. This feature allows you to assign all the modules in a project using a single pattern.

The third option is to set up a file that you can *touch* to cause the reloads to be performed:

```
PerlSetVar ReloadTouchFile /tmp/reload_modules
```

Now when you're happy with your changes, simply go to the command line and type:

```
panic% touch /tmp/reload_modules
```

If you set this, and don't *touch* the file, the reloads won't happen (regardless of how the modules have been registered).

This feature is very convenient in a production server environment, but compared to a full restart, the benefits of preloaded modules memory-sharing are lost, since each child will get its own copy of the reloaded modules.

Note that `Apache::Reload` might have a problem with reloading single modules containing multiple packages that all use pseudo-hashes. The solution: don't use pseudo-hashes. Pseudo-hashes will be removed from newer versions of Perl anyway.

Just like with `Apache::StatInc`, if you have modules loaded from directories that are not in `@INC`, `Apache::Reload` will fail to find the files. This is because `@INC` is reset to its original value even if it gets temporarily modified in the script. The solution is to extend `@INC` at server startup to include all the directories from which you load files that aren't in the standard `@INC` paths.

Using dynamic configuration files

Sometimes you may want an application to monitor its own configuration file and reload it when it is altered. But you don't want to restart the server for these changes to take effect. The solution is to use dynamic configuration files.

Dynamic configuration files are especially useful when you want to provide administrators with a configuration tool that modifies an application on the fly. This approach eliminates the need to provide shell access to the server. It can also prevent typos, because the administration program can verify the submitted modifications.

It's possible to get away with `Apache::Reload` and still have a similar small overhead for the `stat()` call, but this requires the involvement of a person who can modify *httpd.conf* to configure `Apache::Reload`. The method described next has no such requirement.

Writing configuration files. We'll start by describing various approaches to writing configuration files, and their strengths and weaknesses.

If your configuration file contains only a few variables, it doesn't matter how you write the file. In practice, however, configuration files often grow as a project develops. This is especially true for projects that generate HTML files, since they tend to demand many easily configurable settings, such as the location of headers, footers, templates, colors, and so on.

A common approach used by CGI programmers is to define all configuration variables in a separate file. For example:

```
$cgi_dir  = '/home/httpd/perl';
$cgi_url  = '/perl';
$docs_dir = '/home/httpd/docs';
$docs_url = '/';
$img_dir  = '/home/httpd/docs/images';
$img_url  = '/images';
# ... many more config params here ...
$color_hint   = '#777777';
$color_warn   = '#990066';
$color_normal = '#000000';
```

The `use strict;` pragma demands that all variables be declared. When using these variables in a mod_perl script, we must declare them with `use vars` in the script, so we start the script with:

```
use strict;
use vars qw($cgi_dir $cgi_url $docs_dir $docs_url
            # ... many more config params here ....
            $color_hint  $color_warn $color_normal
           );
```

It is a nightmare to maintain such a script, especially if not all features have been coded yet—we have to keep adding and removing variable names. Since we're writing clean code, we also start the configuration file with `use strict;`, so we have to list the variables with `use vars` here as well—a second list of variables to maintain. Then, as we write many different scripts, we may get name collisions between configuration files.

The solution is to use the power of Perl's packages and assign a unique package name to each configuration file. For example, we might declare the following package name:

```
package Book::Config0;
```

Now each configuration file is isolated into its own namespace. But how does the script use these variables? We can no longer just `require( )` the file and use the variables, since they now belong to a different package. Instead, we must modify all our scripts to use the configuration variables' fully qualified names (e.g., referring to `$Book::Config0::cgi_url` instead of just `$cgi_url`).

You may find typing fully qualified names tedious, or you may have a large repository of legacy scripts that would take a while to update. If so, you'll want to import the required variables into any script that is going to use them. First, the configuration package has to export those variables. This entails listing the names of all the variables in the `@EXPORT_OK` hash. See Example 6-21.

Example 6-21. Book/Config0.pm

```
package Book::Config0;
use strict;

BEGIN {
  use Exporter ();

  @Book::HTML::ISA       = qw(Exporter);
  @Book::HTML::EXPORT    = qw( );
  @Book::HTML::EXPORT_OK = qw($cgi_dir $cgi_url $docs_dir $docs_url
                              # ... many more config params here ....
                              $color_hint $color_warn $color_normal);
}

use vars qw($cgi_dir $cgi_url $docs_dir $docs_url
            # ... many more config params here ....
            $color_hint  $color_warn $color_normal
           );

$cgi_dir  = '/home/httpd/perl';
$cgi_url  = '/perl';
$docs_dir = '/home/httpd/docs';
$docs_url = '/';
$img_dir  = '/home/httpd/docs/images';
$img_url  = '/images';
# ... many more config params here ...
$color_hint   = "#777777";
$color_warn   = "#990066";
$color_normal = "#000000";
```

A script that uses this package will start with this code:

```
use strict;
use Book::Config0 qw($cgi_dir $cgi_url $docs_dir $docs_url
                     # ... many more config params here ....
                     $color_hint  $color_warn $color_normal
                   );
use vars          qw($cgi_dir $cgi_url $docs_dir $docs_url
                     # ... many more config params here ....
                     $color_hint  $color_warn $color_normal
                   );
```

Whoa! We now have to update at least three variable lists when we make a change in naming of the configuration variables. And we have only one script using the configuration file, whereas a real-life application often contains many different scripts.

There's also a performance drawback: exported variables add some memory overhead, and in the context of mod_perl this overhead is multiplied by the number of server processes running.

There are a number of techniques we can use to get rid of these problems. First, variables can be grouped in named groups called *tags*. The tags are later used as arguments to the `import()` or `use()` calls. You are probably familiar with:

```
use CGI qw(:standard :html);
```

We can implement this quite easily, with the help of `export_ok_tags()` from `Exporter`. For example:

```
BEGIN {
  use Exporter ();
  use vars qw( @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS );
  @ISA         = qw(Exporter);
  @EXPORT      = ();
  @EXPORT_OK   = ();

  %EXPORT_TAGS = (
      vars => [qw($firstname $surname)],
      subs => [qw(reread_conf untaint_path)],
  );
  Exporter::export_ok_tags('vars');
  Exporter::export_ok_tags('subs');
}
```

In the script using this configuration, we write:

```
use Book::Config0 qw(:subs :vars);
```

Subroutines are exported exactly like variables, since symbols are what are actually being exported. Notice we don't use `export_tags()`, as it exports the variables automatically without the user asking for them (this is considered bad style). If a module automatically exports variables with `export_tags()`, you can avoid unnecessary imports in your script by using this syntax:

```
use Book::Config0 ();
```

You can also go even further and group tags into other named groups. For example, the `:all` tag from `CGI.pm` is a group tag of all other groups. It requires a little more effort to implement, but you can always save time by looking at the solution in `CGI.pm`'s code. It's just a matter of an extra code to expand all the groups recursively.

As the number of variables grows, however, your configuration will become unwieldy. Consider keeping all the variables in a single hash built from references to other scalars, anonymous arrays, and hashes. See Example 6-22.

Example 6-22. Book/Config1.pm

```
package Book::Config1;
use strict;

BEGIN {
  use Exporter ();

  @Book::Config1::ISA       = qw(Exporter);
  @Book::Config1::EXPORT    = qw( );
  @Book::Config1::EXPORT_OK = qw(%c);
}

use vars qw(%c);

%c = (
   dir => {
       cgi  => '/home/httpd/perl',
       docs => '/home/httpd/docs',
       img  => '/home/httpd/docs/images',
      },
   url => {
       cgi  => '/perl',
       docs => '/',
       img  => '/images',
      },
   color => {
         hint   => '#777777',
         warn   => '#990066',
         normal => '#000000',
        },
  );
```

Good Perl style suggests keeping a comma at the end of each list. This makes it easy to add new items at the end of a list.

Our script now looks like this:

```
use strict;
use Book::Config1 qw(%c);
use vars          qw(%c);
print "Content-type: text/plain\n\n";
print "My url docs root: $c{url}{docs}\n";
```

The whole mess is gone. Now there is only one variable to worry about.

The one small downside to this approach is auto-vivification. For example, if we write `$c{url}{doc}` by mistake, Perl will silently create this element for us with the value undef. When we `use strict;`, Perl will tell us about any misspelling of this kind for a simple scalar, but this check is not performed for hash elements. This puts the onus of responsibility back on us, since we must take greater care.

The benefits of the hash approach are significant. Let's make it even better by getting rid of the Exporter stuff completely, removing all the exporting code from the configuration file. See Example 6-23.

Example 6-23. Book/Config2.pm

```
package Book::Config2;
use strict;
use vars qw(%c);

%c = (
   dir => {
       cgi  => '/home/httpd/perl',
       docs => '/home/httpd/docs',
       img  => '/home/httpd/docs/images',
      },
   url => {
       cgi  => '/perl',
       docs => '/',
       img  => '/images',
      },
   color => {
         hint   => '#777777',
         warn   => '#990066',
         normal => '#000000',
        },
  );
```

Our script is modified to use fully qualified names for the configuration variables it uses:

```
use strict;
use Book::Config2 ();
print "Content-type: text/plain\n\n";
print "My url docs root: $Book::Config2::c{url}{docs}\n";
```

To save typing and spare the need to use fully qualified variable names, we'll use a magical Perl feature to alias the configuration variable to a script's variable:

```
use strict;
use Book::Config2 ();
use vars qw(%c);
*c = \%Book::Config2::c;
print "Content-type: text/plain\n\n";
print "My url docs root: $c{url}{docs}\n";
```

We've aliased the *c glob with a reference to the configuration hash. From now on, %Book::Config2::c and %c refer to the same hash for all practical purposes.

One last point: often, redundancy is introduced in configuration variables. Consider:

```
$cgi_dir  = '/home/httpd/perl';
$docs_dir = '/home/httpd/docs';
$img_dir  = '/home/httpd/docs/images';
```

It's obvious that the base path */home/httpd* should be moved to a separate variable, so only that variable needs to be changed if the application is moved to another location on the filesystem.

```
$base     = '/home/httpd';
$cgi_dir  = "$base/perl";
$docs_dir = "$base/docs";
$img_dir  = "$docs_dir/images";
```

This cannot be done with a hash, since we cannot refer to its values before the definition is completed. That is, this will not work:

```
%c = (
   base => '/home/httpd',
   dir => {
       cgi  => "$c{base}/perl",
       docs => "$c{base}/docs",
       img  => "$c{base}{docs}/images",
      },
  );
```

But nothing stops us from adding additional variables that are lexically scoped with `my()`. The following code is correct:

```
my $base = '/home/httpd';
%c = (
   dir => {
       cgi  => "$base/perl",
       docs => "$base/docs",
       img  => "$base/docs/images",
      },
  );
```

We've learned how to write configuration files that are easy to maintain, and how to save memory by avoiding importing variables in each script's namespace. Now let's look at reloading those files.

Reloading configuration files. First, lets look at a simple case, in which we just have to look after a simple configuration file like the one below. Imagine a script that tells you who is the patch pumpkin of the current Perl release.* (*Pumpkin* is a whimsical term for the person with exclusive access to a virtual "token" representing a certain authority, such as applying patches to a master copy of some source.)

```
use CGI ();
use strict;

my $firstname = "Jarkko";
my $surname = "Hietaniemi";
my $q = CGI->new;
```

* These are the recent pumpkins: Chip Salzenberg for 5.004, Gurusamy Sarathy for 5.005 and 5.6, Jarkko Hietaniemi for 5.8, Hugo van der Sanden for 5.10.

```
print $q->header(-type=>'text/html');
print $q->p("$firstname $surname holds the patch pumpkin" .
            "for this Perl release.");
```

The script is very simple: it initializes the CGI object, prints the proper HTTP header, and tells the world who the current patch pumpkin is. The name of the patch pumpkin is a hardcoded value.

We don't want to modify the script every time the patch pumpkin changes, so we put the `$firstname` and `$surname` variables into a configuration file:

```
$firstname = "Jarkko";
$surname = "Hietaniemi";
1;
```

Note that there is no package declaration in the above file, so the code will be evaluated in the caller's package or in the `main::` package if none was declared. This means that the variables `$firstname` and `$surname` will override (or initialize) the variables with the same names in the caller's namespace. This works for global variables only—you cannot update variables defined lexically (with `my( )`) using this technique.

Let's say we have started the server and everything is working properly. After a while, we decide to modify the configuration. How do we let our running server know that the configuration was modified without restarting it? Remember, we are in production, and a server restart can be quite expensive. One of the simplest solutions is to poll the file's modification time by calling `stat( )` before the script starts to do real work. If we see that the file was updated, we can force a reconfiguration of the variables located in this file. We will call the function that reloads the configuration `reread_conf( )` and have it accept the relative path to the configuration file as its single argument.

`Apache::Registry` executes a `chdir( )` to the script's directory before it starts the script's execution. So if your CGI script is invoked under the `Apache::Registry` handler, you can put the configuration file in the same directory as the script. Alternatively, you can put the file in a directory below that and use a path relative to the script directory. However, you have to make sure that the file will be found, somehow. Be aware that `do( )` searches the libraries in the directories in `@INC`.

```
use vars qw(%MODIFIED);
sub reread_conf {
    my $file = shift;
    return unless defined $file;
    return unless -e $file and -r _;
    my $mod = -M _;
    unless (exists $MODIFIED{$file} and $MODIFIED{$file} == $mod) {
        unless (my $result = do $file) {
            warn "couldn't parse $file: $@" if $@;
            warn "couldn't read $file: $!" unless defined $result;
            warn "couldn't run $file"      unless         $result;
        }
        $MODIFIED{$file} = $mod; # Update the MODIFICATION times
    }
}
```

Notice that we use the == comparison operator when checking the file's modification timestamp, because all we want to know is whether the file was changed or not.

When the require(), use(), and do() operators successfully return, the file that was passed as an argument is inserted into %INC. The hash element key is the name of the file, and the element's value is the file's path. When Perl sees require() or use() in the code, it first tests %INC to see whether the file is already there and thus loaded. If the test returns true, Perl saves the overhead of code rereading and recompiling; however, calling do() will load or reload the file regardless of whether it has been previously loaded.

We use do(), not require(), to reload the code in this file because although do() behaves almost identically to require(), it reloads the file unconditionally. If do() cannot read the file, it returns undef and sets $! to report the error. If do() can read the file but cannot compile it, it returns undef and sets an error message in $@. If the file is successfully compiled, do() returns the value of the last expression evaluated.

The configuration file can be broken if someone has incorrectly modified it. Since we don't want the whole service using that file to be broken that easily, we trap the possible failure to do() the file and ignore the changes by resetting the modification time. If do() fails to load the file, it might be a good idea to send an email about the problem to the system administrator.

However, since do() updates %INC like require() does, if you are using Apache::StatINC it will attempt to reload this file before the reread_conf() call. If the file doesn't compile, the request will be aborted. Apache::StatINC shouldn't be used in production anyway (because it slows things down by stat()ing all the files listed in %INC), so this shouldn't be a problem.

Note that we assume that the entire purpose of this function is to reload the configuration if it was changed. This is fail-safe, because if something goes wrong we just return without modifying the server configuration. The script should not be used to initialize the variables on its first invocation. To do that, you would need to replace each occurrence of return() and warn() with die().

We've used the above approach with a huge configuration file that was loaded only at server startup and another little configuration file that included only a few variables that could be updated by hand or through the web interface. Those variables were initialized in the main configuration file. If the webmaster breaks the syntax of this dynamic file while updating it by hand, it won't affect the main (write-protected) configuration file and won't stop the proper execution of the programs. In the next section, we will see a simple web interface that allows us to modify the configuration file without the risk of breaking it.

Example 6-24 shows a sample script using our reread_conf() subroutine.

Example 6-24. reread_conf.pl

```
use vars qw(%MODIFIED $firstname $surname);
use CGI ();
use strict;

my $q = CGI->new;
print $q->header(-type => 'text/plain');
my $config_file = "./config.pl";
reread_conf($config_file);
print $q->p("$firstname $surname holds the patch pumpkin" .
            "for this Perl release.");

sub reread_conf {
    my $file = shift;
    return unless defined $file;
    return unless -e $file and -r _;
    my $mod = -M _;
    unless ($MODIFIED{$file} and $MODIFIED{$file} == $mod) {
        unless (my $result = do $file) {
            warn "couldn't parse $file: $@" if $@;
            warn "couldn't read $file: $!"  unless defined $result;
            warn "couldn't run $file"       unless $result;
        }
        $MODIFIED{$file} = $mod; # Update the MODIFICATION time
    }
}
```

You should be using `(stat $file)[9]` instead of `-M $file` if you are modifying the `$^T` variable. This is because `-M` returns the modification time relative to the Perl interpreter startup time, set in `$^T`. In some scripts, it can be useful to reset `$^T` to the time of the script invocation with `"local $^T = time()"`. That way, `-M` and other `-X` file status tests are performed relative to the script invocation time, not the time the process was started.

If your configuration file is more sophisticated—for example, if it declares a package and exports variables—the above code will work just as well. Variables need not be `import()`ed again: when `do()` recompiles the script, the originally imported variables will be updated with the values from the reloaded code.

Dynamically updating configuration files. The CGI script below allows a system administrator to dynamically update a configuration file through a web interface. This script, combined with the code we have just seen to reload the modified files, gives us a system that is dynamically reconfigurable without having to restart the server. Configuration can be performed from any machine that has a browser.

Let's say we have a configuration file like the one in Example 6-25.

Example 6-25. Book/MainConfig.pm

```
package Book::MainConfig;

use strict;
use vars qw(%c);

%c = (
      name     => "Larry Wall",
      release  => "5.000",
      comments => "Adding more ways to do the same thing :)",

      other    => "More config values",

      colors   => { foreground => "black",
                    background => "white",
                  },

      machines => [qw( primary secondary tertiary )],

);
```

We want to make the variables `name`, `release`, and `comments` dynamically configurable. We'll need a web interface with an input form that allows modifications to these variables. We'll also need to update the configuration file and propagate the changes to all the currently running processes.

Let's look at the main stages of the implementation:

1. Create a form with preset current values of the variables.
2. Let the administrator modify the variables and submit the changes.
3. Validate the submitted information (numeric fields should hold numbers within a given range, etc.).
4. Update the configuration file.
5. Update the modified value in the current process's memory.
6. Display the form as before with the (possibly changed) current values.

The only part that seems hard to implement is a configuration file update, for a couple of reasons. If updating the file breaks it, the whole service won't work. If the file is very big and includes comments and complex data structures, parsing the file can be quite a challenge.

So let's simplify the task. If all we want is to update a few variables, why don't we create a tiny configuration file containing just those variables? It can be modified through the web interface and overwritten each time there is something to be changed, so that we don't have to parse the file before updating it. If the main configuration file is changed, we don't care, because we don't depend on it any more.

The dynamically updated variables will be duplicated in the main file and the dynamic file. We do this to simplify maintenance. When a new release is installed,

the dynamic configuration file won't exist—it will be created only after the first update. As we just saw, the only change in the main code is to add a snippet to load this file if it exists and was changed.

This additional code must be executed after the main configuration file has been loaded. That way, the updated variables will override the default values in the main file. See Example 6-26.

Example 6-26. manage_conf.pl

```
# remember to run this code in taint mode
use strict;
use vars qw($q %c $dynamic_config_file %vars_to_change %validation_rules);

use CGI ();

use lib qw(.);
use Book::MainConfig ();
*c = \%Book::MainConfig::c;

$dynamic_config_file = "./config.pl";

# load the dynamic configuration file if it exists, and override the
# default values from the main configuration file
do $dynamic_config_file if -e $dynamic_config_file and -r _;

# fields that can be changed and their captions
%vars_to_change =
  (
   'name'     => "Patch Pumpkin's Name",
   'release'  => "Current Perl Release",
   'comments' => "Release Comments",
  );

# each field has an associated regular expression
# used to validate the field's content when the
# form is submitted
%validation_rules =
  (
   'name'     => sub { $_[0] =~ /^[\w\s\.]+$/;   },
   'release'  => sub { $_[0] =~ /^\d+\.[\d_]+$/; },
   'comments' => sub { 1;                        },
  );

# create the CGI object, and print the HTTP and HTML headers
$q = CGI->new;
print $q->header(-type=>'text/html'),
      $q->start_html();

# We always rewrite the dynamic config file, so we want all the
# variables to be passed, but to save time we will only check
# those variables that were changed.  The rest will be retrieved from
# the 'prev_*' values.
```

Example 6-26. manage_conf.pl (continued)

```
my %updates = ( );
foreach (keys %vars_to_change) {
    # copy var so we can modify it
    my $new_val = $q->param($_) || '';

    # strip a possible ^M char (Win32)
    $new_val =~ s/\cM//g;

    # push to hash if it was changed
    $updates{$_} = $new_val
        if defined $q->param("prev_" . $_)
           and $new_val ne $q->param("prev_" . $_);
}

# Note that we cannot trust the previous values of the variables
# since they were presented to the user as hidden form variables,
# and the user could have mangled them. We don't care: this can't do
# any damage, as we verify each variable by rules that we define.

# Process if there is something to process. Will not be called if
# it's invoked the first time to display the form or when the form
# was submitted but the values weren't modified (we'll know by
# comparing with the previous values of the variables, which are
# the hidden fields in the form).

process_changed_config(%updates) if %updates;

show_modification_form( );

# update the config file, but first validate that the values are
# acceptable
sub process_changed_config {
    my %updates = @_;

    # we will list here all variables that don't validate
    my %malformed = ( );

    print $q->b("Trying to validate these values<br>");
    foreach (keys %updates) {
        print "<dt><b>$_</b> => <pre>$updates{$_}</pre>";

        # now we have to handle each var to be changed very carefully,
        # since this file goes immediately into production!
        $malformed{$_} = delete $updates{$_}
            unless $validation_rules{$_}->($updates{$_});

    }

    if (%malformed) {
        print $q->hr,
            $q->p($q->b(qq{Warning! These variables were changed
                           to invalid values. The original
```

Example 6-26. manage_conf.pl (continued)

```
                        values will be kept.})
                ),
            join ",<br>",
                map { $q->b($vars_to_change{$_}) . " : $malformed{$_}\n"
                    } keys %malformed;
    }

    # Now complete the vars that weren't changed from the
    # $q->param('prev_var') values
    map { $updates{$_} = $q->param('prev_' . $_)
              unless exists $updates{$_} } keys %vars_to_change;

    # Now we have all the data that should be written into the dynamic
    # config file

    # escape single quotes "'" while creating a file
    my $content = join "\n",
        map { $updates{$_} =~ s/(['\\])/\\$1/g;
              '$c{' . $_ . "} = '" . $updates{$_} . "';\n"
            } keys %updates;

    # add '1;' to make require() happy
    $content .= "\n1;";

    # keep the dummy result in $res so it won't complain
    eval {my $res = $content};
    if ($@) {
        print qq{Warning! Something went wrong with config file
                 generation!<p> The error was :</p> <br><pre>$@</pre>};
        return;
    }

    print $q->hr;

    # overwrite the dynamic config file
    my $fh = Apache::gensym( );
    open $fh, ">$dynamic_config_file.bak"
        or die "Can't open $dynamic_config_file.bak for writing: $!";
    flock $fh, 2; # exclusive lock
    seek $fh, 0, 0; # rewind to the start
    truncate $fh, 0; # the file might shrink!
    print $fh $content;
    close $fh;

    # OK, now we make a real file
    rename "$dynamic_config_file.bak", $dynamic_config_file
        or die "Failed to rename: $!";

    # rerun it to update variables in the current process! Note that
    # it won't update the variables in other processes. Special
    # code that watches the timestamps on the config file will do this
    # work for each process. Since the next invocation will update the
```

Example 6-26. manage_conf.pl (continued)

```
    # configuration anyway, why do we need to load it here? The reason
    # is simple: we are going to fill the form's input fields with
    # the updated data.
    do $dynamic_config_file;

}

sub show_modification_form {

    print $q->center($q->h3("Update Form"));

    print $q->hr,
        $q->p(qq{This form allows you to dynamically update the current
           configuration. You don't need to restart the server in
           order for changes to take an effect}
             );

    # set the previous settings in the form's hidden fields, so we
    # know whether we have to do some changes or not
    $q->param("prev_$_", $c{$_}) for keys %vars_to_change;

    # rows for the table, go into the form
    my @configs = ();

    # prepare text field entries
    push @configs,
        map {
          $q->td( $q->b("$vars_to_change{$_}:") ),
          $q->td(
           $q->textfield(
                 -name      => $_,
                 -default   => $c{$_},
                 -override  => 1,
                 -size      => 20,
                 -maxlength => 50,
                )
          ),
        } qw(name release);

    # prepare multiline textarea entries
    push @configs,
        map {
          $q->td( $q->b("$vars_to_change{$_}:") ),
          $q->td(
           $q->textarea(
                -name     => $_,
                -default  => $c{$_},
                -override => 1,
                -rows     => 10,
                -columns  => 50,
                -wrap     => "HARD",
                )
```

Example 6-26. manage_conf.pl (continued)

```
          ),
        } qw(comments);

    print $q->startform(POST => $q->url), "\n",
          $q->center(
              $q->table(map {$q->Tr($_), "\n",} @configs),
              $q->submit('', 'Update!'), "\n",
          ),
          map ({$q->hidden("prev_" . $_, $q->param("prev_".$_)) . "\n" }
               keys %vars_to_change), # hidden previous values
          $q->br, "\n",
          $q->endform, "\n",
          $q->hr, "\n",
          $q->end_html;

}
```

For example, on July 19 2002, Perl 5.8.0 was released. On that date, Jarkko Hietaniemi exclaimed:

```
The pumpking is dead! Long live the pumpking!
```

Hugo van der Sanden is the new pumpking for Perl 5.10. Therefore, we run *manage_conf.pl* and update the data. Once updated, the script overwrites the previous *config.pl* file with the following content:

```
$c{release}  =  '5.10';

$c{name}  =  'Hugo van der Sanden';

$c{comments}  =  'Perl rules the world!';

1;
```

Instead of crafting your own code, you can use the `CGI::QuickForm` module from CPAN to make the coding less tedious. See Example 6-27.

Example 6-27. manage_conf.pl

```
use strict;
use CGI qw( :standard :html3 ) ;
use CGI::QuickForm;
use lib qw(.);
use Book::MainConfig ();
*c = \%Book::MainConfig::c;

my $TITLE = 'Update Configuration';
show_form(
    -HEADER => header . start_html( $TITLE ) . h3( $TITLE ),
    -ACCEPT => \&on_valid_form,
    -FIELDS => [
        {
            -LABEL      => "Patch Pumpkin's Name",
```

Example 6-27. manage_conf.pl (continued)

```
            -VALIDATE   => sub { $_[0] =~ /^[\w\s\.]+$/;   },
            -default    => $c{name},
        },
        {
            -LABEL      => "Current Perl Release",
            -VALIDATE   => sub { $_[0] =~ /^\d+\.[\d_]+$/; },
            -default    => $c{release},
        },
        {
            -LABEL      => "Release Comments",
            -default    => $c{comments},
        },
        ],
    );

sub on_valid_form {
    # save the form's values
}
```

That's it. `show_form( )` creates and displays a form with a submit button. When the user submits, the values are checked. If all the fields are valid, `on_valid_form( )` is called; otherwise, the form is re-presented with the errors highlighted.

Handling the "User Pressed Stop Button" Case

When a user presses the Stop or Reload button, the current socket connection is broken (aborted). It would be nice if Apache could always immediately detect this event. Unfortunately, there is no way to tell whether the connection is still valid unless an attempt to read from or write to the connection is made.

Note that no detection technique will work if the connection to the backend mod_perl server is coming from a frontend mod_proxy (as discussed in Chapter 12). This is because mod_proxy doesn't break the connection to the backend when the user has aborted the connection.

If the reading of the request's data is completed and the code does its processing without writing anything back to the client, the broken connection won't be noticed. When an attempt is made to send at least one character to the client, the broken connection will be noticed and the `SIGPIPE` signal (Broken Pipe) will be sent to the process. The program can then halt its execution and perform all its cleanup requirements.

Prior to Apache 1.3.6, `SIGPIPE` was handled by Apache. Currently, Apache does not handle `SIGPIPE`, but mod_perl takes care of it.

Under mod_perl, `$r->print` (or just `print()`) returns a true value on success and a false value on failure. The latter usually happens when the connection is broken.

If you want behavior similar to the old `SIGPIPE` (as it was before Apache version 1.3.6), add the following configuration directive:

```
PerlFixupHandler Apache::SIG
```

When Apache's `SIGPIPE` handler is used, Perl may be left in the middle of its `eval()` context, causing bizarre errors when subsequent requests are handled by that child. When `Apache::SIG` is used, it installs a different `SIGPIPE` handler that rewinds the context to make sure Perl is in a normal state before the new request is served, preventing these bizarre errors. But in general, you don't need to use `Apache::SIG`.

If you use `Apache::SIG` and you would like to log when a request was canceled by a `SIGPIPE` in your Apache *access_log*, you must define a custom `LogFormat` in your *httpd.conf*. For example:

```
PerlFixupHandler Apache::SIG
LogFormat "%h %l %u %t \"%r\" %s %b %{SIGPIPE}e"
```

If the server has noticed that the request was canceled via a `SIGPIPE`, the log line will end with `1`. Otherwise, it will just be a dash. For example:

```
127.0.0.1 - - [09/Jan/2001:10:27:15 +0100]
"GET /perl/stopping_detector.pl HTTP/1.0" 200 16 1
127.0.0.1 - - [09/Jan/2001:10:28:18 +0100]
"GET /perl/test.pl HTTP/1.0"              200 10 -
```

Detecting Aborted Connections

Now let's use the knowledge we have acquired to trace the execution of the code and watch all the events as they happen. Let's take a simple `Apache::Registry` script that purposely hangs the server process, like the one in Example 6-28.

Example 6-28. stopping_detector.pl

```
my $r = shift;
$r->send_http_header('text/plain');

print "PID = $$\n";
$r->rflush;

while (1) {
    sleep 1;
}
```

The script gets a request object `$r` by `shift()`ing it from the `@_` argument list (passed by the `handler()` subroutine that was created on the fly by `Apache::Registry`). Then the script sends a `Content-Type` header telling the client that we are going to send a plain-text response.

Next, the script prints out a single line telling us the ID of the process that handled the request, which we need to know in order to run the tracing utility. Then we flush Apache's STDOUT buffer. If we don't flush the buffer, we will never see this information printed (our output is shorter than the buffer size used for print(), and the script intentionally hangs, so the buffer won't be auto-flushed).*

Then we enter an infinite while loop that does nothing but sleep(), emulating code that doesn't generate any output. For example, it might be a long-running mathematical calculation, a database query, or a search for extraterrestrial life.

Running *strace -p PID*, where *PID* is the process ID as printed on the browser, we see the following output printed every second:

```
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8)  = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8)  = 0
nanosleep({1, 0}, {1, 0})                 = 0
time([978969822])                         = 978969822
time([978969822])                         = 978969822
```

Alternatively, we can run the server in single-server mode. In single-server mode, we don't need to print the process ID, since the PID is the process of the single mod_perl process that we're running. When the process is started in the background, the shell program usually prints the PID of the process, as shown here:

```
panic% httpd -X &
 [1] 20107
```

Now we know what process we have to attach to with *strace* (or a similar utility):

```
panic% strace -p 20107
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8)  = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8)  = 0
nanosleep({1, 0}, {1, 0})                 = 0
time([978969822])                         = 978969822
time([978969822])                         = 978969822
```

We see the same output as before.

Let's leave *strace* running and press the Stop button. Did anything change? No, the same system calls trace is printed every second, which means that Apache didn't detect the broken connection.

Now we are going to write \0 (NULL) characters to the client in an attempt to detect the broken connection as soon as possible after the Stop button is pressed. Since these are NULL characters, they won't be seen in the output. Therefore, we modify the loop code in the following way:

* Buffering is used to reduce the number of system calls (which do the actual writing) and therefore improve performance. When the buffer (usually a few kilobytes in size) is getting full, it's flushed and the data is written.

```
    while (1) {
        $r->print("\0");
        last if $r->connection->aborted;
        sleep 1;
    }
```

We add a `print( )` statement to print a `NULL` character, then we check whether the connection was aborted, with the help of the `$r->connection->aborted` method. If the connection is broken, we break out of the loop.

We run this script and run *strace* on it as before, but we see that it still doesn't work—the script doesn't stop when the Stop button is pressed.

The problem is that we aren't flushing the buffer. The `NULL` characters won't be printed until the buffer is full and is autoflushed. Since we want to try writing to the connection pipe all the time, we add an `$r->rflush( )` call. Example 6-29 is a new version of the code.

Example 6-29. stopping_detector2.pl

```
my $r = shift;
$r->send_http_header('text/plain');

print "PID = $$\n";
$r->rflush;

while (1) {
    $r->print("\0");
    $r->rflush;
    last if $r->connection->aborted;
    sleep 1;
}
```

After starting the *strace* utility on the running process and pressing the Stop button, we see the following output:

```
    rt_sigprocmask(SIG_BLOCK, [CHLD], [ ], 8)  = 0
    rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
    rt_sigprocmask(SIG_SETMASK, [ ], NULL, 8)  = 0
    nanosleep({1, 0}, {1, 0})               = 0
    time([978970895])                       = 978970895
    alarm(300)                              = 0
    alarm(0)                                = 300
    write(3, "\0", 1)                       = -1 EPIPE (Broken pipe)
    --- SIGPIPE (Broken pipe) ---
    chdir("/usr/src/httpd_perl")            = 0
    select(4, [3], NULL, NULL, {0, 0})      = 1 (in [3], left {0, 0})
    time(NULL)                              = 978970895
    write(17, "127.0.0.1 - - [08/Jan/2001:19:21"..., 92) = 92
    gettimeofday({978970895, 554755}, NULL) = 0
    times({tms_utime=46, tms_stime=5, tms_cutime=0,
      tms_cstime=0}) = 8425400
    close(3)                                = 0
```

```
rt_sigaction(SIGUSR1, {0x8099524, [ ], SA_INTERRUPT|0x4000000},
  {SIG_IGN}, 8) = 0alarm(0)                                  = 0
rt_sigprocmask(SIG_BLOCK, NULL, [ ], 8)  = 0
rt_sigaction(SIGALRM, {0x8098168, [ ], SA_RESTART|0x4000000},
  {0x8098168, [ ], SA_INTERRUPT|0x4000000}, 8) = 0
fcntl(18, F_SETLKW, {type=F_WRLCK, whence=SEEK_SET,
  start=0, len=0}) = 0
```

Apache detects the broken pipe, as you can see from this snippet:

```
write(3, "\0", 1)                       = -1 EPIPE (Broken pipe)
--- SIGPIPE (Broken pipe) ---
```

Then it stops the script and does all the cleanup work, such as access logging:

```
write(17, "127.0.0.1 - - [08/Jan/2001:19:21"..., 92) = 92
```

where 17 is a file descriptor of the opened *access_log* file.

The Importance of Cleanup Code

Cleanup code is a critical issue with aborted scripts. For example, what happens to locked resources, if there are any? Will they be freed or not? If not, scripts using these resources and the same locking scheme might hang forever, waiting for these resources to be freed.

And what happens if a file was opened and never closed? In some cases, this might lead to a file-descriptor leakage. In the long run, many leaks of this kind might make your system unusable: when all file descriptors are used, the system will be unable to open new files.

First, let's take a step back and recall what the problems and solutions for these issues are under mod_cgi. Under mod_cgi, the resource-locking issue is a problem only if you use external lock files and use them for lock indication, instead of using `flock()`. If the script running under mod_cgi is aborted between the lock and the unlock code, and you didn't bother to write cleanup code to remove old, dead locks, you're in big trouble.

The solution is to place the cleanup code in an `END` block:

```
END {
    # code that ensures that locks are removed
}
```

When the script is aborted, Perl will run the `END` block while shutting down.

If you use `flock()`, things are much simpler, since all opened files will be closed when the script exits. When the file is closed, the lock is removed as well—all the locked resources are freed. There are systems where `flock()` is unavailable; on those systems, you can use Perl's emulation of this function.

With mod_perl, things can be more complex when you use global variables as file-handles. Because processes don't exit after processing a request, files won't be closed

unless you explicitly `close()` them or reopen them with the `open()` call, which first closes the file. Let's see what problems we might encounter and look at some possible solutions.

Critical section

First, we want to take a little detour to discuss the "critical section" issue. Let's start with a resource-locking scheme. A schematic representation of a proper locking technique is as follows:

1. Lock a resource

   <critical section starts>
2. Do something with the resource

   <critical section ends>
3. Unlock the resource

If the locking is exclusive, only one process can hold the resource at any given time, which means that all the other processes will have to wait. The code between the locking and unlocking functions cannot be interrupted and can therefore become a service bottleneck. That's why this code section is called critical. Its execution time should be as short as possible.

Even if you use a shared locking scheme, in which many processes are allowed to concurrently access the resource, it's still important to keep the critical section as short as possible, in case a process requires an exclusive lock.

Example 6-30 uses a shared lock but has a poorly designed critical section.

Example 6-30. critical_section_sh.pl

```
use Fcntl qw(:flock);
use Symbol;

my $fh = gensym;
open $fh, "/tmp/foo" or die $!;

# start critical section
flock $fh, LOCK_SH;  # shared lock, appropriate for reading
seek $fh, 0, 0;
my @lines = <$fh>;
for (@lines) {
    print if /foo/;
}
close $fh; # close unlocks the file
# end critical section
```

The code opens the file for reading, locks and rewinds it to the beginning, reads all the lines from the file, and prints out the lines that contain the string "foo".

The gensym( ) function imported by the Symbol module creates an anonymous glob data structure and returns a reference to it. Such a glob reference can be used as a file or directory handle. Therefore, it allows lexically scoped variables to be used as filehandles.

Fcntl imports file-locking symbols, such as LOCK_SH, LOCK_EX, and others with the :flock group tag, into the script's namespace. Refer to the Fcntl manpage for more information about these symbols.

If the file being read is big, it will take a relatively long time for this code to complete printing out the lines. During this time, the file remains open and locked with a shared lock. While other processes may access this file for reading, any process that wants to modify the file (which requires an exclusive lock) will be blocked waiting for this section to complete.

We can optimize the critical section as follows. Once the file has been read, we have all the information we need from it. To make the example simpler, we've chosen to just print out the matching lines. In reality, the code might be much longer.

We don't need the file to be open while the loop executes, because we don't access it inside the loop. Closing the file before we start the loop will allow other processes to obtain exclusive access to the file if they need it, instead of being blocked for no reason.

Example 6-31 is an improved version of the previous example, in which we only read the contents of the file during the critical section and process it afterward, without creating a possible bottleneck.

Example 6-31. critical_section_sh2.pl

```
use Fcntl qw(:flock);
use Symbol;

my $fh = gensym;
open $fh, "/tmp/foo" or die $!;

# start critical section
flock $fh, LOCK_SH;
seek $fh, 0, 0;
my @lines = <$fh>;
close $fh; # close unlocks the file
# end critical section

for (@lines) {
    print if /foo/;
}
```

Example 6-32 is a similar example that uses an exclusive lock. The script reads in a file and writes it back, prepending a number of new text lines to the head of the file.

Example 6-32. critical_section_ex.pl

```
use Fcntl qw(:flock);
use Symbol;

my $fh = gensym;
open $fh, "+>>/tmp/foo" or die $!;

# start critical section
flock $fh, LOCK_EX;
seek $fh, 0, 0;
my @add_lines =
  (
   qq{Complete documentation for Perl, including FAQ lists,\n},
   qq{should be found on this system using 'man perl' or\n},
   qq{'perldoc perl'. If you have access to the Internet, point\n},
   qq{your browser at http://www.perl.com/, the Perl Home Page.\n},
  );

my @lines = (@add_lines, <$fh>);
seek $fh, 0, 0;
truncate $fh, 0;
print $fh @lines;
close $fh; # close unlocks the file
# end critical section
```

Since we want to read the file, modify it, and write it back without anyone else changing it in between, we open it for reading and writing with the help of "+>>" and lock it with an exclusive lock. You cannot safely accomplish this task by opening the file first for reading and then reopening it for writing, since another process might change the file between the two events. (You could get away with "+<" as well; please refer to the *perlfunc* manpage for more information about the `open( )` function.)

Next, the code prepares the lines of text it wants to prepend to the head of the file and assigns them and the content of the file to the `@lines` array. Now we have our data ready to be written back to the file, so we `seek( )` to the start of the file and `truncate( )` it to zero size. Truncating is necessary when there's a chance the file might shrink. In our example, the file always grows, so in this case there is actually no need to truncate it; however, it's good practice to always use `truncate( )`, as you never know what changes your code might undergo in the future, and `truncate( )` doesn't significantly affect performance.

Finally, we write the data back to the file and close it, which unlocks it as well.

Did you notice that we created the text lines to be prepended as close to the place of usage as possible? This complies with good "locality of code" style, but it makes the critical section longer. In cases like this, you should sacrifice style in order to make the critical section as short as possible. An improved version of this script with a shorter critical section is shown in Example 6-33.

Example 6-33. critical_section_ex2.pl

```
use Fcntl qw(:flock);
use Symbol;

my @lines =
  (
   qq{Complete documentation for Perl, including FAQ lists,\n},
   qq{should be found on this system using 'man perl' or\n},
   qq{'perldoc perl'. If you have access to the Internet, point\n},
   qq{your browser at http://www.perl.com/, the Perl Home Page.\n},
  );

my $fh = gensym;
open $fh, "+>>/tmp/foo" or die $!;

# start critical section
flock $fh, LOCK_EX;
seek $fh, 0, 0;
push @lines, <$fh>;

seek $fh, 0, 0;
truncate $fh, 0;
print $fh @lines;
close $fh; # close unlocks the file
# end critical section
```

There are two important differences. First, we prepared the text lines to be prepended *before* the file is locked. Second, rather than creating a new array and copying lines from one array to another, we appended the file directly to the `@lines` array.

Safe resource locking and cleanup code

Now let's get back to this section's main issue, safe resource locking. If you don't make a habit of closing all files that you open, you may encounter many problems (unless you use the `Apache::PerlRun` handler, which does the cleanup for you). An open file that isn't closed can cause file-descriptor leakage. Since the number of file descriptors available is finite, at some point you will run out of them and your service will fail. This will happen quite fast on a heavily used server.

You can use system utilities to observe the opened and locked files, as well as the processes that have opened (and locked) the files. On FreeBSD, use the *fstat* utility. On many other Unix flavors, use *lsof*. On systems with a */proc* filesystem, you can see the opened file descriptors under */proc/PID/fd/*, where PID is the actual process ID.

However, file-descriptor leakage is nothing compared to the trouble you will give yourself if the code terminates and the file remains locked. Any other process requesting a lock on the same file (or resource) will wait indefinitely for it to become unlocked. Since this will not happen until the server reboots, all processes trying to use this resource will hang.

Example 6-34 is an example of such a terrible mistake.

Example 6-34. flock.pl

```
use Fcntl qw(:flock);
open IN, "+>>filename" or die "$!";
flock IN, LOCK_EX;
# do something
# quit without closing and unlocking the file
```

Is this safe code? No—we forgot to close the file. So let's add the `close()`, as in Example 6-35.

Example 6-35. flock2.pl

```
use Fcntl qw(:flock);
open IN, "+>>filename" or die "$!";
flock IN, LOCK_EX;
# do something
close IN;
```

Is it safe code now? Unfortunately, it is not. If the user aborts the request (for example, by pressing the browser's Stop or Reload buttons) during the critical section, the script will be aborted before it has had a chance to `close()` the file, which is just as bad as if we forgot to close it.

In fact, if the same process runs the same code again, an `open()` call will `close()` the file first, which will unlock the resource. This is because `IN` is a global variable. But it's quite possible that the process that created the lock will not serve the same request for a while, since it might be busy serving other requests. During that time, the file will be locked for other processes, making them hang. So relying on the same process to reopen the file is a bad idea.

This problem happens only if you use global variables as file handles. Example 6-36 has the same problem.

Example 6-36. flock3.pl

```
use Fcntl qw(:flock);
use Symbol ();
use vars qw($fh);
$fh = Symbol::gensym();
open $fh, "+>>filename" or die "$!";
flock $fh, LOCK_EX;
# do something
close $fh;
```

`$fh` is still a global variable, and therefore the code using it suffers from the same problem.

The simplest solution to this problem is to always use lexically scoped variables (created with `my()`). The lexically scoped variable will always go out of scope (assuming

that it's not used in a closure, as explained in the beginning of this chapter), whether the script gets aborted before close( ) is called or you simply forgot to close( ) the file. Therefore, if the file was locked, it will be closed and unlocked. Example 6-37 is a good version of the code.

Example 6-37. flock4.pl

```
use Fcntl qw(:flock);
use Symbol ();
my $fh = Symbol::gensym();
open $fh, "+>>filename" or die "$!";
flock $fh, LOCK_EX;
# do something
close $fh;
```

If you use this approach, please don't conclude that you don't have to close files anymore because they are automatically closed for you. Not closing files is bad style and should be avoided.

Note also that Perl 5.6 provides a Symbol.pm-like functionality as a built-in feature, so you can write:

```
open my $fh, ">/tmp/foo" or die $!;
```

and $fh will be automatically vivified as a valid filehandle. You don't need to use Symbol::gensym and Apache::gensym anymore, if backward compatibility is not a requirement.

You can also use IO::* modules, such as IO::File or IO::Dir. These are much bigger than the Symbol module (as a matter of fact, these modules use the Symbol module themselves) and are worth using for files or directories only if you are already using them for the other features they provide. Here is an example of their usage:

```
use IO::File;
use IO::Dir;
my $fh = IO::File->new(">filename");
my $dh = IO::Dir->new("dirname");
```

Alternatively, there are also the lighter FileHandle and DirHandle modules.

If you still have to use global filehandles, there are a few approaches you can take to clean up in the case of abnormal script termination.

If you are running under Apache::Registry and friends, the END block will perform the cleanup work for you. You can use END in the same way for scripts running under mod_cgi, or in plain Perl scripts. Just add the cleanup code to this block, and you are safe.

For example, if you work with DBM files, it's important to flush the DBM buffers by calling a sync( ) method:

```
END {
    # make sure that the DB is flushed
```

```
    $dbh->sync( );
}
```

Under mod_perl, the above code will work only for `Apache::Registry` and `Apache::PerlRun` scripts. Otherwise, execution of the `END` block is postponed until the process terminates. If you write a handler in the mod_perl API, use the `register_cleanup( )` method instead. It accepts a reference to a subroutine as an argument. You can rewrite the DBM synchronization code in this way:

```
$r->register_cleanup(sub { $dbh->sync( ) });
```

This will work under `Apache::Registry` as well.

Even better would be to check whether the client connection has been aborted. Otherwise, the cleanup code will always be executed, and for normally terminated scripts, this may not be what you want. To perform this check, use:

```
$r->register_cleanup(
  # make sure that the DB is flushed
  sub {
      $dbh->sync( ) if Apache->request->connection->aborted( );
  }
);
```

Or, if using an `END` block, use:

```
END {
    # make sure that the DB is flushed
    $dbh->sync( ) if Apache->request->connection->aborted( );
}
```

Note that if you use `register_cleanup( )`, it should be called at the beginning of the script or as soon as the variables you want to use in this code become available. If you use it at the end of the script, and the script happens to be aborted before this code is reached, no cleanup will be performed.

For example, `CGI.pm` registers a cleanup subroutine in its `new( )` method:

```
sub new {
  # code snipped
  if ($MOD_PERL) {
      Apache->request->register_cleanup(\&CGI::_reset_globals);
      undef $NPH;
  }
  # more code snipped
}
```

Another way to register a section of cleanup code for mod_perl API handlers is to use `PerlCleanupHandler` in the configuration file:

```
<Location /foo>
    SetHandler perl-script
    PerlHandler        Apache::MyModule
    PerlCleanupHandler Apache::MyModule::cleanup( )
    Options ExecCGI
</Location>
```

`Apache::MyModule::cleanup` performs the cleanup.

Handling Server Timeout Cases and Working with $SIG{ALRM}

Similar to the case where a user aborts the script execution by pressing the Stop button, the browser itself might abort the script if it hasn't returned any output after a certain timeout period (usually a few minutes).

Sometimes scripts perform very long operations that might take longer than the client's timeout.

This can happen when performing full searches of a large database with no full search support. Another example is a script interacting with external applications whose prompt reponse time isn't guaranteed. Consider a script that retrieves a page from another site and does some processing on it before it gets presented to the user. Obviously, nothing guarantees that the page will be retrieved fast, if at all.

In this situation, use `$SIG{ALRM}` to prevent the timeouts:

```
my $timeout = 10; # seconds
eval {
  local $SIG{ALRM} =
      sub { die "Sorry, timed out. Please try again\n" };
  alarm $timeout;
  # some operation that might take a long time to complete
  alarm 0;
};
die $@ if $@;
```

In this code, we run the operation that might take a long time to complete inside an `eval` block. First we initialize a localized `ALRM` signal handler, which resides inside the special `%SIG` hash. If this handler is triggered, it will call `die()`, and the `eval` block will be aborted. You can then do what you want with it—in our example, we chose to abort the execution of the script. In most cases, you will probably want to report to the user that the operation has timed out.

The actual operation is placed between two `alarm()` calls. The first call starts the clock, and the second cancels it. The clock is running for 10 seconds in our example. If the second `alarm()` call doesn't occur within 10 seconds, the `SIGALRM` signal is sent and the handler stored in `$SIG{ALRM}` is called. In our case, this will abort the `eval` block.

If the operation between the two `alarm()`s completes in under 10 seconds, the alarm clock is stopped and the `eval` block returns successfully, without triggering the `ALRM` handler.

Notice that only one timer can be used at a given time. `alarm( )`'s returned value is the amount of time remaining in the previous timer. So you can actually roughly measure the execution time as a side effect.

It is usually a mistake to intermix `alarm( )` and `sleep( )` calls. `sleep( )` may be internally implemented in your system with `alarm( )`, which will break your original `alarm( )` settings, since every new `alarm( )` call cancels the previous one.

Finally, the actual time resolution may be imprecise, with the timeout period being accurate to plus or minus one second. You may end up with a timeout that varies between 9 and 11 seconds. For granularity finer than one second, you can use Perl's four-argument version of `select( )`, leaving the first three arguments undefined. Other techniques exist, but they will not help with the task in question, in which we use `alarm( )` to implement timeouts.

Generating Correct HTTP Headers

An HTTP response header consists of at least two fields: HTTP response and MIME-type header `Content-Type`:

```
HTTP/1.0 200 OK
Content-Type: text/plain
```

After adding a newline, you can start printing the content. A more complete response includes the date timestamp and server type. For example:

```
HTTP/1.0 200 OK
Date: Tue, 10 Apr 2001 03:01:36 GMT
Server: Apache/1.3.19 (Unix) mod_perl/1.25
Content-Type: text/plain
```

To notify clients that the server is configured with `KeepAlive Off`, clients must be told that the connection will be closed after the content has been delivered:

```
Connection: close
```

There can be other headers as well, such as caching control headers and others specified by the HTTP protocol. You can code the response header with a single `print( )` statement:

```
print qq{HTTP/1.1 200 OK
  Date: Tue, 10 Apr 2001 03:01:36 GMT
  Server: Apache/1.3.19 (Unix) mod_perl/1.25
  Connection: close
  Content-Type: text/plain

  };
```

or with a "here"-style `print( )`:

```
print <<'EOT';
  HTTP/1.1 200 OK
  Date: Tue, 10 Apr 2001 03:01:36 GMT
```

```
Server: Apache/1.3.19 (Unix) mod_perl/1.25
Connection: close
Content-type: text/plain

EOT
```

Don't forget to include two newlines at the end of the HTTP header. With the help of `Apache::Util::ht_time()`, you can get the right timestamp string for the `Date:` field.

If you want to send non-default headers, use the `header_out()` method. For example:

```
$r->header_out("X-Server" => "Apache Next Generation 10.0");
$r->header_out("Date" => "Tue, 10 Apr 2001 03:01:36 GMT");
```

When the headers setting is completed, the `send_http_header()` method will flush the headers and add a newline to designate the start of the content.

```
$r->send_http_header;
```

Some headers have special aliases. For example:

```
$r->content_type('text/plain');
```

is the same as:

```
$r->header_out("Content-Type" => "text/plain");
```

but additionally sets some internal flags used by Apache. Whenever special-purpose methods are available, you should use those instead of setting the header directly.

A typical handler looks like this:

```
use Apache::Constants qw(OK);
$r->content_type('text/plain');
$r->send_http_header;
return OK if $r->header_only;
```

To be compliant with the HTTP protocol, if the client issues an HTTP `HEAD` request rather than the usual `GET`, we should send only the HTTP header, the document body. When Apache receives a `HEAD` request, `header_only()` returns true. Therefore, in our example the handler returns immediately after sending the headers.

In some cases, you can skip the explicit content-type setting if Apache figures out the right MIME type based on the request. For example, if the request is for an HTML file, the default *text/html* will be used as the content type of the response. Apache looks up the MIME type in the *mime.types* file. You can always override the default content type.

The situation is a little bit different with `Apache::Registry` and similar handlers. Consider a basic CGI script:

```
print "Content-type: text/plain\n\n";
print "Hello world";
```

By default, this won't work, because it looks like normal text, and no HTTP headers are sent. You may wish to change this by adding:

```
PerlSendHeader On
```

in the Apache::Registry <Location> section of your configuration. Now the response line and common headers will be sent in the same way they are by mod_cgi. Just as with mod_cgi, even if you set PerlSendHeader On, the script still needs to send the MIME type and a terminating double newline:

```
print "Content-type: text/html\n\n";
```

The PerlSendHeader On directive tells mod_perl to intercept anything that looks like a header line (such as Content-Type: text/plain) and automatically turn it into a correctly formatted HTTP header, much like CGI scripts running under mod_cgi. This feature allows you to keep your CGI scripts unmodified.

You can use $ENV{PERL_SEND_HEADER} to find out whether PerlSendHeader is On or Off.

```
if ($ENV{PERL_SEND_HEADER}) {
    print "Content-type: text/html\n\n";
}
else {
    my $r = Apache->request;
    $r->content_type('text/html');
    $r->send_http_header;
}
```

Note that you can always use the code in the else part of the above example, whether the PerlSendHeader directive is On or Off.

If you use CGI.pm's header( ) function to generate HTTP headers, you do not need to activate this directive because CGI.pm detects *mod_perl* and calls send_http_header( ) for you.

There is no free lunch—you get the mod_cgi behavior at the expense of the small but finite overhead of parsing the text that is sent. Note that mod_perl makes the assumption that individual headers are not split across print( ) statements.

The Apache::print( ) routine must gather up the headers that your script outputs in order to pass them to $r->send_http_header. This happens in src/modules/perl/Apache.xs (print( )) and Apache/Apache.pm (send_cgi_header( )). There is a shortcut in there—namely, the assumption that each print( ) statement contains one or more complete headers. If, for example, you generate a Set-Cookie header using multiple print( ) statements, like this:

```
print "Content-type: text/plain\n";
print "Set-Cookie: iscookietext\; ";
print "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
print "path=\/\; ";
print "domain=\.mmyserver.com\; ";
print "\n\n";
print "Hello";
```

the generated Set-Cookie header is split over a number of print( ) statements and gets lost. The above example won't work! Try this instead:

```
my $cookie = "Set-Cookie: iscookietext\; ";
$cookie .= "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
$cookie .= "path=\/\; ";
$cookie .= "domain=\.mmyserver.com\; ";
print "Content-type: text/plain\n",
print "$cookie\n\n";
print "Hello";
```

Using special-purpose cookie generator modules (for example, Apache::Cookie or CGI::Cookie) is an even cleaner solution.

Sometimes when you call a script you see an ugly "Content-Type: text/html" displayed at the top of the page, and often the HTML content isn't rendered correctly by the browser. As you have seen above, this generally happens when your code sends the headers twice.

If you have a complicated application in which the header might be sent from many different places depending on the code logic, you might want to write a special subroutine that sends a header and keeps track of whether the header has already been sent. You can use a global variable to flag that the header has already been sent, as shown in Example 6-38.

Example 6-38. send_header.pl

```
use strict;
use vars qw($header_printed);
$header_printed = 0;

print_header("text/plain");
print "It worked!\n";
print_header("text/plain");

sub print_header {
    return if $header_printed;

    my $type = shift || "text/html";
    $header_printed = 1;
    my $r = Apache->request;
    $r->content_type($type);
    $r->send_http_header;
}
1;
```

$header_printed serves as a Boolean variable, specifying whether the header was sent or not. It gets initialized to false (0) at the beginning of each code invocation. Note that the second invocation of print_header( ) within the same request will immediately return, since $header_printed will become true after print_header( ) is executed for the first time in the same request.

You can continue to improve this subroutine even further to handle additional headers, such as cookies.

Method Handlers: The Browse and See, Browse and View Example

Let's look at an example of the method-handler concepts presented in Chapter 4. Suppose you need to implement a handler that allows browsing the files in the document root and beneath. Directories should be browsable (so you can move up and down the directory tree), but files should not be viewable (so you can see the available files, but you cannot click to view them).

So let's write a simple file browser. We know what customers are like, so we suspect that the customer will ask for similar customized modules pretty soon. To avoid having to duplicate our work later, we decide to start writing a base class whose methods can easily be overridden as needed. Our base class is called `Apache::BrowseSee`.

We start the class by declaring the package and using the `strict` pragma:

```
package Apache::BrowseSee;
use strict;
```

Next, we import common constants (e.g., `OK`, `NOT_FOUND`, etc.), load the `File::Spec::Functions` and `File::Basename` modules, and import a few path-manipulation functions that we are going to use:

```
use Apache::Constants qw(:common);
use File::Spec::Functions qw(catdir canonpath curdir updir);
use File::Basename 'dirname';
```

Now let's look at the functions. We start with the simple constructor:

```
sub new { bless {}, shift;}
```

The real entry point, the handler, is prototyped as `($$)`. The handler starts by instantiating its object, if it hasn't already been done, and storing the `$r` object, so we don't need to pass it to the functions as an argument:

```
sub handler ($$) {
    my($self, $r) = @_;
    $self = $self->new unless ref $self;
    $self->{r} = $r;
```

Next we retrieve the `path_info` element of the request record:

```
    $self->{dir} = $r->path_info || '/';
```

For example, if the request was */browse/foo/bar*, where */browse* is the location of the handler, the `path_info` element will be */foo/bar*. The default value `/` is used when the path is not specified.

Then we reset the entries for *dirs* and *files*:

```
$self->{dirs}  = {};
$self->{files} = {};
```

This is needed because it's possible that the `$self` object is created outside the handler (e.g., in the startup file) and may persist between requests.

Now an attempt to fetch the contents of the directory is made:

```
eval { $self->fetch( ) };
return NOT_FOUND if $@;
```

If the `fetch( )` method dies, the error message is assigned to `$@` and we return `NOT_FOUND`. You may choose to approach it differently and return an error message explaining what has happened. You may also want to log the event before returning:

```
warn($@), return NOT_FOUND if $@;
```

Normally this shouldn't happen, unless a user messes with the arguments (something you should always be on the lookout for, because they *will* do it).

When the `fetch( )` function has completed successfully, all that's left is to send the HTTP header and start of the HTML via the `head( )` method, render the response, send the end of the HTML via `tail( )`,* and finally to return the `OK` constant to tell the server that the request has been fully answered:

```
    $self->head;
    $self->render;
    $self->tail;

    return OK;
}
```

The response is generated by three functions. The `head( )` method is a very simple one—it sends the HTTP header `text/html` and prints an HTML preamble using the current directory name as a title:

```
sub head {
    my $self = shift;
    $self->{r}->send_http_header("text/html");
    print "<html><head><title>Dir: $self->{dir}</title><head><body>";
}
```

The `tail( )` method finishes the HTML document:

```
sub tail {
    my $self = shift;
    print "</body></html>";
}
```

* This could perhaps be replaced by a templating system. See Appendix D for more information about the Template Toolkit.

The fetch( ) method reads the contents of the directory stored in the object's *dir* attribute (relative to the document root) and then sorts the contents into two groups, directories and files:

```
sub fetch {
    my $self = shift;
    my $doc_root = Apache->document_root;
    my $base_dir = canonpath( catdir($doc_root, $self->{dir}));

    my $base_entry = $self->{dir} eq '/' ? '' : $self->{dir};
    my $dh = Apache::gensym( );
    opendir $dh, $base_dir or die "Cannot open $base_dir: $!";
    for (readdir $dh) {
        next if $_ eq curdir( );  # usually '.'

        my $full_dir = catdir $base_dir, $_;
        my $entry = "$base_entry/$_";
        if (-d $full_dir) {
            if ($_ eq updir( )) { # '..'
                $entry = dirname $self->{dir};
                next if catdir($base_dir, $entry) eq $doc_root;
            }
            $self->{dirs}{$_} = $entry;
        }
        else {
            $self->{files}{$_} = $entry;
        }
    }
    closedir $dh;
}
```

By using canonpath( ), we make sure that nobody messes with the path_info element, by eliminating successive slashes and "/."s on Unix and taking appropriate actions on other operating systems. It's important to use File::Spec and other cross-platform functions when developing applications.

While looping through the directory entries, we skip over the current directory entry using the curdir( ) function imported from File::Spec::Functions (which is equivalent to . on Unix) and handle the parent directory entry specially by matching the updir( ) function (which is equivalent to .. on Unix). The function dirname( ) gives us the parent directory, and afterward we check that this directory is different from the document root. If it's the same, we skip this entry.

Note that since we use the path_info element to pass the directory relative to the document root, we rely on Apache to handle the case when users try to mess with the URL and add .. to reach files they aren't supposed to reach.

Finally, let's look at the render( ) method:

```
sub render {
    my $self = shift;
    print "<p>Current Directory: <i>$self->{dir}</i><br>";
```

```
        my $location = $self->{r}->location;
        print qq{<a href="$location$self->{dirs}{$_}">$_</a><br>}
            for sort keys %{ $self->{dirs} || {} };
        print qq{$_<br>}
            for sort keys %{ $self->{files} || {} };
    }
```

The `render( )` method actually takes the files and directories prepared in the `fetch( )` method and displays them to the user. First the name of the current directory is displayed, followed by the directories and finally the files. Since the module should allow browsing of directories, we hyperlink them. The files aren't linked, since we are in "see but don't touch" mode.*

Finally, we finish the package with `1;` to make sure that the module will be successfully loaded. The `__END__` token allows us to put various notes and POD documentation after the program, where Perl won't complain about them.

```
    1;
    __END__
```

Example 6-39 shows how the whole package looks.

Example 6-39. Apache/BrowseSee.pm

```
package Apache::BrowseSee;
use strict;

use Apache::Constants qw(:common);
use File::Spec::Functions qw(catdir canonpath curdir updir);
use File::Basename 'dirname';

sub new { bless {}, shift;}

sub handler ($$) {
    my($self, $r) = @_;
    $self = $self->new unless ref $self;

    $self->{r}     = $r;
    $self->{dir}   = $r->path_info || '/';
    $self->{dirs}  = {};
    $self->{files} = {};

    eval { $self->fetch( ) };
    return NOT_FOUND if $@;

    $self->head;
    $self->render;
    $self->tail;
```

* In your real code you should also escape HTML- and URI-unsafe characters in the filenames (e.g., <, >, &, ", ', etc.) by using the `Apache::Util::escape_html` and `Apache::Util::escape_uri` functions.

Example 6-39. Apache/BrowseSee.pm (continued)

```
    return OK;
}

sub head {
    my $self = shift;
    $self->{r}->send_http_header("text/html");
    print "<html><head><title>Dir: $self->{dir}</title><head><body>";
}

sub tail {
    my $self = shift;
    print "</body></html>";
}

sub fetch {
    my $self = shift;
    my $doc_root = Apache->document_root;
    my $base_dir = canonpath( catdir($doc_root, $self->{dir}));

    my $base_entry = $self->{dir} eq '/' ? '' : $self->{dir};
    my $dh = Apache::gensym( );
    opendir $dh, $base_dir or die "Cannot open $base_dir: $!";
    for (readdir $dh) {
        next if $_ eq curdir( );

        my $full_dir = catdir $base_dir, $_;
        my $entry = "$base_entry/$_";
        if (-d $full_dir) {
            if ($_ eq updir( )) {
                $entry = dirname $self->{dir};
                next if catdir($base_dir, $entry) eq $doc_root;
            }
            $self->{dirs}{$_} = $entry;
        }
        else {
            $self->{files}{$_} = $entry;
        }
    }
    closedir $dh;
}

sub render {
    my $self = shift;
    print "Current Directory: <i>$self->{dir}</i><br>";

    my $location = $self->{r}->location;
    print qq{<a href="$location$self->{dirs}{$_}">$_</a><br>}
        for sort keys %{ $self->{dirs} || {} };
    print qq{$_<br>}
        for sort keys %{ $self->{files} || {} };
}
```

Example 6-39. Apache/BrowseSee.pm (continued)

```
1;
__END__
```

This module should be saved as *Apache/BrowseSee.pm* and placed into one of the directories in `@INC`. For example, if */home/httpd/perl* is in your `@INC`, you can save it in */home/httpd/perl/Apache/BrowseSee.pm*.

To configure this module, we just add the following snippet to *httpd.conf*:

```
PerlModule Apache::BrowseSee
<Location /browse>
    SetHandler perl-script
    PerlHandler Apache::BrowseSee->handler
</Location>
```

Users accessing the server from */browse* can now browse the contents of your server from the document root and beneath but cannot view the contents of the files (see Figure 6-2).

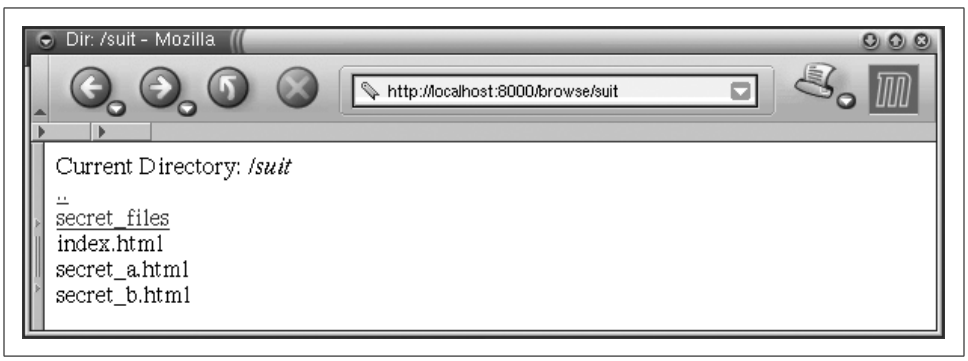

Figure 6-2. The files can be browsed but not viewed

Now let's say that as soon as we get the module up and running, the client comes back and tells us he would like us to implement a very similar application, except that files should now be viewable (clickable). This is because later he wants to allow only authorized users to read the files while letting everybody see what he has to offer.

We knew that was coming, remember? Since we are lazy and it's not exciting to write the same code again and again, we will do the minimum amount of work while still keeping the client happy. This time we are going to implement the `Apache::BrowseRead` module:

```
package Apache::BrowseRead;
use strict;
use base qw(Apache::BrowseSee);
```

We place the new module into *Apache/BrowseRead.pm*, declare a new package, and tell Perl that this package inherits from `Apache::BrowseSee` using the `base` pragma. The last line is roughly equivalent to:

```
BEGIN {
    require Apache::BrowseSee;
    @Apache::BrowseRead::ISA = qw(Apache::BrowseSee);
}
```

Since this class is going to do the same job as `Apache::BrowseSee`, apart from rendering the file listings differently, all we have to do is override the `render( )` method:

```
sub render {
    my $self = shift;
    print "<p>Current Directory: <i>$self->{dir}</i><br>";

    my $location = $self->{r}->location;
    print qq{<a href="$location$self->{dirs}{$_}">$_</a><br>}
        for sort keys %{ $self->{dirs} || {} };
    print  qq{<a href="$self->{files}{$_}">$_</a><br>}
        for sort keys %{ $self->{files} || {} };
}
```

As you can see, the only difference here is that we link to the real files now.

We complete the package as usual with `1;` and `__END__`:

```
1;
__END__
```

Example 6-40 shows the whole package.

Example 6-40. Apache/BrowseRead.pm

```
package Apache::BrowseRead;
use strict;
use base qw(Apache::BrowseSee);

sub render {
    my $self = shift;
    print "<p>Current Directory: <i>$self->{dir}</i><br>";

    my $location = $self->{r}->location;
    print qq{<a href="$location$self->{dirs}{$_}">$_</a><br>}
        for sort keys %{ $self->{dirs} || {} };
    print  qq{<a href="$self->{files}{$_}">$_</a><br>}
        for sort keys %{ $self->{files} || {} };
}
1;
__END__
```

Finally, we should add a new configuration section in *httpd.conf*:

```
PerlModule Apache::BrowseRead
<Location /read>
    SetHandler perl-script
    PerlHandler Apache::BrowseRead->handler
</Location>
```

Now, when accessing files through */read*, we can browse and view the contents of the files (see Figure 6-3). Once we add some authentication/authorization methods, we will have a server where everybody can browse, but only privileged users can read.

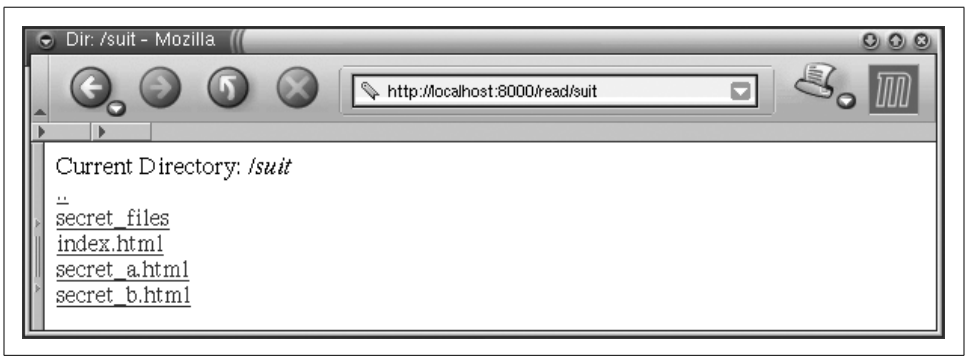

Figure 6-3. The files can be browsed and read

You might be wondering why you would write a special module to do something Apache itself can already do for you. First, this was an example on using method handlers, so we tried to keep it simple while showing some real code. Second, this example can easily be adapted and extended—for example, it can handle virtual files that don't exist on the filesystem but rather are generated on the fly and/or fetched from the database, and it can easily be changed to do whatever *you* (or your client) want to do, instead of what Apache allows.

References

- "Just the FAQs: Coping with Scoping," an article by Mark-Jason Dominus about how Perl handles variables and namespaces, and the difference between `use vars()` and `my()`: *http://www.plover.com/~mjd/perl/FAQs/Namespaces.html*.
- It's important to know how to perform exception handling in Perl code. Exception handling is a general Perl technique; it's not mod_perl-specific. Further information is available in the documentation for the following modules:
  - `Error.pm`, by Graham Barr.
  - `Exception::Class` and `Devel::StackTrace`, by Dave Rolsky.

— `Try.pm`, by Tony Olekshy, available at *http://www.avrasoft.com/perl6/try6-ref5.txt*.

— There is also a great deal of information concerning error handling in the mod_perl online documentation (e.g., *http://perl.apache.org/docs/general/perl_reference/perl_reference.html*).

- Perl Module Mechanics: *http://world.std.com/~swmcd/steven/perl/module_mechanics.html*.

  This page describes the mechanics of creating, compiling, releasing, and maintaining Perl modules, which any mod_perl developer planning on sharing code with others will find useful.