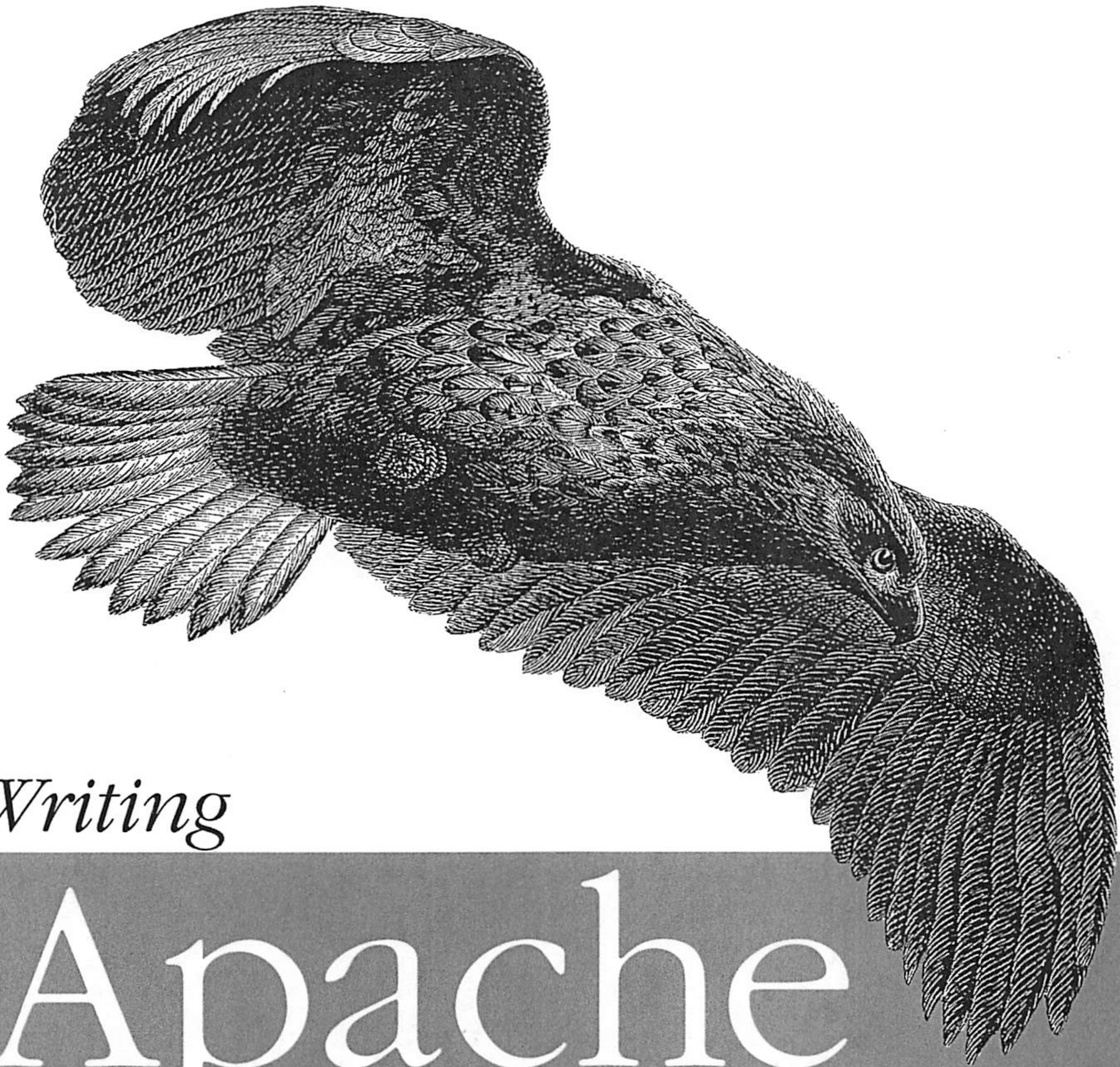
The Apache API and mod_perl



Writing Apache Modules *with Perl and C*

O'REILLY®

Lincoln Stein & Doug MacEachern

AT&T - Exhibit 1008

Writing Apache Modules with Perl and C

Lincoln Stein and Doug MacEachern

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Writing Apache Modules with Perl and C
by Lincoln Stein and Doug MacEachern

Copyright © 1999 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Linda Mui

Production Editor: Melanie Wang

Printing History:

March 1999: First Edition.

The association between the image of a white-tailed eagle and the topic of Apache modules is a trademark of O'Reilly & Associates, Inc. Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-565-92567-0
[LSI]

[2011-11-04]

In this chapter:

- *Content Handlers as File Processors*
- *Virtual Documents*
- *Redirection*
- *Processing Input*
- *Apache::Registry*
- *Handling Errors*
- *Chaining Content Handlers*
- *Method Handlers*

4

Content Handlers

This chapter is about writing content handlers for the Apache response phase, when the contents of the page are actually produced. In this chapter you'll learn how to produce dynamic pages from thin air, how to modify real documents on the fly to produce effects like server-side includes, and how Apache interacts with the MIME-typing system to select which handler to invoke.

Starting with this chapter we shift to using the Apache Perl API exclusively for code examples and function prototypes. The Perl API covers the majority of what C programmers need to use the C-language API. What's missing are various memory management functions that are essential to C programmers but irrelevant in Perl. If you are a C programmer, just have patience and the missing pieces will be filled in eventually. In the meantime, follow along with the Perl examples and enjoy yourself. Maybe you'll even become a convert.

Content Handlers as File Processors

Early web servers were designed as engines for transmitting physical files from the host machine to the browser. Even though Apache does much more, the file-oriented legacy still remains. Files can be sent to the browser unmodified or passed through content handlers to transform them in various ways before sending them on to the browser. Even though many of the documents that you produce with modules have no corresponding physical files, some parts of Apache still behave as if they did.

When Apache receives a request, the URI is passed through any URI translation handlers that may be installed (see Chapter 7, *Other Request Phases*, for information on how to roll your own), transforming it into a file path. The *mod_alias* translation handler (compiled in by default) will first process any *Alias*, *ScriptAlias*,

Redirect, or other *mod_alias* directives. If none applies, the *http_core* default translator will simply prepend the *DocumentRoot* directory to the beginning of the URI.

Next, Apache attempts to divide the file path into two parts: a "filename" part which usually (but not always) corresponds to a physical file on the host's filesystem, and an "additional path information" part corresponding to additional stuff that follows the filename. Apache divides the path using a very simple-minded algorithm. It steps through the path components from left to right until it finds something that doesn't correspond to a directory on the host machine. The part of the path up to and including this component becomes the filename, and everything that's left over becomes the additional path information.

Consider a site with a document root of */home/www* that has just received a request for URI */abc/def/ghi*. The way Apache splits the file path into filename and path information parts depends on what directories it finds in the document root:

Physical Directory	Translated Filename	Additional Path Information
/home/www	*/home/www/abc*	*/def/ghi*
/home/www/abc	*/home/www/abc/def*	*/ghi*
/home/www/abc/def	*/home/www/abc/def/ghi*	empty
/home/www/abc/def/ghi	*/home/www/abc/def/ghi*	empty

Note that the presence of any actual files in the path is irrelevant to this process. The division between the filename and the path information depends only on what directories are present.

Once Apache has decided where the file is in the path, it determines what MIME type it might be. This is again one of the places where you can intervene to alter the process with a custom type handler. The default type handler (*mod_mime*) just compares the filename's extension to a table of MIME types. If there's a match, this becomes the MIME type. If no match is found, then the MIME type is undefined. Again, note that this mapping from filename to MIME type occurs even when there's no actual file there.

There are two special cases. If the last component of the filename happens to be a physical directory, then Apache internally assigns it a "magic" MIME type, defined by the `DIR_MAGIC_TYPE` constant as *httpd/unix-directory*. This is used by the directory module to generate automatic directory listings. The second special case occurs when you have the optional *mod_mime_magic* module installed and the file actually exists. In this case Apache will peek at the first few bytes of the file's contents to determine what type of file it might be. Chapter 7 shows you how to write your own MIME type checker handlers to implement more sophisticated MIME type determination schemes.

After Apache has determined the name and type of the file referenced by the URI, it decides what to do about it. One way is to use information hard-wired into the module's static data structures. The module's `handler_rec` table, which we describe in detail in Chapter 10, *C API Reference Guide, Part I*, declares the module's willingness to handle one or more magic MIME types and associates a content handler with each one. For example, the *mod_cgi* module associates MIME type *application/x-httpd-cgi* with its *cgi_handler()* handler subroutine. When Apache detects that a filename is of type *application/x-httpd-cgi* it invokes *cgi_handler()* and passes it information about the file. A module can also declare its desire to handle an ordinary MIME type, such as *video/quicktime*, or even a wildcard type, such as *video/**. In this case, all requests for URIs with matching MIME types will be passed through the module's content handler unless some other module registers a more specific type.

Newer modules use a more flexible method in which content handlers are associated with files at runtime using explicit names. When this method is used, the module declares one or more content handler names in its `handler_rec` array instead of, or in addition to, MIME types. Some examples of content handler names you might have seen include *cgi-script*, *server-info*, *server-parsed*, *imap-file*, and *perl-script*. Handler names can be associated with files using either *AddHandler* or *SetHandler* directives. *AddHandler* associates a handler with a particular file extension. For example, a typical configuration file will contain this line to associate *.shtml* files with the server-side include handler:

```
AddHandler server-parsed .shtml
```

Now, the *server-parsed* handler defined by *mod_include* will be called on to process all files ending in ".shtml" regardless of their MIME type.

SetHandler is used within *<Directory>*, *<Location>*, and *<Files>* sections to associate a particular handler with an entire section of the site's URI space. In the two examples that follow, the *<Location>* section attaches the *server-parsed* method to all files within the virtual directory */shtml*, while the *<Files>* section attaches *imap-file* to all files that begin with the prefix "map-":

```
<Location /shtml>
  SetHandler server-parsed
</Location>

<Files map-*>
  SetHandler imap-file
</Files>
```

Surprisingly, the *AddHandler* and *SetHandler* directives are not actually implemented in the Apache core. They are implemented by the standard *mod_actions*

module, which is compiled into the server by default. In Chapter 7, we show how to reimplement *mod_actions* using the Perl API.

You'll probably want to use explicitly named content handlers in your modules rather than hardcoded MIME types. Explicit handler names make configuration files cleaner and easier to understand. Plus, you don't have to invent a new magic MIME type every time you add a handler.

Things are slightly different for *mod_perl* users because *two* directives are needed to assign a content handler to a directory or file. The reason for this is that the only real content handler defined by *mod_perl* is its internal *perl-script* handler. You use *SetHandler* to assign *perl-script* the responsibility for a directory or partial URI, and then use a *PerlHandler* directive to tell the *perl-script* handler which Perl module to execute. Directories supervised by Perl API content handlers will look something like this:

```
<Location /graph>
   SetHandler  perl-script
   PerlHandler Apache::Graph
</Location>
```

Don't try to assign *perl-script* to a file extension using something like `AddHandler perl-script .pl`; this is generally useless because you'd need to set *PerlHandler* too. If you'd like to associate a Perl content handler with an extension, you should use the *<Files>* directive. Here's an example:

```
<Files ~ "\.graph$">
   SetHandler  perl-script
   PerlHandler Apache::Graph
</Files>
```

There is no *UnSetHandler* directive to undo the effects of *SetHandler*. However, should you ever need to restore a subdirectory's handler to the default, you can do it with the directive `SetHandler default-handler`, as follows:

```
<Location /graph/tutorial>
   SetHandler default-handler
</Location>
```

Adding a Canned Footer to Pages

To show you how content handlers work, we'll develop a module with the Perl API that adds a canned footer to all pages in a particular directory. You could use this, for example, to automatically add copyright information and a link back to the home page. Later on, we'll turn this module into a full-featured navigation bar.

Example 4-1 gives the code for *Apache::Footer*, and Figure 4-1 shows a screenshot of it in action. Since this is our first substantial module, we'll step through the code section by section.

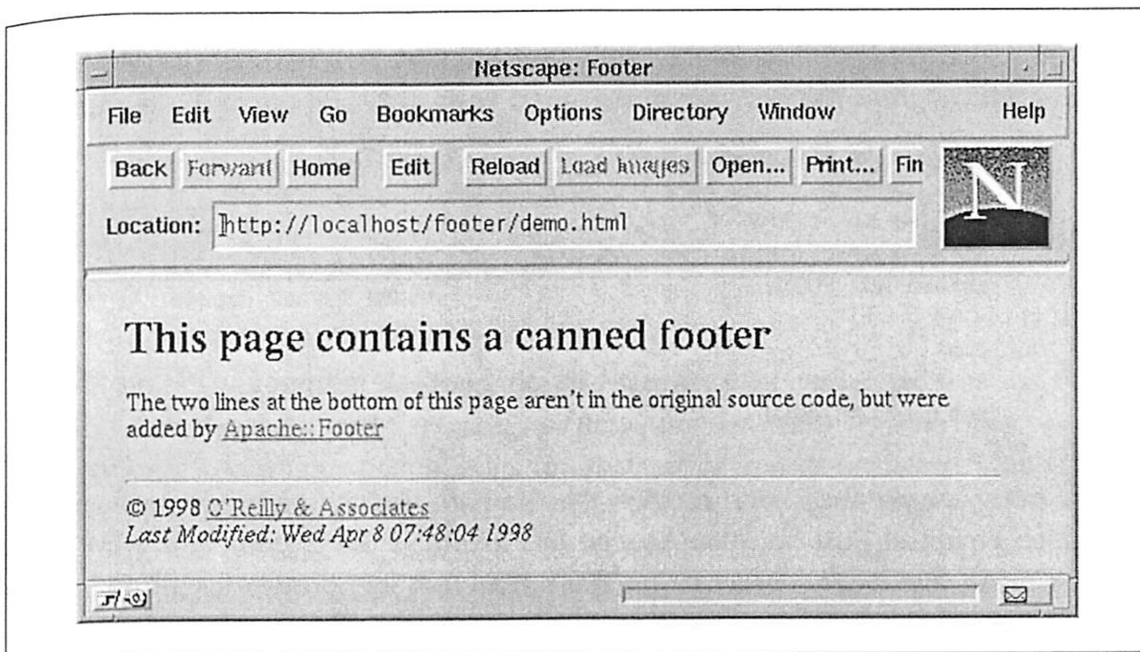


Figure 4-1. The footer on this page was generated automatically by Apache::Footer.

```
package Apache::Footer;

use strict;
use Apache::Constants qw(:common);
use Apache::File ();
```

The code begins by declaring its package name and loading various Perl modules that it depends on. The *use strict* pragma activates Perl checks that prevent us from using global variables before declaring them, disallows the use of function calls without the parentheses, and prevents other unsafe practices. The *Apache::Constants* module defines constants for the various Apache and HTTP result codes; we bring in only those constants that belong to the frequently used *:common* set. *Apache::File* defines methods that are useful for manipulating files.

```
sub handler {
    my $r = shift;
    return DECLINED unless $r->content_type() eq 'text/html';
```

The *handler()* subroutine does all the work of generating the content. It is roughly divided into three parts. In the first part, it fetches information about the requested file and decides whether it wants to handle it. In the second part, it creates the canned footer dynamically from information that it gleans about the file. In the third part, it rewrites the file to include the footer.

In the first part of the process, the handler retrieves the Apache request object and stores it in `$r`. Next it calls the request's *content_type()* method to retrieve its MIME type. Unless the document is of type *text/html*, the handler stops here and returns a `DECLINED` result code to the server. This tells Apache to pass the

document on to any other handlers that have declared their willingness to handle this type of document. In most cases, this means that the document or image will be passed through to the browser in the usual way.

```
my $file = $r->filename;

unless (-e $r->finfo) {
   $r->log_error("File does not exist: $file");
   return NOT_FOUND;
}
unless (-r _) {
   $r->log_error("File permissions deny access: $file");
   return FORBIDDEN;
}
```

At this point we go ahead and recover the file path, by calling the request object's *filename()* method. Just because Apache has assigned the document a MIME type doesn't mean that it actually exists or, if it exists, that its permissions allow it to be read by the current process. The next two blocks of code check for these cases. Using the Perl *–e* file test, we check whether the file exists. If not, we log an error to the server log using the request object's *log_error()* method and return a result code of NOT_FOUND. This will cause the server to return a page displaying the 404 "Not Found" error (exactly what's displayed is under the control of the *Error-Document* directive).

There are several ways to perform file status checks in the Perl API. The simplest way is to recover the file's pathname using the request object's *filename()* method, and pass the result to the Perl *–e* file test:

```
unless (-e $r->filename) {
   $r->log_error("File does not exist: $file");
   return NOT_FOUND;
}
```

A more efficient way, however, is to take advantage of the fact that during its path walking operation Apache already performed a system *stat()* call to collect filesystem information on the file. The resulting status structure is stored in the request object and can be retrieved with the object's *finfo()* method. So the more efficient idiom is to use the test `-e $r->finfo`.

Once *finfo()* is called, the *stat()* information is stored into the magic Perl filehandle _ and can be used for subsequent file testing and *stat()* operations, saving even more CPU time. Using the _ filehandle, we next test that the file is readable by the current process and return `FORBIDDEN` if this isn't the case. This displays a 403 "Forbidden" error.

```
my $modtime = localtime((stat _)[9]);
```

After performing these tests, we get the file modification time by calling *stat()*. We can use the _ filehandle here too, avoiding the overhead of repeating the *stat()* system call. The modification time is passed to the built-in Perl *localtime()* function to convert it into a human-readable string.

```
    my $fh;
    unless ($fh = Apache::File->new($file)) {
       $r->log_error("Couldn't open $file for reading: $!");
       return SERVER_ERROR;
    }
```

At this point, we attempt to open the file for reading using *Apache::File*'s *new()* method. For the most part, *Apache::File* acts just like Perl's *IO::File* object-oriented I/O package, returning a filehandle on success or *undef* on failure. Since we've already handled the two failure modes that we know how to deal with, we return a result code of SERVER_ERROR if the open is unsuccessful. This immediately aborts all processing of the document and causes Apache to display a 500 "Internal Server Error" message.

```
    my $footer = <<END;
<hr>
&copy; 1998 <a href="http://www.ora.com/">O'Reilly &amp; Associates</a><br>
<em>Last Modified: $modtime</em>
END
```

Having successfully opened the file, we build the footer. The footer in this example script is entirely static, except for the document modification date that is computed on the fly.

```
    $r->send_http_header;

    while (<$fh>) {
       s!(</BODY>)!$footer$1!oi;
    } continue {
       $r->print($_);
    }
```

The last phase is to rewrite the document. First we tell Apache to send the HTTP header. There's no need to set the content type first because it already has the appropriate value. We then loop through the document looking for the closing </BODY> tag. When we find it, we use a substitution statement to insert the footer in front of it. The possibly modified line is now sent to the browser using the request object's *print()* method.

```
    return OK;
}

1;
```

At the end, we return an OK result code to Apache and end the handler subroutine definition. Like any other *.pm* file, the module itself must end by returning a true value (usually 1) to signal Perl that it compiled correctly.

If all this checking for the existence and readability of the file before processing seems a bit pedantic, don't worry. It's actually unnecessary for you to do this. Instead of explicitly checking the file, we could have simply returned DECLINED if the attempt to open the file failed. Apache would then pass the URI to the default file handler which will perform its own checks and display the appropriate error messages. Therefore we could have replaced the file tests with the single line:

```
my $fh = Apache::File->new($file) || return DECLINED;
```

Doing the tests inside the module this way makes the checks explicit and gives us a chance to intervene to rescue the situation. For example, we might choose to search for a text file of the same name and present it instead. The explicit tests also improve module performance slightly, since the system wastes a small amount of CPU time when it attempts to open a nonexistent file. If most of the files the module serves do exist, however, this penalty won't be significant.

Example 4-1. Adding a Canned Footer to HTML Pages

```
package Apache::Footer;
# file: Apache/Footer.pm

use strict;
use Apache::Constants qw(:common);
use Apache::File ();

sub handler {
    my $r = shift;
    return DECLINED unless $r->content_type() eq 'text/html';

    my $file = $r->filename;

    unless (-e $r->finfo) {
       $r->log_error("File does not exist: $file");
       return NOT_FOUND;
    }
    unless (-r _) {
       $r->log_error("File permissions deny access: $file");
       return FORBIDDEN;
    }

    my $modtime = localtime((stat _)[9]);

    my $fh;
    unless ($fh = Apache::File->new($file)) {
       $r->log_error("Couldn't open $file for reading: $!");
       return SERVER_ERROR;
    }
```

Example 4-1. Adding a Canned Footer to HTML Pages (continued)

```
    my $footer = <<END;
<hr>
&copy; 1998 <a href=">http://www.ora.com/">O'Reilly &amp; Associates</a><br>
<em>Last Modified: $modtime</em>
END

    $r->send_http_header;

    while (<$fh>) {
       s!(</BODY>)!$footer$1!oi;
    } continue {
       $r->print($_);
    }

    return OK;
}

1;
__END__
```

There are several ways to install and use the *Apache::Footer* content handler. If all the files that needed footers were gathered in one place in the directory tree, you would probably want to attach *Apache::Footer* to that location:

```
<Location /footer>
  SetHandler perl-script
  PerlHandler Apache::Footer
</Location>
```

If the files were scattered about the document tree, it might be more convenient to map *Apache::Footer* to a unique filename extension, such as *.footer*. To achieve this, the following directives would suffice:

```
AddType text/html .footer
<Files ~ "\.footer$">
   SetHandler  perl-script
   PerlHandler Apache::Footer
</Files>
```

Note that it's important to associate MIME type *text/html* with the new extension; otherwise, Apache won't be able to determine its content type during the MIME type checking phase.

If your server is set up to allow per-directory access control files to include file information directives, you can place any of these handler directives inside a *.htaccess* file. This allows you to change handlers without restarting the server. For example, you could replace the *<Location>* section shown earlier with a *.htaccess* file in the directory where you want the footer module to be active:

```
SetHandler perl-script
PerlHandler Apache::Footer
```

A Server-Side Include System

The obvious limitation of the *Apache::Footer* example is that the footer text is hardcoded into the code. Changing the footer becomes a nontrivial task, and using different footers for various parts of the site becomes impractical. A much more flexible solution is provided by Vivek Khera's *Apache::Sandwich* module. This module "sandwiches" HTML pages between canned headers and footers that are determined by runtime configuration directives. The *Apache::Sandwich* module also avoids the overhead of parsing the request document; it simply uses the subrequest mechanism to send the header, body, and footer files in sequence.

We can provide more power than *Apache::Sandwich* by using server-side includes. Server-side includes are small snippets of code embedded within HTML comments. For example, in the standard server-side includes that are implemented in Apache, you can insert the current time and date into the page with a comment that looks like this:

```
Today is <!--#echo var="DATE_LOCAL"-->.
```

In this section, we use *mod_perl* to develop our own system of server-side includes, using a simple but extensible scheme that lets you add new types of includes at a moment's whim. The basic idea is that HTML authors will create files that contain comments of this form:

```
<!--#DIRECTIVE PARAM1 PARAM2 PARAM3 PARAM4...-->
```

A directive name consists of any sequence of alphanumeric characters or underscores. This is followed by a series of optional parameters, separated by spaces or commas. Parameters that contain whitespace must be enclosed in single or double quotes in shell command style. Backslash escapes also work in the expected manner.

The directives themselves are not hardcoded into the module but are instead dynamically loaded from one or more configuration files created by the site administrator. This allows the administrator to create a standard menu of includes that are available to the site's HTML authors. Each directive is a short Perl subroutine. A simple directive looks like this one:

```
sub HELLO { "Hello World!"; }
```

This defines a subroutine named *HELLO()* that returns the string "Hello World!" A document can now include the string in its text with a comment formatted like this one:

```
I said <!--#HELLO-->
```

A more complex subroutine will need access to the Apache object and the server-side include parameters. To accommodate this, the Apache object is passed as the first function argument, and the server-side include parameters, if any, follow.

Here's a function definition that returns any field from the incoming request's HTTP header, using the Apache object's *header_in()* method:

```
sub HTTP_HEADER {
  my ($r,$field) = @_;
  $r->header_in($field);
}
```

With this subroutine definition in place, HTML authors can insert the *User-Agent* field into their document using a comment like this one:

```
You are using the browser <!-- #HTTP_HEADER User-Agent -->.
```

Example 4-2 shows an HTML file that uses a few of these includes, and Figure 4-2 shows what the page looks like after processing.

Example 4-2. An HTML File That Uses Extended Server-Side Includes

```
<html> <head> <title>Server-Side Includes</title></head>
<body bgcolor=white>
<h1>Server-Side Includes</h1>
This is some straight text.<p>

This is a "<!-- #HELLO -->" include.<p>

The file size is <strong><!-- #FSIZE --></strong>, and it was
last modified on <!-- #MODTIME %x --><p>

Today is <!-- #DATE "%A, in <em>anno domini</em> %Y"-->.<p>

The user agent is <em><!--#HTTP_HEADER User-Agent--></em>.<p>

Oops: <!--#OOPS 0--><p>

Here is an included file:
<pre>
<!--#INCLUDE /include.txt 1-->
</pre>

<!--#FOOTER-->
</body> </html>
```

Implementing this type of server-side include system might seem to be something of a challenge, but in fact the code is surprisingly compact (Example 4-3). This module is named *Apache::ESSI,* for "extensible server-side includes."

Again, we'll step through the code one section at a time.

```
package Apache::ESSI;

use strict;
use Apache::Constants qw(:common);
use Apache::File ();
use Text::ParseWords qw(quotewords);
my (%MODIFIED, %SUBSTITUTION);
```

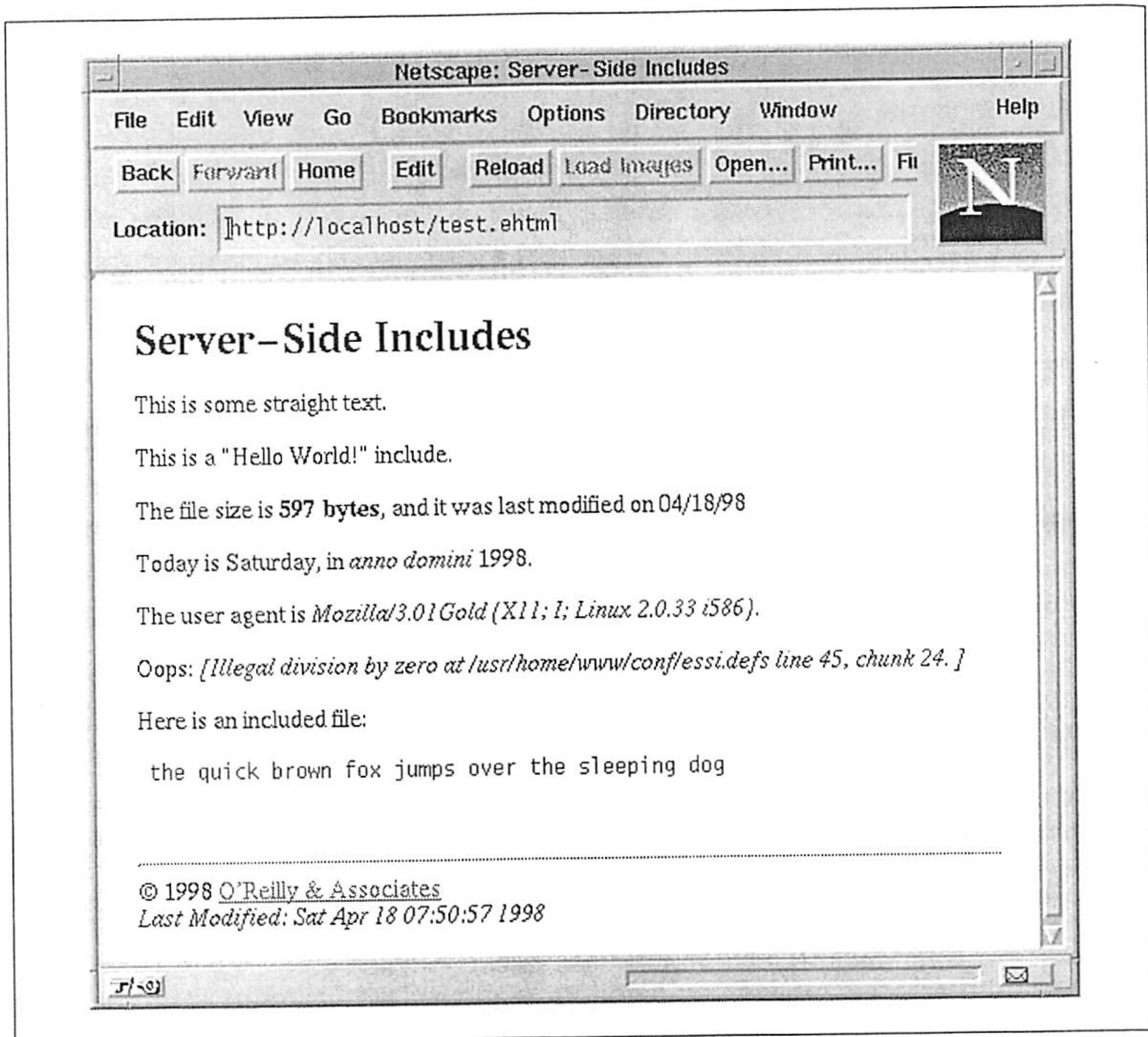


Figure 4-2. A page generated by Apache::ESSI

We start as before by declaring the package name and loading various Perl library modules. In addition to the modules that we loaded in the *Apache::Footer* example, we import the *quotewords()* function from the standard Perl *Text::ParseWords* module. This routine provides command shell–like parsing of strings that contain quote marks and backslash escapes. We also define two lexical variables, `%MODIFIED` and `%SUBSTITUTION`, which are global to the package.

```
sub handler {
    my $r = shift;
    $r->content_type() eq 'text/html' || return DECLINED;
    my $fh = Apache::File->new($r->filename) || return DECLINED;
    my $sub = read_definitions($r)     || return SERVER_ERROR;
    $r->send_http_header;
    $r->print($sub->($r, $fh));
    return OK;
}
```

The *handler()* subroutine is quite short. As in the *Apache::Footer* example, *handler()* starts by examining the content type of the document being requested and declines to handle requests for non-HTML documents. The handler recovers the file's physical path by calling the request object's *filename()* method and attempts to open it. If the file open fails, the handler again returns an error code of `DECLINED`. This avoids *Apache::Footer*'s tedious checking of the file's existence and access permissions, at the cost of some efficiency every time a nonexistent file is requested.

Once the file is opened, we call an internal function named *read_definitions()*. This function reads the server-side includes configuration file and generates an anonymous subroutine to do the actual processing of the document. If an error occurs while processing the configuration file, *read_definitions()* returns *undef* and we return `SERVER_ERROR` in order to abort the transaction. Otherwise, we send the HTTP header and invoke the anonymous subroutine to perform the substitutions on the contents of the file. The result of invoking the subroutine is sent to the client using the request object's *print()* method, and we return a result code of `OK` to indicate that everything went smoothly.

```
sub read_definitions {
    my $r = shift;
    my $def = $r->dir_config('ESSIDefs');
    return unless $def;
    return unless -e ($def = $r->server_root_relative($def));
```

Most of the interesting work occurs in *read_definitions()*. The idea here is to read the server-side include definitions, compile them, and then use them to generate an anonymous subroutine that does the actual substitutions. In order to avoid recompiling this subroutine unnecessarily, we cache its code reference in the package variable `%SUBSTITUTION` and reuse it if we can.

The *read_definitions()* subroutine begins by retrieving the path to the file that contains the server-side include definitions. This information is contained in a per-directory configuration variable named `ESSIDefs`, which is set in the configuration file using the *PerlSetVar* directive and retrieved within the handler with the request object's *dir_config()* method (see the end of the example for a representative configuration file entry). If, for some reason, this variable isn't present, we return *undef*. Like other Apache configuration files, we allow this file to be specified as either an absolute path or a partial path relative to the server root. We pass the path to the request object's *server_root_relative()* method. This convenient function prepends the server root to relative paths and leaves absolute paths alone. We next check that the file exists using the *–e* file test operator and return *undef* if not.

```
    return $SUBSTITUTION{$def} if $MODIFIED{$def} && $MODIFIED{$def} <= -M _;
```

Having recovered the name of the definitions file, we next check the cache to see whether the subroutine definitions are already cached and, if so, whether the file hasn't changed since the code was compiled and cached. We use two hashes for this purpose. The %SUBSTITUTION array holds the compiled code and %MODIFIED contains the modification date of the definition file the last time it was compiled. Both hashes are indexed by the definition file's path, allowing the module to handle the case in which several server-side include definition files are used for different parts of the document tree. If the modification time listed in %MODIFIED is less than or equal to the definition file's current modification date, we return the cached subroutine.

```
my $package = join "::", __PACKAGE__, $def;
$package =~ tr/a-zA-Z0-9_/_/c;
```

The next two lines are concerned with finding a unique namespace in which to compile the server-side include functions. Putting the functions in their own namespace decreases the chance that function side effects will have unwanted effects elsewhere in the module. We take the easy way out here by using the path to the definition file to synthesize a package name, which we store in a variable named $package.

```
eval "package $package; do '$def'";
if($@) {
   $r->log_error("Eval of $def did not return true: $@");
   return;
}
```

We then invoke *eval()* to compile the subroutine definitions into the newly chosen namespace. We use the *package* declaration to set the namespace and *do* to load and run the definitions file. We use *do* here rather than the more common *require* because *do* unconditionally recompiles code files even if they have been loaded previously. If the *eval* was unsuccessful, we log an error and return *undef.*

```
    $SUBSTITUTION{$def} = sub {
        do_substitutions($package, @_);
    };
    $MODIFIED{$def} = -M $def;  # store modification date
    return $SUBSTITUTION{$def};
}
```

Before we exit *read_definitions()*, we create a new anonymous subroutine that invokes the *do_substitutions()* function, store this subroutine in %SUBSTITUTION, and update %MODIFIED with the modification date of the definitions file. We then return the code reference to our caller. We interpose a new anonymous subroutine here so that we can add the contents of the $package variable to the list of variables passed to the *do_substitutions()* function.

```
sub do_substitutions {
    my $package = shift;
```

```
    my($r, $fh) = @_;
    # Make sure that eval() errors aren't trapped.
    local $SIG{__WARN__};
    local $SIG{__DIE__};
    local $/; #slurp $fh
    my $data = <$fh>;
    $data =~ s/<!--\s*\#(\w+)  # start of a function name
               \s*(.*?)        # optional parameters
               \s*-->          # end of comment
             /call_sub($package, $1, $r, $2)/xseg;
    $data;
}
```

When *handler()* invokes the anonymous subroutine, it calls *do_substitutions()* to do the replacement of the server-side include directives with the output of their corresponding routines. We start off by localizing the `$SIG{__WARN__}` and `$SIG{__DIE__}` handlers and setting them back to the default Perl *CORE::warn()* and *CORE::die()* subroutines. This is a paranoid precaution against the use of *CGI::Carp*, which some *mod_perl* users load into Apache during the startup phase in order to produce nicely formatted server error log messages. The subroutine continues by fetching the lines of the page to be processed and joining them in a single scalar value named `$data`.

We then invoke a string substitution function to replace properly formatted comment strings with the results of invoking the corresponding server-side include function. The substitution uses the *e* flag to treat the replacement part as a Perl expression to be evaluated and the *g* flag to perform the search and replace globally. The search half of the function looks like this:

```
/<!--\s*\#(\w+)\s*(.*?)\s*-->/
```

This detects the server-side include comments while capturing the directive name in `$1` and its optional arguments in `$2`.

The replacement of the function looks like this:

```
/call_sub($package, $1, $r, $2)/
```

This just invokes another utility function, *call_sub()*, passing it the package name, the directive name, the request object, and the list of parameters.

```
sub call_sub {
    my($package, $name, $r, $args) = @_;
    my $sub = \&{join '::', $package, $name};
    $r->chdir_file;
    my $res = eval { $sub->($r, quotewords('[ ,]',0,$args)) };
    return "<em>[$@]</em>" if $@;
    return $res;
}
```

The *call_sub()* routine starts off by obtaining a reference to the subroutine using its fully qualified name. It does this by joining the package name to the subroutine

name and then using the funky Perl `\&{...}` syntax to turn this string into a subroutine reference. As a convenience to the HTML author, before invoking the subroutine we call the request object's *chdir_file()* method. This simply makes the current directory the same as the requested file, which in this case is the HTML file containing the server-side includes.

The server-side include function is now invoked, passing it the request object and the optional arguments. We call *quotewords()* to split up the arguments on commas or whitespace. In order to trap fatal runtime errors that might occur during the function's execution, the call is done inside an *eval{}* block. If the call function fails, we return the error message it died with captured within `$@`. Otherwise, we return the value of the call function.

At the bottom of Example 4-3 is an example entry for *perl.conf* (or *httpd.conf* if you prefer). The idea here is to make *Apache::ESSI* the content handler for all files ending with the extension *.ehtml*. We do this with a *<Files>* configuration section that contains the appropriate *SetHandler* and *PerlHandler* directives. We use the *PerlSetVar* directive to point the module to the server-relative definitions file, *conf/essi.defs*.

In addition to the *<Files>* section, we need to ensure that Apache knows that *.ehtml* files are just a special type of HTML file. We use *AddType* to tell Apache to treat *.ehtml* files as MIME type *text/html*.

You could also use *<Location>* or *<Directory>* to assign the *Apache::ESSI* content handler to a section of the document tree, or a different *<Files>* directive to make *Apache::ESSI* the content handler for all HTML files.

Example 4-3. An Extensible Server-Side Include System

```
package Apache::ESSI;
# file: Apache/ESSI.pm

use strict;
use Apache::Constants qw(:common);
use Apache::File ();
use Text::ParseWords qw(quotewords);
my (%MODIFIED, %SUBSTITUTION);

sub handler {
    my $r = shift;
    $r->content_type() eq 'text/html' || return DECLINED;
    my $fh = Apache::File->new($r->filename) || return DECLINED;
    my $sub = read_definitions($r)    || return SERVER_ERROR;
    $r->send_http_header;
    $r->print($sub->($r, $fh));
    return OK;
}
```

Example 4-3. An Extensible Server-Side Include System (continued)

```
sub read_definitions {
    my $r = shift;
    my $def = $r->dir_config('ESSIDefs');
    return unless $def;
    return unless -e ($def = $r->server_root_relative($def));
    return $SUBSTITUTION{$def} if $MODIFIED{$def} && $MODIFIED{$def} <= -M _;

    my $package = join "::", __PACKAGE__, $def;
    $package =~ tr/a-zA-Z0-9_/_/c;

    eval "package $package; do '$def'";

    if($@) {
       $r->log_error("Eval of $def did not return true: $@");
       return;
    }

    $SUBSTITUTION{$def} = sub {
        do_substitutions($package, @_);
    };

    $MODIFIED{$def} = -M $def;  # store modification date
    return $SUBSTITUTION{$def};
}

sub do_substitutions {
    my $package = shift;
    my($r, $fh) = @_;
    # Make sure that eval() errors aren't trapped.
    local $SIG{__WARN__};
    local $SIG{__DIE__};
    local $/; #slurp $fh
    my $data = <$fh>;
    $data =~ s/<!--\s*\#(\w+) # start of a function name
               \s*(.*?)       # optional parameters
               \s*-->         # end of comment
              /call_sub($package, $1, $r, $2)/xseg;
    $data;
}

sub call_sub {
    my($package, $name, $r, $args) = @_;
    my $sub = \&{join '::', $package, $name};
    $r->chdir_file;
    my $res = eval { $sub->($r, quotewords('[ ,]',0,$args)) };
    return "<em>[$@]</em>" if $@;
    return $res;
}

1;
__END__
```

Here are some *perl.conf* directives to go with *Apache::ESSI*:

```
<Files ~ "\.ehtml$">
  SetHandler  perl-script
  PerlHandler Apache::ESSI
  PerlSetVar  ESSIDefs conf/essi.defs
</Files>
AddType text/html .ehtml
```

At this point you'd probably like a complete server-side include definitions file to go with the module. Example 4-4 gives a short file that defines a core set of functions that you can build on top of. Among the functions defined here are ones for inserting the size and modification date of the current file, the date, fields from the browser's HTTP request header, and a function that acts like the C preprocessor *#include* macro to insert the contents of a file into the current document. There's also an include called *OOPS* which divides the number 10 by the argument you provide. Pass it an argument of zero to see how runtime errors are handled.

The *INCLUDE()* function inserts whole files into the current document. It accepts either a physical pathname or a "virtual" path in URI space. A physical path is only allowed if it lives in or below the current directory. This is to avoid exposing sensitive files such as */etc/passwd.*

If the `$virtual` flag is passed, the function translates from URI space to a physical path name using the *lookup_uri()* and *filename()* methods:

```
$file = $r->lookup_uri($path)->filename;
```

The request object's *lookup_uri()* method creates an Apache subrequest for the specified URI. During the subrequest, Apache does all the processing that it ordinarily would on a real incoming request up to, but not including, activating the content handler. *lookup_uri()* returns an *Apache::SubRequest* object, which inherits all its behavior from the Apache request class. We then call this object's *filename()* method in order to retrieve its translated physical file name.

Example 4-4. Server-Side Include Function Definitions

```
# Definitions for server-side includes.
# This file is require'd, and therefore must end with
# a true value.

use Apache::File ();
use Apache::Util qw(ht_time size_string);

# insert the string "Hello World!"
sub HELLO {
    my $r = shift;
    "Hello World!";
}
```

Example 4-4. Server-Side Include Function Definitions (continued)

```
# insert today's date possibly modified by a strftime() format
# string
sub DATE {
    my ($r,$format) = @_;
    return scalar(localtime) unless $format;
    return ht_time(time, $format, 0);
}

# insert the modification time of the document, possibly modified
# by a strftime() format string.
sub MODTIME {
    my ($r,$format) = @_;
    my $mtime = (stat $r->finfo)[9];
    return localtime($mtime) unless $format;
    return ht_time($mtime, $format, 0);
}

# insert the size of the current document
sub FSIZE {
    my $r = shift;
    return size_string -s $r->finfo;
}

# divide 10 by the argument (used to test runtime error trapping)
sub OOPS { 10/$_[1]; }

# insert a canned footer
sub FOOTER {
    my $r = shift;
    my $modtime = MODTIME($r);
    return <<END;
<hr>
&copy; 1998 <a href="http://www.ora.com/">O'Reilly &amp; Associates</a><br>
<em>Last Modified: $modtime</em>
END
}

# insert the named field from the incoming request
sub HTTP_HEADER {
    my ($r,$h) = @_;
    $r->header_in($h);
}

#ensure that path is relative, and does not contain ".."
sub is_below_only { $_[0] !~ m:(^/|(^|/)\.\.(/|$)): }

# Insert the contents of a file.  If the $virtual flag is set
# does a document-root lookup, otherwise treats filename as a
# physical path.
sub INCLUDE {
    my ($r,$path,$virtual) = @_;
    my $file;
```

Example 4-4. Server-Side Include Function Definitions (continued)

```
    if($virtual) {
       $file = $r->lookup_uri($path)->filename;
    }
    else {
       unless(is_below_only($path)) {
           die "Can't include $path\n";
       }
       $file = $path;
    }
    my $fh = Apache::File->new($file) || die "Couldn't open $file: $!\n";
    local $/;
    return <$fh>;
}

1;
```

If you're a fan of server-side includes, you should also check out the Apache *Embperl* and *ePerl* packages. Both packages, along with several others available from the CPAN, build on *mod_perl* to create a Perl-like programming language embedded entirely within server-side includes.

Converting Image Formats

Another useful application of Apache content handlers is converting file formats on the fly. For example, with a little help from the Aladdin Ghostscript interpreter, you can dynamically convert Adobe Acrobat (PDF) files into GIF images when dealing with a browser that doesn't have the Acrobat plug-in installed.*

In this section, we show a content handler that converts image files on the fly. It takes advantage of Kyle Shorter's *Image::Magick* package, the Perl interface to John Cristy's ImageMagick library. *Image::Magick* interconverts a large number of image formats, including JPEG, PNG, TIFF, GIF, MPEG, PPM, and even PostScript. It can also transform images in various ways, such as cropping, rotating, solarizing, sharpening, sampling, and blurring.

The *Apache::Magick* content handler accepts URIs in this form:

```
/path/to/image.ext/Filter1/Filter2?arg=value&arg=value...
```

* At least in theory, you can divine what MIME types a browser prefers by examining the contents of the *Accept* header with `$r->header_in('Accept')`. According to the HTTP protocol, this should return a list of MIME types that the browser can handle along with a numeric preference score. The CGI.pm module even has an *accept()* function that leverages this information to choose the best format for a given document type. Unfortunately, this part of the HTTP protocol has atrophied, and neither Netscape's nor Microsoft's browsers give enough information in the *Accept* header to make it useful for content negotiation.

In its simplest form, the handler can be used to perform image format conversions on the fly. For example, if the actual file is named *bluebird.gif* and you request *bluebird.jpg*, the content handler automatically converts the GIF into a JPEG file and returns it. You can also pass arguments to the converter in the query string. For example, to specify a progressive JPEG image (`interlace = "Line"`) with a quality of 50 percent, you can fetch the file by requesting a URI like this one:

```
/images/bluebird.jpg?interlace=Line&quality=50
```

You can also run one or more filters on the image prior to the conversion. For example, to apply the "Charcoal" filter (which makes the image look like a charcoal sketch) and then put a decorative border around it (the "Frame" filter), you can request the image like this:

```
/images/bluebird.jpg/Charcoal/Frame?quality=75
```

Any named arguments that need to be passed to the filter can be appended to the query string, along with the conversion arguments. In the last example, we can specify a gold-colored frame this way:

```
/images/bluebird.jpg/Charcoal/Frame?quality=75&color=gold
```

This API doesn't allow you to direct arguments to specific filters. Fortunately, most of the filters that you might want to apply together don't have overlapping argument names, and filters ignore any arguments that don't apply to them. The full list of filters and conversion operations can be found at the PerlMagick web site, located at *http://www.wizards.dupont.com/cristy/www/perl.html*. You'll find pointers to the latest ImageMagick code library there as well.

One warning before you use this Apache module on your system: some of the operations can be very CPU-intensive, particularly when converting an image with many colors, such as JPEG, to one that has few colors, such as GIF. You should also be prepared for *Image::Magick*'s memory consumption, which is nothing short of voracious.

Example 4-5 shows the code for *Apache::Magick*.

```
package Apache::Magick;

use strict;
use Apache::Constants qw(:common);
use Image::Magick ();
use Apache::File ();
use File::Basename qw(fileparse);
use DirHandle ();
```

We begin as usual by bringing in the modules we need. We bring in *Apache::Constants*, *File::Basename* for its file path parsing utilities, *DirHandle()* for object-oriented interface to directory reading functions, and the *Image::Magick* module itself.

```
my %LegalArguments = map { $_ => 1 }
qw (adjoin background bordercolor colormap colorspace
    colors compress density dispose delay dither
    display font format iterations interlace
    loop magick mattecolor monochrome page pointsize
    preview_type quality scene subimage subrange
    size tile texture treedepth undercolor);

my %LegalFilters = map { $_ => 1 }
qw(AddNoise Blur Border Charcoal Chop
   Contrast Crop Colorize Comment CycleColormap
   Despeckle Draw Edge Emboss Enhance Equalize Flip Flop
   Frame Gamma Implode Label Layer Magnify Map Minify
   Modulate Negate Normalize OilPaint Opaque Quantize
   Raise ReduceNoise Rotate Sample Scale Segment Shade
   Sharpen Shear Solarize Spread Swirl Texture Transparent
   Threshold Trim Wave Zoom);
```

We then define two hashes, one for all the filter and conversion arguments recognized by *Image::Magick* and the other for the various filter operations that are available. These lists were cut and pasted from the *Image::Magick* documentation. We tried to exclude the ones that were not relevant to this module, such as ones that create multiframe animations, but a few may have slipped through.

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

    # get the name of the requested file
    my $file = $r->filename;

    # If the file exists and there are no transformation arguments
    # just decline the transaction.  It will be handled as usual.
    return DECLINED unless $r->args || $r->path_info || !-r $r->finfo;
```

The *handler()* routine begins as usual by fetching the name of the requested file. We decline to handle the transaction if the file exists, the query string is empty, and the additional path information is empty as well. This is just the common case of the browser trying to fetch an unmodified existing file.

```
    my $source;
    my ($base, $directory, $extension) = fileparse($file, '\.\w+');
    if (-r $r->finfo) { # file exists, so it becomes the source
       $source = $file;
    }
    else {              # file doesn't exist, so we search for it
       return DECLINED unless -r $directory;
       $source = find_image($r, $directory, $base);
    }

    unless ($source) {
       $r->log_error("Couldn't find a replacement for $file");
       return NOT_FOUND;
    }
```

We now use *File::Basename*'s *fileparse()* function to parse the requested file into its basename (the filename without the extension), the directory name, and the extension. We check again whether we can read the file, and if so it becomes the source for the conversion. Otherwise, we search the directory for another image file to convert into the format of the requested file. For example, if the URI requested is *bluebird.jpeg* and we find a file named *bluebird.gif*, we invoke *Image::Magick* to do the conversion. The search is done by an internal subroutine named *find_image()*, which we'll examine later. If successful, the name of the source image is stored in `$source`. If unsuccessful, we log the error with the *log_error()* function and return a `NOT_FOUND` result code.

```
$r->send_http_header;
return OK if $r->header_only;
```

At this point, we send the HTTP header using *send_http_header()*. The next line represents an optimization that we haven't seen before. It may be that the client isn't interested in the content of the image file, but just in its meta-information, such as its length and MIME type. In this case, the browser sends an HTTP HEAD request rather than the usual GET. When Apache receives a HEAD request, it sets *header_only()* to true. If we see that this has happened, we return from the handler immediately with an `OK` status code. Although it wouldn't hurt to send the document body anyway, respecting the HEAD request results in a slight savings in processing efficiency and makes the module compliant with the HTTP protocol.

```
my $q = Image::Magick->new;
my $err = $q->Read($source);
```

Otherwise, it's time to read the source image into memory. We create a new *Image::Magick* object, store it in a variable named `$q`, and then load the source image file by calling its *Read()* method. Any error message returned by *Read()* is stored into a variable called `$err`.

```
my %arguments = $r->args;

# Run the filters
for (split '/', $r->path_info) {
   my $filter = ucfirst $_;
   next unless $LegalFilters{$filter};
   $err ||= $q->$filter(%arguments);
}

# Remove invalid arguments before the conversion
for (keys %arguments) {
   delete $arguments{$_} unless $LegalArguments{$_};
}
```

The next phase of the process is to prepare for the image manipulation. The first thing we do is tidy up the input parameters. We retrieve the query string parameters by calling the request object's *args()* method and store them in a hash named `%arguments`.

We then call the request object's *path_info()* method to retrieve the additional path information. We split the path info into a series of filter names and canonicalize them by capitalizing their initial letters using the Perl built-in operator *ucfirst()*. Each of the filters is applied in turn, skipping over any that aren't on the list of filters that *Image::Magick* accepts. We do an OR assignment into `$err`, so that we maintain the first non-null error message, if any. Having run the files, we remove from the `%arguments` array any arguments that aren't valid in *Image::Magick*'s file format conversion calls.

```
# Create a temporary file name to use for conversion
my($tmpnam, $fh) = Apache::File->tmpfile;
```

Image::Magick needs to write the image to a temporary file. We call the *Apache::File tmpfile()* method to create a suitable temporary file name. If successful, *tmpfile()* returns the name of the temporary file, which we store in the variable `$tmpnam`, and a filehandle open for writing into the file, which we store in the variable `$fh`. The *tmpfile()* method is specially written to avoid a "race condition" in which the temporary file name appears to be unused when the module first checks for it but is created by someone else before it can be opened.

```
# Write out the modified image
open(STDOUT, ">&=" . fileno($fh));
```

The next task is to have *Image::Magick* perform the requested conversion and write it to the temporary file. The safest way to do this would be to pass it the temporary file's already opened filehandle. Unfortunately, *Image::Magick* doesn't accept filehandles; its *Write()* method expects a filename, or the special filename – to write to standard output. However, we can trick it into writing to the filehandle by reopening standard output on the filehandle, which we do by passing the filehandle's numeric file descriptor to *open()* using the rarely seen >&= notation. See the *open()* entry in the *perlfunc* manual page for complete details.

Since STDOUT gets reset before every Perl API transaction, there's no need to save and restore its original value.

```
$extension =~ s/^\.//;
$err ||= $q->Write('filename' => "\U$extension\L:-", %arguments);
if ($err) {
   unlink $tmpnam;
   $r->log_error($err);
   return SERVER_ERROR;
}
close $fh;
```

We now call *Image::Magick*'s *Write()* method with the argument `'filename'=> EXTENSION:-` where *EXTENSION* is the uppercased extension of the document that the remote user requested. We also tack on any conversion arguments that

were requested. For example, if the remote user requested `bluebird.jpg?quality=75`, the call to *Write()* ends up looking like this:

```
$q->Write('filename'=>'JPG:-','quality'=>75);
```

If any errors occurred during this step or the previous ones, we delete the temporary file, log the errors, and return a SERVER_ERROR status code.

```
    # At this point the conversion is all done!
    # reopen for reading
    $fh = Apache::File->new($tmpnam);
    unless ($fh) {
       $r->log_error("Couldn't open $tmpnam: $!");
       return SERVER_ERROR;
    }

    # send the file
    $r->send_fd($fh);

    # clean up and go
    unlink $tmpnam;
    return OK;
}
```

If the call to *Write()* was successful, we need to send the contents of the temporary file to the waiting browser. We could open the file, read its contents, and send it off using a series of *print()* calls, as we've done previously, but in this case there's a slightly easier way. After reopening the file with *Apache::File*'s *new()* method, we call the request object's *send_fd()* method to transmit the contents of the filehandle in one step. The *send_fd()* method accepts all the same filehandle data types as the Perl built-in I/O operators. After sending off the file, we clean up by unlinking the temporary file and returning an OK status.

We'll now turn our attention to the *find_image()* subroutine, which is responsible for searching the directory for a suitable file to use as the image source if the requested file can't be found:

```
sub find_image {
    my ($r, $directory, $base) = @_;
    my $dh = DirHandle->new($directory) or return;
```

The *find_image()* utility subroutine is straightforward. It takes the request object, the parsed directory name, and the basename of the requested file and attempts to search this directory for an image file that shares the same basename. The routine opens a directory handle with *DirHandle->new()* and iterates over its entries.

```
    my $source;
    for my $entry ($dh->read) {
       my $candidate = fileparse($entry, '\.\w+');
       if ($base eq $candidate) {
```

```
                # determine whether this is an image file
                $source = join '', $directory, $entry;
                my $subr = $r->lookup_file($source);
                last if $subr->content_type =~ m:^image/:;
                undef $source;
            }
        }
```

For each entry in the directory listing, we parse out the basename using *fileparse()*. If the basename is identical to the one we're searching for, we call the request object's *lookup_file()* method to activate an Apache subrequest. *lookup_file()* is similar to *lookup_uri()*, which we saw earlier in the context of server-side includes, except that it accepts a physical pathname rather than a URI. Because of this, *lookup_file()* will skip the URI translation phase, but it will still cause Apache to trigger all the various handlers up to, but not including, the content handler.

In this case, we're using the subrequest for the sole purpose of getting at the MIME type of the file. If the file is indeed an image of one sort or another, then we save the request in a lexical variable and exit the loop. Otherwise, we keep searching.

```
        $dh->close;
        return $source;
    }
```

At the end of the loop, `$source` will be undefined if no suitable image file was found, or it will contain the full pathname to the image file if we were successful. We close the directory handle, and return `$source`.

Example 4-5. Apache::Magick Converts Image Formats on the Fly

```
package Apache::Magick;
# file: Apache/Magick.pm

use strict;
use Apache::Constants qw(:common);
use Image::Magick ();
use Apache::File ();
use File::Basename qw(fileparse);
use DirHandle ();

my %LegalArguments = map { $_ => 1 }
qw (adjoin background bordercolor colormap colorspace
    colors compress density dispose delay dither
    display font format iterations interlace
    loop magick mattecolor monochrome page pointsize
    preview_type quality scene subimage subrange
    size tile texture treedepth undercolor);

my %LegalFilters = map { $_ => 1 }
qw(AddNoise Blur Border Charcoal Chop
   Contrast Crop Colorize Comment CycleColormap
   Despeckle Draw Edge Emboss Enhance Equalize Flip Flop
```

Example 4-5. Apache::Magick Converts Image Formats on the Fly (continued)

```
   Frame Gamma Implode Label Layer Magnify Map Minify
   Modulate Negate Normalize OilPaint Opaque Quantize
   Raise ReduceNoise Rotate Sample Scale Segment Shade
   Sharpen Shear Solarize Spread Swirl Texture Transparent
   Threshold Trim Wave Zoom);

sub handler {
    my $r = shift;

    # get the name of the requested file
    my $file = $r->filename;

    # If the file exists and there are no transformation arguments
    # just decline the transaction.  It will be handled as usual.
    return DECLINED unless $r->args || $r->path_info || !-r $r->finfo;

    my $source;
    my ($base, $directory, $extension) = fileparse($file, '\.\w+');
    if (-r $r->finfo) { # file exists, so it becomes the source
       $source = $file;
    }
    else {              # file doesn't exist, so we search for it
       return DECLINED unless -r $directory;
       $source = find_image($r, $directory, $base);
    }

    unless ($source) {
       $r->log_error("Couldn't find a replacement for $file");
       return NOT_FOUND;
    }

    $r->send_http_header;
    return OK if $r->header_only;

    # Read the image
    my $q = Image::Magick->new;
    my $err = $q->Read($source);

    # Conversion arguments are kept in the query string, and the
    # image filter operations are kept in the path info
    my %arguments = $r->args;

    # Run the filters
    for (split '/', $r->path_info) {
       my $filter = ucfirst $_;
       next unless $LegalFilters{$filter};
       $err ||= $q->$filter(%arguments);
    }

    # Remove invalid arguments before the conversion
    for (keys %arguments) {
       delete $arguments{$_} unless $LegalArguments{$_};
    }
```

Example 4-5. Apache::Magick Converts Image Formats on the Fly (continued)

```
    # Create a temporary file name to use for conversion
    my($tmpnam, $fh) = Apache::File->tmpfile;

    # Write out the modified image
    open(STDOUT, ">&=" . fileno($fh));
    $extension =~ s/^\.//;
    $err ||= $q->Write('filename' => "\U$extension\L:-", %arguments);
    if ($err) {
       unlink $tmpnam;
       $r->log_error($err);
       return SERVER_ERROR;
    }
    close $fh;

    # At this point the conversion is all done!
    # reopen for reading
    $fh = Apache::File->new($tmpnam);
    unless ($fh) {
       $r->log_error("Couldn't open $tmpnam: $!");
       return SERVER_ERROR;
    }

    # send the file
    $r->send_fd($fh);

    # clean up and go
    unlink $tmpnam;
    return OK;
}

sub find_image {
    my ($r, $directory, $base) = @_;
    my $dh = DirHandle->new($directory) or return;

    my $source;
    for my $entry ($dh->read) {
       my $candidate = fileparse($entry, '\.\w+');
       if ($base eq $candidate) {
           # determine whether this is an image file
           $source = join '', $directory, $entry;
           my $subr = $r->lookup_file($source);
           last if $subr->content_type =~ m:^image/:;
           undef $source;
       }
    }
    $dh->close;
    return $source;
}

1;
__END__
```

In this chapter:
- *Web Programming Then and Now*
- *The Apache Project*
- *The Apache C and Perl APIs*
- *Ideas and Success Stories*

1

Server-Side Programming with Apache

Before the World Wide Web appeared, client/server network programming was a drag. Application developers had to develop the communications protocol, write the low-level network code to reliably transmit and receive messages, create a user interface at the client side of the connection, and write a server to listen for incoming requests, service them properly, and transmit the results back to the client. Even simple client/server applications were many thousand lines of code, the development pace was slow, and programmers worked in C.

When the web appeared in the early '90s, all that changed. The web provided a simple but versatile communications protocol standard, a universal network client, and a set of reliable and well-written network servers. In addition, the early servers provided developers with a server extension protocol called the Common Gateway Interface (CGI). Using CGI, a programmer could get a simple client/server application up and running in 10 lines of code instead of thousands. Instead of being limited to C or another "systems language," CGI allowed programmers to use whatever development environment they felt comfortable with, whether that be the command shell, Perl, Python, REXX, Visual Basic, or a traditional compiled language. Suddenly client/server programming was transformed from a chore into a breeze. The number of client/server applications increased 100-fold over a period of months, and a new breed of software developer, the "web programmer," appeared.

The face of network application development continues its rapid pace of change. Open the pages of a web developer's magazine today and you'll be greeted by a bewildering array of competing technologies. You can develop applications using server-side include technologies such as PHP or Microsoft's Active Server Pages (ASP). You can create client-side applications with Java, JavaScript, or Dynamic

HTML (DHTML). You can serve pages directly out of databases with products like the Oracle web server or Lotus Domino. You can write high-performance server-side applications using a proprietary server application programming interface (API). Or you can combine server- and client-side programming with integrated development environments like Netscape's LiveWire or NeXT's WebObjects. CGI scripting is still around too, but enhancements like FastCGI and ActiveState's Perl ISAPI are there to improve script performance.

All these choices can be overwhelming, and it isn't always clear which development system offers the best tradeoff between power, performance, compatibility, and longevity. This chapter puts a historical perspective on web application development and shows you how and where the Apache C and Perl APIs fit into the picture.

Web Programming Then and Now

In the beginning was the web server. Specifically, in the very very beginning was CERN *httpd*, a C-language server developed at CERN, the European high-energy physics lab, by Tim Berners-Lee, Ari Luotonen, and Henrik Frystyk Nielsen around 1991. CERN *httpd* was designed to serve static web pages. The server listened to the network for Uniform Resource Locator (URL) requests using what would eventually be called the HTTP/0.9 protocol, translated the URLs into file paths, and returned the contents of the files to the waiting client. If you wanted to extend the functionality of the web server—for example, to hook it up to a bibliographic database of scientific papers—you had to modify the server's source code and recompile.

This was neither very flexible nor very easy to do. So early on, CERN *httpd* was enhanced to launch external programs to handle certain URL requests. Special URLs, recognized with a complex system of pattern matching and string transformation rules, would invoke a command shell to run an external script or program. The output of the script would then be redirected to the browser, generating a web page on the fly. A simple scheme allowed users to pass argument lists to the script, allowing developers to create keyword search systems and other basic applications.

Meanwhile, Rob McCool, of the National Center for Supercomputing Applications at the University of Illinois, was developing another web server to accompany NCSA's browser product, Mosaic. NCSA *httpd* was smaller than CERN *httpd*, faster (or so the common wisdom had it), had a host of nifty features, and was easier than the CERN software to configure and install. It quickly gained ground on CERN *httpd*, particularly in the United States. Like CERN *httpd*, the NCSA product had a facility for generating pages on the fly with external programs but one that

differed in detail from CERN *httpd*'s. Scripts written to work with NCSA *httpd* wouldn't work with CERN *httpd* and vice versa.

The Birth of CGI

Fortunately for the world, the CERN and the NCSA groups did not cling tenaciously to "their" standards as certain latter-day software vendors do. Instead, the two groups got together along with other interested parties and worked out a common standard called the Common Gateway Interface.

CGI was intended to be the duct tape of the web—a flexible glue that could quickly and easily bridge between the web protocols and other forms of information technology. And it worked. By following a few easy conventions, CGI scripts can place user-friendly web frontends on top of databases, scientific analysis tools, order entry systems, and games. They can even provide access to older network services, such as gopher, whois, or WAIS. As the web changed from an academic exercise into big business, CGI came along for the ride. Every major server vendor (with a couple of notable exceptions, such as some of the Macintosh server developers) has incorporated the CGI standard into its product. It comes very close to the "write once, run everywhere" development environment that application developers have been seeking for decades.

But CGI is not the highest-performance environment. The Achilles' heel of a CGI script is that every time a web server needs it, the server must set up the CGI environment, read the script into memory, and launch the script. The CGI protocol works well with operating systems that were optimized for fast process startup and many simultaneous processes, such as Unix dialects, provided that the server doesn't become very heavily loaded. However, as load increases, the process creation bottleneck eventually turns formerly snappy scripts into molasses. On operating systems that were designed to run lightweight threads and where full processes are rather heavyweight, such as Windows NT, CGI scripts are a performance disaster.

Another fundamental problem with CGI scripts is that they exit as soon as they finish processing the current request. If the CGI script does some time-consuming operation during startup, such as establishing a database connection or creating complex data structures, the overhead of reestablishing the state each time it's needed is considerable—and a pain to program around.

Server APIs

An early alternative to the CGI scripting paradigm was the invention of web server APIs (application programming interfaces), mechanisms that the developer can use to extend the functionality of the server itself by linking new modules directly to

the server executable. For example, to search a database from within a web page, a developer could write a module that combines calls to web server functions with calls to a relational database library. Add a dash or two of program logic to transform URLs into SQL, and the web server suddenly becomes a fancy database front-end. Server APIs typically provide extensive access to the innards of the server itself, allowing developers to customize how it performs the various phases of the HTTP transaction. Although this might seem like an esoteric feature, it's quite powerful.

The earliest web API that we know of was built into the Plexus web server, written by Tony Sanders of BSDI. Plexus was a 100 percent pure Perl server that did almost everything that web servers of the time were expected to do. Written entirely in Perl Version 4, Plexus allowed the webmaster to extend the server by adding new source files to be compiled and run on an as-needed basis.

APIs invented later include NSAPI, the interface for Netscape servers; ISAPI, the interface used by Microsoft's Internet Information Server and some other Windows-based servers; and of course the Apache web server's API, the only one of the bunch that doesn't have a cute acronym.

Server APIs provide performance and access to the guts of the server's software, giving them programming powers beyond those of mere mortal CGI scripts. Their drawbacks include a steep learning curve and often a certain amount of risk and inconvenience, not to mention limited portability. As an example of the risk, a bug in an API module can crash the whole server. Because of the tight linkage between the server and its API modules, it's never as easy to install and debug a new module as it is to install and debug a new CGI script. On some platforms, you might have to bring the server down to recompile and link it. On other platforms, you have to worry about the details of dynamic loading. However, the biggest problem of server APIs is their limited portability. A server module written for one API is unlikely to work with another vendor's server without extensive revision.

Server-Side Includes

Another server-side solution uses server-side includes to embed snippets of code inside HTML comments or special-purpose tags. NCSA *httpd* was the first to implement server-side includes. More advanced members of this species include Microsoft's Active Server Pages, Allaire Cold Fusion, and PHP, all of which turn HTML into a miniature programming language complete with variables, looping constructs, and database access methods.

Netscape servers recognize HTML pages that have been enhanced with scraps of JavaScript code (this is distinct from client-side JavaScript, which we talk about later). Embperl, a facility that runs on top of Apache's *mod_perl* module, marries

HTML to Perl, as does PerlScript, an ActiveState extension for Microsoft Internet Information Server.*

The main problem with server-side includes and other HTML extensions is that they're *ad hoc*. No standards exist for server-side includes, and pages written for one vendor's web server will definitely not run unmodified on another's.

Embedded Interpreters

To avoid some of the problems of proprietary APIs and server-side includes, several vendors have turned to using embedded high-level interpretive languages in their servers. Embedded interpreters often come with CGI emulation layers, allowing script files to be executed directly by the server without the overhead of invoking separate processes. An embedded interpreter also eliminates the need to make dramatic changes to the server software itself. In many cases an embedded interpreter provides a smooth path for speeding up CGI scripts because little or no source code modification is necessary.

Examples of embedded interpreters include *mod_pyapache*, which embeds a Python interpreter. When a Python script is requested, the latency between loading the script and running it is dramatically reduced because the interpreter is already in memory. A similar module exists for the TCL language.

Sun Microsystems' "servlet" API provides a standard way for web servers to run small programs written in the Java programming language. Depending on the implementation, a portion of the Java runtime system may be embedded in the web server or the web server itself may be written in Java. Apache's servlet system uses co-processes rather than an embedded interpreter. These implementations all avoid the overhead of launching a new external process for each request.

Much of this book is about *mod_perl*, an Apache module that embeds the Perl interpreter in the server. However, as we shall see, *mod_perl* goes well beyond providing an emulation layer for CGI scripts to give programmers complete access to the Apache API.

Script Co-processing

Another way to avoid the latency of CGI scripts is to keep them loaded and running all the time as a co-process. When the server needs the script to generate a page, it sends it a message and waits for the response.

The first system to use co-processing was the FastCGI protocol, released by Open Market in 1996. Under this system, the web server runs FastCGI scripts as separate

* ActiveState Tool Corp., *http://www.activestate.com/*

processes just like ordinary CGI scripts. However, once launched, these scripts don't immediately exit when they finish processing the initial request. Instead, they go into an infinite loop that awaits new incoming requests, processes them, and goes back to waiting. Things are arranged so that the FastCGI process's input and output streams are redirected to the web server and a CGI-like environment is set up at the beginning of each request.

Existing CGI scripts can be adapted to use FastCGI by making a few, usually painless, changes to the script source code. Implementations of FastCGI are available for Apache, as well as Zeus, Netscape, Microsoft IIS, and other servers. However, FastCGI has so far failed to win wide acceptance in the web development community, perhaps because of Open Market's retreat from the web server market. Fortunately, a group of volunteers have picked up the Apache *mod_fastcgi* module and are continuing to support and advance this freeware implementation. You can find out more about *mod_fastcgi* at the *www.fastcgi.com* website. Commercial implementations of FastCGI are also available from Fast Engines, Inc. (*www.fastengines.com*), which provides the Netscape and Microsoft IIS versions of FastCGI.

Another co-processing system is an Apache module called *mod_jserv*, which you can find at the project homepage, *http://java.apache.org/*. *mod_jserv* allows Apache to run Java servlets using Sun's servlet API. However, unlike most other servlet systems, *mod_jserv* uses something called the "JServ Protocol" to allow the web server to communicate with Java scripts running as separate processes. You can also control these servlets via the Apache Perl API using the *Apache::Servlet* module written by Ian Kluft.

Client-Side Scripting

An entirely different way to improve the performance of web-based applications is to move some or all of the processing from the server side to the client side. It seems silly to send a fill-out form all the way across the Internet and back again if all you need to do is validate that the user has filled in the Zip Code field correctly. This, and the ability to provide more dynamic interfaces, is a big part of the motivation for client-side scripting.

In client-side systems, the browser is more than an HTML rendering engine for the web pages you send it. Instead, it is an active participant, executing commands and even running small programs on your behalf. JavaScript, introduced by Netscape in early 1995, and VBScript, introduced by Microsoft soon afterward, embed a browser scripting language in HTML documents. When you combine browser scripting languages with cascading style sheets, document layers, and other HTML enhancements, you get "Dynamic HTML" (DHTML). The problem with DHTML is that it's a compatibility nightmare. The browsers built by Microsoft

and Netscape implement different sets of DHTML features, and features vary even between browser version numbers. Developers must choose which browser to support, or use mind-bogglingly awkward workarounds to support more than one type of browser. Entire books have been written about DHTML workarounds!

Then there are Java applets. Java burst onto the web development scene in 1995 with an unprecedented level of publicity and has been going strong ever since. A full-featured programming language from Sun Microsystems, Java can be used to write standalone applications, server-side extensions ("servlets," which we discussed earlier), and client-side "applet" applications. Despite the similarity in names, Java and JavaScript share little in common except a similar syntax. Java's ability to run both at the server side and the client side makes Java more suitable for the implementation of complex software development projects than JavaScript or VBScript, and the language is more stable than either of those two.

However, although Java claims to solve client-side compatibility problems, the many slight differences in implementation of the Java runtime library in different browsers has given it a reputation for "write once, debug everywhere." Also, because of security concerns, Java applets are very much restricted in what they can do, although this is expected to change once Sun and the vendors introduce a security model based on unforgeable digital signatures.

Microsoft's ActiveX technology is a repackaging of its COM (Common Object Model) architecture. ActiveX allows dynamic link libraries to be packed up into "controls," shipped across the Internet, and run on the user's computer. Because ActiveX controls are compiled binaries, and because COM has not been adopted by other operating systems, this technology is most suitable for uniform intranet environments that consist of Microsoft Windows machines running a recent version of Internet Explorer.

Integrated Development Environments

Integrated development environments try to give software developers the best of both client-side and server-side worlds by providing a high-level view of the application. In this type of environment, you don't worry much about the details of how web pages are displayed. Instead, you concentrate on the application logic and the user interface.

The development environment turns your program into some mixture of database access queries, server-side procedures, and client-side scripts. Some popular environments of this sort include Netscape's "Live" development systems (LiveWire for client-server applications and LiveConnect for database connectivity),* NeXT's

* As this book was going to press, Netscape announced that it was dropping support for LiveWire, transforming it from a "Live" product into a "dead" one.

object-oriented WebObjects, Allaire's ColdFusion, and the Microsoft FrontPage publishing system. These systems, although attractive, have the same disadvantage as embedded HTML languages: once you've committed to one of these environments, there's no backing out. There's not the least whiff of compatibility across different vendors' development systems.

Making the Choice

Your head is probably spinning with all the possibilities. Which tool should you use for your own application development? The choice depends on your application's requirements and the tradeoffs you're willing to accept. Table 1-1 gives the authors' highly subjective ranking of the different development systems' pros and cons.

Table 1-1. Comparison of Web Development Solutions

	Portability	Performance	Simplicity	Power
CGI	++++	+	+++	++
FastCGI	++	+++	+++	++
Server API	+	++++	+	++++
Server-side includes	++	++	++++	++
DHTML	+	+++	+	++
Client-side Java	++	+++	++	+++
Embedded interpreter	+++	+++	++	++++
Integrated system	+	+++	++	++++

In this table, the "Portability" column indicates how easy it is to move a web application from one server to another in the case of server-side systems, or from one make of web browser to another in the case of client-side solutions. By "Performance," we mean the interactive speed of the application that the user perceives more than raw data processing power of the system. "Simplicity" is our gut feeling for the steepness of the system's learning curve and how convenient the system is to develop in once you're comfortable with it. "Power" is an estimate of the capabilities of the system: how much control it provides over the way the application behaves and its flexibility to meet creative demands.

If your main concern is present and future portability, your best choice is vanilla CGI. You can be confident that your CGI scripts will work properly with all browsers, and that you'll be able to migrate scripts from one server to another with a minimum of hardship. CGI scripts are simple to write and offer a fair amount of flexibility, but their performance is poor.

If you want power and performance at all costs, go with a server API. The applications that you write will work correctly with all browsers, but you'll want to think

twice before moving your programs to a different server. Chances are that a large chunk of your application will need to be rewritten when you migrate from one vendor's API to another's.

FastCGI offers a marked performance improvement but does require you to make some minor modifications to CGI script source code in order to use it.

If you need a sophisticated graphical user interface at the browser side, then some component of your application must be client-side Java or DHTML. Despite its compatibility problems, DHTML is worth considering, particularly when you are running an intranet and have complete control over your users' choice of browsers.

Java applets improve the compatibility situation. So long as you don't try to get too fancy, there's a good chance that an applet will run on more than one version of a single vendor's browser, and perhaps even on browsers from different vendors.

If you're looking for ease of programming and a gentle learning curve, you should consider a server-side include system like PHP or Active Server Pages. You don't have to learn the whole language at once. Just start writing HTML and add new features as you need them. The cost of this simplicity is portability once again. Pages written for one vendor's server-side include system won't work correctly with a different vendor's system, although the HTML framework will still display correctly.

A script interpreter embedded in the web server has much better performance than a standalone CGI script. In many cases, CGI scripts can be moved to embedded interpreters and back again without source code modifications, allowing for portability among different servers. To take the most advantage of the features offered by embedded interpreters, you must usually write server-specific code, which sacrifices portability and adds a bit of complexity to the application code.

The Apache Project

This book is devoted to developing applications with the Apache web server API, so we turn our attention now to the short history of the Apache project.

The Apache project began in 1995 when a group of eight volunteers, seeing that web software was becoming increasingly commercialized, got together to create a supported open source web server. Apache began as an enhanced version of the public-domain NCSA server but steadily diverged from the original. Many new features have been added to Apache over the years: significant features include the ability for a single server to host multiple virtual web sites, a smorgasbord of authentication schemes, and the ability for the server to act as a caching proxy. In some cases, Apache is way ahead of the commercial vendors in the features wars. For example, at the time this book was written only the Apache web server had implemented the HTTP/1.1 Digest Authentication scheme.

Internally the server has been completely redesigned to use a modular and extensible architecture, turning it into what the authors describe as a "web server toolkit." In fact, there's very little of the original NCSA *httpd* source code left within Apache. The main NCSA legacy is the configuration files, which remain backward-compatible with NCSA *httpd*.

Apache's success has been phenomenal. In less than three years, Apache has risen from relative obscurity to the position of market leader. Netcraft, a British market research company that monitors the growth and usage of the web, estimates that Apache servers now run on over 50 percent of the Internet's web sites, making it by far the most popular web server in the world. Microsoft, its nearest rival, holds a mere 22 percent of the market.* This is despite the fact that Apache has lacked some of the conveniences that common wisdom holds to be essential, such as a graphical user interface for configuration and administration.

Apache has been used as the code base for several commercial server products. The most successful of these, C2Net's Stronghold, adds support for secure communications with Secure Socket Layer (SSL) and a form-based configuration manager. There is also WebTen by Tenon Intersystems, a Macintosh PowerPC port, and the Red Hat Secure Server, an inexpensive SSL-supporting server from the makers of Red Hat Linux.

Another milestone was reached in November of 1997 when the Apache Group announced its port of Apache to the Windows NT and 95 operating systems (Win32). A fully multithreaded implementation, the Win32 port supports all the features of the Unix version and is designed with the same modular architecture as its brother. Freeware ports to OS/2 and the AmigaOS are also available.

In the summer of 1998, IBM announced its plans to join with the Apache volunteers to develop a version of Apache to use as the basis of its secure Internet commerce server system, supplanting the servers that it and Lotus Corporation had previously developed.

Why use Apache? Many web sites run Apache by accident. The server software is small, free, and well documented and can be downloaded without filling out pages of licensing agreements. The person responsible for getting his organization's web site up and running downloads and installs Apache just to get his feet wet, intending to replace Apache with a "real" server at a later date. But that date never comes. Apache does the job and does it well.

* Impressive as they are, these numbers should be taken with a grain or two of salt. Netcraft's survey techniques count only web servers connected directly to the Internet. The number of web servers running intranets is not represented in these counts, which might inflate or deflate Apache's true market share.

However, there are better reasons for using Apache. Like other successful open source products such as Perl, the GNU tools, and the Linux operating system, Apache has some big advantages over its commercial rivals.

It's fast and efficient
: The Apache web server core consists of 25,000 lines of highly tuned C code. It uses many tricks to eke every last drop of performance out of the HTTP protocol and, as a result, runs faster and consumes less system resources than many commercial servers. Its modular architecture allows you to build a server that contains just the functionality that you need and no more.

It's portable
: Apache runs on all Unix variants, including the popular freeware Linux operating system. It also runs on Microsoft Windows systems (95, 98, and NT), OS/2, and even the bs2000 mainframe architecture.

It's well supported
: Apache is supported by a cast of thousands. Beyond the core Apache Group developers, who respond to bug reports and answer technical questions via email, Apache is supported by a community of webmasters with hundreds of thousands of hours of aggregate experience behind them. Questions posted to the Usenet newsgroup *comp.infosystems.www.servers.unix* are usually answered within hours. If you need a higher level of support, you can purchase Stronghold or another commercial version of Apache and get all the benefits of the freeware product, plus trained professional help.

It won't go away
: In the software world, a vendor's size or stock market performance is no guarantee of its staying power. Companies that look invincible one year become losers the next. In 1988, who would have thought the Digital Equipment whale would be gobbled up by the Compaq minnow just 10 years later? Good community software projects don't go away. Because the source code is available to all, someone is always there to pick up the torch when a member of the core developer group leaves.

It's stable and reliable
: All software contains bugs. When a commercial server contains a bug there's an irresistible institutional temptation for the vendor to cover up the problem or offer misleading reassurances to the public. With Apache, the entire development process is open to the public. The source code is all there for you to review, and you can even eavesdrop on the development process by subscribing to the developer's mailing list. As a result, bugs don't remain hidden for long, and they are usually fixed rapidly once uncovered. If you get really desperate, you can dig into the source code and fix the problem yourself. (If you do so, please send the fix back to the community!)

It's got features to burn
: Because of its modular architecture and many contributors, Apache has more features than any other web server on the market. Some of its features you may never use. Others, such as its powerful URL rewriting facility, are peerless and powerful.

It's extensible
: Apache is open and extensible. If it doesn't already have a feature you want, you can write your own server module to implement it. In the unlikely event that the server API doesn't support what you want to do, you can dig into the source code for the server core itself. The entire system is open to your inspection; there are no black boxes or precompiled libraries for you to work around.

It's easy to administer
: Apache is configured with plain-text configuration files and controlled with a simple command-line tool. This sounds like a deficiency when compared to the fancy graphical user interfaces supplied with commercial servers, but it does have some advantages. You can save old copies of the configuration files or even commit them to a source code control system, allowing you to keep track of all the configuration changes you've made and to return to an older version if something breaks. You can easily copy the configuration files from one host machine to another, effectively cloning the server. Lastly, the ability to control the server from the command line lets you administer the server from anywhere that you can telnet from—you don't even need web connectivity.

  This being said, Apache does provide simple web-based interfaces for viewing the current configuration and server status. A number of people are working on administrative GUIs, and there is already a web interface for remotely managing web user accounts (the *user_manage* tool available at *http://stein.cshl.org/~lstein/user_manage*).

It makes you part of a community
: When you install an Apache server you become part of a large virtual community of Apache webmasters, authors, and developers. You will never feel that the software is something whose use has been grudgingly granted to you by a corporate entity. Instead, the Apache server is owned by its community. By using the Apache server, you automatically own a bit of it too and are contributing, if even in only a small way, to its continued health and development. Welcome to the club!

The Apache C and Perl APIs

The Apache module API gives you access to nearly all of the server's internal processing. You can inspect what it's doing at each step of the HTTP transaction cycle

and intervene at any of the steps to customize the server's behavior. You can arrange for the server to take custom actions at startup and exit time, add your own directives to its configuration files, customize the process of translating URLs into file names, create custom authentication and authorization systems, and even tap into the server's logging system. This is all done via modules—self-contained pieces of code that can either be linked directly into the server executable, or loaded on demand as a dynamic shared object (DSO).

The Apache module API was intended for C programmers. To write a traditional compiled module, you prepare one or more C source files with a text editor, compile them into object files, and either link them into the server binary or move them into a special directory for DSOs. If the module is implemented as a DSO, you'll also need to edit the server configuration file so that the module gets loaded at the appropriate time. You'll then launch the server and begin the testing and debugging process.

This sounds like a drag, and it is. It's even more of a drag because you have to worry about details of memory management and configuration file processing that are tangential to the task at hand. A mistake in any one of these areas can crash the server.

For this reason, the Apache server C API has generally been used only for substantial modules which need high performance, tiny modules that execute very frequently, or anything that needs access to server internals. For small to medium applications, one-offs, and other quick hacks, developers have used CGI scripts, FastCGI, or some other development system.

Things changed in 1996 when Doug MacEachern introduced *mod_perl*, a complete Perl interpreter wrapped within an Apache module. This module makes almost the entire Apache API available to Perl programmers as objects and method calls. The parts that it doesn't export are C-specific routines that Perl programmers don't need to worry about. Anything that you can do with the C API you can do with *mod_perl* with less fuss and bother. You don't have to restart the server to add a new *mod_perl* module, and a buggy module is less likely to crash the server.

We have found that for the vast majority of applications *mod_perl* is all you need. For those cases when you need the raw processing power or the small memory footprint that a compiled module gives you, the C and Perl forms of the API are close enough so that you can prototype the application in *mod_perl* first and port it to C later. You may well be surprised to find that the "prototype" is all you really need!

This book uses *mod_perl* to teach you the Apache API. This keeps the examples short and easy to understand, and shows you the essentials without bogging down

in detail. Toward the end of the book we show you how to port Apache modules written in Perl into C to get the memory and execution efficiency of a compiled language.

Ideas and Success Stories

To give you an impression of the power and versatility of the Apache API, here are some examples of what people have done with it. Some of the modules described here have been incorporated into Apache and are now part of the standard distribution. Others are third-party modules that have been developed to solve particular mission-critical tasks.

A movie database
: The Internet Movie Database (*http://www.imdb.com/*) uses *mod_perl* to make queries against a vast database of film and television movies. The system rewrites URLs on the fly in order to present pages in the language of the user's choice and to quickly retrieve the results of previously cached searches. In 1998, the site won the coveted Webby award for design and service.

No more URL spelling errors
: URLs are hard things to type, and many HTML links are broken because of a single typo in a long URL. The most frequent errors are problems with capitalization, since many HTML authors grew up in a case-insensitive MS-DOS/Windows world before entering the case-sensitive web.

  mod_speling [*sic*], part of the standard Apache distribution, is a C-language module that catches and fixes typographical errors on the fly. If no immediate match to a requested URL is found, it checks for capitalization variations and a variety of character insertions, omissions, substitutions, and transpositions, trying to find a matching valid document on the site. If one is found, it generates a redirect request, transparently forwarding the browser to the correct resource. Otherwise, it presents the user with a menu of closest guesses to choose from.

An on-campus housing renewal system
: At Texas A&M University, students have to indicate each academic year whether they plan to continue living in campus-provided housing. For the 1997–1998 academic year, the university decided to move the process from its current error-prone manual system to a web-based solution. The system was initially implemented using ActiveWare's PerlScript to drive a set of Microsoft Internet Information Server Active Server Pages, but with less than two weeks to go before deployment it was clear that the system would be too slow to handle the load. The system was hurriedly rewritten to use *mod_perl* on top of

the NT version of Apache, resulting in a measured 60-fold increase in performance. The system went online in the nick of time and functioned without a hitch, serving 400,000 documents generated on the fly to 10,000 people over the course of the four-day registration period.

Scripting languages embedded in HTML

The PHP system (*http://www.php.net/*) is a powerful scripting language that processes programs embedded within HTML documents. The language provides support for persistent connections to ODBC and Unix databases, on-the-fly graphics, and LDAP searches. The language is implemented both as a CGI script that can run on top of any server and as a high-performance C-language module for Apache.

The ePerl (*http://www.engelschall.com/sw/eperl/*) and Embperl (*http://perl.apache.org/embperl/*) systems are like PHP, but use *mod_perl* to embed snippets of Perl code directly inside HTML pages. They can do anything that Perl can do, including opening network connections to other Internet services, accessing databases, and generating dynamic documents based on user input.

An advertising banner server

No web application needs higher performance than banner ad servers, which are pummeled by millions of requests per day. One banner ad vendor, whose conventional CGI-based system was topping out at 1.5 banners per second, moved its system to *mod_perl* and experienced a greater than 10-fold performance boost. The vendor is now serving 10 million banners a week from a single host.

A dynamic map server

The *www.stadtplandienst.de* site uses the *mod_perl* API with the ImageMagick graphics library to create dynamic searchable tourist maps for Berlin and other German cities. The system is fast and responsive, despite the computationally intensive nature of its job and its frequently heavy load.

A commodities trading system

Lind-Waldock & Co. (*http://www.lind-waldock.com/*), the world's largest discount commodities trading firm, uses *mod_perl* running under the Stronghold version of Apache to generate live and delayed quotes, dynamic charts, and late-breaking news, as well as a frontend to their online order entry system. The system is tightly integrated with the company's relational database system for customer authentication and transaction processing.

Brian Fitzpatrick, a member of the consulting team that designed and implemented the system, was pleasantly surprised at how smooth the process was: "*mod_perl* allowed us to work the web server and code around our design—not the other way around."

A document management system

The Advanced Computer Communications company maintains more than 1500 documents in various formats scattered among multiple NFS-mounted file systems in its internal network. Their document management system periodically indexes the scattered documents by document name, creation date, and content, then uses the *mod_perl* interface to the Apache API to allow users to search and retrieve documents of interest to them. The system automatically performs document format conversion. Some are sent to the browser for download, others are precompressed with PKZIP to reduce transmission time, and still others are converted into formats that can be displayed directly in the browser window.

These applications represent only a few of the possible uses for the Apache module API. What you can do with it is limited only by your imagination. The rest of this book shows you how to turn your ideas into reality.