

CHAPTER 1

Introducing CGI and mod_perl

This chapter provides the foundations on which the rest of the book builds. In this chapter, we give you:




- A history of CGI and the HTTP protocol.
- An explanation of the Apache 1.3 Unix model, which is crucial to understanding how mod_perl 1.0 works.
- An overall picture of mod_perl 1.0 and its development.
- An overview of the difference between the Apache C API, the Apache Perl API (i.e., the mod_perl API), and CGI compatibility. We will also introduce the `Apache::Registry` and `Apache::PerlRun` modules.
- An introduction to the mod_perl API and handlers.

A Brief History of CGI

When the World Wide Web was born, there was only one web server and one web client. The *httpd* web server was developed by the Centre d'Etudes et de Recherche Nucléaires (CERN) in Geneva, Switzerland. *httpd* has since become the generic name of the binary executable of many web servers. When CERN stopped funding the development of *httpd*, it was taken over by the Software Development Group of the National Center for Supercomputing Applications (NCSA). The NCSA also produced Mosaic, the first web browser, whose developers later went on to write the Netscape client.

Mosaic could fetch and view static documents* and images served by the *httpd* server. This provided a far better means of disseminating information to large numbers of people than sending each person an email. However, the glut of online resources soon made search engines necessary, which meant that users needed to be able to

* A static document is one that exists in a constant state, such as a text file that doesn't change.




submit data (such as a search string) and servers needed to process that data and return appropriate content.


Search engines were first implemented by extending the web server, modifying its source code directly. Rewriting the source was not very practical, however, so the NCSA developed the *Common Gateway Interface* (CGI) specification. CGI became a standard for interfacing external applications with web servers and other information servers and generating dynamic information.

A CGI program can be written in virtually any language that can read from `STDIN` and write to `STDOUT`, regardless of whether it is interpreted (e.g., the Unix shell), compiled (e.g., C or C++), or a combination of both (e.g., Perl). The first CGI programs were written in C and needed to be compiled into binary executables. For this reason, the directory from which the compiled CGI programs were executed was named *cgi-bin*, and the source files directory was named *cgi-src*. Nowadays most servers come with a preconfigured directory for CGI programs called, as you have probably guessed, *cgi-bin*.

The HTTP Protocol



Interaction between the browser and the server is governed by the *HyperText Transfer Protocol* (HTTP), now an official Internet standard maintained by the World Wide Web Consortium (W3C). HTTP uses a simple request/response model: the client establishes a TCP* connection to the server and sends a request, the server sends a response, and the connection is closed. Requests and responses take the form of *messages*. A message is a simple sequence of text lines.



HTTP messages have two parts. First come the *headers*, which hold descriptive information about the request or response. The various types of headers and their possible content are fully specified by the HTTP protocol. Headers are followed by a blank line, then by the message *body*. The body is the actual content of the message, such as an HTML page or a GIF image. The HTTP protocol does not define the content of the body; rather, specific headers are used to describe the content type and its encoding. This enables new content types to be incorporated into the Web without any fanfare.

HTTP is a stateless protocol. This means that requests are not related to each other. This makes life simple for CGI programs: they need worry about only the current request.

The Common Gateway Interface Specification

If you are new to the CGI world, there's no need to worry—basic CGI programming is very easy. Ninety percent of CGI-specific code is concerned with reading data

* TCP/IP is a low-level Internet protocol for transmitting bits of data, regardless of its use.

submitted by a user through an HTML form, processing it, and returning some response, usually as an HTML document.

In this section, we will show you how easy basic CGI programming is, rather than trying to teach you the entire CGI specification. There are many books and online tutorials that cover CGI in great detail (see <http://hoohoo.ncsa.uiuc.edu/>). Our aim is to demonstrate that if you know Perl, you can start writing CGI scripts almost immediately. You need to learn only two things: how to accept data and how to generate output.

The HTTP protocol makes clients and servers understand each other by transferring all the information between them using headers, where each header is a key-value pair. When you submit a form, the CGI program looks for the headers that contain the input information, processes the received data (e.g., queries a database for the keywords supplied through the form), and—when it is ready to return a response to the client—sends a special header that tells the client what kind of information it should expect, followed by the information itself. The server can send additional headers, but these are optional. Figure 1-1 depicts a typical request-response cycle.

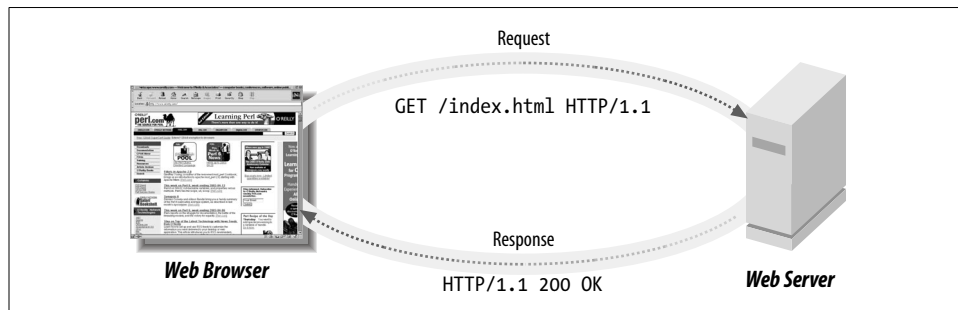


Figure 1-1. Request-response cycle

Sometimes CGI programs can generate a response without needing any input data from the client. For example, a news service may respond with the latest stories without asking for any input from the client. But if you want stories for a specific day, you have to tell the script which day's stories you want. Hence, the script will need to retrieve some input from you.

To get your feet wet with CGI scripts, let's look at the classic "Hello world" script for CGI, shown in Example 1-1.

Example 1-1. "Hello world" script

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

print "Content-type: text/plain\n\n";
print "Hello world!\n";
```

We start by sending a Content-type header, which tells the client that the data that follows is of plain-text type. *text/plain* is a Multipurpose Internet Mail Extensions (MIME) type. You can find a list of widely used MIME types in the *mime.types* file, which is usually located in the directory where your web server's configuration files are stored.* Other examples of MIME types are *text/html* (text in HTML format) and *video/mpeg* (an MPEG stream).

According to the HTTP protocol, an empty line must be sent after all headers have been sent. This empty line indicates that the actual response data will start at the next line.†

Now save the code in *hello.pl*, put it into a *cgi-bin* directory on your server, make the script executable, and test the script by pointing your favorite browser to:

`http://localhost/cgi-bin/hello.pl`

It should display the same output as Figure 1-2.



Figure 1-2. Hello world

A more complicated script involves parsing input data. There are a few ways to pass data to the scripts, but the most commonly used are the GET and POST methods. Let's write a script that expects as input the user's name and prints this name in its response. We'll use the GET method, which passes data in the request URI (uniform resource indicator):

`http://localhost/cgi-bin/hello.pl?username=Doug`

When the server accepts this request, it knows to split the URI into two parts: a path to the script (*http://localhost/cgi-bin/hello.pl*) and the "data" part (*username=Doug*, called the *QUERY_STRING*). All we have to do is parse the data portion of the URI and extract the key *username* and value *Doug*. The GET method is used mostly for hard-coded queries, where no interactive input is needed. Assuming that portions of your

* For more information about Internet media types, refer to RFCs 2045, 2046, 2047, 2048, and 2077, accessible from <http://www.rfc-editor.org/>.

† The protocol specifies the end of a line as the character sequence Ctrl-M and Ctrl-J (carriage return and new-line). On Unix and Windows systems, this sequence is expressed in a Perl string as `\015\012`, but Apache also honors `\n`, which we will use throughout this book. On EBCDIC machines, an explicit `\r\n` should be used instead.

site are dynamically generated, your site's menu might include the following HTML code:

```
<a href="/cgi-bin/display.pl?section=news">News</a><br>
<a href="/cgi-bin/display.pl?section=stories">Stories</a><br>
<a href="/cgi-bin/display.pl?section=links">Links</a><br>
```

Another approach is to use an HTML form, where the user fills in some parameters. The HTML form for the "Hello user" script that we will look at in this section can be either:

```
<form action="/cgi-bin/hello_user.pl" method="POST">
<input type="text" name="username">
<input type="submit">
</form>
```

or:

```
<form action="/cgi-bin/hello_user.pl" method="GET">
<input type="text" name="username">
<input type="submit">
</form>
```

Note that you can use either the GET or POST method in an HTML form. However, POST should be used when the query has side effects, such as changing a record in a database, while GET should be used in simple queries like this one (simple URL links are GET requests).*

Formerly, reading input data required different code, depending on the method used to submit the data. We can now use Perl modules that do all the work for us. The most widely used CGI library is the CGI.pm module, written by Lincoln Stein, which is included in the Perl distribution. Along with parsing input data, it provides an easy API to generate the HTML response.

Our sample "Hello user" script is shown in Example 1-2.

Example 1-2. "Hello user" script

```
#!/usr/bin/perl

use CGI qw(:standard);
my $username = param('username') || "unknown";

print "Content-type: text/plain\n\n";
print "Hello $username!\n";
```

Notice that this script is only slightly different from the previous one. We've pulled in the CGI.pm module, importing a group of functions called :standard. We then used its param() function to retrieve the value of the username key. This call will return the

* See *Axioms of Web Architecture* at <http://www.w3.org/DesignIssues/Axioms.html#state>.

name submitted by any of the three ways described above (a form using either POST, GET, or a hardcoded name with GET; the last two are essentially the same). If no value was supplied in the request, `param()` returns `undef`.

```
my $username = param('username') || "unknown";
```

`$username` will contain either the submitted username or the string "unknown" if no value was submitted. The rest of the script is unchanged—we send the MIME header and print the "Hello `$username`!" string.*

As we've just mentioned, `CGI.pm` can help us with output generation as well. We can use it to generate MIME headers by rewriting the original script as shown in Example 1-3.

Example 1-3. "Hello user" script using `CGI.pm`

```
#!/usr/bin/perl

use CGI qw(:standard);
my $username = param('username') || "unknown";

print header("text/plain");
print "Hello $username!\n";
```

To help you learn how `CGI.pm` copes with more than one parameter, consider the code in Example 1-4.

Example 1-4. `CGI.pm` and `param()` method

```
#!/usr/bin/perl

use CGI qw(:standard);
print header("text/plain");

print "The passed parameters were:\n";
for my $key ( param() ) {
    print "$key => ", param($key), "\n";
}
```

Now issue the following request:

```
http://localhost/cgi-bin/hello_user.pl?a=foo&b=bar&c=foobar
```

The browser will display:

```
The passed parameters were:
a => foo
b => bar
c => foobar
```

* All scripts shown here generate plain text, not HTML. If you generate HTML output, you have to protect the incoming data from cross-site scripting. For more information, refer to the CERT advisory at <http://www.cert.org/advisories/CA-2000-02.html>.

Separating key=value Pairs

Note that `&` or `;` usually is used to separate the *key=value* pairs. The former is less preferable, because if you end up with a `QUERY_STRING` of this format:

```
id=foo&reg=bar
```

some browsers will interpret `®` as an SGML entity and encode it as `®`. This will result in a corrupted `QUERY_STRING`:

```
id=foo&reg;=bar
```

You have to encode `&` as `&` if it is included in HTML. You don't have this problem if you use `;` as a separator:

```
id=foo;reg=bar
```

Both separators are supported by `CGI.pm`, `Apache::Request`, and `mod_perl`'s `args()` method, which we will use in the examples to retrieve the request parameters.

Of course, the code that builds `QUERY_STRING` has to ensure that the values don't include the chosen separator and encode it if it is used. (See RFC2854 for more details.)

Now generate this form:

```
<form action="/cgi-bin/hello_user.pl" method="GET">
<input type="text" name="firstname">
<input type="text" name="lastname">
<input type="submit">
</form>
```

If we fill in only the `firstname` field with the value `Doug`, the browser will display:

```
The passed parameters were:
firstname => Doug
lastname =>
```

If in addition the `lastname` field is `MacEachern`, you will see:

```
The passed parameters were:
firstname => Doug
lastname => MacEachern
```

These are just a few of the many functions `CGI.pm` offers. Read its manpage for detailed information by typing `perldoc CGI` at your command prompt.

We used this long `CGI.pm` example to demonstrate how simple basic CGI is. You shouldn't reinvent the wheel; use standard tools when writing your own scripts, and you will save a lot of time. Just as with Perl, you can start creating really cool and powerful code from the very beginning, gaining more advanced knowledge over time. There is much more to know about the CGI specification, and you will learn about some of its advanced features in the course of your web development practice. We will cover the most commonly used features in this book.

For now, let CGI.pm or an equivalent library handle the intricacies of the CGI specification, and concentrate your efforts on the core functionality of your code.

Apache CGI Handling with mod_cgi

The Apache server processes CGI scripts via an Apache module called mod_cgi. (See later in this chapter for more information on request-processing phases and Apache modules.) mod_cgi is built by default with the Apache core, and the installation procedure also preconfigures a *cgi-bin* directory and populates it with a few sample CGI scripts. Write your script, move it into the *cgi-bin* directory, make it readable and executable by the web server, and you can start using it right away.

Should you wish to alter the default configuration, there are only a few configuration directives that you might want to modify. First, the ScriptAlias directive:

```
ScriptAlias /cgi-bin/ /home/httpd/cgi-bin/
```

ScriptAlias controls which directories contain server scripts. Scripts are run by the server when requested, rather than sent as documents.

When a request is received with a path that starts with */cgi-bin*, the server searches for the file in the */home/httpd/cgi-bin* directory. It then runs the file as an executable program, returning to the client the generated output, not the source listing of the file.

The other important part of *httpd.conf* specifies how the files in *cgi-bin* should be treated:

```
<Directory /home/httpd/cgi-bin>
  Options FollowSymLinks
  Order allow,deny
  Allow from all
</Directory>
```

The above setting allows the use of symbolic links in the */home/httpd/cgi-bin* directory. It also allows anyone to access the scripts from anywhere.

mod_cgi provides access to various server parameters through environment variables. The script in Example 1-5 will print these environment variables.

Example 1-5. Checking environment variables

```
#!/usr/bin/perl

print "Content-type: text/plain\n\n";
for (keys %ENV) {
  print "$_ => $ENV{$_}\n";
}
```

Save this script as *env.pl* in the directory *cgi-bin* and make it executable and readable by the server (that is, by the username under which the server runs). Point your

browser to `http://localhost/cgi-bin/env.pl` and you will see a list of parameters similar to this one:

```
SERVER_SOFTWARE => Server: Apache/1.3.24 (Unix) mod_perl/1.26
                  mod_ssl/2.8.8 OpenSSL/0.9.6
GATEWAY_INTERFACE => CGI/1.1
DOCUMENT_ROOT => /home/httpd/docs
REMOTE_ADDR => 127.0.0.1
SERVER_PROTOCOL => HTTP/1.0
REQUEST_METHOD => GET
QUERY_STRING =>
HTTP_USER_AGENT => Mozilla/5.0 Galeon/1.2.1 (X11; Linux i686; U;) Gecko/0
SERVER_ADDR => 127.0.0.1
SCRIPT_NAME => /cgi-bin/env.pl
SCRIPT_FILENAME => /home/httpd/cgi-bin/env.pl
```

Your code can access any of these variables with `$ENV{"somekey"}`. However, some variables can be spoofed by the client side, so you should be careful if you rely on them for handling sensitive information. Let's look at some of these environment variables.

```
SERVER_SOFTWARE => Server: Apache/1.3.24 (Unix) mod_perl/1.26
                  mod_ssl/2.8.8 OpenSSL/0.9.6
```

The `SERVER_SOFTWARE` variable tells us what components are compiled into the server, and their version numbers. In this example, we used Apache 1.3.24, `mod_perl` 1.26, `mod_ssl` 2.8.8, and `OpenSSL` 0.9.6.

```
GATEWAY_INTERFACE => CGI/1.1
```

The `GATEWAY_INTERFACE` variable is very important; in this example, it tells us that the script is running under `mod_cgi`. When running under `mod_perl`, this value changes to `CGI-Perl/1.1`.

```
REMOTE_ADDR => 127.0.0.1
```

The `REMOTE_ADDR` variable tells us the remote address of the client. In this example, both client and server were running on the same machine, so the client is `localhost` (whose IP is `127.0.0.1`).

```
SERVER_PROTOCOL => HTTP/1.0
```

The `SERVER_PROTOCOL` variable reports the HTTP protocol version upon which the client and the server have agreed. Part of the communication between the client and the server is a negotiation of which version of the HTTP protocol to use. The highest version the two can understand will be chosen as a result of this negotiation.

```
REQUEST_METHOD => GET
```

The now-familiar `REQUEST_METHOD` variable tells us which request method was used (GET, in this case).

```
QUERY_STRING =>
```

The `QUERY_STRING` variable is also very important. It is used to pass the query parameters when using the GET method. `QUERY_STRING` is empty in this example, because we didn't pass any parameters.

```
HTTP_USER_AGENT => Mozilla/5.0 Galeon/1.2.1 (X11; Linux i686; U;) Gecko/0
```

The `HTTP_USER_AGENT` variable contains the user agent specifications. In this example, we are using Galeon on Linux. Note that this variable is very easily spoofed.

Spooing HTTP_USER_AGENT

If the client is a custom program rather than a widely used browser, it can mimic its bigger brother's signature. Here is an example of a very simple client using the LWP library:

```
#!/usr/bin/perl -w
use LWP::UserAgent;

my $ua = new LWP::UserAgent;
$ua->agent("Mozilla/5.0 Galeon/1.2.1 (X11; Linux i686; U;) Gecko/0");
my $req = new HTTP::Request('GET', 'http://localhost/cgi-bin/env.pl');

my $res = $ua->request($req);
print $res->content if $res->is_success;
```

This script first creates an instance of a user agent, with a signature identical to Galeon's on Linux. It then creates a request object, which is passed to the user agent for processing. The response content is received and printed.

When run from the command line, the output of this script is strikingly similar to what we obtained with the browser. It notably prints:

```
HTTP_USER_AGENT => Mozilla/5.0 Galeon/1.2.1 (X11; Linux i686; U;) Gecko/0
```

So you can see how easy it is to fool a naïve CGI programmer into thinking we've used Galeon as our client program.

```
SERVER_ADDR => 127.0.0.1
SCRIPT_NAME => /cgi-bin/env.pl
SCRIPT_FILENAME => /home/httpd/cgi-bin/env.pl
```

The `SERVER_ADDR`, `SCRIPT_NAME`, and `SCRIPT_FILENAME` variables tell us (respectively) the server address, the name of the script as provided in the request URI, and the real path to the script on the filesystem.

Now let's get back to the `QUERY_STRING` parameter. If we submit a new request for `http://localhost/cgi-bin/env.pl?foo=ok&bar=not_ok`, the new value of the query string is displayed:

```
QUERY_STRING => foo=ok&bar=not_ok
```

This is the variable used by CGI.pm and other modules to extract the input data.

Keep in mind that the query string has a limited size. Although the HTTP protocol itself does not place a limit on the length of a URI, most server and client software does. Apache currently accepts a maximum size of 8K (8192) characters for the entire URI. Some older client or proxy implementations do not properly support URIs larger than 255 characters. This is true for some new clients as well—for example, some WAP phones have similar limitations.

Larger chunks of information, such as complex forms, are passed to the script using the POST method. Your CGI script should check the `REQUEST_METHOD` environment variable, which is set to `POST` when a request is submitted with the POST method. The script can retrieve all submitted data from the `STDIN` stream. But again, let CGI.pm or similar modules handle this process for you; whatever the request method, you won't have to worry about it because the key/value parameter pairs will always be handled in the right way.

The Apache 1.3 Server Model

Now that you know how CGI works, let's talk about how Apache implements `mod_cgi`. This is important because it will help you understand the limitations of `mod_cgi` and why `mod_perl` is such a big improvement. This discussion will also build a foundation for the rest of the performance chapters of this book.

Forking

Apache 1.3 on all Unix flavors uses the *forking* model.* When you start the server, a single process, called the *parent process*, is started. Its main responsibility is starting and killing child processes as needed. Various Apache configuration directives let you control how many child processes are spawned initially, the number of spare idle processes, and the maximum number of processes the parent process is allowed to fork.

Each child process has its own lifespan, which is controlled by the configuration directive `MaxRequestsPerChild`. This directive specifies the number of requests that should be served by the child before it is instructed to step down and is replaced by another process. Figure 1-3 illustrates.

When a client initiates a request, the parent process checks whether there is an idle child process and, if so, tells it to handle the request. If there are no idle processes, the parent checks whether it is allowed to fork more processes. If it is, a new process is forked to handle the request. Otherwise, the incoming request is queued until a child process becomes available to handle it.

* In Chapter 24 we talk about Apache 2.0, which introduces a few more server models.

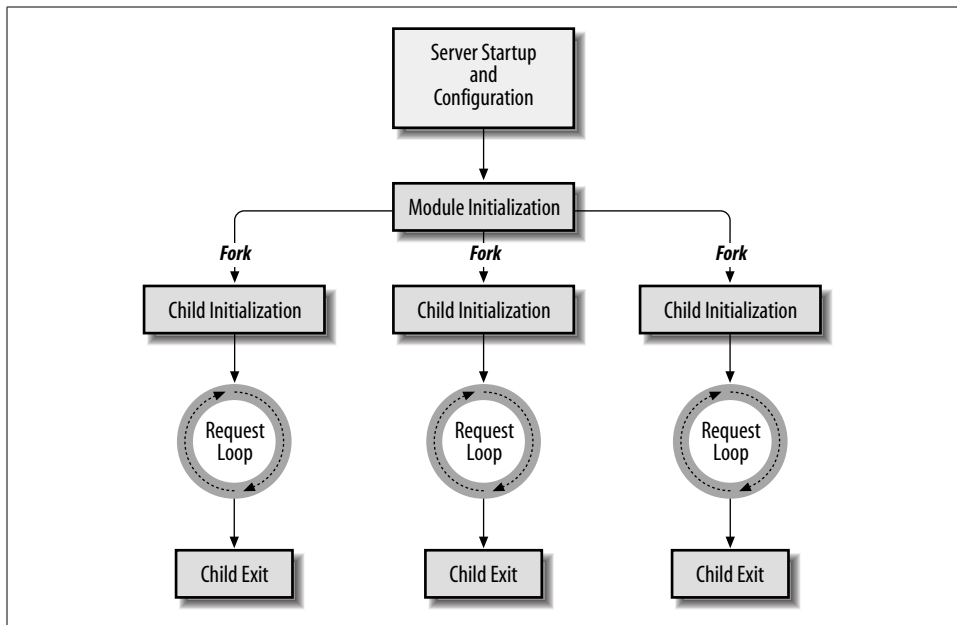


Figure 1-3. The Apache 1.3 server lifecycle

The maximum number of queued requests is configurable by the ListenBacklog configuration directive. When this number is reached, a client issuing a new request will receive an error response informing it that the server is unreachable.

This is how requests for static objects, such as HTML documents and images, are processed. When a CGI request is received, an additional step is performed: mod_cgi in the child Apache process forks a new process to execute the CGI script. When the script has completed processing the request, the forked process exits.

CGI Scripts Under the Forking Model

One of the benefits of this model is that if something causes the child process to die (e.g., a badly written CGI script), it won't cause the whole service to fail. In fact, only the client that initiated the request will notice there was a problem.

Many free (and non-free) CGI scripts are badly written, but they still work, which is why no one tries to improve them. Examples of poor CGI programming practices include forgetting to close open files, using uninitialized global variables, ignoring the warnings Perl generates, and forgetting to turn on taint checks (thus creating huge security holes that are happily used by crackers to break into online systems).

Why do these sloppily written scripts work under mod_cgi? The reason lies in the way mod_cgi invokes them: every time a Perl CGI script is run, a new process is forked, and a new Perl interpreter is loaded. This Perl interpreter lives for the span of

the request's life, and when the script exits (no matter how), the process and the interpreter exit as well, cleaning up on the way. When a new interpreter is started, it has no history of previous requests. All the variables are created from scratch, and all the files are reopened if needed. Although this detail may seem obvious, it will be of paramount importance when we discuss `mod_perl`.

Performance Drawbacks of Forking

There are several drawbacks to `mod_cgi` that triggered the development of improved web technologies. The first problem lies in the fact that a new process is forked and a new Perl interpreter is loaded for each CGI script invocation. This has several implications:

- It adds the overhead of forking, although this is almost insignificant on modern Unix systems.
- Loading the Perl interpreter adds significant overhead to server response times.
- The script's source code and the modules that it uses need to be loaded into memory and compiled each time from scratch. This adds even more overhead to response times.
- Process termination on the script's completion makes it impossible to create persistent variables, which in turn prevents the establishment of persistent database connections and in-memory databases.
- Starting a new interpreter removes the benefit of memory sharing that could be obtained by preloading code modules at server startup. Also, database connections can't be pre-opened at server startup.

Another drawback is limited functionality: `mod_cgi` allows developers to write only content handlers within CGI scripts. If you need to access the much broader core functionality Apache provides, such as authentication or URL rewriting, you must resort to third-party Apache modules written in C, which sometimes make the production server environment somewhat cumbersome. More components require more administration work to keep the server in a healthy state.

The Development of `mod_perl 1.0`

Of the various attempts to improve on `mod_cgi`'s shortcomings, `mod_perl` has proven to be one of the better solutions and has been widely adopted by CGI developers. Doug MacEachern fathered the core code of this Apache module and licensed it under the Apache Software License, which is a certified open source license.

`mod_perl` does away with `mod_cgi`'s forking by embedding the Perl interpreter into Apache's child processes, thus avoiding the forking `mod_cgi` needed to run Perl programs. In this new model, the child process doesn't exit when it has processed a request. The Perl interpreter is loaded only once, when the process is started. Since

the interpreter is persistent throughout the process's lifetime, all code is loaded and compiled only once, the first time it is needed. All subsequent requests run much faster, because everything is already loaded and compiled. Response processing is reduced to simply running the code, which improves response times by a factor of 10–100, depending on the code being executed.

But Doug's real accomplishment was adding a `mod_perl` API to the Apache core. This made it possible to write complete Apache modules in Perl, a feat that used to require coding in C. From then on, `mod_perl` enabled the programmer to handle all phases of request processing in Perl.

The `mod_perl` API also allows complete server configuration in Perl. This has made the lives of many server administrators much easier, as they now benefit from dynamically generating the configuration and are freed from hunting for bugs in huge configuration files full of similar directives for virtual hosts and the like.*

To provide backward compatibility for plain CGI scripts that used to be run under `mod_cgi`, while still benefiting from a preloaded Perl interpreter and modules, a few special handlers were written, each allowing a different level of proximity to pure `mod_perl` functionality. Some take full advantage of `mod_perl`, while others do not.

`mod_perl` embeds a copy of the Perl interpreter into the Apache `httpd` executable, providing complete access to Perl functionality within Apache. This enables a set of `mod_perl`-specific configuration directives, all of which start with the string `Perl`. Most, but not all, of these directives are used to specify handlers for various phases of the request.

It might occur to you that sticking a large executable (Perl) into another large executable (Apache) creates a very, very large program. `mod_perl` certainly makes `httpd` significantly bigger, and you will need more RAM on your production server to be able to run many `mod_perl` processes. However, in reality, the situation is not as bad as it first appears. `mod_perl` processes requests much faster, so the number of processes needed to handle the same request rate is much lower relative to the `mod_cgi` approach. Generally, you need slightly more available memory, but the speed improvements you will see are well worth every megabyte of memory you can add. Techniques that can reduce memory requirements are covered in Chapter 10.

According to <http://netcraft.com/>, as of January 2003, `mod_perl` has been used on more than four million web sites. Some of these sites have been using `mod_perl` since its early days. You can see an extensive list of sites that use `mod_perl` at <http://perl.apache.org/outstanding/sites.html> or http://perl.apache.org/outstanding/success_stories/. The latest usage statistics can be viewed at <http://perl.apache.org/outstanding/stats/>.

* `mod_vhost_alias` offers similar functionality.

Running CGI Scripts with mod_perl

Since many web application developers are interested in the content delivery phase and come from a CGI background, mod_perl includes packages designed to make the transition from CGI simple and painless. Apache::PerlRun and Apache::Registry run unmodified CGI scripts, albeit much faster than mod_cgi.*

The difference between Apache::Registry and Apache::PerlRun is that Apache::Registry caches all scripts, and Apache::PerlRun doesn't. To understand why this matters, remember that if one of mod_perl's benefits is added speed, another is persistence. Just as the Perl interpreter is loaded only once, at child process startup, your scripts are loaded and compiled only once, when they are first used. This can be a double-edged sword: persistence means global variables aren't reset to initial values, and file and database handles aren't closed when the script ends. This can wreak havoc in badly written CGI scripts.

Whether you should use Apache::Registry or Apache::PerlRun for your CGI scripts depends on how well written your existing Perl scripts are. Some scripts initialize all variables, close all file handles, use taint mode, and give only polite error messages. Others don't.

Apache::Registry compiles scripts on first use and keeps the compiled scripts in memory. On subsequent requests, all the needed code (the script and the modules it uses) is already compiled and loaded in memory. This gives you enormous performance benefits, but it requires that scripts be well behaved.

Apache::PerlRun, on the other hand, compiles scripts at each request. The script's namespace is flushed and is fresh at the start of every request. This allows scripts to enjoy the basic benefit of mod_perl (i.e., not having to load the Perl interpreter) without requiring poorly written scripts to be rewritten.

A typical problem some developers encounter when porting from mod_cgi to Apache::Registry is the use of uninitialized global variables. Consider the following script:

```
use CGI;
$q = CGI->new();
$topsecret = 1 if $q->param("secret") eq 'Muahaha';
# ...
if ($topsecret) {
    display_topsecret_data();
}
else {
    security_alert();
}
```

* Apache::RegistryNG and Apache::RegistryBB are two new experimental modules that you may want to try as well.

This script will always do the right thing under `mod_cgi`: if `secret=Muahaha` is supplied, the top-secret data will be displayed via `display_topsecret_data()`, and if the authentication fails, the `security_alert()` function will be called. This works only because under `mod_cgi`, all globals are undefined at the beginning of each request.

Under `Apache::Registry`, however, global variables preserve their values between requests. Now imagine a situation where someone has successfully authenticated, setting the global variable `$topsecret` to a true value. From now on, anyone can access the top-secret data without knowing the secret phrase, because `$topsecret` will stay true until the process dies or is modified elsewhere in the code.

This is an example of sloppy code. It will do the right thing under `Apache::PerlRun`, since all global variables are undefined before each iteration of the script. However, under `Apache::Registry` and `mod_perl` handlers, all global variables must be initialized before they can be used.

The example can be fixed in a few ways. It's a good idea to always use the `strict` mode, which requires the global variables to be declared before they are used:

```
use strict;
use CGI;
use vars qw($top $q);
# init globals
$top = 0;
$q = undef;
# code
$q = CGI->new();
$topsecret = 1 if $q->param("secret") eq 'Muahaha';
# ...
```

But of course, the simplest solution is to avoid using globals where possible. Let's look at the example rewritten without globals:

```
use strict;
use CGI;
my $q = CGI->new();
my $topsecret = $q->param("secret") eq 'Muahaha' ? 1 : 0;
# ...
```

The last two versions of the example will run perfectly under `Apache::Registry`.

Here is another example that won't work correctly under `Apache::Registry`. This example presents a simple search engine script:

```
use CGI;
my $q = CGI->new();
print $q->header('text/plain');
my @data = read_data();
my $pat = $q->param("keyword");
foreach (@data) {
    print if /$pat/o;
}
```


The example retrieves some data using `read_data()` (e.g., lines in the text file), tries to match the keyword submitted by a user against this data, and prints the matching lines. The `/o` regular expression modifier is used to compile the regular expression only once, to speed up the matches. Without it, the regular expression will be recompiled as many times as the size of the `@data` array.

Now consider that someone is using this script to search for something inappropriate. Under `Apache::Registry`, the pattern will be cached and won't be recompiled in subsequent requests, meaning that the next person using this script (running in the same process) may receive something quite unexpected as a result. Oops.

The proper solution to this problem is discussed in Chapter 6, but `Apache::PerlRun` provides an immediate workaround, since it resets the regular expression cache before each request.

So why bother to keep your code clean? Why not use `Apache::PerlRun` all the time? As we mentioned earlier, the convenience provided by `Apache::PerlRun` comes at a price of performance deterioration.

In Chapter 9, we show in detail how to benchmark the code and server configuration. Based on the results of the benchmark, you can tune the service for the best performance. For now, let's just show the benchmark of the short script in Example 1-6.

Example 1-6. readdir.pl

```
use strict;

use CGI ();
use IO::Dir ();

my $q = CGI->new;
print $q->header("text/plain");
my $dir = IO::Dir->new(".");
print join "\n", $dir->read;
```

The script loads two modules (`CGI` and `IO::Dir`), prints the HTTP header, and prints the contents of the current directory. If we compare the performance of this script under `mod_cgi`, `Apache::Registry`, and `Apache::PerlRun`, we get the following results:

Mode	Requests/sec

Apache::Registry	473
Apache::PerlRun	289
mod_cgi	10

Because the script does very little, the performance differences between the three modes are very significant. `Apache::Registry` thoroughly outperforms `mod_cgi`, and you can see that `Apache::PerlRun` is much faster than `mod_cgi`, although it is still about twice as slow as `Apache::Registry`. The performance gap usually shrinks a bit as more code is added, as the overhead of `fork()` and code compilation becomes less

significant compared to execution times. But the benchmark results won't change significantly.

Jumping ahead, if we convert the script in Example 1-6 into a `mod_perl` handler, we can reach 517 requests per second under the same conditions, which is a bit faster than `Apache::Registry`. In Chapter 13, we discuss why running the code under the `Apache::Registry` handler is a bit slower than using a pure `mod_perl` content handler.

It can easily be seen from this benchmark that `Apache::Registry` is what you should use for your scripts to get the most out of `mod_perl`. But `Apache::PerlRun` is still quite useful for making an easy transition to `mod_perl`. With `Apache::PerlRun`, you can get a significant performance improvement over `mod_cgi` with minimal effort.

Later, we will see that `Apache::Registry`'s caching mechanism is implemented by compiling each script in its own namespace. `Apache::Registry` builds a unique package name using the script's name, the current URI, and the current virtual host (if any). `Apache::Registry` prepends a package statement to your script, then compiles it using Perl's `eval` function. In Chapter 6, we will show how exactly this is done.

What happens if you modify the script's file after it has been compiled and cached? `Apache::Registry` checks the file's last-modification time, and if the file has changed since the last compile, it is reloaded and recompiled.

In case of a compilation or execution error, the error is logged to the server's error log, and a server error is returned to the client.

Apache 1.3 Request Processing Phases

To understand `mod_perl`, you should understand how request processing works within Apache. When Apache receives a request, it processes it in 11 phases. For every phase, a standard default handler is supplied by Apache. You can also write your own Perl handlers for each phase; they will override or extend the default behavior. The 11 phases (illustrated in Figure 1-4) are:

Post-read-request

This phase occurs when the server has read all the incoming request's data and parsed the HTTP header. Usually, this stage is used to perform something that should be done once per request, as early as possible. Modules' authors usually use this phase to initialize per-request data to be used in subsequent phases.

URI translation

In this phase, the requested URI is translated to the name of a physical file or the name of a virtual document that will be created on the fly. Apache performs the translation based on configuration directives such as `ScriptAlias`. This translation can be completely modified by modules such as `mod_rewrite`, which register themselves with Apache to be invoked in this phase of the request processing.

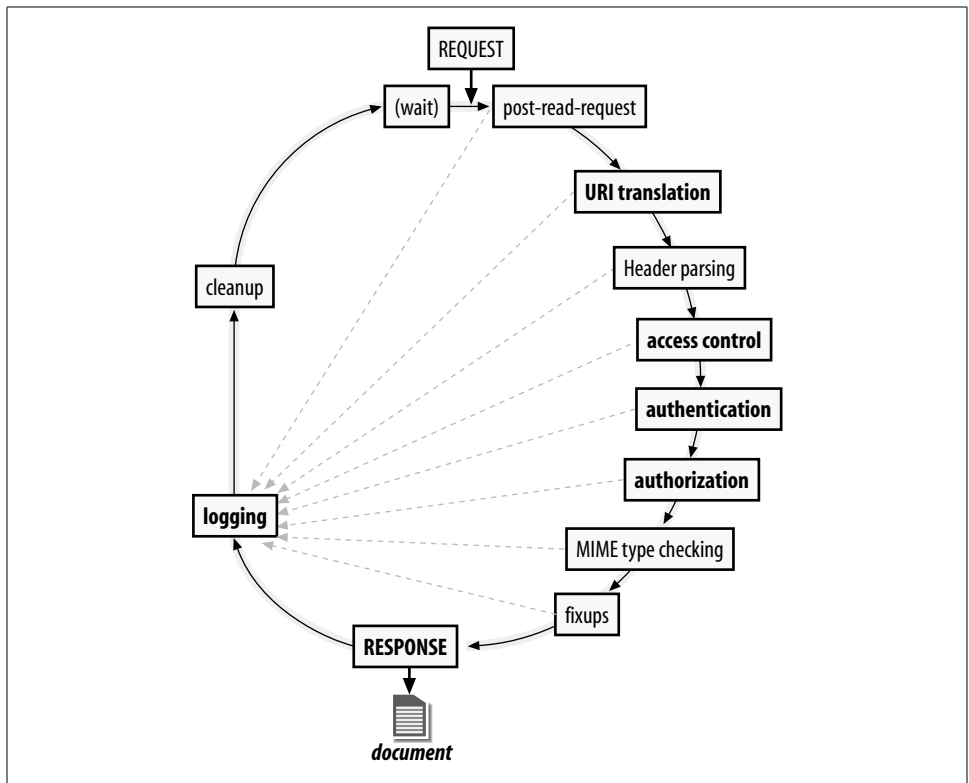


Figure 1-4. Apache 1.3 request processing phases

Header parsing

During this phase, you can examine and modify the request headers and take a special action if needed—e.g., blocking unwanted agents as early as possible.

Access control

This phase allows the server owner to restrict access to specific resources based on various rules, such as the client’s IP address or the day of week.

Authentication

Sometimes you want to make sure that a user really is who he claims to be. To verify his identity, challenge him with a question that only he can answer. Generally, the question is a login name and password, but it can be any other challenge that allows you to distinguish between users.

Authorization

The service might have various restricted areas, and you might want to allow the user to access some of these areas. Once a user has passed the authentication process, it is easy to check whether a specific location can be accessed by that user.

MIME type checking

Apache handles requests for different types of files in different ways. For static HTML files, the content is simply sent directly to the client from the filesystem. For CGI scripts, the processing is done by `mod_cgi`, while for `mod_perl` programs, the processing is done by `mod_perl` and the appropriate Perl handler. During this phase, Apache actually decides on which method to use, basing its choice on various things such as configuration directives, the filename's extension, or an analysis of its content. When the choice has been made, Apache selects the appropriate content handler, which will be used in the next phase.

Fixup

This phase is provided to allow last-minute adjustments to the environment and the request record before the actual work in the content handler starts.

Response

This is the phase where most of the work happens. First, the handler that generates the response (a content handler) sends a set of HTTP headers to the client. These headers include the Content-type header, which is either picked by the MIME-type-checking phase or provided dynamically by a program. Then the actual content is generated and sent to the client. The content generation might entail reading a simple file (in the case of static files) or performing a complex database query and HTML-ifying the results (in the case of the dynamic content that `mod_perl` handlers provide).

This is where `mod_cgi`, `Apache::Registry`, and other content handlers run.

Logging

By default, a single line describing every request is logged into a flat file. Using the configuration directives, you can specify which bits of information should be logged and where. This phase lets you hook custom logging handlers—for example, logging into a relational database or sending log information to a dedicated master machine that collects the logs from many different hosts.

Cleanup

At the end of each request, the modules that participated in one or more previous phases are allowed to perform various cleanups, such as ensuring that the resources that were locked but not freed are released (e.g., a process aborted by a user who pressed the Stop button), deleting temporary files, and so on.

Each module registers its cleanup code, either in its source code or as a separate configuration entry.

At almost every phase, if there is an error and the request is aborted, Apache returns an error code to the client using the default error handler (or a custom one, if provided).

Apache 1.3 Modules and the `mod_perl 1.0` API

The advantage of breaking up the request process into phases is that Apache gives a programmer the opportunity to “hook” into the process at any of those phases.

Apache has been designed with modularity in mind. A small set of core functions handle the basic tasks of dealing with the HTTP protocol and managing child processes. Everything else is handled by modules. The core supplies an easy way to plug modules into Apache at build time or runtime and enable them at runtime.

Modules for the most common tasks, such as serving directory indexes or logging requests, are supplied and compiled in by default. `mod_cgi` is one such module. Other modules are bundled with the Apache distribution but are not compiled in by default: this is the case with more specialized modules such as `mod_rewrite` or `mod_proxy`. There are also a vast number of third-party modules, such as `mod_perl`, that can handle a wide variety of tasks. Many of these can be found in the Apache Module Registry (<http://modules.apache.org/>).

Modules take control of request processing at each of the phases through a set of well-defined hooks provided by Apache. The subroutine or function in charge of a particular request phase is called a *handler*. These include authentication handlers such as `mod_auth_dbi`, as well as content handlers such as `mod_cgi`. Some modules, such as `mod_rewrite`, install handlers for more than one request phase.

Apache also provides modules with a comprehensive set of functions they can call to achieve common tasks, including file I/O, sending HTTP headers, or parsing URIs. These functions are collectively known as the Apache Application Programming Interface (API).

Apache is written in C and currently requires that modules be written in the same language. However, as we will see, `mod_perl` provides the full Apache API in Perl, so modules can be written in Perl as well, although `mod_perl` must be installed for them to run.

mod_perl 1.0 and the mod_perl API

Like other Apache modules, `mod_perl` is written in C, registers handlers for request phases, and uses the Apache API. However, `mod_perl` doesn't directly process requests. Rather, it allows you to write handlers in Perl. When the Apache core yields control to `mod_perl` through one of its registered handlers, `mod_perl` dispatches processing to one of the registered Perl handlers.

Since Perl handlers need to perform the same basic tasks as their C counterparts, `mod_perl` exposes the Apache API through a `mod_perl` API, which is a set of Perl functions and objects. When a Perl handler calls such a function or method, `mod_perl` translates it into the appropriate Apache C function.

Perl handlers extract the last drop of performance from the Apache server. Unlike `mod_cgi` and `Apache::Registry`, they are not restricted to the content generation phase and can be tied to any phase in the request loop. You can create your own custom authentication by writing a `PerlAuthenHandler`, or you can write specialized logging code in a `PerlLogHandler`.

Handlers are not compatible with the CGI specification. Instead, they use the `mod_perl` API directly for every aspect of request processing.

`mod_perl` provides access to the Apache API for Perl handlers via an extensive collection of methods and variables exported by the Apache core. This includes methods for dealing with the request (such as retrieving headers or posted content), setting up the response (such as sending HTTP headers and providing access to configuration information derived from the server's configuration file), and a slew of other methods providing access to most of Apache's rich feature set.

Using the `mod_perl` API is not limited to `mod_perl` handlers. `Apache::Registry` scripts can also call API methods, at the price of forgoing CGI compatibility.

We suggest that you refer to the book *Writing Apache Modules with Perl and C*, by Lincoln Stein and Doug MacEachern (O'Reilly), if you want to learn more about API methods.

References

- The CGI specification: <http://hoohoo.ncsa.uiuc.edu/cgi/>
- The HTTP/1.1 standard: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
- Various information about CGI at the W3C site: <http://www.w3.org/CGI/>
- MIME Media Types: <http://www.ietf.org/rfc/rfc2046.txt>
- The Apache Modules Registry: <http://modules.apache.org/>
- *Writing Apache Modules with Perl and C*, by Lincoln Stein and Doug MacEachern (O'Reilly); selected chapters available online at <http://www.modperl.com/>
- *mod_perl Developer's Cookbook*, by Geoffrey Young, Paul Lindner, and Randy Kobes (Sams Publishing); selected chapters available online at <http://www.modperlcookbook.org/>.
- *CGI Programming with Perl*, by Scott Guelich, Shishir Gundavaram, Gunther Birznieks (O'Reilly)