Visual®

Read Less-Learn More®

Adam McDaniel

Perl and Apache

Sample code available on the companion Web site

Your visual blueprint™ for developing dynamic Web content

Perl and Apache

Your visual blueprint™ for developing dynamic Web content

by Adam McDaniel

WILEY
Wiley Publishing, Inc.

Perl and Apache: Your visual blueprint™ for developing dynamic Web content

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256

www.wiley.com

Published simultaneously in Canada

Library of Congress Control Number: 2010934753

ISBN: 978-0-470-55680-1

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Trademark Acknowledgments

Contact Us

For general information on our other products and services please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993 or fax 317-572-4002.

For technical support please visit www.wiley.com/techsupport.

Credits

Acquisitions Editor
Aaron Black

Project Editor
Jade Williams

Technical Editor
Allen Wyatt

Copy Editor
Marylouise Wiack

Editorial Director
Robyn Siesky

Editorial Manager
Rosemarie Graham

Business Manager
Amy Knies

Senior Marketing Manager
Sandy Smith

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Barry Pruett

Project Coordinator
Patrick Redmond

Graphics and Production Specialists
Andrea Hornberger
Jennifer Mayberry

Quality Control Technician
Jessica Kramer

Proofreading
Tricia Liebig

Indexing
Potomac Indexing, LLC

Media Development Project Manager
Laura Moss

Media Development Assistant Project Manager
Jenny Swisher

Screen Artist
Ana Carrillo
Jill Proll
Ron Terry

Illustrator
Cheryl Grubbs

About the Author

Adam McDaniel has been desgining, developing, modifying, and maintaining computer programs of one language or another since 1993, and has been an active proponent of Perl since being introduced to the langauge in 1998. In early 1999, Adam led a team of developers implementing an E-Commerce fulfillment engine written entirely in Perl for a virtual shopping mall. Afterwards, he worked for Hitachi ID Systems for over 8 years, during which he designed and implemented security recommondations and software for various Fortune 500 companies across the United States and Europe.

Always interested in new technologies and architectures, development credits include an open-source offline HTML reader for the Palm OS platform, contributions to the Linux Kernel, plus countless utility and speciality programs. In 2006, Adam produced the Array.org Netbook Kernel software download and Web site, allowing users to download an optimized build of the Linux kernel, specific for the Ubuntu Linux distribution. This site, implemented using Perl and Apache, became hugely popular resulting in millions unique visitors in just a few months. It actually prompted him to move away from security architecture and design and into a new field: Linux distribution architecture. Today, Adam works as the Lead OS Architect for Jolicloud, a Paris-based Linux distribution that specializes in a cloud-based user interface designed for netbooks, tablets, and other portable computers.

Author's Acknowledgments

This book is actually the product of many significant people, projects, and events, without all of which, this project would never have been possible.

First and foremost, I must thank my wife, Shauna, for her un-ending patience, understanding, encouragement, and love, both silent and vocal, which she happily supplied while I toiled away endlessly on this project. I must also thank my editors at Wiley Publishing, espcially Aaron Black and Jade Williams, for their expert guideance, advice, and patience, despite their occasional prodding over e-mail.

This book could not be possible without hundreds of thousands of developers who have dedicated their time and expertise to open-source software. Projects like the Linux Kernel, Perl, Apache, and everything in-between serves as an excellent model of design, efficiency, and dedication to people like me and other technology enthusiasts.

Finally, regarding significant events, I have to thank our unpredictable Canadian winter weather. In January 2008, we experienced seven days of -40 degree weather; this caused a water pipe to burst and flood my basement with 18 inches of water while I was away from home. Had my basement not flooded, I would never have replaced an old waterlogged laptop with a brand new EeePC netbook through the insurance claim. Without that, I would have never become interested in netbook hardware, nor in customizing the Linux kernel for that hardware. And I certainly would have never created my Array.org Netbook Kernel Web site, through which Aaron Black would never have contacted me, and this book would never have come into existence.

Dedication

This book is dedicated to my father, William McDaniel, who knew about this project, but never got the chance to see the final result.

How to Use This Book

Who This Book Is For

This book is for advanced computer users who want to take their knowledge of this particular technology or software application to the next level.

The Conventions in This Book

1 Steps

This book uses a step-by-step format to guide you easily through each task. Numbered steps are actions you must do; bulleted steps clarify a point, step, or optional feature; and indented steps give you the result.

2 Notes

Notes give additional information — special conditions that may occur during an operation, a situation that you want to avoid, or a cross reference to a related area of the book.

3 Icons and Buttons

Icons and buttons show you exactly what you need to click to perform a step.

4 Extra or Apply It

An Extra section provides additional information about the preceding task — insider information and tips for ease and efficiency. An Apply It section takes the code from the preceding task one step further and allows you to take full advantage of it.

5 Bold

Bold type shows text or numbers you must type.

6 Italics

Italic type introduces and defines a new term.

7 Courier Font

`Courier font` indicates the use of scripting language code such as statements, operators, or functions, and code such as objects, methods, or properties.

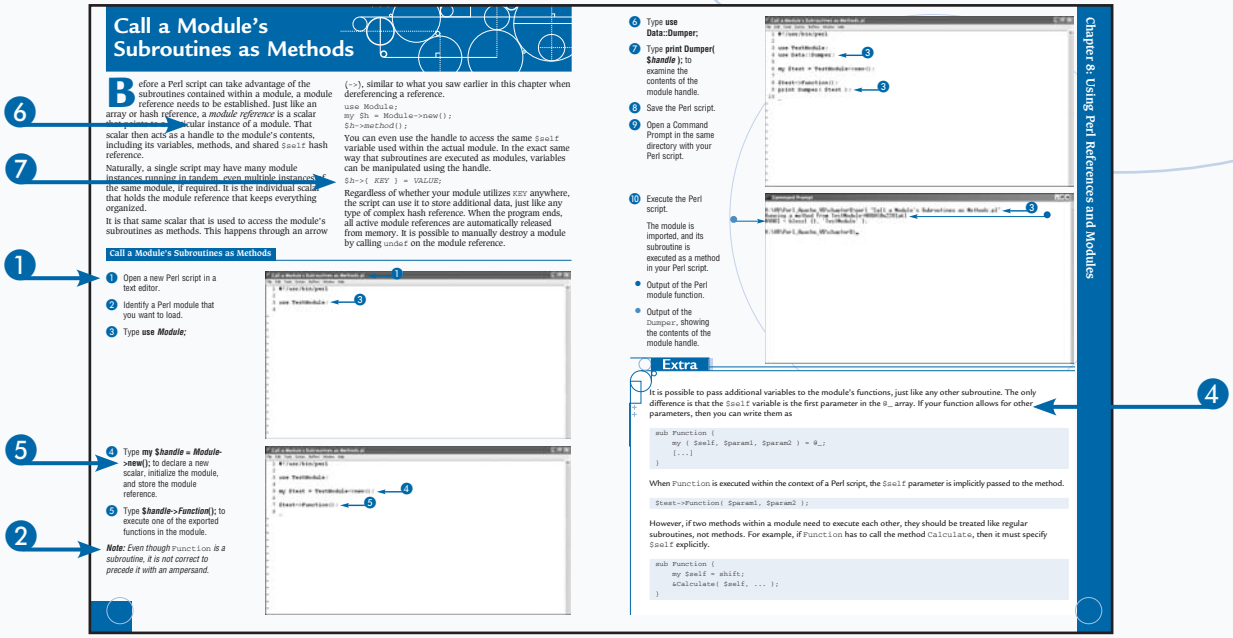


TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

Introducing Apache and Perl

Since the inception of the World Wide Web in 1989, users, academics, and professionals have been inspired by this new canvas to present information over the Internet. The jump from a text-based interface to a graphical interface would capture the world's imagination on presenting information to the masses.

Content-owners can now store information in a series of files on a server, with end-users accessing that data at their convenience. In the earlier days of the Internet, the server-side information was stored as simple static text files and images, meaning that files were only changed when someone manually made a change and uploaded the new file to the server. As a result, most Web sites did not change very often. With the introduction of the Common Gateway Interface (CGI), Web sites could now use programs in place of static files to dynamically create on-demand content that was unique for every user.

Anyone wanting to participate on the graphical Internet requires a client-side program, called a *Web browser*. This program is installed onto a local workstation, and requires an outgoing connection to the public Internet. The browser establishes a communication link through the network to a server-side counterpart, called a *Web server*, submits a request, and waits for a response. It is the server's job to interpret the request being made, assess the requester's credentials, open the file or execute a program using CGI, and transmit the results back to the user. Once the browser receives the data, it must decode the transfer and render a graphical representation of the text and images so that the user can interpret the information.

Many Web browser programs are freely available for download, depending on the user's choice of operating system. Popular options include Mozilla Firefox, Google Chrome, Microsoft Internet Explorer, and Apple Safari. For the content-owner, a Web site is delivered to the user's browser by various Web server programs, like Apache HTTP Server or Microsoft Internet Information Service (IIS). Again, the options available depend on the choice of operating system on the server.

The program that utilizes CGI typically runs on the same computer as the Web server. There are multiple languages available today that can interact with CGI, including PHP, Java, and even C and C++, but this book focuses on the Practical Extraction and Report Language, more commonly known as Perl.

The Apache HTTP Server

The Apache HTTP Server, widely viewed today as the *de facto* Web server for Unix and Windows platforms, handles more than 90 percent of World Wide Web traffic.

A History of Apache

First released in 1995, Apache evolved from the remains of the now defunct NCSA HTTPd program, which was the first Web server created to support the Hypertext Transfer Protocol, or HTTP.

Early Web servers were only capable of relaying static content directly from files stored on the Web server's hard drive. Eventually, the CGI protocol was standardized and support was added to Apache.

Versions of Apache

Because Apache has been in development for many years, new versions are constantly being released as *major version milestones*. The latest major release, Apache 2.2, supports a wide range of configuration features, performance enhancements, and third-party modules.

Earlier releases, such as Apache 2.0 and 1.3, are still available and supported. However, you should only consider using an older version if you have a specific reason to do so. If you are just starting out with Apache, use the latest stable release available, Apache 2.2.

The End-User Experience

To provide a pleasurable and esthetically pleasing Web site, the Web site author composes HTML content that describes the site's text, images, and layout. Early Web browsers lacked many of today's client-side technologies that are used to create dynamic content, such as Flash, Java, and JavaScript. Instead, they relied on the Web server to provide the entire end-user experience.

Because today's Internet has many options for dynamic content, a Web site author may choose any type of technology that is available: client-side, server-side, or even both. Intelligently mixing technologies that complement each other can create a memorable site that your users will want to visit again.

The Perl Programming Language

Following its own development path, Perl matured independently from Apache as a multi-purpose scripting language for Unix. The Perl programming language uses a syntax structure that is very similar to C in design, yet free and malleable in implementation. The Perl language is classified as a *third-generation*, or *high-level*, programming language; the programmer does not need to worry about complex memory allocation or architecture-specific, low-level CPU interaction. The program file source code, or *script*, is scanned and interpreted, not compiled, by the Perl interpreter at run time.

A History of Perl

When Perl was originally developed, it was not intended to help Web servers deliver dynamic content. Perl 1.0 was released in 1987 as a tool to read, print, sort, report, and interpret large amounts of data efficiently; it quickly became a useful tool for programmers and system administrators. Today's generation, Perl 5, is widely viewed as the most common and most stable version available. The next generation of the language, Perl 6, is currently under development and available as an experimental release.

Versions of Perl

Like Apache, Perl has experienced several major release milestones. The latest stable release, Perl 5.12, is a staple program on virtually all recent releases of Unix-based operating system distributions.

The latest experimental release of Perl 6 introduces major changes to Perl 5 syntax and internals. Because both generations will remain in active development, there is no need to switch your programming focus to Perl 6 after you learn Perl 5, however converting a script is not complicated.

The New, Dynamic Internet

By 1993, Perl was being used in tandem with Web servers to supply content over the Web by executing a program. With the new CGI protocol facilitating program execution on a Web server, Web sites could now provide the means to display dynamically changing content to Internet users.

Shortly after CGI was adopted, Web site authors were creating programs using Perl to support more complex features online. It was now possible to automate Web site features that changed very often, such as news reports, stock quotes, and sports scores, all of which previously required a human to sit at a computer and update files on a server.

Even more enhanced real-time features were developed, such as user authentication, data validation, and database access. This allowed Web sites to produce larger portals that were designed to only allow registered users access to secure information.

Introducing the Common Gateway Interface

The Common Gateway Interface, or CGI, is a protocol used by the Web server to communicate with other programs that are stored locally on the Web server. These programs use CGI to identify unique information about the user's session. When a user directs her Web browser to your Web site, you can instruct the Web server's CGI *handler* to execute a custom program, like a Perl script, to generate dynamic HTML code, and relay it back to the user's browser.

The CGI acronym has many meanings. More commonly it refers to the Computer Generated Imagery created for television and movies. However, in the context of the Internet Web browser, server, and this book, CGI only refers to this communication protocol.

The output of a typical CGI program is most often HTML code, but it can be of any file type. See the section, "Understanding CGI from the Web Browser's Point of View," for a description of how this works.

CGI Process Flow

When a user requests a Web page, the Web server decides whether or not to launch a CGI process based upon the filename the user requests. If he requests a static file, such as `index.html` or `report.txt`, the server relays the file back to the user as-is. If he requests a dynamic Perl script, such as `index.pl` or `report.pl`, the server launches the CGI process, executes the Perl interpreter, and relays the program's output back to the user.

HTTP Headers

Regardless of what the user is requesting or receiving, all communication traffic handled by the Web server is prefixed by a series of *HTTP headers*. These headers supply information that is vital for CGI, but they are only visible to the Web browser, server, and CGI programming languages like Perl. The user never sees these headers when the browser renders the Web page.

HTTP Request Headers

The Web browser sends HTTP request headers to the Web server on every page request. This tells the server which URL is being requested, what languages and encryption protocols are supported, any established cookies, and any submitted content from the user through an HTML form.

CGI scripts have access to these request headers through the server's environment variables. Perl CGI scripts interpret environment variables by way of a special variable called `%ENV`.

HTTP Response Headers

The Web server sends HTTP response headers back to the Web browser, followed by the requested content. This informs the browser on how much data content to expect, how it is formatted, and any new cookies that the browser must store.

CGI scripts have limited control of the HTTP response headers. To conform to HTTP standards, the Web server provides the majority of response headers automatically but still expects the CGI script to provide the *content-type* HTTP response header. This allows the CGI script to forewarn the Web browser, advising whether the output data it is sending is HTML code, a JPEG image, or a downloadable Zip file.

HTML Forms

In order to collect information from the user, data is submitted through an HTML form. You can use this form to collect any type of information, and bind the user's typed answer to a specific identifier. When the user clicks the Submit button, a new URL is requested and the data is relayed to the server.

On the server-side, you need to configure a CGI script to collect the information being submitted and process it. You must configure the HTML form to use a specific method to encode and relay its information to the server. The two most popular methods are GET and POST.

GET Method

The GET method appends the submitted content onto the end of the URL receiving the request. Due to the limited length of URLs in some browsers, which only support 256 characters, not much data can be submitted with GET. Also, the actual data is visible in the URL's Address bar when the new page loads.

POST Method

The POST method sends the submitted content immediately after the HTTP request headers. This allows for more data to be sent than GET, and it is hidden from the browser's Address bar. The main difference of the POST method is that bookmarks and page reloads may not work as the user expects; for example, the actual submitted information might not be maintained if the user refreshes the bookmarked URL at a later date.

Common Environment Variables

Several environment variables are available to you that are passed in the context of CGI. The HTTP request headers and the Web browser populate most of these variables, but the Web server also provides some of them. You can use them within a Perl CGI script to control the interaction with the user within the CGI protocol.

VARIABLE NAME	DESCRIPTION
SERVER_NAME	The server name responding, according to the browser
REQUEST_METHOD	The method of the request (GET or POST)
HTTP_USER_AGENT	A string identifying the user's browser
HTTP_COOKIE	A string of active cookies relative to the user's session
QUERY_STRING	Any additional data passed after the question-mark (?) in the URL for GET requests
SCRIPT_NAME	The script being executed, from the perspective of the server
REQUEST_URI	The script being executed, from the perspective of the browser
REQUEST_ADDR	The user's IP address that originated the request
SERVER_ADDR	The server's IP address that received the request

Additional environment variables may be available; this depends on the type of the request, the content, the Web server, the Web browser, and any other technologies such as secure sockets layer (SSL) encryption, or server-side includes (SSI).

Understanding CGI from the End-User's Point of View

When a user directs her Web browser to a particular page, she may never know if she is requesting a dynamic CGI program or static file, or have any idea about the CGI programming language being used on the server. Even if she does recognize that the page is dynamically generated, it is nearly impossible to identify what program is being used behind the CGI interface on the Web server.

Any astute user who has an idea about CGI development and Web servers may identify several clues from a Web site, thus identifying what software it runs. Even as you start generating CGI pages, you may start to notice some of these subtle clues on other Web sites and infer what they use to generate and display their content and services.

The problem with these "subtle clues" is that malicious users may be able to take advantage of your Web site, after identifying what software it runs, and leverage any number of known security attacks against your server.

Ultimately, these types of attacks are fairly easy to circumvent. See Chapter 22 for simple tips on preventing common attacks.

The HTML Interface

When most users visit your Web site, they only experience and interact with it through their Web browser's window. By default, this is all most users can see; however, users who are interested in how your Web site is constructed can still access the actual HTML code and syntax used.

Accessing the HTML Source Code

The Web site's HTML code is the only source-level content the user can access directly. Most Web browsers support an option to view the page's source code by simply right-clicking on the page and selecting View Source.

There is nothing you can do to prevent this. Regardless of whether a page is static or dynamic, the user can always view the HTML content.

Naturally, if a CGI program generated this content, there should be no indication of what program you used, or the contents of the CGI program's source code.

Accessing the CGI Source Code

When you properly configure a CGI protocol on the Web server, any requests by the user to access a CGI program by its URL are met with the program's output, not the program's file content.

In the case of a Perl script, a user may access the URL as `http://servername/cgi-bin/search.pl`. The CGI handler knows that any files in the `cgi-bin` directory should be executed, and not read. If your Perl script exists in another directory, one that does not have the CGI handler enabled, then the full source is visible.

For this reason, you need to properly secure your server hardware, and ensure that only appropriate people have access to the Web site source code on the server. In the previous example, if someone were to carelessly copy `search.pl` into another directory that lacks a CGI handler, all sensitive data within would be exposed online.

Prompting for User-Submitted Data

A CGI program needs to be developed that accepts any user-submitted data provided by an HTML form. When the browser first displays the form to the user, the form defines which URL should receive the data, and the method to use. It is that URL that the browser goes to when the users clicks the Submit button. The CGI program responding to that URL must collect the data, process it, and display an appropriate message back to the user.

Building an HTML Form

Most HTML forms follow the same structure. A form element introduces the form and surrounds several input elements, which collect data in the browser. The following is just an introduction to basic HTML forms. For more information, consult the W3C HTML specification.

Start a New Form

An HTML form always begins with `<form method=method action=url>`. The `method` is either GET or POST, and the action `url` should be a CGI script that opens when the user submits the form.

Single-Line Text Input

You can use the HTML element `<input type=text name=text>` to create a single-line text input field. The `name` attribute is used later by the CGI script as a key to the value provided by the user. You can use an additional attribute, `value=text`, to pre-populate the form with a default value in the field.

Multi-Line Text Input

To handle multiple lines of input from the user, use `<textarea name=text></textarea>`. The `name` attribute works the same way as the single-line input, but any default text should instead be defined between the opening and closing `textarea` elements. Additional attributes such as `rows=num` and `cols=num` can control the dimensions of the multi-line text input box.

Hidden Text Input

Use the HTML tag `<input type=hidden name=text value=text>` to pass additional information in the HTML form, but not allow the user to see it or change it directly. This can be useful if one CGI program originally generated the HTML form, and it needs to send additional data to another CGI program that will process the form.

Submit Buttons

All forms should have some sort of Submit button. This is what the user clicks to send the form to the Web server. You can often use `<input type=submit>`, but a button with the literal text, "Submit Query," displays. To change the button text, add the attribute `value=text`.

It may look weird in that this is the only input element that may have a `value` attribute, but not a `name` attribute. If you include a `name`, then the CGI program will receive the button's value as an input field. This can be particularly useful if you have multiple buttons on the same form, each with different processing functionality.

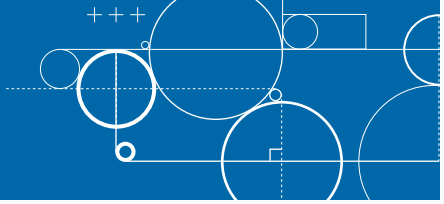
Finish the Form

Complete the HTML form with a closing `</form>` tag. This instructs the browser that the form is complete.

Viewing Data Returned by the Server

Once the data has been populated by the user, and submitted to a CGI program, the program should display something useful or intelligent to the user to indicate the results of their request. This may be as simple as displaying a "Thank You" message, maybe even stating that all information was received correctly. Or, if the user failed to correctly populate a field, the CGI program needs to inform the user which field was incorrect and why, and re-display the form. It is always good etiquette to have your CGI program pre-populate the fields that were correctly submitted with the original value, and to highlight the fields that were incomplete or incorrect.

Understanding CGI from the Web Browser's Point of View



The Web browser actually receives a lot more information from the Web server than it displays. The user is completely unaware of the subtle interactions between the browser and server, including various CGI communications. As a dynamic Web site author, you need to be aware of what is going on here so that you can better interact with your user's browser and provide a dynamic Web site experience.

Environment Variables

All CGI programs have access to several environment variables related to each Web page request. The Web server provides some of these variables, while the Web browser provides others.

Apache comes with a useful CGI script called `printenv.pl`, which you can use to see all environment variables that are in use, and to validate that CGI program execution is working correctly. To enable this script, you must first enable the CGI handler in Apache; see Chapter 10 for more information.

The Web browser provides the CGI with several environment variables including the following: the current URL (`HTTP_REQUEST_URI`), the referring URL (`HTTP_REFERER`), the browser's language (`HTTP_ACCEPT_LANGUAGE`), and the browser's software (`HTTP_USER_AGENT`). By reading these environment variables, your CGI program knows what URL the user is requesting, what URL he is coming from, what languages are supported, and the browsing software version, respectively.

Cookies

Cookies are portions of data that a Web server or CGI program can assign to a user's Web browser, allowing it to "remember" a user from an earlier Web site visit. When a CGI program wants to remember a user, it sends an initial cookie to that user's browser using the HTTP response headers. This could include Web site preference information such as the user's preferred language, or a uniquely generated authentication token.

The Web browser's job is to accept and store the newly assigned cookie. The Web browser does not care about the cookie's content, only that it must relay that same cookie back to the Web server on any subsequent Web page request using the HTTP request headers.

If that same CGI program receives its cookie back again, it knows that this user has visited before, and retrieves the data that it asked the browser to store. It is possible for a user to configure her Web browser to reject new cookies, or manually delete existing cookies. Doing so makes it impossible for the CGI program to remember the user, forcing it to treat the user like a first-time visitor.

MIME Types

All content delivered by Web servers is preceded with a special Multipurpose Internet Mail Extension (MIME) type header. This introduces the HTTP content coming from the Web server to the user's browser. As the name implies, MIME originated in the realm of e-mail, allowing for multiple message content blocks to be bundled into a single e-mail message. This allows the client to choose how to display the content by announcing its content type. The HTTP protocol uses a subset of MIME and requires all content delivered by a Web server to specify an appropriate *content-type* HTTP response header called an *Internet Media Type*.

INTERNET MEDIA TYPE / MIME TYPE	DESCRIPTION	COMMON FILE EXTENSIONS
`text/plain`	A plain-text file with no special formatting	.txt
`text/html`	An HTML-formatted Web page	.html .htm
`image/jpeg`	An image saved in the JPEG format	.jpeg .jpg
`application/zip`	A compressed ZIP archive	.zip

MIME Types *(continued)*

At first glance, MIME types in HTTP headers seem redundant; the Web browser should be able to identify the file type based upon its extension. When the user requests the address `http://mysite.com/index.html`, she is obviously viewing an HTML file as the Web page. However, if the user requests a CGI address such as `http://mysite.com/index.pl`, the Web browser does not know what format `.pl` files represent, and neither does the Web server. Instead, the Web server blindly executes the program, relaying all output back to the browser. The CGI program provides the content-type response header in its output, and the Web server relays this to the browser. So, if the CGI program announces content-type `text/html`, then the browser knows to render the CGI's output as an HTML page.

Processing User-Submitted Data

When an HTML form displays on the browser, the user is prompted to populate its fields and click a button to submit the data to the server. Each field has an identifier that is used as a lookup to match the field's value. The Web browser's role is to collect the information from the fields and encode them in a way that can be transmitted safely to the Web server.

Encoding the HTML Form's Values

The data being sent is encapsulated within a format that uses special characters to separate fields and values. If the user happens to type in a character that is sensitive to this format, a macro must replace that character so that it does not interfere with the expected formatting.

Fortunately, all alphanumeric characters are generally safe as-is, but some non-alphanumeric characters must be converted into their *percent-hexadecimal-ASCII* format by the Web browser. For example, the equals sign (=) is represented in ASCII as value 61, or in hexadecimal as 3D. If a user types an equals sign into an HTML form, the browser encodes the character as `%3D`.

Some of the characters that require conversion include the equals sign, plus sign (`%2B`), carriage return (`%0D`), line feed (`%0A`), question mark (`%3F`), and ampersand (`%26`).

The only exception to this rule is the space character. While `%20` is perfectly legal, Web browsers often convert it into a literal plus sign (+).

Receiving Data from the Web Server

The Web server never handles form data directly. Instead, a CGI program is responsible for processing the data. The program simply reverses the process performed by the Web browser: it splits the `name=value` pairs by the ampersand, and then decodes all of the *percent-hexadecimal-ASCII* values back into their original character values.

Sending Data to the Web Server

Before the data is sent to the Web server, the Web browser serializes the HTML form's fields into a string of `name=value` pairs, joining each pair with an ampersand (`&`). Because each `value` has had any sensitive non-alphanumeric characters encoded, these characters cannot affect the overlying structure of each pair, and the user's original value is maintained.

FORM FIELD NAME	USER VALUE	ENCODED STRING
name	John Smith	name=John+Smith&age=40&children=Chris+%26+Jason
age	40	
children	Chris & Jason	

If the HTML form specifies the GET method to send the data, the URL is appended with a question mark (?) followed by the encoded string of names and values.

If the HTML form specifies the POST method, the URL is not appended. Instead, the encoded string is sent after the standard HTTP request headers.

Understanding CGI from the Web Server's Point of View

The Web server's role is to facilitate the flow of information between the Web browser and the online content hosted by the server. This may require the Web server to execute a CGI program to provide the dynamic features of a Web site. Fortunately, because the CGI protocol has been standard for years, it does not matter which Web server or Web browser is used.

This section, in fact this entire book, uses Apache as the example Web server. If you compare it to another Web server such as Microsoft Internet Information Server (IIS), you will see that the majority of the concepts revolving around CGI are basically the same. The only real difference is the implementation and configuration of the actual Web server.

The Web server's role is basically transparent to the Web browser and CGI program. Its sole purpose is to forward request information being sent from the browser to the CGI program, and to relay response information from the CGI program back to the browser.

Receiving Data from the Browser

When the user goes to a Web site, his browser generates an HTTP GET or POST request and sends it to the site's Web server. This request consists of the specific site URL and the user's current HTTP environment settings, along with any cookies, secure socket layer (SSL) encryption status, and any other relevant information.

This information is bundled up into an HTTP request message and sent over the Internet. The Web server receives the HTTP request, reads the information that it deems relevant, and identifies if either a static file or a CGI program is needed to complete the request. So, if the user requests a static HTML file, then a CGI program does not run. The contents of the static file are relayed back to the user, encapsulated in an HTTP response message.

If the user requests a file that identifies itself as a CGI program, then Apache recognizes this, locates the program on the Web server's hard drive, and executes it appropriately. The program's output is sent back to the Web server, which it bundles with a similar HTTP response message.

Executing a CGI Program

A CGI program can be either a compiled binary or a standalone script. The Web server needs to know how to execute the program so that the intended output is sent back to the user.

In order for the Web server to identify if a CGI program is required, the server must have an understanding on the various ways a CGI program can be called. For example, it could be called directly, by the user specifying a Perl script in the Web browser's URL, or indirectly, hidden within an SHTML file using *server-side includes*. Once the CGI program is identified and located, it is executed.

Executing a CGI Program *(continued)*

Forwarding Data to the CGI Program

The Web server relays the CGI data to the CGI program in multiple ways. Regardless of the HTTP method used, the program's environment settings are populated with CGI data. Also, if the HTTP POST method is used, additional information can be found on standard-input.

It is the CGI program's responsibility to interpret, parse, and analyze the CGI data into usable segments. In the case of a Perl CGI script, it should relay this information to the programmer in a way that is easily accessible.

The information being sent to the CGI program includes any request data, cookies, HTML form fields and values, the user's environment settings, the Web server's environment settings, and any other information relevant to the request.

Receiving Data from the CGI Program

Once the CGI program is launched and performs its function, it sends its generated output back to the Web server by way of standard-output. In other words, the CGI program simply *prints* the output back to the Web server.

Prior to sending any of the actual data, though, the first line of the CGI program's output must define a content-type MIME header. Because a CGI program can technically output any type of content, be it an HTML-formatted Web page, a JPEG image, or a Zip file, it must introduce the content type to the Web browser. This is required because the CGI program cannot output the extension to the actual file format, such as HTML, JPEG, or XML. The extension simply does not exist in the HTTP response message.

By providing the content-type, the Web server knows the formatting of the CGI program's output, and can relay this information back to the Web browser, which then handles it appropriately given the content. Ultimately, the Web server does not care about the format of the data it receives from the CGI program. Its sole purpose is to send that data back to the user's Web browser.

Sending Data to the End User

After the initial request for the Web page, the user sits and waits for his Web browser to render the page. The Web browser waits for the Web server to return the results of the request. The Web server in turn waits for the CGI program to return its output. All this time, the CGI program may be using the server's CPU processing power, calculating some complex task. Only when the CGI program finishes does the process unravel: the program's output is sent to the Web server, then to the Web browser, and then to the user.

As data flows back to the user, it changes slightly with each step. The Web server actually appends additional HTTP response header fields to what it receives from the CGI program, completing the HTTP response message.

Naturally, problems can happen in this process. Most often, it may be due to a problem in the actual CGI program's execution. If the program crashes, then no output is sent back to Apache, which in turn sends a generic error message back to the user, something like "Error 500: Internal server error."

Or, if the CGI program takes too long to respond, Apache simply gives up waiting. Instead, it sends a timeout error back to the user, perhaps with some bundled text asking the user to try again later.

Regardless of the response from the CGI program, Apache needs to send something back to the user's Web browser as its HTTP response message. Apache constructs the response data, based upon conditions set out by the request. It tacks on the data received from the CGI program if applicable, or the contents of the file requested, sending everything back to the requesting user's Web browser over the Internet.

Understanding CGI from the CGI Program's Point of View

A Web site's author may choose to build her site with HTML content split into several individual files, such as `index.html` or `aboutus.html`. Static files are relatively easy to construct, but makes it difficult to provide any type of dynamic content. Instead, the author may create one or more CGI programs to dynamically generate each Web page.

The CGI approach has the advantage of controlling what information displays, and making it unique to the end-user given his particular session. This may be as simple as including the current date and time in the top-right corner of the Web site, or as complex as an e-commerce store with a shopping cart and a credit-card processing checkout.

The CGI handler provides the conduit between a CGI program, such as a Perl script, and the user. The program must collect the incoming information provided by the Web server through the CGI handler including the user's environment, cookies, and session. The program must analyze this information, process it, and generate output from it, like a dynamic Web page.

CGI Handler Executes the Perl Interpreter

For every user who requests a Web page that is dynamically generated, the Apache CGI handler must launch a new instance of the Perl interpreter into memory to process the request. The CGI handler also instructs the Perl interpreter which Perl script it needs to execute, and finally forwards the incoming CGI data to the script's run-time input.

Naturally, if the user references a Perl script filename in the URL, Apache does not necessarily know that it is executing a Perl script, or a binary file written in machine language. Because a script file is nothing more than text that follows a specific syntax, it cannot be executed directly by the operating system like a compiled binary.

In the Unix world, when a program file is launched on the command line, the operating system relies the *system shell* to identify if the file is binary and can be executed directly, or if it requires a helper program. To do this, the shell reads the first line of the file, looking for a *shebang* (`#!`) interpreter directive. This is why all Perl scripts begin with the following line:

```
#!/usr/bin/perl
```

The shell sees this and knows that it must execute `/usr/bin/perl` first, as this is the program that can properly interpret the contents of the script file.

In the Windows world, the same basic thing happens, except the `explorer.exe` "shell" does not look for an interpreter directive. Instead, it recognizes Perl scripts by way of a `.pl` extension. It knows that `.pl` files first require the program `C:\Perl\bin\perl.exe`. When executing CGI scripts on Windows, Apache does not use `explorer.exe`, but instead assumes the role of a Unix system shell. Apache opens the file being referenced and looks for a shebang interpreter directive, which must describe the location of the Perl interpreter binary installed on Windows:

```
#!C:/Perl/bin/perl.exe
```

Reading Data Sent from the CGI Handler

Once the Perl interpreter is running, and it has parsed the Perl script and executed it, the first thing the script should do is read introductory information about the HTTP request session from the CGI handler.

The CGI handler supplies most of its information to Perl by way of environment variables into the Perl interpreter session. The CGI handler provides the URL requested, the user's IP address, the Web browser name and version, all cookies related to the Web site, and so on. All of this information is gleaned from the Perl session's global environment variable.

Sending Data Back to the CGI Handler

It is the Perl script's job to enhance the user's Web-browsing experience. Usually this happens by way of dynamically generated HTML, but it could be any type of formatted data, composed by the Perl script's logic. The Perl script simply "prints" this information on its standard-output handle, with Apache listening intently to this handle while the Perl script is running.

However, prior to printing any HTML code, the Perl script must first print any outgoing HTTP headers. Apache relays these HTTP headers back to the user's Web browser in the HTTP response message. In essence, these headers allow the Perl script to communicate directly to the user's browser, within the guidelines of the HTTP specifications.

At a minimum, the Perl script must print the content-type MIME header. This is absolutely required, and enforced by Apache. If it is missing, Apache sends a generic error code back to the user, regardless of the Perl script's output.

Additional headers may also be provided to further fine-tune the HTTP session. Cookies, for example, are also created by way of the outgoing HTTP headers. After the Perl script defines any outgoing headers, it must print one blank line, followed by the actual generated content.

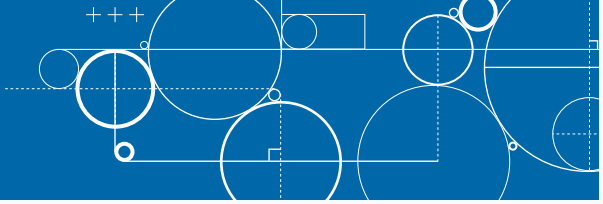
Shutting Down

Once the Perl script has finished, and its output has been sent back to Apache for relaying to the user's Web browser, the Perl interpreter closes its connection to the CGI handler and shuts down.

Normally, Apache waits until the Perl interpreter has exited before sending *any* data back to the user. Apache actually buffers the Perl script's output and only sends it once the CGI program is complete. Most of the time this is not an issue; however, if you have a Perl script that takes several seconds, or even minutes, to run, the user will be waiting for some time before the browser displays the results of the Web page request. If the delay is unavoidable, you can at least instruct Apache to provide *some* content back to the user, such as a "Please wait..." message, by flushing the output buffer in the middle of program execution.

Given enough traffic on a dynamic Web site, constantly starting up and shutting down the same group of CGI programs can be rather demanding on the CPU. In the case of a Perl script, the Perl interpreter is the actual binary program that is being launched on each request. For every single request, the interpreter parses the script into machine code, executes it, and exits. It is possible to configure Apache to actually embed Perl directly into itself. This means that the Perl interpreter is persistently running in RAM within every process of Apache, so you can avoid the actual startup and shutdown of Perl; the interpreter simply remains idle until a CGI request is received. This feature is provided by an extension module for Apache called *mod_perl*. See Chapter 23 for more information on implementing mod_perl on your Web site.

Compare Perl to Other CGI Languages



Although this book focuses on using Perl as a CGI language, Perl is certainly not your only option. Technically speaking, any programming language that can access environment variables, read incoming data, and write outgoing data will support the Apache CGI handler.

Not all programming languages are created equal. Some have libraries specially designed to access CGI data, while others have modules that make constructing HTML very easy. Some have a language syntax that is very easy to remember, and others have third-party modules that make complex calculations very simple.

You may find that as you experiment with different languages, some work better than others. There is no rule that says you must select one specific language for an entire Web site. Feel free to mix and match languages and technologies and see for yourself what works best.

PHP

PHP is an interpretive language that was designed to specialize in CGI development. While influenced by other languages such as C, JavaScript, and even Perl, PHP makes building dynamic Web sites very easy.

PHP's strength comes mainly from its ability to embed itself directly within an HTML file. Static Web site content is represented as standard HTML in a PHP script. Content that is dynamic is written as PHP code, contained within special `<? ... ?>` tags.

A lot of the mundane CGI handler interactions are automated by PHP. For example, the PHP interpreter takes care of tasks such as parsing HTML forms for values, and accessing details from the CGI environment. At a minimum, Perl requires you to either import its CGI library, or to produce 15 to 20 lines of code by hand, to accomplish the same thing.

PHP does have its flaws. For example, its ability to parse, sort, and organize raw data is not as efficient as that of Perl. Also, its library of third-party modules is not as robust or mature as the Perl CPAN repository. For simple CGI programming, however, many developers find PHP to be more than adequate.

Active Server Pages

Active Server Pages, or ASP, is a Microsoft framework for dynamically executing server-side code in-between standard HTML. First released in 1996, ASP refers to the engine on the Web server; the underlying language is actually VBScript.

At first glance, ASP-formatted files look like PHP files, using the tags `<% ... %>` instead of PHP's `<? ... ?>`. However, the feature set available to VBScript is much more robust than PHP, especially with the ASP engine providing a series of object handles that you can use to manage the application, as well as request, response, server, and session classes.

The VBScript language is limited to running on Microsoft Windows-based hosts; however, it is not limited to IIS. A third-party module is available for making ASP and VBScript Web sites run under Apache on Windows.

VBScript supports many standard functions found in most programming languages, such as built-in file, date, math, string, and binary methods. ActiveX objects provide additional functionality, but design problems in ActiveX and VBScript have led to problems with malicious third-party code, and several security advisories and patches.

ASP.NET replaced ASP in 2002. The shift to ASP.NET meant that it is no longer an interpretive language; code must now be compiled natively within the .NET framework. While this does mean more overhead in terms of memory utilization, the code is generally faster and more responsive than the original ASP. Many ASP.NET developers have chosen to continue using *classic* ASP, especially to develop simple Web sites.

Ruby on Rails

Ruby on Rails is an application framework for developing dynamic Web sites using the Ruby programming language. Ruby is an interpretive language that has been around for years, and is influenced by other languages such as Perl and Python. However, Ruby on Rails is a fairly recent framework that is gaining in popularity. It makes key assumptions about common features and functionality, and strives to maximize code re-use.

Directly comparing Ruby on Rails to Perl is not exactly fair. Ruby on Rails enforces a specific architecture method to group the Web site development into interaction, display, and structure components. By forcing the programmer to separate her code into these three components, Ruby on Rails developers argue that Web sites can be built more efficiently and quickly. Perl CGI does not mandate that you design using this model, but you are welcome to if you prefer this type of organization.

Some developers of larger Web sites have criticized Ruby on Rails for not scaling efficiently, even opting to mix technologies such as Perl on more resource-intensive operations.

C, C++, C#, and .NET

Programs written in C, C++, C#, or .NET are compiled into binary-code, which is executed natively by the operating system. With these languages, there is no interpretive program involved. Instead, these programs need to be compiled once, and then executed by the Web server each time it receives a CGI request.

This method of developing CGI programs is very different from the other examples in this section; however, natively compiled programs do have their own unique strengths and weaknesses that you should consider.

One huge advantage to using one of these languages is that the program is already converted into a format that the CPU can quickly and efficiently execute. This means that larger programs can run much faster and are generally more responsive than interpretive-based languages.

Strictly speaking, using an interpreter means that your source code is re-compiled for every single CGI request that the Web server receives. In addition, the actual execution of the interpreter affects the server's memory and CPU resources.

The disadvantage to using a native language is that there is more work involved in creating and debugging a program, as compared to a similar program that you would write using an interpretive language. It is more tedious because the programmer must manually compile the binary object before installing it in a location where the Web server can access it. With an interpretive language, the compile-and-install steps are not applicable.

The programmer must also worry about lower-level memory management, and system-level hardware interaction — something that is taken care of automatically by a higher-level interpretive language such as Perl or PHP.

Ultimately, for established Web sites that are larger, busier, and make significant resource demands on server hardware, a natively compiled binary is a better long-term solution, provided that you can invest the extra effort into writing and maintaining the program. For a newer Web site with little traffic or resource demands, using an interpretive language such as Perl will help you develop the site more quickly.

Compare Apache to Other Web Servers

Just as many other languages have implementations of the CGI protocol, so do many Web servers. As of the March 2010 Netcraft survey of Web server software, Apache represented a 54 percent market share of all Web sites surveyed — over 112 million servers. Naturally, the software must be good if so many people are using it; however, some find it overly complicated and difficult to set up, especially with its text-based configuration files. Programs such as Microsoft Internet Information Server (IIS) have stolen market share from Apache partly because they are easier to configure and deploy.

Unless you run your own server hardware on the Internet, you may not have much of a choice regarding which Web server to run on your Web-hosting provider's servers. Apache is the standard on practically all Unix-based hosting providers. Windows-based hosting providers tend to only provide IIS as an option. However, it is strongly recommended that you install a Web server locally on your workstation for development and testing purposes.

When choosing a Web server, at a minimum you need something that supports CGI. Additional support for features such as server-side includes, secure socket layer encryption, virtual domain hosting, and basic/digest authentication are all certainly nice to have, but your specific application may not require them.

Internet Information Server

Microsoft first released IIS 1.0 in 1995 with Windows NT 3.51. Since then, the program has been steadily upgraded and improved with each new release of Windows. The most recent major version is IIS 7.5 for Windows Server 2008.

As of March 2010, about 24 percent, or roughly 50 million Web sites, use IIS. This is impressive because only Windows servers can deploy the program; however, IIS's level of adoption has bounced between 20 percent and 40 percent since late 1997. Since peaking in late 2007, its market share has dropped to a new four-year low.

IIS and Apache have been competing for market share for many years. In fact, the two programs have become so synonymous with their respective operating systems that many Web site developers have simply accepted either program as the default option — immediately after selecting a development platform. The key advantage Apache has over IIS, with respect to market share, is that Apache can run on either Windows or Unix servers, whereas IIS is limited to Windows-server platforms only.

Unlike Apache, IIS's license and source code are proprietary, and not freely available. Rather than charging for it, Microsoft has been shipping it out as a standard component of all Windows installs since Windows XP.

Strictly speaking, as a CGI Web server, IIS is fully compatible with Apache in terms of handling Perl or other interpretive languages. In other words, if you use IIS on your personal workstation to develop and test your Web site, and Apache on your Web-hosting provider's server to deploy it publically, your CGI scripts should be equally usable at both sites, regardless of any minor compatibility adjustments to the CGI source code.

Depending on whom you ask, several studies have been released that argue that Apache is better than IIS and vice versa when compared on similar hardware. The only real disadvantage to IIS may be its performance serving CGI scripts under high-volume conditions; after all, IIS is designed to run ASP, a competitor to the CGI standard. You can use a third-party program called FastCGI to address this CGI deficiency on IIS. FastCGI runs underneath IIS and acts as sub-server for all CGI traffic deferred by IIS. FastCGI is also available for Apache.