An Overview of the World Wide Web

No book written about Apache, the most widely used Web server software on the Internet today, would be complete without a discussion of the World Wide Web (WWW) itself—how it came into existence and how it works. A key part of mastering any complex software application is understanding the technology underlying that system, and the technology that underlies Apache is the World Wide Web. This chapter is an introductory overview of a vast topic. The chapter begins with a history of the World Wide Web, introducing the Apache Web server, and then moves through an explanation of how the Web works, with a short introductory tour of the inner workings of the HyperText Markup Language (HTML) and the HyperText Transfer Protocol (HTTP). We’ll look at important features of the HTTP/1.1 version of the protocol, and use three common tools to observe the protocols in action.
A Brief History of the Web

The World Wide Web (referred to throughout this book simply as the Web) is the result of years of evolutionary change. No one person or group can be credited with its creation. Indeed, it is unlikely that the original designers of the Web had notions as grand as the eventual reality of their accomplishment. Although based on the concept of embedded links between documents, called hypertext links, which has its beginnings in the mid-1940s, the Web is generally considered the idea of one man, Tim Berners-Lee.

In 1989, Berners-Lee submitted a proposal for a research project to CERN (Conseil Européen pour la Recherche Nucléaire) in Geneva, Switzerland. Berners-Lee’s proposal outlined a hypertext-based system that we would all recognize as today’s Web, but it didn’t discuss the technical foundation of that system, and didn’t address the need to develop network protocols to support the system. The paper basically proposed extending the HyperCard system that was available for the Apple Macintosh computer to a local area network–based system. The Web actually had a very humble beginning; the proposal did not, for example, foresee the expansion of the proposed system to global proportions.

The following year, 1990, a NeXT Cube workstation was purchased by CERN, and work began on the first graphical hypertext delivery system—the first Web browser. The CERN labs also distributed technical details that allowed developers to create their own Web servers. The first Web sites were initially set up as experimental, or “proof of principle” sites, mostly by academic and research institutions with the resources to develop them. Most of these were very simple servers, consisting of a few hundred lines of C code, based on source code obtained from CERN. In November 1992, the CERN list of “reasonably reliable servers” consisted of only 26 servers, at sites around the world.

All that changed in 1993. CERN was making available its own “reasonably reliable” server, with instructions on how to port and compile it to different types of hardware. In the United States, the National Center for Supercomputing Applications (NCSA), located at the University of Illinois at Urbana-Champaign, released NCSA Mosaic. The development and free distribution of the Mosaic browser was the catalyst that caused a sudden and sustained increase in the proliferation of Web servers on the Internet. NCSA also offered its own version of a Web server that was freely downloadable and relatively easy to install. The NCSA server was more widely adopted than the CERN server and led the way with the addition of new features.
By 1994, the most widely used Web server software in the world was NCSA httpd. NCSA also had the lead in the development of the HTTP client software with its Mosaic browser. The Web looked like it belonged to NCSA—a nonprofit organization, something that is almost inconceivable from our viewpoint a mere eight years later. Progress on the NCSA server project stalled when its developer, Rob McCool, left NCSA in mid-1994. Since the source code for the NCSA server was widely available, many developers were already working on improvements and bug fixes. This trend toward decentralized, uncoordinated development continued into 1995, the year in which the Apache server was born.

The Apache server was assembled and released in early 1995 as a set of patches to the NCSA httpd 1.3 Web server. The name Apache is derived directly from this beginning as a patchy server. Get it? Numerous individual programmers, loosely bound into a consortium initially called the Apache Group, contributed the original source code patches that made up the first Apache server. In the true spirit and style of what is best about the Internet and open-source development, they collaborated by sharing ideas, criticism, encouragement, and camaraderie via e-mail and Usenet newsgroups.

Less than a year after the Apache group was formed, the Apache server replaced NCSA httpd as the most-used Web server. There are several reasons why Apache was so rapidly accepted and widely installed so soon after its initial availability. First and foremost, Apache was functionally identical to, and administered exactly like, NCSA httpd. With virtually no alterations to the filesystem or configuration files, Apache could be used as a drop-in replacement for NCSA. And there were good reasons to do so: Apache was faster, it was more reliable, it enjoyed wide support—and it was cool. Apache administrators immediately became part of a development effort that was on the leading edge of a new technology that was changing every facet of computing. Later developments only increased the superiority of Apache over the noncommercial, freely distributed servers available at the time. The most significant change was probably its support for add-on modules; this was achieved by exposing the internal workings of the server engine to third-party programs through a set of Application Programming Interfaces (APIs). This allowed anyone, anywhere, to customize Apache to meet their own specific needs, but more important, it led to the development of modules that can be freely obtained and added to the server to extend its capabilities. Many of these modules have been adopted for inclusion as part of the Apache distribution, though the use of most is optional.

With the decline in use of the two most prominent HTTP servers of the mid-1990s (new development for both CERN and NCSA HTTP servers was completely abandoned), Apache grew to become the most widely used Web server software on the Internet. As shown on the timeline in Figure 1.1, it achieved this status in April 1996, according to the Netcraft Web Survey (www.netcraft.com), and has held this position continuously since.
Figure 1.1 A timeline of the Web and Apache

Today most competing servers are commercially developed and supported, predominately those from Netscape and Microsoft. The growth of commercial servers doesn’t reflect superiority over Apache as much as it points to the increasing use of the Web for commercial purposes. Many large companies, integrating their Web commerce engines with legacy databases and enterprise resource planning (ERP) systems, insist on using only commercial software. Traditionally, noncommercial software has been seen as experimental and not production-ready, incapable of performance and reliability equal to software developed for commercial resale. It’s also possible to sue a commercial software vendor, even if it is difficult to win such a lawsuit.
Apache and software servers like Apache that use Linux have slowly overcome corporate prejudice to make major inroads in changing industry perception of noncommercial open-source software. In September 2001, Apache received a tremendous vote of confidence from the Gartner Research Group when it released a report urging corporations to abandon the use of the most prevalent commercial Web server in favor of Apache, which it considered far more securable. As PCs did in the early 1980s, Apache and Linux continue to filter steadily into Fortune 1000 firms.

**The Apache License**

What does it mean to describe Apache as open-source software? Open-source software is often associated with the GNU Public License (GPL), mainly because the GPL is the licensing agreement used by Linux and many applications written for Linux. Apache, however, does not use the GPL and has a special license of its own. The Apache license, which can be retrieved from [http://httpd.apache.org/docs-2.0/LICENSE](http://httpd.apache.org/docs-2.0/LICENSE), doesn’t specifically prohibit any use of Apache and, unlike the GPL, does not require that modifications to Apache be made public. The Apache license exists mainly to limit the liability of the Apache Software Foundation (formerly known as the Apache Group) for damages or loss resulting from the use of Apache software. It also requires that the Foundation is properly credited in any commercial use of Apache or products based on the Apache server.

The essential thing to remember about the Apache license is that it expresses a copyright held by the Foundation for the Apache source code. Apache is not in the public domain. Control of the Apache source code remains ultimately in the hands of the Apache Software Foundation. Although it is possible that the Foundation could suddenly decide to pull the rug out from under hundreds of thousands of site administrators by requiring licensing or even prohibiting use of the software for commercial purposes, the likelihood of that happening is extremely remote. You’ll find more precedents of commercial software vendors suddenly deciding to change their corporate practices or policies to the detriment of their clients, deciding, for example, that it is no longer economically feasible to support particular software. How many times has a software support technician told you, “You need to buy an upgrade to fix that problem”? Using open-source software can often protect your company from the vagaries of software vendors in a ruthless marketplace.
How the Web Works

The rapid adoption of the Web can be largely attributed to the accessibility of its technology. From the very first, Web browsers have been freely distributed, and it is highly unlikely that use of the Web would have exploded in the mid-1990s had the situation been any different. The Web provides an interface to information that is simple and intuitive, providing links to millions of sites around the world. We now have access to vast repositories of information, and much of it is free for the taking.

Was it our relentless search for intellectual self-improvement that enticed most of us into downloading our first Web browser onto our PCs, or an insatiable need for information to help us do our jobs more efficiently? Hardly. The truth is that the character of the early Web sites had a lot to do with the Web’s instant popularity. What Web user wasn’t first attracted by the interesting new combinations of text and color graphics (not to mention the fact that the personal computer gave a sense of security that the boss didn’t know what we were really doing with our computers on company time)?

We were checking out something cool and always looking for something even cooler. It’s not surprising that time spent using the new information-distribution system became known by a simple nontechnical term: “browsing” the Web. The system of hyperlinks that allows one page to reference others, each of which leads to other (hopefully related) pages is what opened the Web to a vast, mostly nontechnical audience. The Web was an astounding success because, even on a 14-inch monitor with a pixel resolution that is a joke by today’s standards, nearly everyone said the same thing when they saw their first Web page: “Cool!” No competing scheme for exchanging information that ignored the “cool” factor stood a chance against the Web.

At the heart of the design of the Web is the concept of the hyperlink. The clickable links on a Web page can point to resources located anywhere in the world. The designers of the first hypertext information system started with this concept. For this concept to work on a major scale, three pieces of the Web had to be invented. First, there had to be a universally accepted method of uniquely defining each Web resource. This naming scheme is the Uniform Resource Locator (URL), described in the accompanying sidebar. The second piece was a scheme for formatting Web-delivered documents so that a named resource could become a clickable link in another document. This formatting scheme is the HyperText Markup Language (HTML). The third piece of the Web is some means to bring everything together into one huge information system. That piece of the puzzle is the network communication protocol that links any client workstation to any of millions of Web servers: the HyperText Transfer Protocol (HTTP).

A hyperlink embedded in an HTML-formatted page is only one way to use a URL, but it is the hyperlink that gave rise to the Web. If we had to resort to exchanging URLs by
writing them on napkins, there would be no Web. Most of us think of the Web in terms of visiting Web sites, but the mechanism is not one of going somewhere, it is one of retrieving a resource (usually a Web page) across a network using the unique identifier for the resource: its URL.

URLs can also be manually entered into a text box provided for that purpose in a Web browser or saved as a bookmark for later point-and-click retrieval. Most e-mail programs today allow URLs to be included in the message body so that the recipient can simply click them to retrieve the named resource. Some e-mail packages allow you to embed images in the message body using URLs. When the message is read, the image is retrieved separately; it could reside on any Internet server, not necessarily the sender’s machine.

What Is a URL?

A Uniform Resource Locator (or URL) is a means of identifying a resource that is accessible through the Internet. Although the distinction is academic, a URL is a special case of a Uniform Resource Identifier (URI) that is understood by Web servers. A URI is any string that uniquely identifies an Internet resource.

Each URL is composed of three parts: a mechanism (or protocol) for retrieving the resource, the hostname of a server that can provide the resource, and a name for the resource. The resource name is usually a filename preceded by a partial path, which in Apache is relative to the path defined as the DocumentRoot. Here’s an example of a URL:

http://www.apache.org/docs/misc/FAQ.html

This URL identifies a resource on a server whose Internet name is www.apache.org. The resource has the filename FAQ.html and probably resides in a directory named misc, which is a subdirectory of docs (a subdirectory of the directory the server knows as DocumentRoot) although as we’ll see later, there are ways to redirect requests to other parts of the filesystem. The URL also identifies HTTP as the protocol to be used to retrieve the files. The http:// protocol is so widely used that it is the default if nothing is entered for the mechanism. The only other common retrieval method you’re likely to see in a URL is ftp://, although your particular browser probably supports a few others, including news:// and gopher://.

A URL can also invoke a program such as a CGI script written in Perl, which might look like this:

http://jackal.hiwaay.net/cgi-bin/comments.cgi
It was the Web browser, with its ability to render attractive screens from HTML-formatted documents, that initially caught the eye of the public. Beneath the pretty graphical interface of the browser, however, the Web is an information-delivery system consisting of client and server software components that communicate over a network. These components communicate using HTTP. The following sections describe this client/server relationship and the HTTP protocol used to move Web data around the world. This provides an introduction to the subject of the book, Apache, which is the foremost implementation of the HTTP server component.

What Is a Web Server?

Essentially, a Web server is a software application that listens for client connections on a specific network port. When a connection is made, the Web server then waits for a request from the client application. The client is usually a Web browser, but it could also be a Web site indexing utility or perhaps an interactive telnet session. The resource request, usually a request to send the contents of a file stored on the server, is always phrased in some version of HTTP.

Although the Web server’s primary purpose is to distribute information from a central computer, modern Web servers perform other tasks as well. Before the file transfer, most modern Web servers send descriptive information about the requested resource, instructing the client how to interpret or format the resource. Many Web servers perform user authentication and data encryption to permit applications like online credit-card purchasing. Another common feature of Web servers is that they provide database access on behalf of the client, eliminating the need for the client to use a full-featured database client application. Apache provides all of these features.

The HyperText Transfer Protocol (HTTP)

The Web consists of all the Web servers on the Internet and the millions of client systems that are capable of establishing temporary connections to them. The essential glue that holds the Web together is the set of interoperability standards that permit these clients and servers to exchange information across the Internet. These well-defined standard methods of communicating across a network are called protocols. To understand the Web, it is important to understand the protocols that establish and define it.

What is a protocol? Traditionally, the word refers to the rules of social behavior followed by dignitaries and heads of states. In computer networking, the term also refers to rules of behavior—those that apply to the two sides of a network connection. In this sense, the HTTP protocol defines the behavior expected of the client (browser) and server components of an HTTP connection. A browser can be written only if it knows
what to expect from the servers it connects to, and that behavior is defined by the protocol specification (HTTP).

Generally, when an HTTP/1.1 server like Apache receives a request from a client browser, it will perform one of two actions. It will either respond to the request by sending a document (either static or dynamically generated by a program) or refuse to respond to the request, sending instead a numeric status code indicating why. If the numeric status code is in the range 300–399, it indicates to the browser that the server is redirecting the request to an alternate location.

A Web server cannot force a browser to retrieve a resource from an alternate location. It sends a status code showing that the server couldn’t respond to the browser’s request, along with a Location: directive indicating an alternate location for the resource. The browser is politely asked to redirect its request to this location. The important thing to keep in mind is that the server does not control the browser’s behavior, but simply suggests or requests a certain action. That’s the essence of a protocol, which is simply a codification of the acceptable (proper) and expected behavior of the components of a system.

The one protocol that all Web servers and browsers must support is HTTP. HTTP is actually not very complex as protocols go. The first version of HTTP (now referred to as version 0.9, or HTTP/0.9, although at the time there was no official versioning of the protocol) was extremely simple, designed only to transfer raw data across the Internet. The early Web servers that implemented this now-obsolete version of HTTP responded to simple requests like:

```
GET /welcome.html
```

Upon receiving this request, a server responded by sending a document stored in the file welcome.html, if it existed in the server’s defined DocumentRoot directory, or an error response if it did not.

Today’s Web servers still respond to HTTP/0.9 requests, but only the very oldest browsers in existence still form their requests in that manner. HTTP/0.9 was officially laid to rest in May 1996 with the release of Request for Comments (RFC) 1945 (“Hypertext Transfer Protocol—HTTP/1.0”), which formally defined HTTP version 1.0. The most important addition to the HTTP protocol in version 1.0 was the use of _headers_ that describe the data being transferred. It is these headers that instruct the browser how to treat the data. The most common header used on the Web is certainly this one:

```
Content-Type: text/html
```

This header instructs the browser to treat the data that follows it as text formatted using HTML. HTML formatting codes embedded in the text describe how the browser
will render the page. Most people think of HTML when they think of the Web. We’re all familiar with how an HTML document appears in a browser, with its tables, images, clickable buttons, and, most importantly, clickable links to other locations. The use of HTML is not limited to applications written for the Web. All of the most popular electronic mail clients in use today support the formatting of message bodies in HTML.

The important thing to remember is that the Web’s most commonly used formatting specification (HTML) and the network transfer protocol used by all Web servers and browsers (HTTP) are independent. Neither relies exclusively on the other or insists on its use. Of the two, HTTP is the specification most tightly associated with the Web and needs to be part of all World Wide Web server and browser software.

**NOTE** The operation of the Web is standardized by a number of documents called Requests for Comments (RFCs). While many of these are considered “informational” documents and have no status as standards or specifications, those that have been accepted by the Internet Engineering Task Force (IETF) are the accepted specifications that are used in the development of network applications. While RFCs are available from a number of sites, probably the best source is [www.rfc-editor.org](http://www.rfc-editor.org), which is funded by the Internet Society.

### New Features in HTTP/1.1

The current version of HTTP is version 1.1, which is described and defined by RFC 2616 (“Hypertext Transfer Protocol—HTTP/1.1”). The official date of this document is June 1999, but work on the specification has been ongoing for years, and features embodied in the specification slowly found their way into mainstream servers and clients during that time. Version 1.1 includes several important new features that have been requested for years. You can expect HTTP/1.1 to be fully supported in all versions of Apache starting with version 1.3.4.

Most of the changes to HTTP in version 1.1 were made to the way HTTP client and server programs communicate, and are designed primarily to enhance performance, especially using caching proxies (Chapter 12, “Proxying and Performance Tuning”). Most features of HTTP/1.1 operate almost unchanged from HTTP/1.0, but some of the changes are quite visible and important to the Web site administrator.

One of the features of HTTP/1.1, hostname identification, is a way for the server to determine which of several virtual hosts should receive the request. In Chapter 6, “Virtual Hosting,” we’ll see how this eliminates the need for Web site hosting services to reserve unique IP addresses for each virtual Web site on a single host server. Hostname identification was one of the most requested changes in HTTP/1.1.
HTTP/1.1 supports a feature called content negotiation, in which an exchange of new HTTP/1.1 headers allows the browser and server to negotiate a common set of settings. This is useful, for example, in cases where a Web server provides resources in several versions (called representations or variants). The content negotiation feature of HTTP/1.1 allows the browser to automatically indicate a preferred language for the requested resource, or perhaps an alternate format for a document like PDF or PostScript. Content negotiation is covered in Chapter 15, “Metainformation and Content Negotiation.”

Four new request methods, described in detail in the next section, were added to HTTP/1.1: OPTIONS, TRACE, DELETE, and PUT (Table 1.1). Ever encounter a Web site that lets you upload a file from your Web browser? Probably not. That feature isn’t seen more often because it requires that both server and browser support the PUT request method introduced in HTTP/1.1. (Actually, it is possible to upload files to a Web server using the POST method and CGI in earlier versions of HTTP, but HTTP/1.1 is the first to support two-way file transfers.) Few Web site administrators are willing to rely on a new feature that would exclude a significant number of potential customers who are using outdated browsers. Soon, most browsers in use on the Internet will support HTTP/1.1. Increased server support for HTTP/1.1 has been introduced in each new release of Apache since work began on the specification several years ago.

Regardless of the browser you choose, ensuring compatibility with the very latest release has become almost essential. Although I had to have my fingers pried from Netscape Navigator 3.04, which I used for several years, I will never again use a Web browser even a few revisions old, in order to have the latest features of most modern browsers. On computers running Microsoft Windows operating systems, the choice of Web browser has been narrowed to Microsoft’s own Internet Explorer (IE), by far the most popular browser; no competitor comes close to the popularity it enjoys. In fact, many of the Web browser screenshots in this book were made from the IE browser.

The real threat to the dominance of the Internet Explorer browser is from the increasing number of people using Linux as a desktop operating system. IE isn’t available for Linux and probably never will be, but open-source browsers like Konqueror (popular with users of the KDE desktop) and Mozilla (more commonly used with the Gnome desktop) are excellent.

**HTTP Request Methods**

All HTTP requests begin with a header that specifies the request method. The most common method is the one used to request a resource from the server. This is the GET method. It is used to retrieve a resource from a Web server whenever you type a URL in
the text box of your browser. The GET method is also used to invoke scripts, and it has provision for parameters to be appended to the method to allow data to be sent to the server. The primary use of the GET method, therefore, is resource retrieval.

Table 1.1 shows the eight methods supported by HTTP/1.1. With the exception of the first three, GET, HEAD, and POST, all of these methods were added in HTTP/1.1 and were not part of HTTP/1.0 and earlier. Not all of these are retrieval methods; the PUT and POST methods are used to send data from the client to the server.

### Table 1.1 HTTP/1.1 Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieves the resource identified in the request URL.</td>
</tr>
<tr>
<td>HEAD</td>
<td>Identical to GET except that the server does not return a message body to the client. Essentially, this returns only the HTTP header information.</td>
</tr>
<tr>
<td>POST</td>
<td>Instructs the server to receive information from the client; used most often to receive information entered into fields of a Web form.</td>
</tr>
<tr>
<td>PUT</td>
<td>Allows the client to send the resource identified in the request URL to the server. The server, if it will accept the PUT, opens a file into which it saves the information it receives from the client.</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Used to request information about the communication options provided by the server. This allows the client to negotiate a suitable set of communication parameters with the server.</td>
</tr>
<tr>
<td>TRACE</td>
<td>Initiates a loopback of the request message for testing purposes, allowing the client to see exactly what is being seen by the server.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Requests that the server delete the resource identified in the request URL.</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Instructs a Web proxy to tunnel a connection from the client to the server, rather than proxying the request.</td>
</tr>
</tbody>
</table>

**Observing the HTTP Protocol in Action**

The quickest path to understanding how a basic HTTP retrieval works is to connect directly to a Web server and enter the HTTP request manually. Observing the protocol interactions between a client and server or manually requesting a resource and observing
the server’s response to your request shows the full range of HTTP protocol interactions. You can do this with a few different tools:

**telnet**  A terminal emulation protocol that is commonly used in TCP/IP-based networks and on the Internet. It is normally used to log onto a remote computer and run a terminal session remotely, but it can also be used to connect to any TCP (Transmission Control Protocol) socket on a remote server to manually control the connection. Since telnet is considered an inherent part of the TCP/IP communications protocol, any computer that supports TCP/IP network communications should supply a reasonably useful telnet client utility.

**lwp-request**  A Perl tool that allows you to control an HTTP connection manually.

**HttpSniffer.pl**  A Perl tool you can use to observe the HTTP connection between a client and server.

### Using telnet

You can connect directly to a Web server and enter the HTTP request manually with the Linux `telnet` command, which allows you to connect to a specific TCP port on the remote system. Not only will this allow you to see the complete exchange of messages between the client and server, it also gives you complete control of the session and provides a valuable tool for troubleshooting your Web server.

Enter the following `telnet` command at the shell prompt, replacing `somehost.com` with the name of any server accessible from your workstation and known to be running a Web server:

```
telnet somehost.com 80
```

This command instructs `telnet` to connect to TCP port 80, which is the well-known port reserved for HTTP connections. You should receive some confirmation of a successful connection, but you will not receive data immediately from the remote server. If the process listening on port 80 of the remote system is an HTTP server (as it should be), it sends nothing upon receiving a connection, because it is waiting for a request from the client. This behavior is defined by the HTTP specification.

The examples that follow are actual traces from my Linux server, which hosts a fully operational Apache server. I `telnet` to `localhost`, which is a special reserved hostname for the local system. You can do the same, if the system on which you are executing `telnet` also hosts an HTTP server. (If you stay with me through Chapter 5, “Apache Modules,” you’ll have a working system on which to test these commands.) Until then, you can connect to any Web server on the Internet to perform these tests.
At this point, `telnet` has an open connection to the remote HTTP server, which is waiting for a valid HTTP request. The simplest request you can enter is

```bash
GET /
```

This requests the default Web page for the directory defined as the server root. A properly configured HTTP server should respond with a valid page. Our request, which makes no mention of the HTTP version we wish to use, will cause the server to assume we are using HTTP/0.9. This shouldn’t cause problems with any server, but it is considered an obsolete form. All requests in HTTP/1.0 and subsequent versions should contain the HTTP version of the requester (or browser software):

```bash
GET / HTTP/1.0
```

The server, which assumes you are a client that understands only HTTP/0.9, simply sends the requested resource (in this case, the default page for my Web site). In Listing 1.1, I’ve issued the same request, but this time my `GET` line specifies HTTP/1.0 as the version of HTTP I’m using. Notice this time that the server will not respond as soon you type the request and press Enter. It waits for additional information (this is normal HTTP/1.0 behavior). Two carriage-return/line-feed character pairs are required to indicate the end of an HTTP/1.0 request.

**Listing 1.1** Testing Apache with `telnet`

```bash
$ telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^[].'
GET /
```

```html
<html>
<head>
<title> Simple Test Page </title>
</head>
<body>
<h1> Simple Test Page </h1>
</body>
</html>
```
Using **lwp-request**

If you’ve installed the collection of Perl modules and utility scripts collectively known as libwww-perl, you can use the lwp-request script that comes with that package to test HTTP connections. With this script, you can specify different request methods and display options. The following example illustrates the use of the `-e` argument to display response headers (more on headers shortly) with the `-d` argument to suppress the content in the response:

```
$ lwp-request -e -d http://localhost/
```

Be sure to explore the other options available for lwp-request. For example, you can use the `-H` option to specify arbitrary request headers. This can be especially useful when experimenting with HTTP. For example, you can add `Referer:` and `Host:` headers to your request with this command:

```
lwp-request -H 'Referer: http://another.url.com/' \ 
   -H 'Host: vhost1.hiwaay.net' http://jackal.hiwaay.net/
```
libwww-perl consists of several scripts, supported by the following standard Perl modules (available separately, although most easily installed as part of the libwww-perl bundle):

- **URI**  Support for Uniform Resource Identifiers
- **Net::FTP**  Support for the FTP protocol
- **MIME::Base64**  Required for authentication headers
- **Digest::MD5**  Required for digest authentication
- **HTML::HeadParser**  Support for HTML headers

Even though you may not actually use the functionality of all of these modules, they must be properly installed on your machine to use the utility scripts provided with libwww-perl. Use the following commands to install everything at once on a Linux system on which you have the CPAN.pm module:

```perl
# perl -MCPAN -e shell;
cpan> install Bundle::LWP
```

Among the utilities provided with libwww-perl, the most important (and the one most useful for examining the exchange of headers in an HTTP transaction) is lwp-request. Another that I find very useful, however, is lwp-download, which can be used to retrieve a resource from a remote server. Note that besides the HTTP shown in this example, you can also use FTP, which gives you a simple, interesting way to automate FTP downloads:

```bash
# lwp-download http://jackal.hiwaay.net
Saving to 'index.html'...
3.85 KB received
```

---

### CPAN

The best way to maintain the latest versions of all Perl modules is to use the CPAN.pm module. This powerful module is designed to ensure that you have the latest available versions of Perl modules registered with the Comprehensive Perl Archive Network, or CPAN (http://cpan.org). CPAN archives virtually everything that has to do with Perl, including software as source code and binary ports, along with documentation, code samples, and newsgroup postings. The CPAN site is mirrored at over 100 sites around the world, for speed and reliability. You generally choose the one nearest you geographically.

The CPAN.pm Perl module completely automates the processes of comparing your installed modules against the latest available in the CPAN archives, downloading modules, building modules (using the enclosed makefiles), and installing them.
The module is intelligent enough to connect to any one of the CPAN mirror sites, and (using FTP) can download lists of the latest modules for comparison against your local system to see whether you have modules installed that need upgrading. Once you install it, CPAN.pm even updates itself! Not only does the module automate the process of updating and installing modules, it makes the process almost bulletproof. I have never experienced problems with the module.

Another powerful Perl tool for observing the HTTP protocol is HttpSniffer.pl. Although not as convenient as lwp-request because it does require setup and a separate client component (usually a Web browser), HttpSniffer.pl allows you to “snoop” on a real-world HTTP exchange. As you’ll read next, it is more useful when you need to examine header exchanges with a browser (during content negotiation, for example).

Using HttpSniffer.pl

HttpSniffer.pl acts as an HTTP tunnel, connecting directly to a remote server, forwarding connections from client browsers, and displaying the headers (or writing them to a log file) exchanged between the client and server.

Download HttpSniffer.pl directly from its author’s Web site at www.schmerg.com. You can run the program on any platform running Perl 5.004 (or later). Figure 1.2 shows a typical session. The command window in the foreground shows how I invoked HttpSniffer.pl, pointing it at my Web server, jackal.hiwaay.net, with the -r argument. HttpSniffer.pl, by default, receives connections on TCP port 8080 and forwards them to the specified remote host. The browser in the background (running on the same computer as HttpSniffer.pl) is pointed at the URL http://localhost:8080. It appears to receive a page directly from jackal.hiwaay.net, but the connection is actually made by HttpSniffer.pl, which displays both the client request HTTP headers and the server response HTTP headers. The pages retrieved from jackal.hiwaay.net by HttpSniffer.pl are returned to the requesting browser.

HttpSniffer.pl is not only an invaluable debugging tool, it is also the best way to learn the purpose of HTTP headers by watching the actual headers that are part of an HTTP exchange. If you have access to a proxy server, on a remote server or through Apache’s mod_proxy (discussed in Chapter 12), you can point HttpSniffer.pl at the proxy, and then configure your client browser to connect to HttpSniffer.pl as an HTTP proxy server. That way, you can use your browser to connect to any remote host as you normally would, and all requests will be redirected (or proxied) by HttpSniffer.pl. Be prepared for lots of output, though. Generally, you should invoke HttpSniffer.pl with a line like the following (the -l argument causes all of the output from the command to be written into the text file specified):

```
# HttpSniffer.pl -r jackal.hiwaay.net -l /tmp/httpheaders.txt
```
Figure 1.2  HttpSniffer.pl at work

The only problem with HttpSniffer.pl and lwp-request is that they are not available on every Linux system. But telnet is. For that reason, I use telnet in the examples in this chapter; every Linux administrator has access to it and can duplicate these examples. However, if you have HttpSniffer.pl or lwp-request, I encourage you to use them for testing.

The Response Code Header

In Listing 1.1, notice that HTTP sends a group of headers before sending the requested resource. The first header identifies the version of HTTP supported by the server and sends a request response code. The response code is in two parts, a number and a comment: 200 is the response code for a fully successful request, the OK is the comment provided as a convenience for human viewers only.

In Listing 1.1, the server replies with the response code 200, indicating that everything went well. Of course, that is not always the case. HTTP response codes fall into five categories, with a range of codes for each category:

<table>
<thead>
<tr>
<th>Code Range</th>
<th>Response Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>100–199</td>
<td>Informational</td>
</tr>
</tbody>
</table>
The response categories contain more than 40 individual response codes. Each is accompanied by a short comment that is intended to make the code understandable to the user. To see a full list of these codes, go to the HTML Writers Guild at www.hwg.org/lists/hwg-servers/response_codes.html.

When using telnet to test an HTTP connection, it is best to replace the GET request method with HEAD. This prevents the server from actually sending the requested resource; it sends only the headers in reply. For telnet tests, the headers are what you’re interested in. The resource is best viewed with a real browser.

The request shown in Listing 1.2, specifying HTTP/1.1, has a very different result from the first test.

**Listing 1.2** The Headers in a Failed Test

```
$ telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET / HTTP/1.1

HTTP/1.1 400 Bad Request
Date: Wed, 20 Feb 2002 19:06:18 GMT
Server: Apache/2.0.32 (Unix) DAV/2
Content-Length: 370
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
client sent HTTP/1.1 request without hostname (see RFC2616 section 14.23):
</p>
```
The response code header clearly indicates that our request failed. This is because HTTP/1.1 requires the client browser to furnish a hostname if it chooses to use HTTP/1.1. Note that the choice of HTTP version is always the client's. This hostname will usually be the same as the hostname of the Web server. (Chapter 6 discusses virtual hosting, in which a single Web server answers requests for multiple hostnames.)

In addition to warning the client about a failed request, the server makes note of all request failures in its own log file. The failed request in Listing 1.2 causes the following error to be logged by the server:

```
[Wed Feb 20 13:06:19 2002] [error] [client 127.0.0.1] client sent HTTP/1.1 request without hostname (see RFC2616 section 14.23): /
```

NOTE Logging is an important topic that is covered extensively later in this book. Chapter 11, “Apache Logging,” is a complete discussion of connection and error logging in Apache. The path and filename of the log are defined in the Apache configuration, as we’ll see in Chapter 4, “The Apache Core Directives.”

Request redirection is an essential technique for many Web servers as resources are moved or retired. (Chapter 9 shows how to use Apache’s tools for aliasing and redirection.) Listing 1.3 illustrates a redirected request.

**Listing 1.3 A Redirected Request**

```
# telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^[ ]'.
GET /manual HTTP/1.0

HTTP/1.1 301 Moved Permanently
Date: Wed, 20 Feb 2002 19:12:36 GMT
Server: Apache/2.0.32 (Unix) DAV/2
Location: http://jackal.hiwaay.net/manual/
Content-Length: 316
Connection: close
Content-Type: text/html; charset=iso-8859-1
```
If the browser specifies HTTP/1.1 in the request line, the very next line *must* identify a hostname for the request, as in Listing 1.4.

**Listing 1.4 Using the HTTP/1.1 Host Command**

```
# telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.jackal.hiwaay.net

HTTP/1.1 200 OK
Date: Wed, 20 Feb 2002 19:13:38 GMT
Server: Apache/2.0.32 (Unix) DAV/2
Last-Modified: Wed, 20 Feb 2002 18:05:13 GMT
ETag: "53c25-7d-6e514840"
Accept-Ranges: bytes
Content-Length: 125
Content-Type: text/html; charset=ISO-8859-1

<HTML>
<HEAD>
<TITLE> Simple Test Page </TITLE>
</HEAD>
<BODY>
<H1> Simple Test Page </H1>
</BODY>
</HTML>
```

Connection closed by foreign host.
If our server answers requests for several virtual hosts, the Host: header of the request would identify the virtual host that should respond to the request. Better support for virtual site hosting is one of the major enhancements to the HTTP protocol in version 1.1.

The Other Headers
The response code header is always the first header sent by the server, and is usually followed by a number of other headers that convey additional information about the HTTP message or the resource it contains (usually referred to as the message body). For example, the test shown in Listing 1.4 produced seven additional headers after the response code header: Date, Server, Last Modified, ETag, Accept-Ranges, Content-Length, and Content-Type. The following sections briefly outline these and other HTTP headers.

General Headers
Headers that carry information about the messages being transmitted between client and server are lumped into the category of general headers. These headers do not provide information about the content of the messages being transmitted between the client and server. Instead, they carry information that applies to the entire session and to both client request and server response portions of the transaction. They include:

- Cache-Control: Specifies directives to proxy servers (Chapter 12).
- Connection: Allows the sender to specify options for this network connection.
- Date: Standard representation of the date and time the message was sent.
- Pragma: Used to convey non-HTTP information to any recipient that understands the contents of the header. The contents are not part of HTTP.
- Trailer: Indicates a set of header fields that can be found in the trailer of a multiple-part message.
- Transfer-Encoding: Indicates any transformations that have been applied to the message body in order to correctly transfer it.
- Upgrade: Used by the client to specify additional communication protocols it supports and would like to use if the server permits.
- Via: Tacked onto the message by proxies or gateways to show that they handled the message.
- Warning: Specifies additional information about the status or transformation of a message, which might not be reflected in the message itself.

Request Headers
Request headers are used to pass information from HTTP client to server; these headers always follow the one mandatory line in a request, which contains the URI of the
request itself. Request headers act as modifiers for the actual request, allowing the client to include additional information that qualifies the request, usually specifying what constitutes an acceptable response:

- **Accept** Lists all MIME media types the client is capable of accepting.
- **Accept-Charset** Lists all character sets the client is capable of accepting.
- **Accept-Encoding** Lists all encodings (particularly compression schemes) the client is capable of accepting.
- **Accept-Language** Lists all languages the client is willing to accept.
- **Authorization** Provides the user's credentials to access the requested resource (usually a username/password pair).
- **Expect** Indicates server behaviors that are required by the client.
- **From** An Internet e-mail address for the person controlling the requesting user agent (browser).
- **Host** Indicates an Internet hostname and port number for the resource being requested. Used by HTTP/1.1 clients to specify a single virtual host among many on a server.
- **If-Match** A client that has one or more resources previously obtained from the server can verify that one of those resources is current by including a list of their associated tags in this header.
- **If-Modified-Since** Specifies a date found in a previously received entity to check it for currency.
- **If-None-Match** Similar to the If-Match: header, but used to verify that none of the previously received resources is current.
- **If-Range** When a client has a partial copy of a resource in its cache, it can use this header to retrieve the rest of the resource if it hasn’t been modified.
- **If-Unmodified-Since** Used by caching engines to specify that the resource should be sent only if not modified since a specified date.
- **Max-Forwards** Specifies the number of times this client request can be forwarded by proxies and gateways.
- **Proxy-Authorization** Supplies the credentials that the client must supply to use a proxy server.
- **Range** Specifies the retrieval of a portion of a resource, usually specified as a range of bytes.
- **Referer** Specifies the URI of the resource from which the request URI was obtained (usually from a hyperlink in another Web page).


TE Indicates what transfer encodings the client is willing to accept and whether it will accept headers in trailer fields in chunked transfer-coding.

User-Agent Contains information about the user agent (browser) originating the request.

Response Headers
The server uses response headers to pass information in addition to the request response to the requesting client. Response headers usually provide information about the response message itself, and not necessarily about the resource being sent to satisfy a client request. Increasingly, response headers serve to provide information used by caching gateways or proxy server engines. The response headers will be an important part of the discussion on proxy caching (Chapter 12). They include:

Accept-Ranges Specifies units (usually bytes) in which the server will accept range requests.

Age The server’s estimated time (in seconds) required to fulfill this request.

Etag Contains the current value of the requested entity tag.

Location Contains a URI to which the client request should be redirected.

Proxy-Authenticate Indicates the authentication schema and parameters applicable to the proxy for this request.

Retry-After Used by the server to indicate how long a URI is expected to be unavailable.

Server Contains information about the software used by the origin server to handle the request. Apache identifies itself using this header.

Vary Indicates that the resource has multiple sources that may vary according to the supplied list of request headers.

WWW-Authenticate Used with a 401-Unauthorized response code to indicate that the requested URI needs authentication, and specifies the authorization scheme required (usually a username/password pair) and the name of the authorization realm.

Entity Headers
Entity headers contain information directly related to the resource being provided to the client in fulfillment of the request—in other words, the response message content or body. This information is used by the client to determine how to render the resource or which application to invoke to handle it (for example, Adobe Acrobat Reader). Entity headers contain metainformation (or information about information, the subject of Chapter 15):

Allow Informs the client of valid methods associated with the resource.
**Content-Encoding** Indicates the encoding (usually compression) scheme applied to the contents.

**Content-Language** Indicates the natural language of the contents.

**Content-Length** Contains the size of the body of the HTTP message.

**Content-Location** Supplies the resource location for the resource in the message body, usually used when the resource should be requested using another URI.

**Content-MD5** Contains the MD5 digest of the message body, used to verify the integrity of the resource.

**Content-Range** Sent with a partial body to specify where in the complete resource this portion fits.

**Content-Type** Describes the MIME media type of the contents.

**Expires** Specifies a date and time after which the resource should be considered obsolete.

**Last-Modified** Specifies the date and time at which the original document or resource was last modified.

**NOTE** More details about these and other headers are available in the HTTP specification, RFC 2616 (see www.rfc-editor.org).

### In Sum

This chapter looked at the World Wide Web, its origins and history, and described briefly how it functions. An essential part of the design of the Web is the standard set of protocols that allow applications to interoperate with any of the millions of other systems that make up the Web. The protocol that enables the Web to exist is the HyperText Transfer Protocol (HTTP), which defines how data is communicated between Web clients and servers. This chapter demonstrated a couple of ways to view HTTP headers and listed those headers that are defined by the HTTP specification (or RFC). The chapter concluded with a discussion of the important enhancements to the HTTP protocol that were added in its current version, HTTP/1.1. This information provides the foundation for understanding what Apache does and how it does it.

Although Apache is quite well established as the leading Web server on the Internet, it is by no means the only Web server to compete for that status. The next chapter provides a brief look at the most important of its competitors, in order to place Apache in its proper context as it stands out as the best of the breed, even among champion contenders. I’ll also discuss the important changes that have been made to Apache 2.0, the latest major upgrade to the Apache Web server. These changes will ensure that Apache maintains its dominance of the Internet Web server market.