

Copyright © 2000, 1996 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.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly & Associates, Inc., is independent of Sun Microsystems.

The O'Reilly logo is a registered trademark 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. The use of the mouse image in association with the topic of CGI Programming is a trademark of O'Reilly & Associates, Inc.

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.

Preface

The first edition of *CGI Programming on the World Wide Web* was published in early 1996. The Web was very different then: the number of web hosts was 100,000, Netscape Navigator 2.0 (the first JavaScript™-enabled browser) was released, and Java was less than a year old and was used primarily for applets. The Web was still young, but it was developing quickly.

In 1996, CGI was the only stable and well-understood method for creating dynamic content on the Web. However, very few sites exploited its full potential. In the first edition, Shishir wrote:

Today's computer users expect custom answers to particular questions. Gone are the days when people were satisfied by the computing center staff passing out a single, general report to all users. Instead, each salesperson, manager, and engineer wants to enter specific queries and get up-to-date responses. And if a single computer can do that, why not the Web?

This is the promise of CGI. You can display sales figures for particular products month by month, as requested by your staff, using beautiful pie charts or plots. You can let customers enter keywords in order to find information on your products.

In 1996, these were bold claims. Today, they describe business as usual. That promise of CGI has certainly been fulfilled.

This book is about more than writing CGI scripts. It is about programming for the Web. Although we focus on CGI programming with Perl (thus the title change for this edition), many of the concepts we cover are common to all server-side web development. Even if you find yourself working with alternative technologies down the road, the effort you invest learning CGI now will continue to yield value later.

0.1. What's in the Book

Because CGI has changed so much in the last few years, it is only appropriate that this new edition reflect the changes. Thus, most of this book has been rewritten. New topics include CGI.pm, HTML templates, security, JavaScript, XML, search engines, style suggestions, and compatible, high-performance alternatives to CGI. Previous topics, such as session management, email, dynamic images, and relational databases, have been expanded and updated. Finally, we modified our presentation of CGI to begin with a discussion of HTTP, the underlying language of the Web. An understanding of HTTP provides a foundation for a more thorough understanding of CGI.

Despite the changes, the original goal of this book remains the same: to teach you everything you need to know to become a good CGI developer. This is not a learn-by-example book -- it isn't built around a

handful of CGI scripts with a discussion of how each script works. There are already lots of books like that available for CGI. While these books can certainly be useful, especially if one of the examples matches a particular challenge you are facing, they often teach *how* without explaining *why*. The aim of this book is to cover the fundamentals so that you can create CGI scripts to tackle any challenge. Don't worry, though, because we'll look at lots of examples. But our examples will serve to illustrate the discussion, rather than the other way around.

We should admit up front that there is a Unix bias in this book. Both Perl and CGI were originally conceived for the Unix platform, so it remains the most popular platform for Perl and CGI development. Of course, Perl and CGI support numerous other systems, including Microsoft's popular 32-bit Windows systems: Windows 95, Windows 98, Windows NT, and Windows 2000 (hereafter collectively referred to as *Win32*). Throughout this book, we will focus on Unix, but we'll also point out those things you need to be aware of when developing for non-Unix-compatible systems.

We use the Apache web server throughout our examples. There are several reasons: it is the most popular web server used today, it is available for the most platforms, it is free, it is open source, and it supports modules (such as *mod_perl* and *mod_fastcgi*) that improve both the power and the performance of Perl for web development.

Chapter 1. Getting Started

Contents:

[History](#)

[Introduction to CGI](#)

[Alternative Technologies](#)

[Web Server Configuration](#)

Like the rest of the Internet, the *Common Gateway Interface*, or *CGI*, has come a very long way in a very short time. Just a handful of years ago, CGI scripts were more of a novelty than practical; they were associated with hit counters and guestbooks, and were written largely by hobbyists. Today, CGI scripts, written by professional web developers, provide the logic to power much of the vast structure the Internet has become.

1.1. History

Despite the attention it now receives, the Internet is not new. In fact, the precursor to today's Internet began thirty years ago. The Internet began its existence as the ARPAnet, which was funded by the United States Department of Defense to study networking. The Internet grew gradually during its first 25 years, and then suddenly blossomed.

The Internet has always contained a variety of protocols for exchanging information, but when web browsers such as NCSA Mosaic and, later, Netscape Navigator appeared, they spurred an explosive growth. In the last six years, the number of web hosts alone has grown from under a thousand to more than ten million. Now, when people hear the term Internet, most think of the Web. Other protocols, such as those for email, FTP, chat, and news, certainly remain popular, but they have become secondary to the Web, as more people are using web sites as their gateway to access these other services.

The Web was by no means the first technology available for publishing and exchanging information, but there was something different about the Web that prompted its explosive growth. We'd love to tell you that CGI was the sole factor for the Web's early growth over protocols like FTP and Gopher. But that wouldn't be true. Probably the real reason the Web gained popularity initially was because it came with pictures. The Web was designed to present multiple forms of media: browsers supported inlined images almost from the start, and HTML supported rudimentary layout control that made information easier to present and read. This control continued to increase as Netscape added support for new extensions to HTML with each successive release of the browser.

Thus initially, the Web grew into a collection of personal home pages and assorted web sites containing a variety of miscellaneous information. However, no one really knew what to *do* with it, especially businesses. In 1995, a common refrain in corporations was "Sure the Internet is great, but how many people have actually made money online?" How quickly things change.

1.1.1. How CGI Is Used Today

Today, e-commerce has taken off and dot-com startups are appearing everywhere. Several technologies have been fundamental to this progress, and CGI is certainly one of the most important. CGI allows the Web to *do* things, to be more than a collection of static resources. A *static* resource is something that does not change from request to request, such as an HTML file or a graphic. A *dynamic* resource is one that contains information that may vary with each request, depending on any number of conditions including a changing data source (like a database), the identity of the user, or input from the user. By supporting dynamic content, CGI allows web servers to provide online applications that users from around the world on various platforms can all access via a standard client: a web browser.

It is difficult to enumerate all that CGI can do, because it does so much. If you perform a search on a web site, a CGI application is probably processing your information. If you fill out a registration form on the Web, a CGI application is probably processing your information. If you make an online purchase, a CGI application is probably validating your credit card and logging the transaction. If you view a chart online that dynamically displays information graphically, chances are that a CGI application created that chart. Of course, over the last few years other technologies have appeared to handle dynamic tasks like these; we'll look at some of those in a moment. However, CGI remains the most popular way to do these tasks and more.

Chapter 2. The Hypertext Transport Protocol

Contents:

[URLs](#)

[HTTP](#)

[Browser Requests](#)

[Server Responses](#)

[Proxies](#)

[Content Negotiation](#)

[Summary](#)

The Hypertext Transport Protocol (HTTP) is the common language that web browsers and web servers use to communicate with each other on the Internet. CGI is built on top of HTTP, so to understand CGI fully, it certainly helps to understand HTTP. One of the reasons CGI is so powerful is because it allows you to manipulate the metadata exchanged between the web browser and server and thus perform many useful tricks, including:

- Serve content of varying type, language, or other encoding according to the client's needs.
- Check the user's previous location.
- Check the browser type and version and adapt your response to it.
- Specify how long the client can cache a page before it is considered outdated and should be reloaded.

We won't cover all of the details of HTTP, just what is important for our understanding of CGI. Specifically, we'll focus on the request and response process: how browsers ask for and receive web pages.

If you are interested in understanding more about HTTP than we provide here, visit the World Wide Web Consortium's web site at <http://www.w3.org/Protocols/>. On the other hand, if you are eager to get started writing CGI scripts, you may be tempted to skip this chapter. We encourage you not to. Although you can certainly learn to write CGI scripts without learning HTTP, without the bigger picture you may end up memorizing what to do instead of understanding why. This is certainly the most challenging chapter, however, because we cover a lot of material without many examples. So if you find it a little dry and want to peek ahead to the fun stuff, we'll forgive you. Just be sure to return here later.

2.1. URLs

During our discussion of HTTP and CGI, we will be often be referring to *URLs*, or *Uniform Resource Locators*. If you have used the Web at all, then you are probably familiar with URLs. In web terms, a *resource* represents anything available on the web, whether it be an HTML page, an image, a CGI script, etc. URLs provide a standard way to locate these resources on the Web.

Note that URLs are not actually specific to HTTP; they can refer to resources in many protocols. Our discussion here will focus strictly on HTTP URLs.

What About URIs?

You may have also encountered the term URI and wondered about the difference between a URI and a URL. Actually, the terms are often interchangeable because all URLs are URIs. Uniform Resource Identifiers (URIs) are a more generalized class which includes URLs as well as Uniform Resource Names (URNs). A URN provides a name that sticks to an object even though the location of the object may move around. You can think of it this way: your name is similar to a URN, while your address is similar to a URL. Both serve to identify you in some way, and in this manner both are URIs.

Because URNs are just a concept and are not used on the Web today, you can safely think of URIs and URLs as interchangeable terms and not let the terminology throw you. Since we are not interested in other forms of URIs, we will try to avoid confusion altogether by just using the term URL in the text.

2.1.1. Elements of a URL

HTTP URLs consist of a scheme, a host name, a port number, a path, a query string, and a fragment identifier, any of which may be omitted under certain circumstances (see [Figure 2-1](#)).

Figure 2-1. Components of a URL

HTTP URLs contain the following elements:

Scheme

The scheme represents the protocol, and for our purposes will either be `http` or `https`. `https` represents a connection to a secure web server. Refer to [the sidebar "The Secure Sockets Layer"](#) later in this chapter.

Host

The host identifies the machine running a web server. It can be a domain name or an IP address, although it is a bad idea to use IP addresses in URLs and is strongly discouraged. The problem is that IP addresses often change for any number of reasons: a web site may move from one machine to another, or it may relocate to another network. Domain names can remain constant in these cases, allowing these changes to remain hidden from the user.

Port number

The port number is optional and may appear in URLs only if the host is also included. The host and port are separated by a colon. If the port is not specified, port 80 is used for `http` URLs and port 443 is used for `https` URLs.

It is possible to configure a web server to answer other ports. This is often done if two different web servers need to operate on the same machine, or if a web server is operated by someone who does not have sufficient rights on the machine to start a server on these ports (e.g., only `root` may bind to ports below 1024 on Unix machines). However, servers using ports other than the standard 80 and 443 may be inaccessible to users behind firewalls. Some firewalls are configured to restrict access to all but a narrow set of ports representing the defaults for certain allowed protocols.

Path information

Path information represents the location of the resource being requested, such as an HTML file or a CGI script. Depending on how your web server is configured, it may or may not map to some actual file path on your system. As we mentioned last chapter, the URL path for CGI scripts generally begin with `/cgi/` or `/cgi-bin/` and these paths are mapped to a similarly-named directory in the web server, such as `/usr/local/apache/cgi-bin`.

Note that the URL for a script may include path information beyond the location of the script itself. For example, say you have a CGI at:

```
http://localhost/cgi/browse_docs.cgi
```

You can pass extra path information to the script by appending it to the end, for example:

```
http://localhost/cgi/browse_docs.cgi/docs/product/description.text
```

Here the path `/docs/product/description.text` is passed to the script. We explain how to access and use this additional path information in more detail in the next chapter.

Query string

A query string passes additional parameters to scripts. It is sometimes referred to as a search string or an index. It may contain name and value pairs, in which each pair is separated from the next pair by an ampersand (&), and the name and value are separated from each other by an equals sign (=). We discuss how to parse and use this information in your scripts in the next chapter.

Query strings can also include data that is not formatted as name-value pairs. If a query string does not contain an equals sign, it is often referred to as an index. Each argument should be separated from the next by an encoded space (encoded either as + or %20; see [Section 2.1.3, "URL Encoding"](#) below). CGI scripts handle indexes a little differently, as we will see in the next chapter.

Fragment identifier

Fragment identifiers refer to a specific section in a resource. Fragment identifiers are not sent to web servers, so you cannot access this component of the URLs in your CGI scripts. Instead, the browser fetches a resource and then applies the fragment identifier to locate the appropriate section in the resource. For HTML documents, fragment identifiers refer to anchor tags within the document:

```
<a name="anchor" >Here is the content you're after...</a>
```

The following URL would request the full document and then scroll to the section marked by the anchor tag:

```
http://localhost/document.html#anchor
```

Web browsers generally jump to the bottom of the document if no anchor for the fragment identifier is found.

2.1.2. Absolute and Relative URLs

Many of the elements within a URL are optional. You may omit the scheme, host, and port number in a URL if the URL is used in a context where these elements can be assumed. For example, if you include a URL in a link on an HTML page and leave out these elements, the browser will assume the link applies to a resource on the same machine as the link. There are two classes of URLs:

Absolute URL

URLs that include the hostname are called absolute URLs. An example of an absolute URL is *http://localhost/cgi/script.cgi*.

Relative URL

URLs without a scheme, host, or port are called relative URLs. These can be further broken down into full and relative paths:

Full paths

Relative URLs with an absolute path are sometimes referred to as *full paths* (even though they can also include a query string and fragment identifier). Full paths can be distinguished from URLs with relative paths because they always start with a forward

slash. Note that in all these cases, the paths are virtual paths, and do not necessarily correspond to a path on the web server's filesystem. An example of an absolute path is */index.html*.

Relative paths

Relative URLs that begin with a character other than a forward slash are *relative paths*. Examples of relative paths include *script.cgi* and *../images/photo.jpg*.

2.1.3. URL Encoding

Many characters must be encoded within a URL for a variety of reasons. For example, certain characters such as `?`, `#`, and `/` have special meaning within URLs and will be misinterpreted unless encoded. It is possible to name a file *doc#2.html* on some systems, but the URL *http://localhost/doc#2.html* would not point to this document. It points to the fragment *2.html* in a (possibly nonexistent) file named *doc*. We must encode the `#` character so the web browser and server recognize that it is part of the resource name instead.

Characters are encoded by representing them with a percent sign (`%`) followed by the two-digit hexadecimal value for that character based upon the ISO Latin 1 character set or ASCII character set (these character sets are the same for the first eight bits). For example, the `#` symbol has a hexadecimal value of `0x23`, so it is encoded as `%23`.

The following characters must be encoded:

- Control characters: ASCII `0x00` through `0x1F` plus `0x7F`
- Eight-bit characters: ASCII `0x80` through `0xFF`
- Characters given special importance within URLs: `;` `/` `?` `:` `@` `&` `=` `+` `$` `,`
- Characters often used to delimit (quote) URLs: `<` `>` `#` `%` `"`
- Characters considered unsafe because they may have special meaning for other protocols used to transmit URLs (e.g., SMTP): `{` `}` `|` `\` `^` `[` `]` ```

Additionally, spaces should be encoded as `+` although `%20` is also allowed. As you can see, most characters must be encoded; the list of allowed characters is actually much shorter:

- Letters: `a-z` and `A-Z`
- Digits: `0-9`
- The following characters: `-` `_` `.` `!` `~` `*` `'` `(` `)`

It is actually permissible and not uncommon for any of the allowed characters to also be encoded by some software. Thus, any application that decodes a URL must decode every occurrence of a percentage sign followed by any two hexadecimal digits.

The following code encodes text for URLs:

```
sub url_encode {
    my $text = shift;
    $text =~ s/([a-z0-9_!~*'() -])/sprintf "%%02X", ord($1)/ei;
    $text =~ tr/ /+/;
    return $text;
}
```

Any character not in the allowed set is replaced by a percentage sign and its two-digit hexadecimal equivalent. The three percentage signs are necessary because percentage signs indicate format codes for `sprintf`, and literal percentage signs must be indicated by two percentage signs. Our format code thus includes a percentage sign, `%%`, plus the format code for two hexadecimal digits, `%02X`.

Code to decode URL encoded text looks like this:

```
sub url_decode {
    my $text = shift;
    $text =~ tr/\+/ /;
    $text =~ s/%([a-f0-9][a-f0-9])/chr( hex( $1 ) )/ei;
    return $text;
}
```

Here we first translate any plus signs to spaces. Then we scan for a percentage sign followed by two hexadecimal digits and use Perl's `chr` function to convert the hexadecimal value into a character.

Neither the encoding nor the decoding operations can be safely repeated on the same text. Text encoded twice differs from text encoded once because the percentage signs introduced in the first step would themselves be encoded in the second. Likewise, you cannot encode or decode entire URLs. If you were to decode a URL, you could no longer reliably parse it, for you may have introduced characters that would be misinterpreted such as `/` or `?`. You should always parse a URL to get the components you want before decoding them; likewise, encode components before building them into a full URL.

Note that it's good to understand how a wheel works but reinventing it would be pointless. Even though you have just seen how to encode and decode text for URLs, you shouldn't do so yourself. The `URI::URL` module (actually it is a collection of modules), available on CPAN (see [Appendix B, "Perl Modules"](#)), provides many URL-related modules and functions. One of the included modules, `URI::Escape`, provides the `url_escape` and `url_unescape` functions. Use them. The subroutines in these modules have been vigorously tested, and future versions will reflect any changes to HTTP as it evolves. [\[2\]](#) Using standard subroutines will also make your code much clearer to those who may have to maintain your code later (this includes you).

[2]Don't think this could happen? What if we told you the tilde character~() was not always allowed in URLs? This restriction was removed after it became common practice for some web servers to accept a tilde plus username in the path to indicate a user's personal web directory.

If, despite these warnings, you still insist on writing your own decoding code yourself, at least place it in appropriately named subroutines. Granted, some of these actions take only a line or two of code, but the code is quite cryptic, and these operations should be clearly labeled.

Chapter 3. The Common Gateway Interface

Contents:

[The CGI Environment](#)

[Environment Variables](#)

[CGI Output](#)

[Examples](#)

Now that we have explored HTTP in general, we can return to our discussion of CGI and see how our scripts interact with HTTP servers to produce dynamic content. After you have read this chapter, you'll understand how to write basic CGI scripts and fully understand all of our previous examples. Let's get started by looking at a script now.

This script displays some basic information, including CGI and HTTP revisions used for this transaction and the name of the server software:

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

print <<END_OF_HTML;
Content-type: text/html

<HTML>
<HEAD>
  <TITLE>About this Server</TITLE>
</HEAD>
<BODY>
<H1>About this Server</H1>
<HR>
<PRE>
  Server Name:          $ENV{SERVER_NAME}
  Listening on Port:    $ENV{SERVER_PORT}
  Server Software:     $ENV{SERVER_SOFTWARE}
  Server Protocol:     $ENV{SERVER_PROTOCOL}
  CGI Version:         $ENV{GATEWAY_INTERFACE}
</PRE>
<HR>
</BODY>
</HTML>
END_OF_HTML
```

When you request the URL for this CGI script, it produces the output shown in [Figure 3-1](#).



Figure 3-1. Output from server_info.cgi

This simple example demonstrates the basics about how scripts work with CGI:

- The web server passes information to CGI scripts via environment variables, which the script accesses via the `%ENV` hash.
- CGI scripts produce output by printing an HTTP message on `STDOUT`.
- CGI scripts do not need to output full HTTP headers. This script outputs only one HTTP header, *Content-type*.

These details define what we will call the *CGI environment*. Let's explore this environment in more detail.

3.1. The CGI Environment

CGI establishes a particular environment in which CGI scripts operate. This environment includes such things as what current working directory the script starts in, what variables are preset for it, where the standard file handles are directed, and so on. In return, CGI requires that scripts be responsible for defining the content of the HTTP response and at least a minimal set of HTTP headers.

When CGI scripts are executed, their current working directory is typically the directory in which they reside on the web server; at least this is the recommended behavior according to the CGI standard, though it is not supported by all web servers (e.g., Microsoft's IIS). CGI scripts are generally executed with limited permissions. On Unix systems, CGI scripts execute with the same permission as the web server which is generally a special user such as *nobody*, *web*, or *www*. On other operating systems, the web server itself may need to be configured to set the permissions that CGI scripts have. In any event, CGI scripts should not be able to read and write to all areas of the file system. You may think this is a problem, but it is actually a good thing as you will learn in our security discussion in [Chapter 8](#),

["Security"](#).

3.1.1. File Handles

Perl scripts generally start with three standard file handles predefined: STDIN, STDOUT, and STDERR. CGI Perl scripts are no different. These file handles have particular meaning within a CGI script, however.

3.1.1.1. STDIN

When a web server receives an HTTP request directed to a CGI script, it reads the HTTP headers and passes the content body of the message to the CGI script on STDIN. Because the headers have already been removed, STDIN will be empty for GET requests that have no body and contain the encoded form data for POST requests. Note that there is no end-of-file marker, so if you try to read more data than is available, your CGI script will hang, waiting for more data on STDIN that will never come (eventually, the web server or browser should time out and kill this CGI script but this wastes system resources). Thus, you should never try to read from STDIN for GET requests. For POST requests, you should always refer to the value of the *Content-Length* header and read only that many bytes. We'll see how to read this information in [Chapter 4, "Forms and CGI"](#) in [Chapter 4, "Forms and CGI"](#).

3.1.1.2. STDOUT

Perl CGI scripts return their output to the web server by printing to STDOUT. This may include some HTTP headers as well as the content of the response, if present. Perl generally buffers output on STDOUT and sends it to the web server in chunks. The web server itself may wait until the entire output of the script has finished before sending it onto the client. For example, the iPlanet (formerly Netscape) Enterprise Server buffers output, while Apache (1.3 and higher) does not.

3.1.1.3. STDERR

CGI does not designate how web servers should handle output to STDERR, and servers implement this in different ways, but they almost always produces a *500 Internal Server Error* reply. Some web servers, like Apache, append STDERR output to the web server's error log, which includes other errors such as authorization failures and requests for documents not on the server. This is very helpful for debugging errors in CGI scripts.

Other servers, such as those by iPlanet, do not distinguish between STDOUT and STDERR; they capture both as output from the script and return them to the client. Nevertheless, outputting data to STDERR will typically produce a server error because Perl does not buffer STDERR, so data printed to

STDERR often arrives at the web server before data printed to STDOUT. The web server will then report an error because it expects the output to start with a valid header, not the error message. On iPlanet, only the server's error message, and not the complete contents of STDERR, is then logged.

We'll discuss strategies for handling STDERR output in our discussion of CGI script debugging in [Chapter 15, "Debugging CGI Applications"](#).

Chapter 4. Forms and CGI

Contents:

[Sending Data to the Server](#)

[Form Tags](#)

[Decoding Form Input](#)

HTML forms are the user interface that provides input to your CGI scripts. They are primarily used for two purposes: collecting data and accepting commands. Examples of data you collect may include registration information, payment information, and online surveys. You may also collect commands via forms, such as using menus, checkboxes, lists, and buttons to control various aspects of your application. In many cases, your forms will include elements for both: collecting data as well as application control.

A great advantage of HTML forms is that you can use them to create a frontend for numerous gateways (such as databases or other information servers) that can be accessed by any client without worrying about platform dependency.

In order to process data from an HTML form, the browser must send the data via an HTTP request. A CGI script cannot check user input on the client side; the user must press the submit button and the input can only be validated once it has travelled to the server. JavaScript, on the other hand, can perform actions in the browser. It can be used in conjunction with CGI scripts to provide a more responsive user interface. We will see how to do this in [Chapter 7, "JavaScript"](#).

This chapter covers:

- How form data is sent to the server
- How to use HTML tags for writing forms
- How CGI scripts decode the form data

4.1. Sending Data to the Server

In the last couple of chapters, we have referred to the options that a browser can include with an HTTP request. In the case of a GET request, these options are included as the query string portion of the URL passed in the request line. In the case of a POST request, these options are included as the content of the HTTP request. These options are typically generated by HTML forms.

Each HTML form element has an associated name and value, like this checkbox:

```
<INPUT TYPE="checkbox" NAME="send_email" VALUE="yes">
```

If this checkbox is checked, then the option `send_email` with a value of `yes` is sent to the web server. Other form elements, which we will look at in a moment, act similarly. Before the browser can send form option data to the server, the browser must encode it. There are currently two different forms of encoding form data. The default encoding, which has the media type of `application/x-www-form-urlencoded`, is used almost exclusively. The other form of encoding, *multipart/form-data*, is primarily used with forms which allow the user to upload files to the web server. We will look at this in [Section 5.2.4, "File Uploads with CGI.pm"](#).

For now, let's look at how `application/x-www-form-urlencoded` works. As we mentioned, each HTML form element has a name and a value attribute. First, the browser collects the names and values for each element in the form. It then takes these strings and encodes them according to the same rules for encoding URL text that we discussed in [Chapter 2, "The Hypertext Transport Protocol"](#). If you recall, characters that have special meaning for HTTP are replaced with a percentage symbol and a two-digit hexadecimal number; spaces are replaced with `+`. For example, the string "Thanks for the help!" would be converted to "Thanks+for+the+help%21".

Next, the browser joins each name and value with an equals sign. For example, if the user entered "30" when asked for the age, the key-value pair would be "age=30". Each key-value pair is then joined, using the "&" character as a delimiter. Here is an example of an HTML form:

```
<HTML>
<HEAD>
  <TITLE>Mailing List</TITLE>
</HEAD>

<BODY>
<H1>Mailing List Signup</H1>
<P>Please fill out this form to be notified via email about
  updates and future product announcements.</P>

<FORM ACTION="/cgi/register.cgi" METHOD="POST">
  <P>
    Name: <INPUT TYPE="TEXT" NAME="name"><BR>
    Email: <INPUT TYPE="TEXT" NAME="email">
  </P>

  <HR>
  <INPUT TYPE="SUBMIT" VALUE="Submit Registration Info">
</FORM>

</BODY>
</HTML>
```

[Figure 4-1](#) shows how the form looks in Netscape with some sample input.

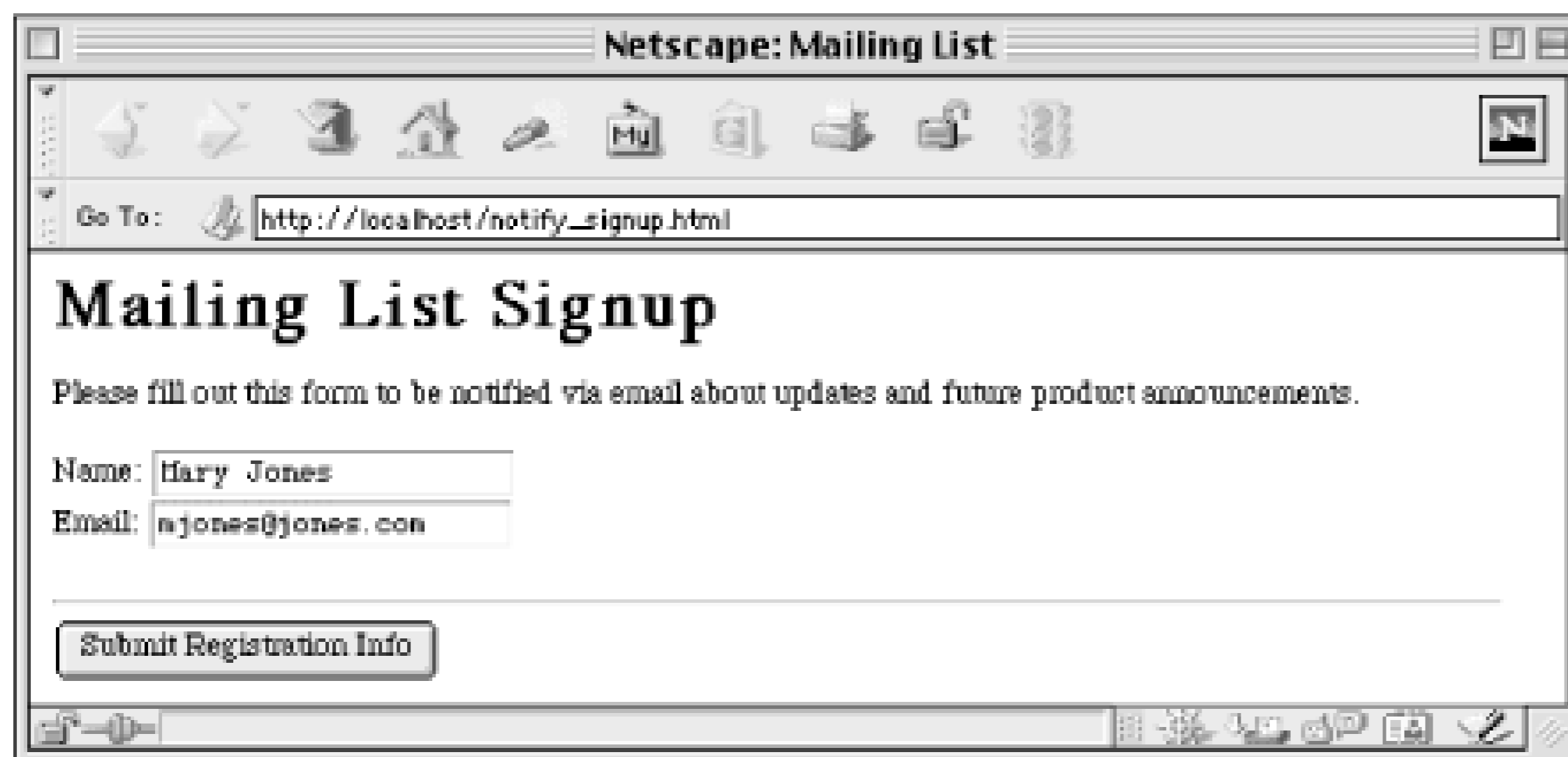


Figure 4-1. Sample HTML form

When this form is submitted, the browser encodes these three elements as:

```
name=Mary+Jones&email=mjones%40jones.com
```

Since the request method is POST in this example, this string would be added to the HTTP request as the content of that message. The HTTP request message would look like this:

```
POST /cgi/register.cgi HTTP/1.1
Host: localhost
Content-Length: 67
Content-Type: application/x-www-form-urlencoded
```

```
name=Mary+Jones&email=mjones%40jones.com
```

If the request method were set to GET, then the request would be formatted this way instead:

```
GET /cgi/register.cgi?name=Mary+Jones&email=mjones%40jones.com HTTP/1.1
Host: localhost
```

Chapter 5. CGI.pm

Contents:

Overview

Handling Input with CGI.pm

Generating Output with CGI.pm

Alternatives for Generating Output

Handling Errors

The CGI.pm module has become the standard tool for creating CGI scripts in Perl. It provides a simple interface for most of the common CGI tasks. Not only does it easily parse input parameters, but it also provides a clean interface for outputting headers and a powerful yet elegant way to output HTML code from your scripts.

We will cover most of the basics here and will revisit CGI.pm later to look at some of its other features when we discuss other components of CGI programming. For example, CGI.pm provides a simple way to read and write to browser cookies, but we will wait to review that until we get to our discussion about maintaining state, in Chapter 11, "Maintaining State" .

If after reading this chapter you are interested in more information, the author of CGI.pm has written an entire book devoted to it: *The Official Guide to Programming with CGI.pm* by Lincoln Stein (John Wiley & Sons).

Because CGI.pm offers so many methods, we'll organize our discussion of CGI.pm into three parts: handling input, generating output, and handling errors. We will look at ways to generate output both with and without CGI.pm. Here is the structure of our chapter:

- Handling Input with CGI.pm
 - *Information about the environment* . CGI.pm has methods that provide information that is similar, but somewhat different from the information available in `%ENV` .
 - *Form input* . CGI.pm automatically parses parameters passed to you via HTML forms and provides a simple method for accessing these parameters.
 - *File uploads* . CGI.pm allows your CGI script to handle HTTP file uploads easily and transparently.
- Generating Output with CGI.pm

- *Generating headers* . CGI.pm has methods to help you output HTTP headers from your CGI script.
- *Generating HTML* . CGI.pm allows you to generate full HTML documents via corresponding method calls.
- Alternatives for Generating Output
 - *Quoted HTML and here documents*. We will compare alternative strategies for outputting HTML.
- Handling Errors
 - *Trapping die*. The standard way to handle errors with Perl, `die` , does not work cleanly with CGI.
 - *CGI::Carp*. The CGI::Carp module distributed with CGI.pm makes it easy to trap `die` and other error conditions that may kill your script.
 - *Custom solutions*. If you want more control when displaying errors to your users, you may want to create a custom subroutine or module.

Let's start with a general overview of CGI.pm.

5.1. Overview

CGI.pm requires Perl 5.003_07 or higher and has been included with the standard Perl distribution since 5.004. You can check which version of Perl you are running with the `-v` option:

```
$ perl -v
```

```
This is perl, version 5.005
```

```
Copyright 1987-1997, Larry Wall
```

```
Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5.0 source kit.
```

You can verify whether CGI.pm is installed and which version by doing this:

```
$ perl -MCGI -e 'print "CGI.pm version $CGI::VERSION\n";'
CGI.pm version 2.56
```

If you get something like the following, then you do not have CGI.pm installed, and you will have to

download and install it. Appendix B, "Perl Modules" , explains how to do this.

```
Can't locate CGI.pm in @INC (@INC contains: /usr/lib/perl5/i386-linux/5.005 /usr
lib/perl5 /usr/lib/perl5/site_perl/i386-linux /usr/lib/perl5/site_perl .).
BEGIN failed--compilation aborted.
```

New versions of CGI.pm are released regularly, and most releases include bug fixes.[6] We therefore recommend that you install the latest version and monitor new releases (you can find a version history at the bottom of the *cgi_docs.html* file distributed with CGI.pm). This chapter discusses features introduced as late as 2.47.

[6]These are not necessarily bugs in CGI.pm; CGI.pm strives to maintain compatibility with new servers and browsers that sometimes include buggy, or at least nonstandard, code.

5.1.1. Denial of Service Attacks

Before we get started, you should make a minor change to your copy of CGI.pm. CGI.pm handles HTTP file uploads and automatically saves the contents of these uploads to temporary files. This is a very convenient feature, and we'll talk about this later. However, file uploads are enabled by default in CGI.pm, and it does not impose any limitations on the size of files it will accept. Thus, it is possible for someone to upload multiple large files to your web server and fill up your disk.

Clearly, the vast majority of your CGI scripts do not accept file uploads. Thus, you should disable this feature and enable it only in those scripts where you wish to use it. You may also wish to limit the size of POST requests, which includes file uploads as well as standard forms submitted via the POST method.

To make these changes, locate CGI.pm in your Perl libraries and then search for text that looks like the following:

```
# Set this to a positive value to limit the size of a POSTing
# to a certain number of bytes:
$POST_MAX = -1;

# Change this to 1 to disable uploads entirely:
$DISABLE_UPLOADS = 0;
```

Set `$DISABLE_UPLOADS` to 1. You may wish to set `$POST_MAX` to a reasonable upper bound as well, such as 100KB. POST requests that are not file uploads are processed in memory, so restricting the size of POST requests avoids someone submitting multiple large POST requests that quickly use up available memory on your server. The result looks like this:

```
# Set this to a positive value to limit the size of a POSTing
# to a certain number of bytes:
$POST_MAX = 102_400; # 100 KB
```

```
# Change this to 1 to disable uploads entirely:
$DISABLE_UPLOADS = 1;
```

If you then want to enable uploads and/or allow a greater size for POST requests, you can override these values in your script by setting `$CGI::DISABLE_UPLOADS` and `$CGI::POST_MAX` after you use the `CGI.pm` module, but before you create a `CGI.pm` object. We will look at how to receive file uploads later in this chapter.

You may need special permission to update your `CGI.pm` file. If your system administrator for some reason will not make these changes, then you must disable file uploads and limit POST requests on a script by script basis. Your scripts should begin like this:

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

use strict;
use CGI;

$CGI::DISABLE_UPLOADS = 1;
$CGI::POST_MAX        = 102_400; # 100 KB

my $q = new CGI;
.
```

Throughout our examples, we will assume that the module has been patched and omit these lines.

5.1.2. The Kitchen Sink

`CGI.pm` is a big module. It provides functions for accessing CGI environment variables and printing outgoing headers. It automatically interprets form data submitted via POST, via GET, and handles multipart-encoded file uploads. It provides many utility functions to do common CGI-related tasks, and it provides a simple interface for outputting HTML. This interface does not eliminate the need to understand HTML, but it makes including HTML inside a Perl script more natural and easier to validate.

Because `CGI.pm` is so large, some people consider it bloated and complain that it wastes memory. In fact, it uses many creative ways to increase the efficiency of `CGI.pm` including a custom implementation of `SelfLoader`. This means that it loads only code that you need. If you use `CGI.pm` only to parse input, but do not use it to produce HTML, then `CGI.pm` does not load the code for producing HTML.

There have also been some alternative, lightweight CGI modules written. One of the lightweight alternatives to `CGI.pm` was begun by David James; he got together with Lincoln Stein and the result is a new and improved version of `CGI.pm` that is even smaller, faster, and more modular than the original. It should be available as `CGI.pm 3.0` by the time you read this book.

5.1.3. Standard and Object-Oriented Syntax

CGI.pm, like Perl, is powerful yet flexible. It supports two styles of usage: a standard interface and an object-oriented interface. Internally, it is a fully object-oriented module. Not all Perl programmers are comfortable with object-oriented notation, however, so those developers can instead request that CGI.pm make its subroutines available for the developer to call directly.

Here is an example. The object-oriented syntax looks like this:

```
use strict;
use CGI;

my $q      = new CGI;
my $name = $q->param( "name" );

print $q->header( "text/html" ),
      $q->start_html( "Welcome" ),
      $q->p( "Hi $name!" ),
      $q->end_html;
```

The standard syntax looks like this:

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

my $name = param( "name" );

print header( "text/html" ),
      start_html( "Welcome" ),
      p( "Hi $name!" ),
      end_html;
```

Don't worry about the details of what the code does right now; we will cover all of it during this chapter. The important thing to notice is the different syntax. The first script creates a CGI.pm object and stores it in `$q` (`$q` is short for *query* and is a common convention for CGI.pm objects, although `$cgi` is used sometimes, too). Thereafter, all the CGI.pm functions are preceded by `$q->`. The second asks CGI.pm to export the standard functions and simply uses them directly. CGI.pm provides several predefined groups of functions, like `:standard`, that can be exported into your CGI script.

The standard CGI.pm syntax certainly has less noise. It doesn't have all those `$q->` prefixes. Aesthetics aside, however, there are good arguments for using the object oriented syntax with CGI.pm.

Exporting functions has its costs. Perl maintains a separate namespace for different chunks of code referred to as packages. Most modules, like CGI.pm, load themselves into their own package. Thus, the functions and variables that modules see are different from the modules and variables you see in your scripts. This is a good thing, because it prevents collisions between variables and functions in different packages that happen to have the same name. When a module exports symbols (whether they are variables or functions),

Perl has to create and maintain an alias of each of these symbols in your program's namespace, the *main* namespace. These aliases consume memory. This memory usage becomes especially critical if you decide to use your CGI scripts with FastCGI or *mod_perl*.

The object-oriented syntax also helps you avoid any possible collisions that would occur if you create a subroutine with the same name as one of CGI.pm's exported subroutines. Also, from a maintenance standpoint, it is clear from looking at the object-oriented script where the code for the header function is: it's a method of a CGI.pm object, so it must be in the CGI.pm module (or one of its associated modules). Knowing where to look for the header function in the second example is much more difficult, especially if your CGI scripts grow large and complex.

Some people avoid the object-oriented syntax because they believe it is slower. In Perl, methods typically are slower than functions. However, CGI.pm is truly an object-oriented module at heart, and in order to provide the function syntax, it must do some fancy footwork to manage an object for you internally. Thus with CGI.pm, the object-oriented syntax is not any slower than the function syntax. In fact, it can be slightly faster.

We will use CGI.pm's object-oriented syntax in most of our examples.

Chapter 6. HTML Templates

Contents:

[Reasons for Using Templates](#)

[Server Side Includes](#)

[HTML::Template](#)

[Embperl](#)

[Mason](#)

The CGI.pm module makes it much easier to produce HTML code from CGI scripts written in Perl. If your goal is to produce self-contained CGI applications that include both the program logic and the interface (HTML), then CGI.pm is certainly the best tool for this. It excels for distributable applications because you do not need to distribute separate HTML files, and it's easy for developers to follow when reading through code. For this reason, we use it in the majority of the examples in this book. However, in some circumstances, there are good reasons for separating the interface from the program logic. In these circumstances, templates may be a better solution.

6.1. Reasons for Using Templates

HTML design and CGI development involve very different skill sets. Good HTML design is typically done by artists or designers in collaboration with marketing folks and people skilled in interface design. CGI development may also involve input from others, but it is very technical in nature. Therefore, CGI developers are often not responsible for creating the interface to their applications. In fact, sometimes they are given non-functional prototypes and asked to provide the logic to drive it. In this scenario, the HTML is already available and translating it into code involves extra work.

Additionally, CGI applications rarely remain static; they require maintenance. Inevitably, bugs are found and fixed, new features are added, the wording is changed, or the site is redesigned with a new color scheme. These changes can involve either the program logic or the interface, but interface changes are often the most common and the most time consuming. Making specific changes to an existing HTML file is generally easier than modifying a CGI script, and many organizations have more people who understand HTML than who understand Perl.

There are many different ways to use HTML templates, and it is very common for web developers to create their own custom solutions. However, the many various solutions can be grouped into a few different approaches. In this chapter, we'll explore each approach by looking at the most powerful and popular solutions for each.

6.1.1. Rolling Your Own

One thing we won't do in this chapter is present a novel template parser or explain how to write your own. The reason is that there are already too many good solutions to warrant this. Of the many web developers out there who have created their own proprietary systems for handling templates, most turn to something else after some time. In fact, one of your authors has experience doing just this.

The first custom template system I developed was like SSI but with control structures added as well as the ability to nest multiple commands in parentheses (commands resembled Excel functions). The template commands were simple, powerful, and efficient, but the underlying code was complicated and difficult to maintain, so at one point I started over. My second solution included a hand-coded, recursive descent parser and an object-oriented, JavaScript-like syntax that was easily extended in Perl. My thinking was that many HTML authors were comfortable with JavaScript already. I was rather proud of it when it was finished, but after a few months of using it, I realized I had created an over-engineered, proprietary solution, and I ported the project to Embperl.

In both of my attempts, I realized the solutions were not worth the effort required to maintain them. In the second case, the code was very maintainable, but even minor maintenance did not seem worth the effort given the high-quality, open source alternatives that are already tested, maintained, and available for all to use. More importantly, custom solutions require other developers and HTML authors to invest time learning systems that they would never encounter elsewhere. No one told me I had to choose a standard solution over a proprietary one, but I discovered the advantages on my own. Sometimes ego must yield to practicality.

So consider the options that are already available and avoid the urge to reinvent the wheel. If you need a particular feature that is not available in another package, consider extending an existing open source solution and give your code back if you think it will benefit others. Of course, in the end what you do is up to you, and you may have a good reason for creating your own solution. You could even point out that none of the solutions presented in this chapter would exist if a few people hadn't decided they should create their own respective solutions, maintain and extend them, and make them available to others.

Chapter 7. JavaScript

Contents:

[Background](#)

[Forms](#)

[Data Exchange](#)

[Bookmarklets](#)

Looking at the title of this chapter, you probably said to yourself, "JavaScript? What does that have to do with CGI programming or Perl?" It's true that JavaScript is not Perl, and it cannot be used to write CGI scripts.^[9] However, in order to develop powerful web applications we need to learn much more than CGI itself. Therefore, our discussion has already covered HTTP and HTML forms and will later cover email and SQL. JavaScript is yet another tool that, although not fundamental to creating CGI scripts, can help us create better web applications.

[9]Some web servers do support server-side JavaScript, but not via CGI.

In this chapter, we will focus on three specific applications of JavaScript: validating user input in forms; generating semiautonomous clients; and bookmarklets. As we will soon see, all three of these examples use JavaScript on the client side but still rely on CGI scripts on the server side.

This chapter is not intended to be an introduction to JavaScript. Since many web developers learn HTML and JavaScript before turning to Perl and CGI, we will assume you've had some exposure to JavaScript already. If you haven't, or if you are interested in learning more, you may wish to refer to *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly & Associates, Inc.).

7.1. Background

Before we get started, let's discuss the background of JavaScript. As we said, we'll skip the introduction to JavaScript programming, but we should clear up possible confusions about what we mean when we refer to JavaScript and how JavaScript relates to similar technologies.

7.1.1. History

JavaScript was originally developed for Netscape Navigator 2.0. JavaScript has very little to do with Java despite the similarity in names. The languages were developed independently, and JavaScript was originally called *LiveScript*. However Sun Microsystems (the creator of Java) and Netscape struck a

deal, and LiveScript was renamed to JavaScript shortly before its release. Unfortunately, this single marketing decision has confused many who believe that Java and JavaScript are more similar than they are.

Microsoft later created their own JavaScript implementation for Internet Explorer 3.0, which they called *JScript*. Initially, JScript was mostly compatible with JavaScript, but then Netscape and Microsoft developed their languages in different directions. The dynamic behavior provided in the latest versions of these languages is now very different.

Fortunately, there have been efforts to standardize these languages via ECMAScript and DOM. *ECMA Script* is an ECMA standard that defines the syntax and structure of the language that JScript and JavaScript will become. ECMAScript itself is not specific to the Web and is not directly useful as a language because it doesn't do anything; it only defines a few very basic objects. That's where the *Document Object Model* (DOM) comes in. The DOM is a separate standard being developed by the World Wide Web Consortium to define the objects used with HTML and XML documents without respect to a particular programming language.

The end result of these efforts is that JavaScript and JScript should one day adopt both the ECMAScript standard as well as the DOM standard. They will then share a uniform structure and a common model for interacting with documents. At this point they will both become compatible and we can write client-side scripting code that will work across all browsers that support this standard.

Despite the distinction between JavaScript and JScript, most people use the term JavaScript in reference to any implementation of JavaScript or JScript, regardless of browser; we will also use the term JavaScript in this manner.

7.1.2. Compatibility

The biggest issue with JavaScript is the problem we just discussed: browser compatibility. This is not something we typically need to worry about with CGI scripts, which execute on the web server. JavaScript executes in the user's browser, so in order for our code to execute, the browser needs to support JavaScript, JavaScript needs to be enabled (some users turn it off), and the particular implementation of JavaScript in the browser needs to be compatible with our code.

You must decide for yourself whether the benefits that you gain from using JavaScript outweigh these requirements that it places upon the user. Many sites compromise by using JavaScript to provide enhanced functionality to those users who have it, but without restricting access to those users who do not. Most of our examples in this chapter will follow this model. We will also avoid newer language features and confine ourselves to JavaScript 1.1, which is largely compatible between the different browsers that support JavaScript.

Chapter 8. Security

Contents:

[The Importance of Web Security](#)

[Handling User Input](#)

[Encryption](#)

[Perl's Taint Mode](#)

[Data Storage](#)

[Summary](#)

CGI programming offers you something amazing: as soon as your script is online, it is immediately available to the entire world. Anyone from almost anywhere can run the application you created on your web server. This may make you excited, but it should also make you scared. Not everyone using the Internet has honest intentions. Crackers^[12] may attempt to vandalize your web pages in order to show off to friends. Competitors or investors may try to access internal information about your organization and its products.

[12]A *cracker* is someone who attempts to break into computers, snoop network transmissions, and get into other forms of online mischief. This is quite different from a *hacker*, a clever programmer who can find creative, simple solutions to problems. Many programmers (most of whom consider themselves hackers) draw a sharp distinction between the two terms, even though the mainstream media often does not.

Not all security issues involve malevolent users. The worldwide availability of your CGI script means that someone may run your script under circumstances you never imagined and certainly never tested. Your web script should not wipe out files because someone happened to enter an apostrophe in a form field, but this is possible, and issues like these also represent security concerns.

8.1. The Importance of Web Security

Many CGI developers do not take security as seriously as they should. So before we look at how to make CGI scripts more secure, let's look at why we should worry about security in the first place:

1. *On the Internet, your web site represents your public image.* If your web pages are unavailable or have been vandalized, that affects others' impressions of your organization, even if the focus of your organization has nothing to do with web technology.

2. *You may have valuable information on your web server.* You may have sensitive or valuable information available in a restricted area that you may wish to keep unauthorized people from accessing. For example, you may have content or services available to paying members, which you would not want non-paying customers or non-members to access. Even files that are not part of your web server's document tree and are thus not available online to anyone (e.g., credit card numbers) could be compromised.
3. *Someone who has cracked your web server has easier access to the rest of your network.* If you have no valuable information on your web server, you probably cannot say that about your entire network. If someone breaks into your web server, it becomes much easier for them to break into another system on your network, especially if your web server is inside your organization's firewall (which, for this reason, is generally a bad idea).
4. *You sacrifice potential income when your system is down.* If your organization generates revenue directly from your web site, you certainly lose income when your system is unavailable. However, even if you do not fall into this group, you likely offer marketing literature or contact information online. Potential customers who are unable to access this information may look elsewhere when making their decision.
5. *You waste time and resources fixing problems.* You must perform many tasks when your systems are compromised. First, you must determine the extent of the damage. Then you probably need to restore from backups. You must also determine what went wrong. If a cracker gained access to your web server, then you must determine how the cracker managed this in order to prevent future break-ins. If a CGI script damaged files, then you must locate and fix the bug to prevent future problems.
6. *You expose yourself to liability.* If you develop CGI scripts for other companies, and one of those CGI scripts is responsible for a large security problem, then you may understandably be liable. However, even if it is your company for whom you're developing CGI scripts, you may be liable to other parties. For example, if someone cracks your web server, they could use it as a base to stage attacks on other companies. Likewise, if your company stores information that others consider sensitive (e.g., your customers' credit card numbers), you may be liable to them if that information is leaked.

These are only some of the many reasons why web security is so important. You may be able to come up with other reasons yourself. So now that you recognize the importance of creating secure CGI scripts, you may be wondering what makes a CGI script secure. It can be summed up in one simple maxim: *never trust any data coming from the user*. This sounds quite simple, but in practice it's not. In the remainder of this chapter, we'll explore how to do this.

Chapter 9. Sending Email

Contents:

[Security](#)

[Email Addresses](#)

[Structure of Internet Email](#)

[sendmail](#)

[mailx and mail](#)

[Perl Mailers](#)

[procmail](#)

One of the most common tasks your CGI scripts need to perform is sending email. Email is a popular method for exchanging information between people, whether that information comes from other people or from automated systems. You may need to send email updates or receipts to visitors of your web site. You may need to notify members of your organization about certain events like a purchase, a request for information, or feedback about your web site. Email is also a useful tool to notify you when there are problems with your CGI scripts. When you write subroutines that respond to errors in your CGI scripts, it is a very good idea to include code to notify whomever is responsible for maintaining the site about the error.

There are several ways to send email from an application, including using an external mail client, such as `sendmail` or `mail`, or by directly communicating with the remote mail server via Perl. There are also Perl modules that make sending mail especially easy. We'll explore all these options in this chapter by building a sample application that provides a web front end to an emailer.

9.1. Security

Since the subject of security is still fresh in our minds, however, we should take a moment to review security as it relates to email. Sending email is probably one of the largest causes of security errors in CGI scripts.

9.1.1. Mailers and Shells

Most CGI scripts open a pipe to an external mail client such as `sendmail` and `mail`, and pass the email address through the shell as a parameter. Passing any user data through a shell is a very bad thing as we saw in the previous chapter (if you skipped ahead to this chapter, it would be wise to go back and review [Chapter 8, "Security"](#), before continuing). Unless you like living dangerously, you should *never*

pass an email address to an external application via a shell. It is not possible to verify that email addresses contain only certain safe characters either. Contrary to what you may expect, a proper email address can contain *any* valid ASCII character, including control characters and all those troublesome characters that have special meaning in the shell. We'll review what comprises a valid email address in the next section.

9.1.2. False Identities

You have likely received email claiming to be from someone other than the true sender. It happens all the time with unsolicited bulk email (*spam*). Falsifying the return address in an email message is very simple to do, and can even be quite useful. You probably would rather have email messages sent by your web server appear to come from actual individuals or groups within your company than the user (e.g., *nobody*) that the web user runs as. We'll see how to do this in our examples later in this chapter.

So how does this relate to security? Say, for example, you create a web form that allows users to send feedback to members of your organization. You decide to generalize the CGI script responsible for this so you don't have to update it when internal email addresses change. Instead, you insert the email addresses into hidden fields in the feedback form since they're easier to update there. However, you do take security precautions. Because you recognize that it's possible for a cracker to change hidden fields, you are careful not to pass the email addresses through a shell, and you treat them as tainted data. You handled all the details correctly, but you still have a potential security problem -- it's just at a higher level.

If the user can specify the sender, the recipient, and the body of the message, you are allowing them to send any message to anyone anywhere, and the resulting message will originate from your machine. Anyone can falsify the return address in an email message, but it is very difficult to try to mask the message's routing information. A knowledgeable person can look at the headers in an email message and see where that message truly originated, and all the email messages your web server sends out will clearly originate from the machine hosting it.

Thus this feedback page is a security problem because crackers given this much freedom could send damaging or embarrassing email to whomever they wanted, and all the messages would look like they are from your organization. Although this may not seem as serious as a system breach, it is still something you probably would rather avoid.

9.1.3. Spam

Spam, of course, refers to unsolicited junk email. It's those messages that you get from someone you've never heard of advertising weight loss plans, get-rich schemes, and less-than-reputable web sites. None of us like spam, so be certain your web site doesn't contribute to the problem. Avoid creating CGI scripts that are so flexible that they allow the user to specify the recipient and the content of the

message. The previous example of the feedback page illustrates this. As we saw in the last chapter, it is not difficult to create a web client with LWP and a little bit of Perl code. Likewise, it would not be difficult for a spammer to use LWP to repeatedly call your CGI script in order to send out numerous, annoying messages.

Of course, most spammers don't operate this way. The big ones have dedicated equipment, and for those who don't, it's much more convenient to hijack an SMTP server, which is designed to send mail, than having to pass requests through a CGI script. So even if you do create scripts that are wide open to hijacking, the chances that someone will exploit it are slim ... but what if it does happen? You probably do not want to face the mass of angry recipients who have tracked the routing information back to you. When it comes to security, it's always better to play it safe.

Chapter 10. Data Persistence

Contents:

[Text Files](#)

[DBM Files](#)

[Introduction to SQL](#)

[DBI](#)

Many basic web applications can be created that output only email and web documents. However, if you begin building larger web applications, you will eventually need to store data and retrieve it later. This chapter will discuss various ways to do this with different levels of complexity. Text files are the simplest way to maintain data, but they quickly become inefficient when the data becomes complex or grows too large. A DBM file provides much faster access, even for large amounts of data, and DBM files are very easy to use with Perl. However, this solution is also limited when the data grows too complex. Finally, we will investigate relational databases. A relational database management system (RDBMS) provides high performance even with complex queries. However, an RDBMS is more complicated to set up and use than the other solutions.

Applications evolve and grow larger. What may start out as a short, simple CGI script may gain feature upon feature until it has grown to a large, complex application. Thus, when you design web applications, it is a good idea to develop them so that they are easily expandable.

One solution is to make your solutions modular. You should try to abstract the code that reads and writes data so the rest of the code does not know how the data is stored. By reducing the dependency on the data format to a small chunk of code, it becomes easier to change your data format as you need to grow.

10.1. Text Files

One of Perl's greatest strengths is its ability to parse text, and this makes it especially easy to get a web application online quickly using text files as the means of storing data. Although it does not scale to complex queries, this works well for small amounts of data and is very common for Perl CGI applications. We're not going to discuss how to use text files with Perl, since most Perl programmers are already proficient at that task. We're also not going to look at strategies like creating random access files to improve performance, since that warrants a lengthy discussion, and a DBM file is generally a better substitute. We'll simply look at the issues that are particular to using text files with CGI scripts.

10.1.1. Locking

If you write to any files from a CGI script, then you must use some form of file locking. Web servers support numerous concurrent connections, and if two users try to write to the same file at the same time, the result is generally corrupted or truncated data.

10.1.1.1. flock

If your system supports it, using the *flock* command is the easiest way to do this. How do you know if your system supports `flock`? Try it: `flock` will die with a fatal error if your system does not support it. However, `flock` works reliably only on local files; `flock` does not work across most NFS systems, even if your system otherwise supports it.^[19] `flock` offers two different modes of locking: exclusive and shared. Many processes can read from a file simultaneously without problems, but only one process should write to the file at a time (and no other process should read from the file while it is being written). Thus, you should obtain an exclusive lock on a file when writing to it and a shared lock when reading from it. The shared lock verifies that no one else has an exclusive lock on the file and delays any exclusive locks until the shared locks have been released.

[19]If you need to lock a file across NFS, refer to the `File::LockDir` module in *Perl Cookbook* (O'Reilly & Associates, Inc.).

To use `flock`, call it with a filehandle to an open file and a number indicating the type of lock you want. These numbers are system-dependent, so the easiest way to get them is to use the `Fcntl` module. If you supply the `:flock` argument to `Fcntl`, it will export `LOCK_EX`, `LOCK_SH`, `LOCK_UN`, and `LOCK_NB` for you. You can use them as follows:

```
use Fcntl ":flock";

open FILE, "some_file.txt" or die $!;
flock FILE, LOCK_EX;      # Exclusive lock
flock FILE, LOCK_SH;      # Shared lock
flock FILE, LOCK_UN;      # Unlock
```

Closing a filehandle releases any locks, so there is generally no need to specifically unlock a file. In fact, it can be dangerous to do so if you are locking a filehandle that uses Perl's `tie` mechanism. See file locking in the DBM section of this chapter for more information.

Some systems do not support shared file locks and use exclusive locks for them instead. You can use the script in [Example 10-1](#) to test what `flock` supports on your system.

Example 10-1. flock_test.pl

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

use IO::File;
use Fcntl ":flock";

*FH1 = new_tmpfile IO::File or die "Cannot open temporary file: $!\n";

eval { flock FH1, LOCK_SH };
$@ and die "It does not look like your system supports flock: $@\n";

open FH2, ">> &FH1" or die "Cannot dup filehandle: $!\n";

if ( flock FH2, LOCK_SH | LOCK_NB ) {
    print "Your system supports shared file locks\n";
}
else {
    print "Your system only supports exclusive file locks\n";
}
```

If you need to both read and write to a file, then you have two options: you can open the file exclusively for read/write access, or if you only have to do limited writing and what you're writing does not depend on the contents of the file, you can open and close the file twice: once shared for reading and once exclusive for writing. This is generally less efficient than opening the file once, but if you have lots of processes needing to access the file that are doing lots of reading and little writing, it may be more efficient to reduce the time that one process is tying up the file while holding an exclusive lock on it.

Typically when you use `flock` to lock a file, it halts the execution of your script until it can obtain a lock on your file. The `LOCK_NB` option tells `flock` that you do not want it to block execution, but allow your script to continue if it cannot obtain a lock. Here is one way to time out if you cannot obtain a lock on a file:

```
my $count = 0;
my $delay = 1;
my $max   = 15;

open FILE, ">> $filename" or
    error( $q, "Cannot open file: your data was not saved" );

until ( flock FILE, LOCK_SH | LOCK_NB ) {
    error( $q, "Timed out waiting to write to file: " .
        "your data was not saved" ) if $count >= $max;
    sleep $delay;
    $count += $delay;
}
```

In this example, the code tries to get a lock. If it fails, it waits a second and tries again. After fifteen seconds, it gives up and reports an error.

10.1.1.2. Manual lock files

If your system does not support flock, you will need to manually create your own lock files. As the Perl FAQ points out (see *perlfaq5*), this is not as simple as you might think. The problem is that you must check for the existence of a file and create the file as one operation. If you first check whether a lock file exists, and then try to create one if it does not, another process may have created its own lock file after you checked, and you just overwrote it.

To create your own lock file, use the following command:

```
use Fcntl;
.
.
.
sysopen LOCK_FILE, "$filename.lock", O_WRONLY | O_EXCL | O_CREAT, 0644
    or error( $q, "Unable to lock file: your data was not saved" );
```

The O_EXCL function provided by Fcntl tells the system to open the file only if it does not already exist. Note that this will not reliably work on an NFS filesystem.

10.1.2. Write Permissions

In order to create or update a text file, you must have the appropriate permissions. This may sound basic, but it is a common source of errors in CGI scripts, especially on Unix filesystems. Let's review how Unix file permissions work.

Files have both an owner and a group. By default, these match the user and group of the user or process who creates the file. There are three different levels of permissions for a file: the owner's permissions, the group's permissions, and everyone else's permissions. Each of these may have read access, write access, and/or execute access for a file.

Your CGI scripts can only modify a file if *nobody* (or the user your web server runs as) has write access to the file. This occurs if the file is writable by everyone, if it is writable by members of the file's group and *nobody* is a member of that group, or if *nobody* owns the file and the file is writable by its owner.

In order to create or remove a file, *nobody* must have write permission to the directory containing the file. The same rules about owner, group, and other users apply to directories as they do for files. In addition, the execute bit must be set for the directory. For directories, the execute bit determines scan access, which is the ability to change to the directory.

Even though your CGI script may not modify a file, it may be able to replace it. If *nobody* has permission to write to a directory, then it can remove files in the directory in addition to creating new

files, even with the same name. Write permissions on the file do not typically affect the ability to remove or replace the file as a whole.

10.1.3. Temporary Files

Your CGI scripts may need to create temporary files for a number of reasons. You can reduce memory consumption by creating files to hold data as you process it; you gain efficiency by sacrificing performance. You may also use external commands that perform their actions on text files.

10.1.3.1. Anonymous temporary files

Typically, temporary files are anonymous; they are created by opening a handle to a new file and then immediately deleting the file. Your CGI script will continue to have a filehandle to access the file, but the data cannot be accessed by other processes, and the data will be reclaimed by the filesystem once your CGI script closes the filehandle. (Not all systems support this feature.)

As for most common tasks, there is a Perl module that makes managing temporary files much simpler. `IO::File` will create anonymous temporary files for you with the `new_tmpfile` class method; it takes no arguments. You can use it like this: [\[20\]](#)

[20]Actually, if the filesystem does not support anonymous temporary files, then `IO::File` will not create it anonymously, but it's still anonymous to you since you cannot get at the name of the file. `IO::File` will take care of managing and deleting the file for you when its filehandle goes out of scope or your script completes.

```
use IO::File;
.
.
.
my $tmp_fh = new_tmpfile IO::File;
```

You can then read and write to `$tmp_fh` just as you would any other filehandle:

```
print $tmp_fh "</html>\n";

seek $tmp_fh, 0, 0;
while (<$tmp_fh>) {
    print;
}
```

10.1.3.2. Named temporary files

Another option is to create a file and delete it when you are finished with it. One advantage is that you have a filename that can be passed to other processes and functions. Also, using the `IO::File` module is considerably slower than managing the file yourself. However, using named temporary files has two drawbacks. First, greater care must be taken choosing a unique filename so that two scripts will not attempt to use the same temporary file at the same time. Second, the CGI script must delete the file when it is finished, even if it encounters an error and exits prematurely.

The Perl FAQ suggests using the `POSIX` module to generate a temporary filename and an `END` block to ensure it will be cleaned up:

```
use Fcntl;
use POSIX qw(tmpnam);
.
.
.
my $tmp_filename;

# try new temporary filenames until we get one that doesn't already
# exist; the check should be unnecessary, but you can't be too careful
do { $tmp_filename = tmpnam( ) }
    until sysopen( FH, $name, O_RDWR|O_CREAT|O_EXCL );

# install atexit-style handler so that when we exit or die,
# we automatically delete this temporary file
END { unlink( $tmp_filename ) or die "Couldn't unlink $name: $!" }
```

If your system doesn't support `POSIX`, then you will have to create the file in a system-dependent fashion instead.

10.1.4. Delimiters

If you need to include multiple fields of data in each line of your text file, you will likely use delimiters to separate them. Another option is to create fixed-length records, but we won't get into these files here. Common characters to use for delimiting files are commas, tabs, and pipes (`|`).

Commas are primarily used in CSV files, which we will discuss presently. CSV files can be difficult to parse accurately because they can include non-delimiting commas as part of a value. When working with CSV files, you may want to consider the `DBD::CSV` module; this gives you a number of additional benefits, which we will discuss shortly.

Tabs are not generally included within data, so they make convenient delimiters. Even so, you should always check your data and encode or remove any tabs or end-of-line characters before writing to your file. This ensures that your data does not become corrupted if someone happens to pass a newline character in the middle of a field. Remember, even if you are reading data from an HTML form element that would not normally accept a newline character as part of it, you should never trust the user

or that user's browser.

Here is an example of functions you can use to encode and decode data:

```

sub encode_data {
    my @fields = map {
        s/\\/\\\\/g;
        s/\t/\\t/g;
        s/\n/\\n/g;
        s/\r/\\r/g;
        $_;
    } @_;

    my $line = join "\t", @fields;
    return "$line\n";
}

sub decode_data {
    my $line = shift;

    chomp $line;
    my @fields = split /\t/, $line;

    return map {
        s/\\((.))/ $1 eq 't' and "\t" or
        $1 eq 'n' and "\n" or
        $1 eq 'r' and "\r" or
        "$1"/eg;
    } @fields;
}

```

These functions encode tabs and end-of-line characters with the common escape characters that Perl and other languages use (`\t`, `\r`, and `\n`). Because it is introducing additional backslashes as an escape character, it must also escape the backslash character.

The `encode_data` sub takes a list of fields and returns a single encoded scalar that can be written to the file; `decode_data` takes a line read from the file and returns a list of decoded fields. You can use them as shown in [Example 10-2](#).

Example 10-2. `sign_petition.cgi`

```

#!/usr/bin/perl -wT

use strict;
use Fcntl ":flock";
use CGI;
use CGIBook::Error;

```

```

my $DATA_FILE = "/usr/local/apache/data/tab_delimited_records.txt";

my $q          = new CGI;
my $name       = $q->param( "name" );
my $comment    = substr( $q->param( "comment" ), 0, 80 );

unless ( $name ) {
    error( $q, "Please enter your name." );
}

open DATA_FILE, ">> $DATA_FILE" or die "Cannot append to $DATA_FILE: $!";
flock DATA_FILE, LOCK_EX;
seek DATA_FILE, 0, 2;

print DATA_FILE encode_data( $name, $comment );
close DATA_FILE;

print $q->header( "text/html" ),
      $q->start_html( "Our Petition" ),
      $q->h2( "Thank You!" ),
      $q->p( "Thank you for signing our petition. ",
           "Your name has been added below:" ),
      $q->hr,
      $q->start_table,
      $q->tr( $q->th( "Name", "Comment" ) );

open DATA_FILE, $DATA_FILE or die "Cannot read $DATA_FILE: $!";
flock DATA_FILE, LOCK_SH;

while (<DATA_FILE>) {
    my @data = decode_data( $_ );
    print $q->tr( $q->td( @data ) );
}
close DATA_FILE;

print $q->end_table,
      $q->end_html;

sub encode_data {
    my @fields = map {
        s/\\/\\\\/g;
        s/\t/\\t/g;
        s/\n/\\n/g;
        s/\r/\\r/g;
        $_;
    } @_;

    my $line = join "\t", @fields;
    return $line . "\n";
}

```

```

sub decode_data {
    my $line = shift;

    chomp $line;
    my @fields = split /\t/, $line;

    return map {
        s/\\(.)/$1 eq 't' and "\t" or
        $1 eq 'n' and "\n" or
        $1 eq 'r' and "\r" or
        "$1"/eg;
        $_;
    } @fields;
}

```

Note that organizing your code this way gives you another benefit. If you later decide you want to change the format of your data, you do not need to change your entire CGI script, just the `encode_data` and `decode_data` functions.

10.1.5. DBD::CSV

As we mentioned at the beginning of this chapter, it's great to modularize your code so that changing the data format affects only a small chunk of your application. However, it's even better if you don't have to change that chunk either. If you are creating a simple application that you expect to grow, you may want to consider developing your application using CSV files. *CSV (comma separated values)* files are text files formatted such that each line is a record, and fields are delimited by commas. The advantage to using CSV files is that you can use Perl's DBI and DBD::CSV modules, which allow you to access the data via basic SQL queries just as you would for an RDBMS. Another benefit of CSV format is that it is quite common, so you can easily import and export it from other applications, including spreadsheets like Microsoft Excel.

There are drawbacks to developing with CSV files. DBI adds a layer of complexity to your application that you would not otherwise need if you accessed the data directly. DBI and DBD::CSV also allow you to create only simple SQL queries, and it is certainly not as fast as a true relational database system, especially for large amounts of data.

However, if you need to get a project going, knowing that you will move to an RDBMS, and if DBD::CSV meets your immediate requirements, then this strategy is certainly a good choice. We will look at an example that uses DBD::CSV later in this chapter.

Chapter 11. Maintaining State

Contents:

[Query Strings and Extra Path Information](#)

[Hidden Fields](#)

[Client-Side Cookies](#)

HTTP is a stateless protocol. As we discussed in [Chapter 2, "The Hypertext Transport Protocol"](#), the HTTP protocol defines how web clients and servers communicate with each other to provide documents and resources to the user. Unfortunately, as we noted in our discussion of HTTP (see [Section 2.5.1, "Identifying Clients"](#)), HTTP does not provide a direct way of identifying clients in order to keep track of them across multiple page requests. There are ways to track users through indirect methods, however, and we'll explore these methods in this chapter. Web developers refer to the practice of tracking users as *maintaining state*. The series of interactions that a particular user has with our site is a *session*. The information that we collect for a user is *session information*.

Why would we want to maintain state? If you value privacy, the idea of tracking users may raise concerns. It is true that tracking users can be used for questionable purposes. However, there are legitimate instances when you must track users. Take an online store: in order to allow a customer to browse products, add some to a shopping cart, and then check out by purchasing the selected items, the server must maintain a separate shopping cart for each user. In this case, collecting selected items in a user's session information is not only acceptable, but expected.

Before we discuss methods for maintaining state, let's briefly review what we learned earlier about the HTTP transaction model. This will provide a context to understand the options we present later. Each and every HTTP transaction follows the same general format: a request from a client followed by a response from the server. Each of these is divided into a request/response line, header lines, and possibly some message content. For example, if you open your favorite browser and type in the URL:

```
http://www.oreilly.com/catalog/cgi2/index.html
```

Your browser then connects to *www.oreilly.com* on port 80 (the default port for HTTP) and issues a request for */catalog/cgi2/index.html*. On the server side, because the web server is bound to port 80, it answers any requests that are issued through that port. Here is how the request would look from a browser supporting HTTP 1.0:

```
GET /index.html HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
User-Agent: Mozilla/4.5 (Macintosh; I; PPC)
```

The browser uses the GET request method to ask for the document, specifies the HTTP protocol to use, and supplies a number of headers to pass information about itself and the format of the content it will accept. Because the request is sent via GET and not POST, the browser is not passing any content to the server.

Here is how the server would respond to the request:

```
HTTP/1.0 200 OK
Date: Sat, 18 Mar 2000 20:35:35 GMT
Server: Apache/1.3.9 (Unix)
Last-Modified: Wed, 20 May 1998 14:59:42 GMT
Content-Length: 141
Content-Type: text/html
```

(content)

...

In Version 1.0 of HTTP, the server returns the requested document and then closes the connection. Yes, that's right: the server doesn't keep the connection open between itself and the browser. So, if you were to click on a link on the returned page, the browser then issues another request to the server, and so on. As a result, the server has no way of knowing that it's you that is requesting the successive document. This is what we mean by *stateless*, or nonpersistent; the server doesn't maintain or store any request-related information from one transaction to the next. You do know the network address of the client who is connecting to you, but as you'll recall from our earlier discussion of proxies (see [Section 2.5, "Proxies"](#)), multiple users may be making connections via the same proxy.

You may be waiting to hear what's changed in Version 1.1 of HTTP. In fact, a connection may remain open across multiple requests, although the request and response cycle is the same as above. However, you cannot rely on the network connection remaining open since the connection can be closed or lost for any number of reasons, and in any event CGI has not been modified to allow you access any information that would associate requests made across the same connection. So in HTTP 1.1 as in HTTP 1.0, the job of maintaining state falls to us.

Consider our shopping cart example: it should allow consumers to navigate through many pages and selectively place items in their carts. A consumer typically places an item in a cart by selecting a product, entering the desired quantity, and submitting the form. This action sends the data to the web server, which, in turn, invokes the requested CGI application. To the server, it's simply another request. So, it's up to the application to not only keep track of the data between multiple invocations, but also to identify the data as belonging to a particular consumer.

In order to maintain state, we must get the client to pass us some unique identifier with each request. As you can see from the HTTP request example earlier, there are only three different ways the client can pass information to us: via the request line, via a header line, or via the content (in the case of a POST request). Thus, in order to maintain state, we can have the client pass a unique identifier to us via

any of these methods. In fact, the techniques we'll explore will cover all three of these ways:

Query strings and extra path information

It's possible to embed an identifier in the query string or as extra path information within a document's URL. As users traverse through a site, a CGI application generates documents on the fly, passing the identifier from document to document. This allows us to keep track of all the documents requested by each user, and in the order in which they were requested. The browser sends this information to us via the request line.

Hidden fields

Hidden form fields allow us to embed "invisible" name-value information within forms that the user cannot see without viewing the source of the HTML page. Like typical form fields and values, this information is sent to the CGI application when the user presses the submit button. We generally use this technique to maintain the user's selections and preferences when multiple forms are involved. We'll also look at how CGI.pm can do much of this work for us. The browser sends this information to us via the request line or via the message content depending on whether the request was GET or POST, respectively.

Client-side cookies

All modern browsers support client-side cookies, which allow us to store information on the client machine and have it pass it back to us with each request. We can use this to store semi-permanent data on the client-side, which will be available to us whenever the user requests future resources from the server. Cookies are sent back to us by the client in the *Cookie* HTTP header line.

The advantages and disadvantages of each technique are summarized in [Table 11-1](#). We will review each technique separately, so if some of the points in the table are unclear you may want to refer back to this table after reading the sections below. In general, though, you should note that client-side cookies are the most powerful option for maintaining state, but they require something from the client. The other options work regardless of the client, but both have limits in the number of the pages that we can track the user across.

Table 11-1. Summary of the Techniques for Maintaining State

Technique	Scope	Reliability and Performance	Client Requirements
Query strings and extra path information	Can be configured to apply to a particular group of pages or an entire web site, but state information is lost if the user leaves the web site and later returns	Difficult to reliably parse all links in a document; significant performance cost to pass static content through CGI scripts	Does not require any special behavior from the client
Hidden fields	Only works across a series of form submissions	Easy to implement; does not affect performance	Does not require any special behavior from the client
Cookies	Works everywhere, even if the user visits another site and later returns	Easy to implement; does not affect performance	Requires that the client supports (and accepts) cookies

11.1. Query Strings and Extra Path Information

We've passed query information to CGI applications many times throughout this book. In this section, we'll use queries in a slightly less obvious manner, namely to track a user's browsing trail while traversing from one document to the next on the server.

In order to do this, we'll have a CGI script handle every request for a static HTML page. The CGI script will check whether the request URL contains an identifier matching our format. If it doesn't, the script assumes that this is a new user and generates a new identifier. The script then parses the requested HTML document by looking for links to other URLs within our web site and appending a unique identifier to each URL. Thus, the identifier will be passed on with future requests and propagated from document to document. Of course, if we want to track users across CGI applications then we'll also need to parse the output of these CGI scripts. The simplest way to accomplish both goals is to create a general module that handles reading the identifier and parsing the output. This way, we need to write our code only once and can have the script for our HTML pages as well as allow all our other CGI scripts share it.

As you may have guessed, this is not a very efficient process, since a request for each and every HTML document triggers a CGI application to be executed. Tools such as *mod_perl* and FastCGI, discussed in [Chapter 17, "Efficiency and Optimization"](#), help because both of these tools effectively embed the Perl interpreter into the web server.

Another strategy to help improve performance is to perform some processing in advance. If you are

willing to preprocess your documents, you can reduce the amount of work that happens when the customer accesses the document. The majority of the work involved in parsing a document and replacing links is identifying the links. HTML::Parser is a good module, but the work it does is rather complex. If you parse the links and add a special keyword instead of one for a particular user, then later you can look for this keyword and not have to worry about recognizing links. For example, you could parse URLs and add #USERID# as the identifier for each document. The resulting code becomes much simpler. You can effectively handle documents this way:

```
sub parse {
    my( $filename, $id ) = @_;
    local *FH;
    open FH, $filename or die "Cannot open file: $!";

    while (<FH>) {
        s/#USERID#/$id/g;
        print;
    }
}
```

However, when a user traverses through a set of static HTML documents, CGI applications are typically not involved. If that's the case, how do we pass session information from one HTML document to the next, and be able to keep track of it on the server?

The answer to our problem is to configure the server such that when the user requests an HTML document, the server executes a CGI application. The application would then be responsible for transparently embedding special identifying information (such as a query string) into all the hyperlinks within the requested HTML document and returning the newly created content to the browser.

Let's look at how we're actually going to implement the application. It's only a two-step process. To reiterate, the problem we're trying to solve is to determine what documents a particular user requests and how much time he or she spends viewing them. First, we need to identify the set of documents for which we want to obtain the users' browsing history. Once we do that, we simply move these documents to a specific directory under the web server's document root directory.

Next, we need to configure the web server to execute a CGI application each and every time a user requests a document from this directory. We'll use the Apache web server for this example, but the configuration details are very similar for other web servers, as well.

We simply need to insert the following directives into Apache's access configuration file, *access.conf*:

```
<Directory /usr/local/apache/htdocs/store>
    AddType text/html      .html
    AddType Tracker       .html
    Action Tracker        /cgi/track.cgi
</Directory>
```

When a user requests a document from the */usr/local/apache/htdocs/store* directory, Apache executes

the *query_track* application, passing to it the relative URL of the requested document as extra path information. Here's an example. When the user requests a document from the directory for the first time:

```
http://localhost/store/index.html
```

the web server will execute *query_track*, like so:

```
http://localhost/cgi/track.cgi/store/index.html
```

The application uses the `PATH_TRANSLATED` environment variable to get the full path of *index.html*. Then, it opens the file, creates a new identifier for the user, embeds it into each relative URL within the document, and returns the modified HTML stream to the browser. In addition, we log the transaction to a special log file, which you can use to analyze users' browsing habits at a later time.

If you're curious as to what a modified URL looks like, here's an example:

```
http://localhost/store/.CC7e2BMb_H6UdK9KfPtR1g/faq.html
```

The identifier is a modified Base64MD5 message digest, computed using various pieces of information from the request. The code to generate it looks like this:

```
use Digest::MD5;

my $md5 = new Digest::MD5;
my $remote = $ENV{REMOTE_ADDR} . $ENV{REMOTE_PORT};
my $id = $md5->md5_base64( time, $$, $remote );
$id =~ tr|+/=|_|; # Make non-word chars URL-friendly
```

This does a good job of generating a unique key for each request. However, it is not intended to create keys that cannot be cracked. If you are generating session identifiers that provide access to sensitive data, then you should use a more sophisticated method to generate an identifier.

If you use Apache, you do not have to generate a unique identifier yourself if you build Apache with the *mod_unique_id* module. It creates a unique identifier for each request, which is available to your CGI script as `$ENV{UNIQUE_ID}`. *mod_unique_id* is included in the Apache distribution but not compiled by default.

Let's look at how we could construct code to parse HTML documents and insert identifiers. [Example 11-1](#) shows a Perl module that we use to parse the request URL and HTML output.

Example 11-1. CGIBook::UserTracker.pm

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

```

# /-----
# UserTracker Module
#
# Inherits from HTML::Parser
#
#

package CGIBook::UserTracker;

push @ISA, "HTML::Parser";

use strict;
use URI;
use HTML::Parser;

1;

# /-----
# Public methods
#

sub new {
    my( $class, $path ) = @_;
    my $id;

    if ( $ENV{PATH_INFO} and
          $ENV{PATH_INFO} =~ s|^/\.( [a-z0-9_.-]* )/||i ) {
        $id = $1;
    }
    else {
        $id ||= unique_id( );
    }

    my $self = $class->SUPER::new( );
    $self->{user_id} = $id;
    $self->{base_path} = defined( $path ) ? $path : "";

    return $self;
}

sub base_path {
    my( $self, $path ) = @_;
    $self->{base_path} = $path if defined $path;
    return $self->{base_path};
}

sub user_id {
    my $self = shift;
    return $self->{user_id};
}

```

```

# /-----
# Internal (private) subs
#

sub unique_id {
    # Use Apache's mod_unique_id if available
    return $ENV{UNIQUE_ID} if exists $ENV{UNIQUE_ID};

    require Digest::MD5;

    my $md5 = new Digest::MD5;
    my $remote = $ENV{REMOTE_ADDR} . $ENV{REMOTE_PORT};

    # Note this is intended to be unique, and not unguessable
    # It should not be used for generating keys to sensitive data
    my $id = $md5->md5_base64( time, $$, $remote );
    $id =~ tr|+|=|_|; # Make non-word chars URL-friendly
    return $id;
}

sub encode {
    my( $self, $url ) = @_;
    my $uri = new URI( $url, "http" );
    my $id = $self->user_id( );
    my $base = $self->base_path;

    my $path = $uri->path;
    $path =~ s|^$base|$base/.$id| or
        die "Invalid base path configured\n";
    $uri->path( $path );

    return $uri->as_string;
}

# /-----
# Subs to implement HTML::Parser callbacks
#

sub start {
    my( $self, $tag, $attr, $attrseq, $origtext ) = @_;
    my $new_text = $origtext;

    my %relevant_pairs = (
        frameset => "src",
        a         => "href",
        area      => "href",
        form      => "action",
    );
    # Uncomment these lines if you want to track images too
    #     img      => "src",
    #     body     => "background",
}

```

```

);

while ( my( $rel_tag, $rel_attr ) = each %relevant_pairs ) {
    if ( $tag eq $rel_tag and $attr->{$rel_attr} ) {
        $attr->{$rel_attr} = $self->encode( $attr->{$rel_attr} );
        my @attrs = map { "$_=\"$attr->{$_}\" } @$attrseq;
        $new_text = "<$tag @attrs>";
    }
}

# Meta refresh tags have a different format, handled separately
if ( $tag eq "meta" and $attr->{"http-equiv"} eq "refresh" ) {
    my( $delay, $url ) = split ";URL=", $attr->{content}, 2;
    $attr->{content} = "$delay;URL=" . $self->encode( $url );
    my @attrs = map { "$_=\"$attr->{$_}\" } @$attrseq;
    $new_text = "<$tag @attrs>";
}

print $new_text;
}

sub declaration {
    my( $self, $decl ) = @_;
    print $decl;
}

sub text {
    my( $self, $text ) = @_;
    print $text;
}

sub end {
    my( $self, $tag ) = @_;
    print "</$tag>";
}

sub comment {
    my( $self, $comment ) = @_;
    print "<!--$comment-->";
}

```

[Example 11-2](#) shows the CGI application that we use to process static HTML pages.

Example 11-2. query_track.cgi

```

#!/usr/bin/perl -wT

use strict;
use CGIBook::UserTracker;

```

```
local *FILE;
my $track = new CGIBook::UserTracker;
$track->base_path( "/store" );

my $requested_doc = $ENV{PATH_TRANSLATED};
unless ( -e $requested_doc ) {
    print "Location: /errors/not_found.html\n\n";
}

open FILE, $requested_doc or die "Failed to open $requested_doc: $!";

my $doc = do {
    local $/ = undef;
    <FILE>;
};

close FILE;

# This assumes we're only tracking HTML files:
print "Content-type: text/html\n\n";
$track->parse( $doc );
```

Once we have inserted the identifier into all the URLs, we simply send the modified content to the standard output stream, along with the content header.

Now that we've looked at how to maintain state between views of multiple HTML documents, our next step is to discuss persistence when using multiple forms. An online store, for example, is typically broken into multiple pages. We need to be able to identify users as they fill out each page. We'll look at techniques for solving such problems in the next section.

Chapter 12. Searching the Web Server

Contents:

[Searching One by One](#)

[Searching One by One, Take Two](#)

[Inverted Index Search](#)

Allowing users to search for specific information on your web site is a very important and useful feature, and one that can save them from potential frustration trying to locate particular documents. The concept behind creating a search application is rather trivial: accept a query from the user, check it against a set of documents, and return those that match the specified query. Unfortunately, there are several issues that complicate the matter, the most significant of which is dealing with large document repositories. In such cases, it's not practical to search through each and every document in a linear fashion, much like searching for a needle in a haystack. The solution is to reduce the amount of data we need to search by doing some of the work in advance.

This chapter will teach you how to implement different types of search engines, ranging from the trivial, which search documents on the fly, to the most complex, which are capable of intelligent searches.

12.1. Searching One by One

The very first example that we will look at is rather trivial in that it does not perform the actual search, but passes the query to the `fgrep` command and processes the results.

Before we go any further, here's the HTML form that we will use to get the information from the user:

```
<HTML>
<HEAD>
  <TITLE>Simple 'Mindless' Search</TITLE>
</HEAD>
<BODY>
<H1>Are you ready to search?</H1>
<P>
<FORM ACTION="/cgi/grep_search1.cgi" METHOD="GET">
<INPUT TYPE="text" NAME="query" SIZE="20">
<INPUT TYPE="submit" VALUE="GO!">
</FORM>
</BODY>
</HTML>
```

As we mentioned above, the program is quite simple. It creates a pipe to the `fgrep` command and passes it the query, as well as options to perform case-insensitive searches and to return the matching filenames without any text. The program beautifies the output from `fgrep` by converting it to an HTML document and returns it to the browser.

`fgrep` returns the list of matched files in the following format:

```
/usr/local/apache/htdocs/how_to_script.html
/usr/local/apache/htdocs/i_need_perl.html
.
.
```

The program converts this to the following HTML list:

```
<LI><A HREF="/how_to_script.html" >how_to_script.html</A></LI>
<LI><A HREF="/i_need_perl.html">i_need_perl.html</A></LI>
.
.
```

Let's look at the program now, as shown in [Example 12-1](#).

Example 12-1. `grep_search1.cgi`

```
#!/usr/bin/perl -wT
# WARNING: This code has significant limitations; see description

use strict;
use CGI;
use CGIBook::Error;

# Make the environment safe to call fgrep
BEGIN {
    $ENV{PATH} = "/bin:/usr/bin";
    delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
}

my $FGREP          = "/usr/local/bin/fgrep";
my $DOCUMENT_ROOT = $ENV{DOCUMENT_ROOT};
my $VIRTUAL_PATH   = "";

my $q              = new CGI;
my $query          = $q->param( "query" );

$query =~ s/[\^\w ]//g;
$query =~ /([\w ]+)/;
$query = $1;

if ( defined $query ) {
```

```

        error( $q, "Please specify a valid query!" );
    }

my $results = search( $q, $query );

print $q->header( "text/html" ),
      $q->start_html( "Simple Search with fgrep" ),
      $q->h1( "Search for: $query" ),
      $q->ul( $results || "No matches found" ),
      $q->end_html;

sub search {
    my( $q, $query ) = @_;
    local *PIPE;
    my $matches = "";

    open PIPE, "$FGREP -il '$query' $DOCUMENT_ROOT/* |"
        or die "Cannot open fgrep: $!";

    while ( <PIPE> ) {
        chomp;
        s|.|_|;
        $matches .= $q->li(
            $q->a( { href => "$VIRTUAL_PATH/$_" }, $_ )
        );
    }
    close PIPE;
    return $matches;
}

```

We initialize three globals -- `$FGREP`, `$DOCUMENT_ROOT`, and `$VIRTUAL_PATH` -- which store the path to the `fgrep` binary, the search directory, and the virtual path to that directory, respectively. If you do not want the program to search the web server's top-level document directory, you should change `$DOCUMENT_ROOT` to reflect the full path of the directory where you want to enable searches. If you do make such a change, you will also need to modify `$VIRTUAL_PATH` to reflect the URL path to the directory.

Because Perl will pass our `fgrep` command through a shell, we need to make sure that the query we send it is not going to cause any security problems. Let's decide to allow only "words" (represented in Perl as "a-z", "A-Z", "0-9", and "_") and spaces in the search. We proceed to strip out all characters other than words and spaces and pass the result through an additional regular expression to untaint it. We need to do this extra step because, although we know the substitution really did make the data safe, a substitution is not sufficient to untaint the data for Perl. We could have skipped the substitution and just performed the regular expression match, but it means that if someone entered an invalid character, only that part of their query before the illegal character would be included in the search. By doing the substitution first, we can strip out illegal characters and perform a search on everything else.

After all this, if the query is not provided or is empty, we call our familiar *error* subroutine to notify the user of the error. We test whether it is defined first to avoid a warning for using an undefined variable.

We open a PIPE to the *fgrep* command for reading, which is the purpose of the trailing "|". Notice how the syntax is not much different from opening a file. If the pipe succeeds, we can go ahead and read the results from the pipe.

The `-i` options force *fgrep* to perform case-insensitive searches and return the filenames (and not the matched lines). We make sure to quote the string in case the user is searching for a multiple word query.

Finally, the last argument to *fgrep* is a list of all the files that it should search. The shell expands (globs) the wildcard character into a list of all the files in the specified directory. This can cause problems if the directory contains a large number of files, as some shells have internal glob limits. We will fix this problem in the next section.

The `while` loop iterates through the results, setting `$_` to the current record each time through the loop. We strip the end-of-line character(s) and the directory information so we are left with just the filename. Then we create a list item containing a hypertext link to the item.

Finally, we print out our results.

How would you rate this application? It's a simple search engine and it works well on a small collection of files, but it suffers from a few problems:

- It calls an external application (*fgrep*) to handle the search, which makes it nonportable; Windows 95 for instance does not have a *fgrep* application.
- Alphanumeric "symbols" are stripped from the search query, due to security concerns.
- It could very well run into an internal glob limit when used with certain shells; some shells have limits as low as 256 files.
- It does not search multiple directories.
- It does not return content, but simply filename(s), although we could have added this functionality by not specifying the `-l` option.

So, let's try again and create a better search engine.

Chapter 13. Creating Graphics on the Fly

Contents:

[File Formats](#)

[Outputting Image Data](#)

[Generating PNGs with GD](#)

[Additional GD Modules](#)

[PerlMagick](#)

Throughout this book we have seen many examples of CGI scripts generating dynamic output. However, in almost all cases, the output has been HTML. Certainly this is the most common format your scripts will generate. However, CGI scripts can actually generate any type of format, and in this chapter we will look at how we can dynamically generate images.

Generating images dynamically has many uses. One of the most common is to generate graphs. If you have a data source that is continually changing, such as the results of an online survey, a CGI script can generate a graph that presents a visual snapshot of this data.

There are also times when generating images dynamically makes less sense. It is much less efficient to generate an image dynamically than for your web server to serve the image from an image file. Thus, just because some of these tools allow you to generate really cool graphics dynamically doesn't mean you must use them only in a dynamic context. Unless the images you generate are based upon data that changes, save the image to a static file and serve that instead.

This chapter presents a broad overview of the different tools available for generating dynamic images online, and includes references with each for finding more information. The goal of this chapter is to explain techniques for generating images dynamically and familiarize you with the most popular tools available to you. A full description of many of these tools along with others is available in a book of its own, *Programming Web Graphics with Perl and GNU Software* by Shawn Wallace (O'Reilly & Associates, Inc.).

13.1. File Formats

Let's first review the image formats that are used online today. The most common image formats, of course, are GIF and JPEG, which every graphical web browser supports. Other file formats that we will discuss in this chapter include PNG and PDF.

13.1.1. GIF

The *Graphics Interchange Format (GIF)* was created by CompuServe and released as an open standard in 1987. It quickly became a very popular image format and, along with JPEG, became a standard format for images on the Web. GIF files are typically quite small, especially for images with few colors, which makes them well suited for transferring online.

GIF only supports up to 256 colors, but it works well for text and images, such as icons, which do not have many colors but have sharp details. The compression algorithm that GIF uses, LZW, is lossless, which means that no image quality is lost during compression or decompression, allowing GIF files to accurately capture details.

The GIF file format has been extended to support basic animation, which can loop. The moving banner ads that you see online are typically animated GIF files. GIF files can also have a transparent background by specifying a single color in the image that should be displayed as transparent.

13.1.1.1. The LZW patent

Unfortunately, CompuServe and others apparently failed to notice that LZW, the compression algorithm used by GIF, was actually patented by Unisys in 1983. Unisys reportedly discovered that GIF uses LZW in the early 1990s and in 1994 CompuServe and Unisys reached a settlement and announced that developers who write software supporting GIF must pay a licensing fee to Unisys. Note that this does not include web authors who use GIF files or users who browse them on the Web.

This turn of events created quite a stir among developers, especially open source developers. As a result, CompuServe and others developed the PNG format as a LZW-free successor to GIF; we'll discuss PNG below. However, GIF remains a very popular file format, and PNG is not supported by all browsers.

As a result of the LZW licensing issue, the tools we discuss in this chapter provide very limited support for GIF files, as we will see.

13.1.2. PNG

The *Portable Network Graphic (PNG)* format was created as a successor to the GIF format. It adds the following features over GIF:

- PNG uses an efficient compression algorithm that is *not* LZW. In most cases, it achieves slightly better compression than the LZW algorithm.
- PNG supports images in any of three modes: images with a limited palette of 256 or fewer colors,

16-bit grayscale images, and 48-bit true color images.

- PNG supports alpha channels, which allows varying degrees of transparency.
- PNG graphics have a better interlacing algorithm that allows users to make out the contents of the image as it downloads much faster than with a GIF.

For additional differences, as well as an demonstration of the difference between the PNG and GIF interlacing, visit <http://www.cdrom.com/pub/png/pngintro.html>.

Unfortunately, many browsers do not support PNG images. Of those that do, many do not support all of its features, such as multiple levels of transparency. Support for PNG should continue to increase, however, and older browsers that do not support it will eventually be upgraded.

PNG does not support animations.

13.1.3. JPEG

The *Joint Photographic Experts Group (JPEG)* is a standards body created to generate an image format for encoding continuous tone images. Their JPEG standard actually discusses a very general method for still image compression and not a file format. The file format that people typically think of as a JPEG is actually *JFIF*, the *JPEG File Interchange Format*. We will stick with the more familiar term and also refer to a JFIF file as a JPEG file.

JPEG files are ideal for encoding photographs. JPEG supports full, 24-bit color but it uses lossy compression algorithm, which means that each time the file is compressed, detail is lost. Because the encoding for JPEG files is done in blocks, it is most noticeable in images that have very sharp details, such as text and line art. These details may appear blurred in a JPEG file.

JPEG files have no support for animation or transparency.

13.1.4. PDF

Adobe's *Portable Document Format (PDF)* is more than just an image format. It is actually a language derived from PostScript that can include text, basic shapes, line art, and images, as well as numerous other elements. Unlike images, which are typically displayed within an HTML file, PDF files are typically standalone documents, and users use a browser plug-in or external application such as Adobe Acrobat to view them.

Chapter 14. Middleware and XML

Contents:

[Communicating with Other Servers](#)

[An Introduction to XML](#)

[Document Type Definition](#)

[Writing an XML Parser](#)

[CGI Gateway to XML Middleware](#)

CGI programming has been used to make individual web applications from simple guestbooks to complex programs such as a calendar capable of managing the schedules of large groups. Traditionally, these programs have been limited to displaying data and receiving input directly from users.

However, as with all popular technologies, CGI is being pushed beyond these traditional uses. Going beyond CGI applications that interact with users, the focus of this chapter is on how CGI can be a powerful means of communicating with other programs.

We have seen how CGI programs can act as a gateway to a variety of resources such as databases, email, and a host of other protocols and programs. However, a CGI program can also perform some sophisticated processing on the data it gets so that it effectively becomes a data resource itself. This is the definition of CGI *middleware*. In this context, the CGI application sits between the program it is serving data to and the resources that it is interacting with.

The variety of search engines that exist provides a good example of why CGI middleware can be useful. In the early history of the Web, there were only a few search engines to choose from. Now, there are many. The results these engines produce are usually not identical. Finding out about a rare topic is not an easy task if you have to jump from engine to engine to retry the search.

Instead of trying multiple queries, you would probably rather issue one query and get back results from many search engines in a consolidated form with duplicate responses already filtered out. To make this a reality, the search engines themselves must become CGI middleware engines, talking to one CGI script that consolidates the results.

Furthermore, a CGI middleware layer can be used to consolidate databases other than ones on the Internet. For example, a company-wide directory service could be programmed to search several internal phone directory databases such as customer data and human resources data as well as using an Internet phone resource such as <http://www.four11.com/> if the information is lacking internally, as shown in [Figure 14-1](#).

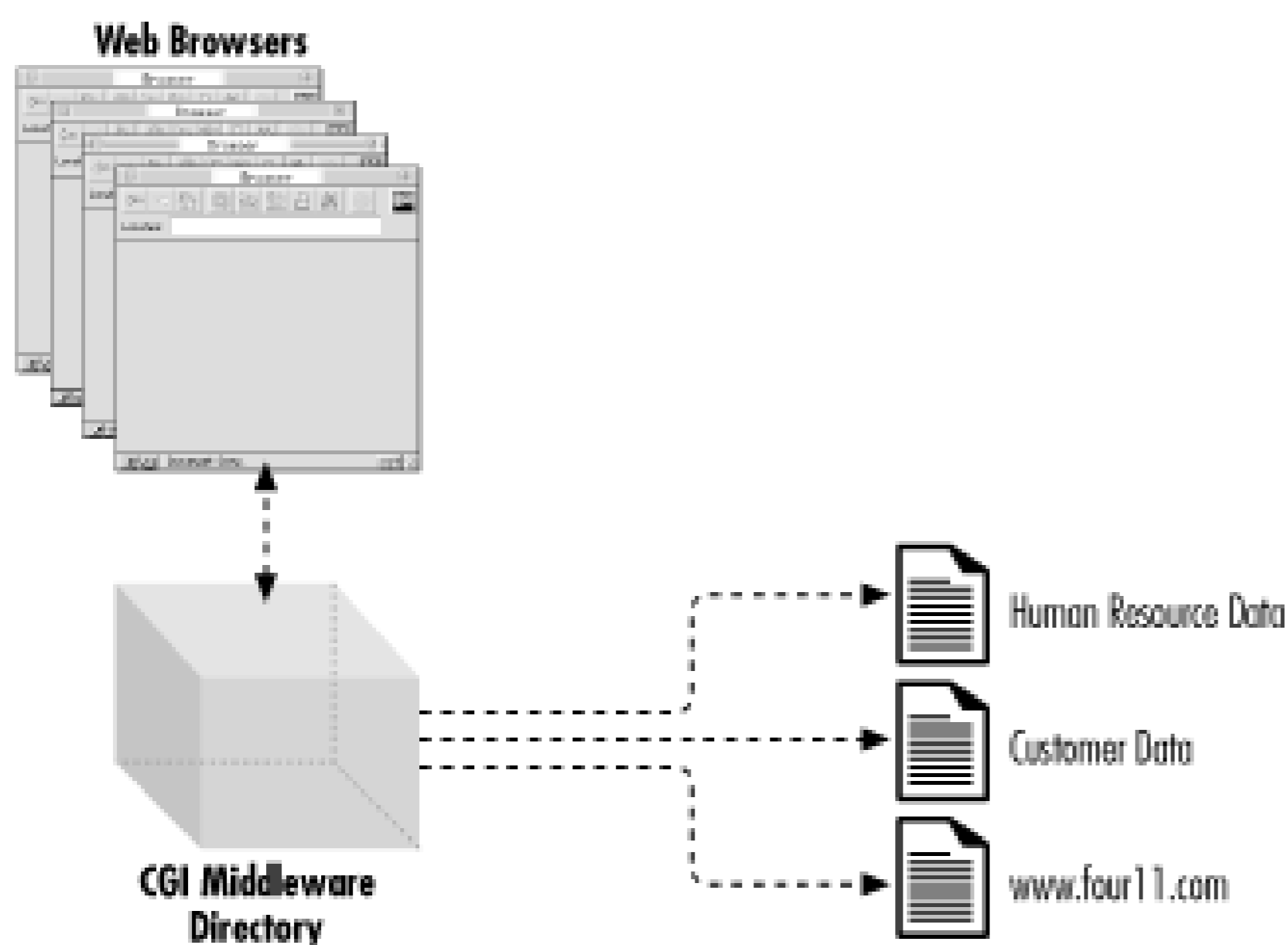


Figure 14-1. Consolidated phone directory interface using CGI middleware

Two technologies to illustrate the use of CGI middleware will be demonstrated later in this chapter. First, we will look at how to perform network connections from your CGI scripts in order to talk to other servers. Then, we introduce eXtensible Markup Language (XML), a platform-independent way of transferring data between programs. We'll show an example using Perl's XML parser.

14.1. Communicating with Other Servers

Let's look at the typical communication scheme between a client and a server. Consider an electronic mail application, for example. Most email applications save the user's messages in a particular file, typically in the `/var/spool/mail` directory. When you send mail to someone on a different host, the mail application must find the recipient's mail file on that server and append your message to it. How does the mail program achieve this task, since it cannot manipulate files on a remote host directly?

The answer to this question is *interprocess communication* (IPC). Typically, there exists a process on the remote host, which acts as a messenger for dealing with email services. When you send a message, the local process on your host communicates with this remote agent across a network to deliver mail. As a result, the remote process is called a server (because it services an issued request), and the local process is referred to as a client. The Web works along the same philosophy: the browser represents the client that issues a request to an HTTP server that interprets and executes the request.

The most important thing to remember here is that the client and the server must speak the same language. In other words, a particular client is designed to work with a specific server. So, for example, an email client, such as Eudora, cannot communicate with a web server. But if you know the stream of data expected by a server, and the output it produces, you can write an application that communicates with the server, as you will see later in this chapter.

14.1.1. Sockets

Most companies have a telephone switchboard that acts as a gateway for calls coming in and going out. A socket can be likened to a telephone switchboard. If you want to connect to a remote host, you need to first create a socket through which the communications would occur. This is similar to dialing "9" to go through the company switchboard to the outside world.

Similarly, if you want to create a server that accepts connections from remote (or local) hosts, you need to set up a socket that listens for connections. The socket is identified on the Internet by the host's IP address and the port that it listens on. Once a connection is established, a new socket is created to handle this connection, so that the original socket can go back and listen for more connections. The telephone switchboard works in the same manner: as it handles outside phone calls, it routes them to the appropriate extension and goes back to accept more calls.

For the sake of discussion, think of a socket simply as a pipe between two locations. You can send and receive information through that pipe. This concept will make it easier for you to understand socket I/O.

14.1.2. IO::Socket

The IO::Socket module, which is included with the standard Perl distribution, makes socket programming simple. [Example 14-1](#) provides a short program that takes a URL from the user, requests the resource via a GET method, then prints the headers and content.

Example 14-1. socket_get.pl

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

use strict;

use IO::Socket;
use URI;

my $location = shift || die "Usage: $0 URL\n";

my $url      = new URI( $location );
my $host     = $url->host;
my $port    = $url->port || 80;
my $path    = $url->path || "/";

my $socket  = new IO::Socket::INET (PeerAddr => $host,
                                   PeerPort => $port,
                                   Proto    => 'tcp')
              or die "Cannot connect to the server.\n";
```

```

$socket->autoflush (1);

print $socket "GET $path HTTP/1.1\n",
             "Host: $host\n\n";
print while (<$socket>);

$socket->close;

```

We use the URI module discussed in [Chapter 2, "The Hypertext Transport Protocol"](#), to break the URL supplied by the user into components. Then we create a new instance of the IO::Socket::INET object and pass it the host, port number, and the communications protocol. And the module takes care of the rest of the details.

We make the socket unbuffered by using the `autoflush` method. Notice in the next set of code that we can use the instance variable `$socket` as a file handle as well. This means that we can read from and write to the socket through this variable.

This is a relatively simple program, but there is an even easier way to retrieve web resources from Perl: LWP.

14.1.3. LWP

LWP, which stands for *libwww-perl*, is an implementation of the W3C's *libwww* package for Perl by Gisle Aas and Martijn Koster, with contributions from a host of others. LWP allows you to create a fully configurable web client in Perl. You can see an example of some of what LWP can do in [Section 8.2.5, "Trusting the Browser"](#).

With LWP, we can write our web agent as shown in [Example 14-2](#).

Example 14-2. `lwp_full_get.pl`

```

#!/usr/bin/perl -wT

use strict;
use LWP::UserAgent;
use HTTP::Request;

my $location = shift || die "Usage: $0 URL\n";

my $agent = new LWP::UserAgent;
my $req = new HTTP::Request GET => $location;
    $req->header('Accept' => 'text/html');

my $result = $agent->request( $req );

```



```
print $result->headers_as_string,  
      $result->content;
```

Here we create a user agent object as well as an HTTP request object. We ask the user agent to fetch the result of the HTTP request and then print out the headers and content of this response.

Finally, let's look at `LWP::Simple`. `LWP::Simple` does not offer the same flexibility as the full LWP module, but it is much easier to use. In fact, we can rewrite our previous example to be even shorter; see [Example 14-3](#).

Example 14-3. `lwp_simple_get.pl`

```
#!/usr/bin/perl -wT  
  
use strict;  
use LWP::Simple;  
  
my $location = shift || die "Usage: $0 URL\n";  
  
getprint( $location );
```

There is a slight difference between this and the previous example. It does not print the HTTP headers, just the content. If we want to access the headers, we would need to use the full LWP module instead.

Chapter 15. Debugging CGI Applications

Contents:

[Common Errors](#)

[Perl Coding Techniques](#)

[Debugging Tools](#)

So far, we've discussed numerous CGI applications, ranging from the trivial to the very complex, but we haven't touched upon the techniques needed to debug them if something goes wrong. Debugging a CGI application is not much different than debugging any other type of application, because, after all, code is code. However, since a CGI application is run by a remote user across the network in a special environment created by the web server, it is sometimes difficult to pinpoint the problems.

This chapter is all about debugging CGI applications. First, we'll examine some of the common errors that developers generally come across when implementing CGI applications. These include incorrect server configuration, permission problems, and violations of the HTTP protocol. Then, we'll explore a few tips, tricks, and tools that will help us track down problems and develop better applications.

15.1. Common Errors

This section can serve as a checklist that you can use to diagnose common problems. Here is a list of common sources of errors:

Source of Problem	Typical Error Message
Application permissions	403 Forbidden
The pound-bang line	403 Forbidden
Line endings	500 Internal Server Error
"Malformed" header	500 Internal Server Error

Let's look at each of these in more detail.

15.1.1. Application Permissions

Typically, web servers are configured to run as *nobody* or another user with minimal access privileges. This is a great preventative step, and one that can possibly salvage your data in the case of an attack. Since the web server process does not have privileges to write to, read from, or execute files in

directories that don't have "world" access, most of your data will stay intact.

However, this also create a few problems for us. First and foremost, we need to set the world execute bit on the CGI applications, so the server can execute them. Here's how you can check the permissions of your applications:

```
$ ls -l /usr/local/apache/cgi-bin/clock
-rwx----- 1 shishir      3624 Oct 17 17:59 clock
```

The first field lists the permissions for the file. This field is divided into three parts: the privileges for the owner, the group, and the world (from left to right), with the first letter indicating the type of the file: either a regular file, or a directory. In this example, the owner has sole permission to read, write, and execute the program.

If you want the server to be able to execute this application, you have to issue the following command:

```
$ chmod 711 clock
-rwx--x--x 1 shishir      3624 Oct 17 17:59 clock*
```

The `chmod` command (change mode) modifies the permissions for the file. The octal code of 711 indicates read (octal 4), write (octal 2), and execute (octal 1) permissions for the owner, and execute permissions for everyone else.

That's not the end of our permission woes. We could run into other problems dealing with file permissions, most notably, the inability to create or update files. We will discuss this in [Section 15.2, "Perl Coding Techniques"](#) later in this chapter.

Despite configuring the server to recognize CGI applications and setting the execute permissions, our applications can still fail to execute, as you'll see next.

15.1.2. The Pound-Bang

If a CGI application is written in Perl, Python, Tcl, or another interpreted scripting language, then it must have a line at the very top that begins with a pound-bang, or `#!`, like this:

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

We've seen this above every script throughout this book. When the web server recognizes a request for a CGI application, it calls the `exec` system function to execute the application. If the application is a compiled executable, the operating system will go ahead and execute it. However, if our application is a script of some sort, then the operating system will look at the first line to see what interpreter to use.

If your scripts are missing the pound-bang line, or if the path you specify is invalid, then you will get an error. On some systems, for example, `perl` is found at `/usr/bin/perl`, while on others it is found at `/usr/local/bin/perl`. On Unix systems, you can use either of the following commands to locate `perl`

(depending on your shell):

```
$ which perl
$ whence perl
```

If neither of these commands work, then look for `perl5` instead of `perl`. If you still cannot locate `perl`, then try either of the following commands. They return anything on your filesystem named `perl`, so they could return multiple results, and the `find` command will search your entire filesystem, so depending on the size of the filesystem, this could take a while:

```
$ locate perl
$ find / -name perl -type f -print 2>/dev/null
```

Another thing to keep in mind: if you have multiple interpreters (i.e., different versions) for the same language, make sure that your scripts reference the one you intend, or else you may see some mysterious effects. For example, on some systems, `perl4` is still installed in addition to `perl5`. Test the path you use with the `-v` flag to get its version.

15.1.3. Line Endings

If you are working with a CGI script that downloaded from another site or edited with a different platform, then it is possible that the line endings do not match those of the current system. For example, `perl` on Unix will complain with multiple syntax errors if you attempt to run a file that is formatted for Windows. You can clean these files up with `perl` from the command line:

```
$ perl -pi -e 's/\r\n/\n/' calendar.cgi
```

15.1.4. "Malformed" Header

As we first discussed in [Chapter 2, "The Hypertext Transport Protocol"](#), and [Chapter 3, "The Common Gateway Interface"](#), and have seen in all the examples since, all CGI applications must return a valid HTTP content-type header, followed by a newline, before the actual content, like this:

```
Content-type: text/html
(other headers)

(content)
```

If you fail to follow this format, then a typical *500 Server Error* will ensue. The partial solution is to return all necessary HTML headers, including content type, as early on in the CGI application as possible. We will look at a very useful technique in the next section that will help us with this task.

However, there are other reasons why we may see such an error. If your CGI application generates errors that are printed to `STDERR`, these error messages may be returned to the web server before all of

the header information. Because Perl buffers output to STDOUT, errors that occur after you have printed the headers may even cause this problem.

What's the moral? Make sure you check your application from the command line before you try to execute it from the Web. If you're using Perl to develop CGI applications, then you can use the `-wCT` switch to check for syntax errors:

```
$ perl -wCT clock.cgi
syntax error in file clock.cgi at line 9, at EOF
clock.cgi had compilation errors.
```

If there are warnings, but no errors, you may see the following:

```
$ perl -wCT clock.cgi
Name "main::opt_g" used only once: possible typo at clock.cgi line 5.
Name "main::opt_u" used only once: possible typo at clock.cgi line 6.
Name "main::opt_f" used only once: possible typo at clock.cgi line 7.
clock.cgi syntax OK
```

Pay attention to the warnings, as well. Perl's syntax checker has really improved over the years, and will alert you of many possible errors, such as using non existent variables, uninitialized variables, or file handles.

And finally, if there are no warnings or errors, you will see:

```
$ perl -wCT clock.cgi
clock.cgi syntax OK
```

To reiterate, make sure your application works from the command line before you even attempt to debug its functionality from the Web.

Chapter 16. Guidelines for Better CGI Applications

Contents:

Architectural Guidelines

Coding Guidelines

Like many forms of programming, CGI programming with Perl is a balance between art and science. As an art form, Perl is such a uniquely expressive language that you have the freedom to accomplish the same tasks in many different ways. However, thinking of Perl as a science, you will want to choose methods based on balancing such real-world requirements as performance, security, and team development.

Furthermore, any program that is useful in one context will generally evolve to be useful in others. This requires that a program be flexible and have the capability to grow. Unfortunately, programs do not grow by themselves. They require the dreaded m-word: maintenance. Maintenance is usually difficult, but it can be made easier by taking steps to make sure that the code is readable as well as flexible.

Because of these concerns, seasoned CGI developers typically end up sticking to a set of guidelines that help their code live up to these expectations. In a corporate setting, these guidelines tend to become the standards through which teams of developers understand how to easily read the code that their neighbors produce.

16.1. Architectural Guidelines

The first step in learning any language consists of being able to accomplish small tasks without the compiler complaining. However, larger programs are made up of more than just syntactically correct statements. The details of how the small parts of a program fit together is just as important as making sure that those same small parts compile successfully.

In other words, a program is literally more than the sum of its parts. Attention must be paid to developing the program in order to accommodate design goals such as flexibility and future maintainability. Sometimes this is referred to as "programming in the large" or "strategic programming." This section emphasizes specific tips on how to architect a CGI application for these design goals.

16.1.1. Plan for Future Growth

Web sites may start small, but they typically grow and evolve over time. You may start out working on a small site without many developers where it is easy to coordinate work. However, as web sites grow and the staff that develops and supports the web site grows, it becomes more critical that it is designed well. Developers should have a development site where they can work on their own copies of the web site without affecting the production web server.

As web sites grow and multiple developers share work on projects, a system to track changes to your applications is crucial. If you are not using a revision control system, you should be planning for one. There are numerous commercial products available for revision control in addition to open source implementations of CVS and RCS. Supporting for a revision control system is an important consideration when making architectural decisions.

You can configure this a number of different ways. Here are a few examples:

- *Web developers share a common development web server.* This is the simplest solution and can work for small groups but quickly becomes unwieldy for large projects. This does not support revision control at a user level, and there is no stable code base because everything is in flux. One developer would be unable to test a component with another developer's code while the second developer is making changes to that code.
- *Web developers have their own directory tree on the web server.* In this example, each developer has a home directory on the web server and can access a copy of web server's contents beneath this directory. This is relatively easy to set up and works if HTML links are relative to the current directory. This supports revision control systems because developers can periodically check in (preferably stable) snapshots of their code. Other developers can update their directories with these snapshots and even develop code in parallel.
- *Web developers have their own copy of the web server running on a separate port.* This requires the most configuration because the web server must be reconfigured each time a port is added for a developer. This works for all relative URLs, whether they contain full paths or paths relative to the current directory. This also supports revision control.

16.1.2. Use Directories to Organize Your Projects

CGI applications often consist of several related files, including one or more CGI scripts, HTML forms, template files -- if you are generating output with templates, data files, configuration files, etc. If your development system is separate from your production server (as it should be), then these systems may have different directory structures.

On your development system you should develop a directory structure that helps you organize this

information easily. On systems that support pointers to directories,[22] it is a good idea to place all the files for a given CGI application within one directory. For example, if you had an web storefront application, you might store the components in subdirectories within */usr/local/projects/web_store* like so:

[22]Such pointers could include symlinks on Unix or aliases on MacOS; Windows shortcuts are not transparent to applications and thus will not work in this context.

```
/usr/local/projects/web_store/
  cgi/
  conf/
  data/
  html/
  templates/
```

You could then create the following symlinks that map this content into the corresponding directories your web server uses:

```
/usr/local/apache/htdocs/web_store    -> /usr/local/projects/web_store/html/
/usr/local/apache/cgi-bin/web_store    -> /usr/local/projects/web_store/cgi/
```

You may also wish to add global directories for data, configuration, and template files:

```
/usr/local/apache/data/web_store      -> /usr/local/projects/web_store/conf/
/usr/local/apache/conf/web_store      -> /usr/local/projects/web_store/data/
/usr/local/apache/templates/web_store -> /usr/local/projects/web_store/templates/
```

Besides making it easier to locate all of the components that are part of the web store application, placing all of your content beneath a common directory such as */usr/local/projects/web_store* makes it easier to manage this application with a revision control system.

Note that it is slower for the web server to follow a symlink than to stay in the document root, so this structure makes more sense on a development system than on a production system.

16.1.3. Use Relative URLs

Your web site will be most flexible if you use relative URLs instead of absolute URLs. In other words, do not include the domain name of your web server when you do not need to. If your development and production web servers have different names, you want your code to work on either system with very little reconfiguration.

Whether these relative URLs contain fully qualified paths or paths that are relative to the current directory depends on how you have configured your development system, as we previously discussed. However, primary navigation elements, such as navigation bars, almost always use fully qualified paths, so configuring your development environment to support this allows the development environment to better mirror the production environment.

16.1.4. Separate Configuration from Your Primary Code

Information that is likely to change in the program or that is dependent upon the environment should be placed in a separate setup file. With Perl, setup files are easy because you can write the file in Perl; they simply need to set one or more global variables. To access these variables in a CGI script, first use Perl's `require` function to import the configuration file.

In some scenarios, each web developer may need different configuration parameters. By storing file paths in a configuration file, web developers can test their applications with their own copies of data and HTML files. However, that does not mean that CGI scripts need to require multiple files; another advantage to using Perl for setup files is that they are easily extended. A CGI script can require a single configuration file that requires other files. This easily supports configuration files for both applications and developers. Likewise, if a CGI application grows so large that a single application configuration file is difficult to manage, you can break it into smaller files and have the primary configuration file require these smaller sections.

16.1.5. Separating Display from Your Primary Code

The display associated with a CGI script is one of the most likely things to change in the lifetime of an application. Most Web sites undergo some look and feel change during their evolution, and an application that will be used across several web sites needs to be flexible enough to accommodate all of their individual cosmetic guidelines. We discussed many of the arguments for separating display from logic in Chapter 6, "HTML Templates" .

However, even beyond keeping HTML separate from code so that HTML maintainers have an easier time, it is a good idea to develop the code that handles display (such as template parsing calls, CGI.pm methods, etc.) separated from the rest of your program logic. This allows you to change the solution you use for generating display with as little effort as possible. You may at some point decide you want to port all your CGI scripts from CGI.pm to templates or vice versa.

Another reason for separating display from the main program logic is that you may not want to limit your program to displaying HTML. As your program evolves, you may want to provide other interfaces. You may wish to convert from basic HTML to the new XHTML standard. Or you might want to add an XML interface to allow other systems programs to grab and process the output of your CGI script as data.

16.1.6. Separating Storage from Your Primary Code

The manner of storing and retrieving data is a key architecture decision that every application

encounters. A simple shopping cart might start out using flat text files to store shopping cart data throughout the user's shopping experience. However, a more sophisticated one will probably want to take advantage of relational databases such as MySQL or Oracle. Other applications may use DBM hash files.

Separating the code that is responsible for data storage from your core program logic is good architectural design. In practice, this can be more difficult to achieve than separating other components of your programs such as display. Often your logic is closely tied to your data. Sometimes you must also make trade-offs with performance; SQL for example is such an expressive language, it is possible to embed logic into your queries, and this is typically much faster and more memory efficient than duplicating this functionality in your programs.

However, it is a good idea to strive towards a separation, especially if your application is using simpler storage mechanisms such as text files. Because applications grow, you may easily find yourself adopting a full RDBMS down the road. The least amount of change required in your code, the better.

One strategy is to simply allow DBI to be your layer of abstraction. If you are not ready for a database, you can use `DBD::CSV` to store your data in text files. Later, if you move to a relational database, most of your code that is built around DBI will not need to change. Keep in mind that not all DBI drivers are equal. `DBD::CSV`, for example, only supports limited SQL queries; while on the other extreme, complex drivers like `DBD::Oracle` allow you to use stored procedures written in Oracle's PL/SQL programming language. Thus, even with DBI, you must balance the portability of writing simple, vanilla SQL against the performance advantages that you can get by taking advantage of particular features available to you with your current storage mechanism, as well as the likelihood that you will want to change storage mechanisms in the future.

16.1.7. Number of Scripts per Application

CGI applications often consist of many different tasks that must work together. For example, in a basic online store you will have code to display a product catalog, code to update a shopping cart, code to display the shopping cart, and code to accept and process payment information. Some CGI developers would argue that all of this should be managed by a single CGI script, possibly breaking some functionality out into modules that can be called by this script. Others would argue that a separate CGI scripts should support each page or functional group of pages, possibly moving common code into modules that can be shared by this script. There are reasons for pursuing either approach; let's take a look at them.

16.1.7.1. Using one CGI program rather than many for each major application

Having one file makes things simple; there is only one file one must edit to make changes. One doesn't need to look through multiple files in order to find a particular section of code. Imagine you saw a

directory with multiple applications:

```
web_store.cgi
order.cgi
display_cart.cgi
maintain_cart.cgi
```

Without delving into the source code, you might pick out that *web_store.cgi* is the primary application. Furthermore, you might conclude that the program probably prints out a front page inviting the user to shop and provides links to the other CGI programs. You would also be able to tell which scripts have to do with ordering, displaying, and maintaining cart information.

However, without actually going into the source code of all these CGI scripts, it is difficult to tell how they relate to one another. For example, can you add or delete shopping cart items from the order page?

Instead, you can make just one CGI program: *web_store.cgi*. This combined script can then import the functionality of order forms, cart data display and maintenance using libraries or modules.

Second, different components often need to share code. It is much simpler for one component to access code in another component if they are in the same file. Moving shared code into modules is certainly an alternative that works well for applications distributed into multiple CGI scripts. However, using modules to share common code requires a greater degree of planning to know what code can be shared and what code will not. A single file is more amenable to making simple changes.

It is possible to use modules with this single CGI program approach. In fact, you can keep file sizes small if you want by making the primary CGI script a basic interface, or a wrapper, that routes requests to other modules. In this scenario, you create multiple modules that handle the different tasks. In many ways it is like having multiple files except that all HTTP requests are directed through a common front-end.

If you write CGI scripts that you distribute so that others may download and install them on their own systems, then you may certainly want to reduce the number of files in your application. In this scenario, the focus is on making the application easy to install and configure. People installing software care more about what the package does than one individual tasks are handled by which component, and it is easier for them to avoid accidentally deleting a file they didn't realize was important if the number of files is minimized.

The final reason you may wish to combine CGI scripts is if you are running FastCGI. FastCGI runs a separate process for each of your CGI scripts, so the fewer scripts you have, the fewer separate processes are running.

16.1.7.2. Using multiple CGI scripts for each major application

There are also several reasons to keep applications distributed. First of all, it does keep files smaller and

more manageable. This also helps with projects that have multiple developers, because reconciling changes made by multiple developers working on the same file at the same time can be complicated to say the least.

Of course, as we stated before, one can keep files small and separated when using the single CGI program approach by shifting code into modules and restricting the single CGI program to being a simple interface that routes requests to the appropriate modules. However, creating a general front-end that uses modules for specific tasks is a rather backward approach for Perl. Typically, Perl modules contain general code that can be shared across multiple programs that do specific tasks. Keeping general code within modules also allows them to be potentially shared across different CGI applications.

It is true that creating multiple files requires more architectural planning when different components need to share the same code because the common code must be placed in a module. You should always plan your architecture carefully and be wary of quick and simple solutions. The problem with quick and simple solutions is that too many of them begin to bloat an application and create an inferior overall solution. It may require a bit more work in the short term to shift code in one component into a module because another component needs to access it; however, in the long run the application may be much more flexible and easier to maintain with this module than it would be if all the code is simply dumped into a common file.

There are some cases when it is clear that code should be kept separate. Some applications, such as our web store example, may have administrative pages where employees can update product information, modify product categories, etc. Those tasks that require a different level of authorization should certainly be kept separate from the public code for security reasons. Administrative code should be placed in a separate subdirectory, such as `/usr/local/apache/cgi-bin/web_store/admin/` that is restricted by the web server.

If you do choose to separate a CGI application into multiple scripts, then you should certainly create a separate directory within `/cgi-bin` for each application. Placing lots of files from lots of different applications together in one directory guarantees confusion later.

16.1.8. Using Submit Buttons to Control Flow

Whether or not you break your applications into multiple scripts, you will still encounter situations where one form may allow the user to choose very different actions. In this case, your CGI script can determine what action to take by looking at the name of the submit button that was chosen. The name and value of submit buttons is only included within form query requests if they were clicked by the user. Thus, you can have multiple submit buttons on the HTML form with different names indicating different paths of logic that the program should follow.

For example, a simple shopping cart CGI script may begin with code like the following:

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

```
use strict;
use CGI;

my $q      = new CGI;
my $quantity = $q->param( "quantity" );
my $item_id = $q->param( "item_id" );
my $cart_id = $q->cookie( "cart_id" );

# Remember to handle exceptional cases
defined( $item_id ) or die "Invalid input: no item id";
defined( $cart_id ) or die "No cookie";

if ( $q->param( "add_item" ) ) {
    add_item( $cart_id, $item_id );
} elsif ( $q->param( "delete_item" ) ) {
    delete_item( $cart_id, $item_id );
} elsif ( $q->param( "update_quantity" ) ) {
    update_quantity( $cart_id, $item_id, $quantity );
} else {
    display_cart( $cart_id );
}

# That's it; subroutines follow...
```

From looking at this section of code, it is easily apparent how the entire script reacts to input and the role of each of the subroutines. If we clicked on a submit button represented in an HTML form with `<INPUT TYPE="submit" NAME="add_item" VALUE="Add Item to Cart">`, the script would call the `add_item` subroutine. Furthermore, it is clear that the default behavior is defined as displaying the shopping cart.

Note that we are branching based on the name of the submit button and not the value; this allows HTML designers to alter the text on the button displayed to users without affecting our script.

Chapter 17. Efficiency and Optimization

Contents:

[Basic Perl Tips, Top Ten](#)

[FastCGI](#)

[mod_perl](#)

Let's face it, CGI applications, run under normal conditions, are not exactly speed demons. In this chapter, we will show you a few tricks that you can use to speed up current applications, and also introduce you to two technologies -- FastCGI and *mod_perl* -- that allow you to develop significantly accelerated CGI applications. If you develop Perl CGI scripts on Win32, then you may also wish to look at ActiveState's PerlEx. Although we do not discuss PerlEx in this chapter, it provides many of the same benefits as *mod_perl*.

First, let's try to understand why CGI applications are so slow. When a user requests a resource from a web server that turns out to be a CGI application, the server has to create another process to handle the request. And when you're dealing with applications that use interpreted languages, like Perl, there is an additional delay incurred in firing up the interpreter, then parsing and compiling the application.

So, how can we possibly improve the performance of Perl CGI applications? We could ask Perl to interpret only the most commonly used parts of our application, and delay interpreting other pieces unless necessary. That certainly would speed up applications. Or, we could turn our application into a server (daemon) that runs in the background and executes on demand. We would no longer have to worry about the overhead of firing up the interpreter and evaluating the code. Or, we could embed the Perl interpreter within the web server itself. Again, we avoid the overhead of having to start a new process, and we don't even suffer the communication delay we would have talking to another daemon.

We'll look at all the techniques mentioned here, in addition to basic Perl tips for writing more efficient applications. Let's start with the basics.

17.1. Basic Perl Tips, Top Ten

Here is a list of ten techniques you can use to improve the performance of your CGI scripts:

10. Benchmark your code.
9. Benchmark modules, too.
8. Localize variables with `my`.

7. Avoid slurping data from files.
6. Clear arrays with `undef` instead of `()`.
5. Use *SelfLoader* where applicable.
4. Use *autouse* where applicable.
3. Avoid the shell.
2. Find existing solutions for your problems.
1. Optimize your regular expressions.

Let's look at each one in more detail.

17.1.1. Benchmark Your Code

Before we can determine how well our program is working, we need to know how to benchmark the critical code. Benchmarking may sound involved, but all it really involves is timing a piece of code, and there are some standard Perl modules to make this very easy to perform. Let's look at a few ways to benchmark code, and you can choose the one that works best for you.

First, here's the simplest way to benchmark:

```
$start = (times)[0];

## your code goes here

$end = (times)[0];

printf "Elapsed time: %.2f seconds!\n", $end - $start;
```

This determines the elapsed user time needed to execute your code in seconds. It is important to consider a few rules when benchmarking:

- Try to benchmark only the relevant piece(s) of code.
- Don't accept the first benchmark value. Benchmark the code several times and take the average.
- If you are comparing different benchmarks, make sure they are tested under comparable conditions. For example, make sure that the load on the machine doesn't differ between tests because another user happened to be running a heavy job during one.

Second, we can use the Benchmark module. The Benchmark module provides us with several functions

that allow us to compare multiple pieces of code and determine elapsed CPU time as well as elapsed real-world time.

Here's the easiest way to use the module:

```
use Benchmark;
$start = new Benchmark;

## your code goes here

$end = new Benchmark;

$elapsed = timediff ($end, $start);
print "Elapsed time: ", timestr ($elapsed), "\n";
```

The result will look similar to the following:

```
Elapsed time: 4 wallclock secs (0.58 usr + 0.00 sys = 0.58 CPU)
```

You can also use the module to benchmark several pieces of code. For example:

```
use Benchmark;
timethese (100, {
    for => <<'end_for',
        my $loop;
        for ($loop=1; $loop <= 100000; $loop++) { 1 }
    end_for
    foreach => <<'end_foreach'
        my $loop;
        foreach $loop (1..100000) { 1 }
    end_foreach
} );
```

Here, we are checking the *for* and *foreach* loop constructs. As a side note, you might be interested to know that, in cases where the loop iterator is great, *foreach* is much less efficient than *for* in versions of Perl older than 5.005.

The resulting output of `timethese` will look something like this:

```
Benchmark: timing 100 iterations of for, foreach...
  for: 49 wallclock secs (49.07 usr + 0.01 sys = 49.08 CPU)
  foreach: 69 wallclock secs (68.79 usr + 0.00 sys = 68.79 CPU)
```

One thing to note here is that `Benchmark` uses the `time` system call to perform the actual timing, and therefore the granularity is still limited to one second. If you want higher resolution timing, you can experiment with the `Time::HiRes` module. Here's an example of how to use the module:

```
use Time::HiRes;
my $start = [ Time::HiRes::gettimeofday( ) ];
```



```
## Your code goes here

my $elapsed = Time::HiRes::tv_interval( $start );
print "Elapsed time: $elapsed seconds!\n";
```

The `gettimeofday` function returns the current time in seconds and microseconds; we place these in a list, and store a reference to this list in `$start`. Later, after our code has run, we call `tv_interval`, which takes `$start` and calculates the difference between the original time and the current time. It returns a floating-point number indicating the number of seconds elapsed.

One caveat: the less time your code takes, the less reliable your benchmarks will be. `Time::HiRes` can be useful for determining how long portions of your program take to run, but do not use it if you want to compare two subroutines that each take less than one second. When comparing code, it is better to use `Benchmark` and have it test your subroutines over many iterations.

17.1.2. Benchmark Modules, Too

CPAN is absolutely wonderful. It contains a great number of highly useful Perl modules. You should take advantage of this resource because the code available on CPAN has been tested and improved by the entire Perl community. However, if you are creating applications where performance is critical, remember to benchmark code included from modules you are using in addition to your own. For example, if you only need a portion of the functionality available in a module, you may benefit by deriving your own version of the module that is tuned for your application. Most modules distributed on CPAN are available according to the same terms as Perl, which allows you to modify code without restriction for your own internal use. However, be sure to verify the licensing terms for a module before you do this, and if you believe your solution would be beneficial to others, notify the module author, and please give back to CPAN.

You should also determine whether using a module make sense. For example, a popular module is `IO::File`, which provides a set of functions to deal with file I/O:

```
use IO::File;
$fh = new IO::File;
if ($fh->open ("index.html")) {
    print <$fh>;
    $fh->close;
}
```

There are advantageous to using an interface like `IO::File`. Unfortunately, due to module loading and method-call overhead, this code is, on the average, ten times slower than:

```
if (open FILE, "index.html") {
    print <FILE>;
    close FILE;
}
```

So the bottom line is, pay very careful attention to modules that you use.

17.1.3. Localize Variables with `my`

You should create lexical variables with the `my` function. Perl keeps track of managing memory usage for you, but it doesn't look ahead to see if you are going to use a variable in the future. In order to create a variable that you need only within a particular block of code, such as a subroutine, declare it with `my`. Then the memory for that variable will be reclaimed at the end of the block.

Note that despite its name, the *local* function doesn't localize variables in the standard sense of the term. Here is an example:

```
sub name {
    local $my_name = shift;
    greeting( );
}

sub greeting {
    print "Hello $my_name, how are you!\n";
}
```

If you run this simple program, you can see that `$my_name` isn't exactly local to the `name` function. In fact, it is also visible in `greeting`. This behavior can produce unexpected results if you are not careful. Thus, most Perl developers avoid using `local` and use `my` instead for everything except global variables, file handles, and Perl's built-in global punctuation variables like `$_` or `$/`.

17.1.4. Avoid Slurping

What is slurping, you ask? Consider the following code:

```
local $/;
open FILE, "large_index.html" or die "Could not open file!\n";
$large_string = <FILE>;
close FILE;
```

Since we undefine the input record separator, one read on the file handle will *slurp* (or read in) the entire file. When dealing with large files, this can be highly inefficient. If what you are doing can be done a line at a time, then use a `while` loop to process only a line at a time:

```
open FILE, "large_index.html" or die "Could not open file!\n";
while (<FILE>) {
    # Split fields by whitespace, output as HTML table row
    print $q->tr( $q->td( [ split ] ) );
}
close FILE;
```

Of course, there are situations when you cannot process a line at a time. For example, you may be looking for data that crosses line boundaries. In this case, you may fall back to slurping for small files. Try benchmarking your code to see what kind of penalty is imposed by slurping in the entire file.

17.1.5. undef Versus ()

If you intend to reuse arrays, especially large ones, it is more efficient to clear them out by equating them to a null list instead of undefining them. For example:

```
...
while (<FILE>) {
    chomp;
    $count++;
    $some_large_array[$count] .= int($_);
}
...

@some_large_array = ( );      ## Good
undef @some_large_array;     ## Not so good
```

If you undefine `@some_large_array` to clear it out, Perl will deallocate the space containing the data. And when you populate the array with new data, Perl will have to reallocate the necessary space again. This can slow things down.

17.1.6. SelfLoader

The SelfLoader module allows you to hide functions and subroutines, so the Perl interpreter does not compile them into internal opcodes when it loads up your application, but compiles them only where there is a need to do so. This can yield great savings, especially if your program is quite large and contains many subroutines that may not all be run for any given request.

Let's look at how to convert your program to use self-loading, and then we can look at the internals of how it works. Here's a simple framework:

```
use SelfLoader;

## step 1: subroutine stubs

sub one;
sub two;
...

## your main body of code
...
```

```

## step 2: necessary/required subroutines

sub one {
    ...
}

__DATA__

## step 3: all other subroutines

sub two {
    ...
}
...
__END__

```

It's a three-step process:

1. Create stubs for all the functions and subroutines in your application.
2. Determine which functions are used often enough that they should be loaded by default.
3. Take the rest of your functions and move them between the `__DATA__` and `__END__` tokens.

Congratulations, Perl will now load these functions only on demand!

Now, how does it actually work? The `__DATA__` token has a special significance to Perl; everything after the token is available for reading through the `DATA` filehandle. When Perl reaches the `__DATA__` token, it stops compiling, and all the subroutines defined after the token do not exist, as far as Perl is concerned.

When you call an unavailable function, `SelfLoader` reads in all the subroutines from the `DATA` filehandle, and caches them in a hash. This is a one-time process, and is performed the first time you call an unavailable function. It then checks to see if the specified function exists, and if so, will `eval` it within the caller's namespace. As a result, that function now exists in the caller's namespace, and any subsequent calls to that function are handled via symbol table lookups.

The costs of this process are the one time reading and parsing of the self-loaded subroutines, and a `eval` for each function that is invoked. Despite this overhead, the performance of large programs with many functions and subroutines can improve dramatically.

17.1.7. `autouse`

If you use many external modules in your application, you may consider using the `autouse` feature to delay loading them until a specific function from a module is used:

```
use autouse DB_File;
```

You have to be very careful when using this feature, since a portion of the chain of execution will shift from compile time to runtime. Also, if a module needs to execute a particular sequence of steps early on in the compile phase, using *autouse* can potentially break your applications.

If the modules you need behave as expected, using *autouse* for modules can yield a big savings when it comes time to "load" your application.

17.1.8. Avoid the Shell

Avoid accessing the shell from your application, unless you have no other choice. Perl has equivalent functions to many Unix commands. Whenever possible, use the functions to avoid the shell overhead. For example, use the *unlink* function, instead of executing the external `rm` command:

```
system( "/bin/rm", $file );          ## External command
unlink $file or die "Cannot remove $file: $!"; ## Internal function
```

It is also much safer to avoid the shell, as we saw in [Chapter 8, "Security"](#). However, there are some instances when you may get better performance using some standard external programs than you can get in Perl. If you need to find all occurrences of a certain term in a very large text file, it may be faster to use `grep` than performing the same task in Perl:

```
system( "/bin/grep", $expr, $file );
```

Note however, that the circumstances under which you might need to do this are rare. First, Perl must do a lot of extra work to invoke a system call, so the performance difference gained by an external command is seldom worth the overhead. Second, if you only were interested in the first match and not all the matches, then Perl gains speed because your script can exit the loop as soon as it finds a match:

```
my $match;
open FILE, $file or die "Could not open $file: $!";
while (<FILE>) {
    chomp;
    if ( /$expr/ ) {
        $match = $_;
        last;
    }
}
```

`grep` will always read the entire file. Third, if you find yourself needing to resort to using `grep` to handle text files, it likely means that the problem isn't so much with Perl as with the structure of your data. You should probably consider a different data format, such as a DBM file or a RDBMS.

Also avoid using the glob `<*>` notation to get a list of files in a particular directory. Perl must invoke a subshell to expand this. In addition to this being inefficient, it can also be erroneous; certain shells have

an internal glob limit, and will return files only up to that limit. Note that Perl 5.6, when released, will solve these limitations by handling globs internally.

Instead, use Perl's `opendir`, `readdir`, and `closedir` functions. Here is an example:

```
@files = </usr/local/apache/htdocs/*.html>;      ## Uses the shell
.....
$directory = "/usr/local/apache/htdocs";        ## A better solution
if (opendir (HTDOCS, $directory)) {
    while ($file = readdir (HTDOCS)) {
        push (@files, "$directory/$file") if ($file =~ /\.html$/);
    }
}
```

17.1.9. Find Existing Solutions for Your Problems

Chances are, if you find yourself stuck with a problem, someone else has encountered it elsewhere and has spent a lot of time developing a solution. And thanks to the spirit of Perl, you can likely borrow it. Throughout this book, we have referred to many modules that are available on CPAN. There are countless more. Take the time to browse through CPAN regularly to see what is available there.

You should also check out the Perl newsgroups. *news:comp.lang.perl.modules* is a good place to go to check in with new module announcements or to get help with particular modules. *news:comp.lang.perl* and *news:comp.lang.perl.misc* are more general newsgroups.

Finally, there are many very good books available that discuss algorithms or useful tricks and tips. The *Perl Cookbook* by Tom Christiansen and Nathan Torkington and *Mastering Algorithms with Perl* by Jon Orwant, Jarkko Hietaniemi, and John Macdonald are full of gems specifically for Perl. Of course, don't overlook books whose focus is not Perl. *Programming Pearls* by John Bentley, *The C Programming Language* by Brian Kernighan and Dennis Ritchie, and *Code Complete* by Steve McConnell are also all excellent references.

17.1.10. Regular Expressions

Regular expressions are an integral part of Perl, and we use them in many CGI applications. There are many different ways that we can improve the performance of regular expressions.

First, avoid using `$&`, `$``, and `$'`. If Perl spots one of these variables in your application, or in a module that you imported, it will make a copy of the search string for possible future reference. This is highly inefficient, and can really bog down your application. You can use the `Devel::SawAmpersand` module, available on CPAN, to check for these variables.

Second, the following type of regular expressions are highly inefficient:

```
while (<FILE>) {
    next if (/^(?:select|update|drop|insert|alter)\b/);
    ...
}
```

Instead, use the following syntax:

```
while (<FILE>) {
    next if (/^select/);
    next if (/^update/);
    ...
}
```

Or, consider building a runtime compile pattern if you do not know what you are searching against at compile time:

```
@keywords = qw (select update drop insert);
$code = "while (<FILE>) {\n";

foreach $keyword (@keywords) {
    $code .= "next if (/^$keyword/);\n";
}

$code .= "}\n";
eval $code;
```

This will build a code snippet that is identical to the one shown above, and evaluate it on the fly. Of course, you will incur an overhead for using *eval*, but you will have to weigh that against the savings you will gain.

Third, consider using *o* modifier in expressions to compile the pattern only once. Take a look at this example:

```
@matches = ( );
...
while (<FILE>) {
    push @matches, $_ if /$query/i;
}
...
```

Code like this is typically used to search for a string in a file. Unfortunately, this code will execute very slowly, because Perl has to compile the pattern each time through the loop. However, you can use the *o* modifier to ask Perl to compile the regex just once:

```
push @matches, $_ if /$query/io;
```

If the value of `$query` changes in your script, this won't work, since Perl will use the first compiled value. The compiled regex features introduced in Perl 5.005 address this; refer to the *perlre* manpage for more information.

Finally, there are often multiple ways that you can build a regular expression for any given task, but some ways are more efficient than others. If you want to learn how to write more efficient regular expressions, we highly recommend Jeffrey Friedl's *Mastering Regular Expressions*.

These tips are general optimization tips. You'll get a lot of mileage from some, and not so much from the others, depending on your application. Now, it's time to look at more complicated ways to optimize our CGI applications.

Appendix A. Works Cited and Further Reading

Contents:

[References](#)

[Additional Reading](#)

[RFCs](#)

[Other Specifications](#)

[Project Home Pages](#)

[Newsgroups](#)

The appendix contains a list of the references cited throughout this book as well as additional recommended material.

A.1. References

Bekman, Stas. *mod_perl Guide*. Available online at <http://perl.apache.org/guide/>.

Christiansen, Tom, and Nathan Torkington. *Perl Cookbook*. O'Reilly & Associates, 1998.

Costales, Bryan, with Eric Allman. *sendmail, Second Edition*. O'Reilly & Associates, 1997.

Deep, John, and Peter Holfelder. *Developing CGI Applications with Perl*. John Wiley & Sons, 1996.

Dobbertin H. "The Status of MD5 After a Recent Attack." *RSA Labs' CryptoBytes*. Vol. 2, No. 2, Summer 1996. Available at <http://www.rsasecurity.com/rsalabs/cryptobytes/>.

Friedl, Jeffrey E. F. *Mastering Regular Expressions*. O'Reilly & Associates, 1997.

Garfinkel, Simson, and Gene Spafford. *Practical Unix and Internet Security, Second Edition*. O'Reilly & Associates, 1996.

Flanagan, David. *JavaScript: The Definitive Guide, Third Edition*. O'Reilly & Associates, 1998.

Laurie, Ben, and Peter Laurie. *Apache: The Definitive Guide, Second Edition*. O'Reilly & Associates, 1999.

Orwant, Jon, Jarkko Hietaniemi, and John Macdonald. *Mastering Algorithms with Perl*. O'Reilly & Associates, 1999.

Robert, Kirrily "Skud". "In Defense of Coding Standards." January 2000. Available at <http://www.perl.com/pub/2000/01/CodingStandards.html>.

Siegel, David. *Secrets of Successful Web Sites: Project Management on the World Wide Web*. Hayden Books, 1997.

Srinivasan, Sriram. *Advanced Perl Programming*. O'Reilly & Associates, 1997.

Stein, Lincoln. *Official Guide to Programming with CGI.pm*. John Wiley & Sons, 1998.

Stein, Lincoln, and Doug MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, 1999.

Wall, Larry, Tom Christiansen, and Jon Orwant. *Programming Perl, Third Edition*. O'Reilly & Associates, 2000.

Wallace, Shawn P. *Programming Web Graphics with Perl and GNU Software*. O'Reilly & Associates, 1999.

Walsh, Nancy. *Learning Perl/Tk*. O'Reilly & Associates, 1999.

Zakon, Robert H. *Hobbes' Internet Timeline (v4.1)*. Available online at <http://www.isoc.org/zakon/>.

Appendix B. Perl Modules

Contents:

[CPAN](#)
[perldoc](#)

This book discusses many Perl modules that may not be included with your system. This appendix contains instructions for installing modules from CPAN. It also discusses how to use *perldoc* to access documentation.

B.1. CPAN

CPAN is the Comprehensive Perl Archive Network, found at <http://www.cpan.org/> and at numerous mirrors around the world (see <http://www.cpan.org/SITES.html>). From CPAN you can download source code and binary distributions of Perl, plus all of the modules we mentioned in this book and many other scripts and modules.

You can browse the very long list of modules at <http://www.cpan.org/modules/00modlist.long.html>. If you know the name of a module you wish to download, then you can generally find it via the first word of the module's name. For example, you can download Digest::MD5 from <http://www.cpan.org/modules/by-module/Digest/>. The filename within that directory is *Digest-MD5-2.09.tar.gz* (note that the version number, 2.09, will likely change by the time you read this book).

B.1.1. Installing Modules

All Perl modules distributed on CPAN follow a consistent install process, but some modules are easier to install than others. Some have dependencies on other modules, and some include C source code that must be compiled and often linked to other libraries on your system.

You may have difficulty compiling the modules that contain C code. Most commercial distributions of Unix do not include an ANSI C compiler. You can generally obtain a prebuilt binary of the `gcc` compiler instead. Check software archive sites specific to your platform (for example, <http://www.sun.com/sunsite/> for Solaris and <http://hpux.cae.wisc.edu/> for HP/UX). Linux and BSD systems should already have the tools you need.

If you are using ActiveState's release of Perl on Win32, then you can use the Perl Package Manager to download pre-built binary modules from ActiveState. Visit <http://www.activestate.com/PPM/> for more

information.

The simplest way to install modules on Unix and compatible systems is to use the CPAN.pm module. You can invoke it like this, typically as the superuser:

```
# perl -MCPAN -e shell
```

It creates an interactive shell, from which you to get information about modules on CPAN and install or update modules on your system. The first time you run CPAN, it will prompt you for configuration information that tells it what tools are available for downloading modules, and what CPAN mirrors to use.

Once CPAN is configured, you can install a module by simply typing `install` followed by the name of the module:

```
cpan> install Digest::MD5
```

CPAN will fetch the requested module and install it. CPAN recognizes dependencies on other modules and will automatically install required modules for you. There are several other commands available besides `install`; you can get a full list by entering a question mark at the prompt.

Occasionally, CPAN will not be able to install a module for you. In that case, you will have to install a module manually. On Unix and compatible systems, you should use the following steps after you have downloaded a module:

```
$ gzip -dc Digest-MD5-2.09.tar.gz | tar xvf -
$ cd Digest-MD5-2.09
$ perl Makefile.PL
$ make
$ make test
$ su
# make install
```

If `make` or `make test` fails, then you will need to find and fix the problem. Check the documentation included with the module for assistance. If the module you have downloaded contains C code that links to other libraries, verify that the versions of your libraries match what the Perl module expects. You might also search past newsgroup postings for anyone who already encountered and solved the same problem. You can use <http://www.deja.com/usenet/> for this; navigate to the advanced news search and search *comp.lang.perl.modules* for related terms.

If you have verified any version dependencies, cannot find any answers in the documentation, cannot find any answers in past newsgroup postings, and cannot solve the problem yourself, then post a polite, detailed message to *news:comp.lang.perl.modules* explaining the problem and asking for assistance. You probably should not use *deja.com* for this, however. Unfortunately, some of the most knowledgeable and helpful Perl coders filter out news messages posted from *deja.com* (for the same reason, you may want to avoid sending your message from a Microsoft mail application, too).

Index

[Symbols](#) | [A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal featured on the cover of *CGI Programming with Perl*, Second Edition, is a mouse, a rodent of the family Muridae. True, or long-tailed, mice belong to the youngest group in the animal kingdom, approximately 15 million years old. Over 200 species of mice exist, but the most common is the house mouse. The house mouse is the second most widely distributed mammal on Earth, behind only humans. Despite their name, house mice often live in fields, but they usually live near human dwellings. House mice eat almost anything, but they prefer grains and grain products.

Mice reach sexual maturity at two to three months of age. After a gestation period of 20 to 21 days, they deliver a litter averaging six blind, bald, helpless babies. House-dwelling mice can bear young continually, but if overpopulation becomes a problem some female mice will remain infertile.

Mice are often considered to be pests, or worse. They can cause serious crop damage, as well as food contamination. In addition, mice can carry viral, bacterial, and parasitic disease. Despite all this, mice were worshipped in parts of Asia Minor and Greece in ancient times. Today, mice continue to hold an important part in popular culture, often appearing as the heroes of cartoons and books that are ostensibly intended for children, such as *Stuart Little*, *Pinky and the Brain*, and, of course, Mickey Mouse.

Nicole Arigo was the production editor and copyeditor for *CGI Programming with Perl*, Second Edition. Emily Quill proofread the book. Melanie Wang, Mary Anne Weeks Mayo, and Jane Ellin provided quality control. Ellen Troutman Zaig wrote the index.

Edie Freedman designed the cover of this book, using a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

Alicia Cech and David Futato designed the interior layout based on a series design by Nancy Priest. Mike Sierra implemented the design in FrameMaker 5.5.6. The text and heading fonts are ITC Garamond Light and Garamond Book. The illustrations that appear in the book were produced by Robert Romano and Rhon Porter using Macromedia FreeHand 8 and Adobe Photoshop 5. This colophon was written by Clairemarie Fisher O'Leary.

[Previous](#)

[Home](#)

Index

[Book Index](#)