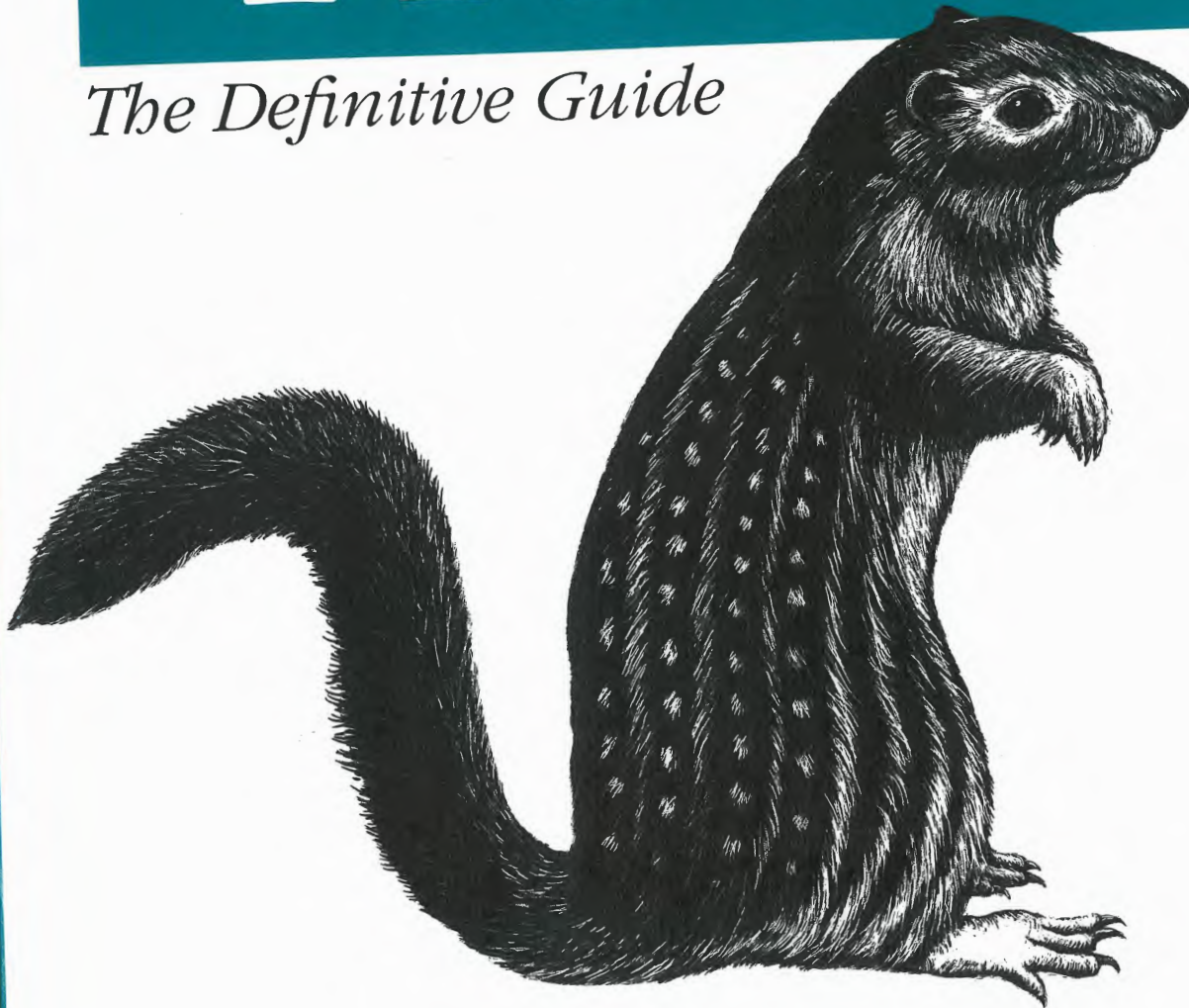


Understanding Web Internals

HTTP

The Definitive Guide



O'REILLY®

*David Gourley & Brian Totty
with Marjorie Sayer, Sailu Reddy & Anshu Aggarwal*

HTTP

The Definitive Guide

David Gourley and Brian Totty
with Marjorie Sayer, Sailu Reddy, and Anshu Aggarwal

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

HTTP: The Definitive Guide

by David Gourley and Brian Totty
with Marjorie Sayer, Sailu Reddy, and Anshu Aggarwal

Copyright © 2002 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol,
CA 95472.

O'Reilly Media, Inc. books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Linda Mui
Production Editor: Rachel Wheeler
Cover Designer: Ellie Volckhausen
Interior Designers: David Futato and Melanie Wang

Printing History:

September 2002: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *HTTP: The Definitive Guide*, the image of a thirteen-lined ground squirrel, and related trade dress are trademarks of O'Reilly Media, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

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

ISBN: 978-1-56592-509-0
[LSI]

[2011-01-27]

Table of Contents

Preface	xiii
---------------	------

Part I. HTTP: The Web's Foundation

1. Overview of HTTP	3
HTTP: The Internet's Multimedia Courier	3
Web Clients and Servers	4
Resources	4
Transactions	8
Messages	10
Connections	11
Protocol Versions	16
Architectural Components of the Web	17
The End of the Beginning	21
For More Information	21
2. URLs and Resources	23
Navigating the Internet's Resources	24
URL Syntax	26
URL Shortcuts	30
Shady Characters	35
A Sea of Schemes	38
The Future	40
For More Information	41
3. HTTP Messages	43
The Flow of Messages	43
The Parts of a Message	44

Methods	53
Status Codes	59
Headers	67
For More Information	73
4. Connection Management	74
TCP Connections	74
TCP Performance Considerations	80
HTTP Connection Handling	86
Parallel Connections	88
Persistent Connections	90
Pipelined Connections	99
The Mysteries of Connection Close	101
For More Information	104

Part II. HTTP Architecture

5. Web Servers	109
Web Servers Come in All Shapes and Sizes	109
A Minimal Perl Web Server	111
What Real Web Servers Do	113
Step 1: Accepting Client Connections	115
Step 2: Receiving Request Messages	116
Step 3: Processing Requests	120
Step 4: Mapping and Accessing Resources	120
Step 5: Building Responses	125
Step 6: Sending Responses	127
Step 7: Logging	127
For More Information	127
6. Proxies	129
Web Intermediaries	129
Why Use Proxies?	131
Where Do Proxies Go?	137
Client Proxy Settings	141
Tricky Things About Proxy Requests	144
Tracing Messages	150
Proxy Authentication	156

Proxy Interoperation	157
For More Information	160
7. Caching	161
Redundant Data Transfers	161
Bandwidth Bottlenecks	161
Flash Crowds	163
Distance Delays	163
Hits and Misses	164
Cache Topologies	168
Cache Processing Steps	171
Keeping Copies Fresh	175
Controlling Cachability	182
Setting Cache Controls	186
Detailed Algorithms	187
Caches and Advertising	194
For More Information	196
8. Integration Points: Gateways, Tunnels, and Relays	197
Gateways	197
Protocol Gateways	200
Resource Gateways	203
Application Interfaces and Web Services	205
Tunnels	206
Relays	212
For More Information	213
9. Web Robots	215
Crawlers and Crawling	215
Robotic HTTP	225
Misbehaving Robots	228
Excluding Robots	229
Robot Etiquette	239
Search Engines	242
For More Information	246
10. HTTP-NG	247
HTTP's Growing Pains	247
HTTP-NG Activity	248

Modularize and Enhance	248
Distributed Objects	249
Layer 1: Messaging	250
Layer 2: Remote Invocation	250
Layer 3: Web Application	251
WebMUX	251
Binary Wire Protocol	252
Current Status	252
For More Information	253

Part III. Identification, Authorization, and Security

11. Client Identification and Cookies	257
The Personal Touch	257
HTTP Headers	258
Client IP Address	259
User Login	260
Fat URLs	262
Cookies	263
For More Information	276
12. Basic Authentication	277
Authentication	277
Basic Authentication	281
The Security Flaws of Basic Authentication	283
For More Information	285
13. Digest Authentication	286
The Improvements of Digest Authentication	286
Digest Calculations	291
Quality of Protection Enhancements	299
Practical Considerations	300
Security Considerations	303
For More Information	306
14. Secure HTTP	307
Making HTTP Safe	307
Digital Cryptography	309

Symmetric-Key Cryptography	313
Public-Key Cryptography	315
Digital Signatures	317
Digital Certificates	319
HTTPS: The Details	322
A Real HTTPS Client	328
Tunneling Secure Traffic Through Proxies	335
For More Information	336

Part IV. Entities, Encodings, and Internationalization

15. Entities and Encodings	341
Messages Are Crates, Entities Are Cargo	342
Content-Length: The Entity's Size	344
Entity Digests	347
Media Type and Charset	348
Content Encoding	351
Transfer Encoding and Chunked Encoding	354
Time-Varying Instances	359
Validators and Freshness	360
Range Requests	363
Delta Encoding	365
For More Information	369
16. Internationalization	370
HTTP Support for International Content	370
Character Sets and HTTP	371
Multilingual Character Encoding Primer	376
Language Tags and HTTP	384
Internationalized URIs	389
Other Considerations	392
For More Information	392
17. Content Negotiation and Transcoding	395
Content-Negotiation Techniques	395
Client-Driven Negotiation	396
Server-Driven Negotiation	397
Transparent Negotiation	400

Transcoding	403
Next Steps	405
For More Information	406

Part V. Content Publishing and Distribution

18. Web Hosting	411
Hosting Services	411
Virtual Hosting	413
Making Web Sites Reliable	419
Making Web Sites Fast	422
For More Information	423
19. Publishing Systems	424
FrontPage Server Extensions for Publishing Support	424
WebDAV and Collaborative Authoring	429
For More Information	446
20. Redirection and Load Balancing	448
Why Redirect?	449
Where to Redirect	449
Overview of Redirection Protocols	450
General Redirection Methods	452
Proxy Redirection Methods	462
Cache Redirection Methods	469
Internet Cache Protocol	473
Cache Array Routing Protocol	475
Hyper Text Caching Protocol	478
For More Information	481
21. Logging and Usage Tracking	483
What to Log?	483
Log Formats	484
Hit Metering	492
A Word on Privacy	495
For More Information	495

Part VI. Appendixes

A. URI Schemes	499
B. HTTP Status Codes	505
C. HTTP Header Reference	508
D. MIME Types	533
E. Base-64 Encoding	570
F. Digest Authentication	574
G. Language Tags	581
H. MIME Charset Registry	602
Index	617

Overview of HTTP

The world's web browsers, servers, and related web applications all talk to each other through HTTP, the Hypertext Transfer Protocol. HTTP is the common language of the modern global Internet.

This chapter is a concise overview of HTTP. You'll see how web applications use HTTP to communicate, and you'll get a rough idea of how HTTP does its job. In particular, we talk about:

- How web clients and servers communicate
- Where resources (web content) come from
- How web transactions work
- The format of the messages used for HTTP communication
- The underlying TCP network transport
- The different variations of the HTTP protocol
- Some of the many HTTP architectural components installed around the Internet

We've got a lot of ground to cover, so let's get started on our tour of HTTP.

HTTP: The Internet's Multimedia Courier

Billions of JPEG images, HTML pages, text files, MPEG movies, WAV audio files, Java applets, and more cruise through the Internet each and every day. HTTP moves the bulk of this information quickly, conveniently, and reliably from web servers all around the world to web browsers on people's desktops.

Because HTTP uses reliable data-transmission protocols, it guarantees that your data will not be damaged or scrambled in transit, even when it comes from the other side of the globe. This is good for you as a user, because you can access information without worrying about its integrity. Reliable transmission is also good for you as an Internet application developer, because you don't have to worry about HTTP communications

being destroyed, duplicated, or distorted in transit. You can focus on programming the distinguishing details of your application, without worrying about the flaws and foibles of the Internet.

Let's look more closely at how HTTP transports the Web's traffic.

Web Clients and Servers

Web content lives on web servers. Web servers speak the HTTP protocol, so they are often called HTTP servers. These HTTP servers store the Internet's data and provide the data when it is requested by HTTP clients. The clients send HTTP requests to servers, and servers return the requested data in HTTP responses, as sketched in Figure 1-1. Together, HTTP clients and HTTP servers make up the basic components of the World Wide Web.

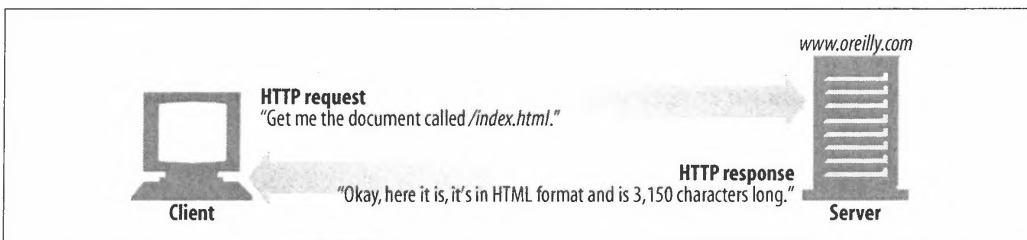


Figure 1-1. Web clients and servers

You probably use HTTP clients every day. The most common client is a web browser, such as Microsoft Internet Explorer or Netscape Navigator. Web browsers request HTTP objects from servers and display the objects on your screen.

When you browse to a page, such as "`http://www.oreilly.com/index.html`," your browser sends an HTTP request to the server `www.oreilly.com` (see Figure 1-1). The server tries to find the desired object (in this case, "`/index.html`") and, if successful, sends the object to the client in an HTTP response, along with the type of the object, the length of the object, and other information.

Resources

Web servers host *web resources*. A web resource is the source of web content. The simplest kind of web resource is a static file on the web server's filesystem. These files can contain anything: they might be text files, HTML files, Microsoft Word files, Adobe Acrobat files, JPEG image files, AVI movie files, or any other format you can think of.

However, resources don't have to be static files. Resources can also be software programs that generate content on demand. These dynamic content resources can generate content based on your identity, on what information you've requested, or on

the time of day. They can show you a live image from a camera, or let you trade stocks, search real estate databases, or buy gifts from online stores (see Figure 1-2).

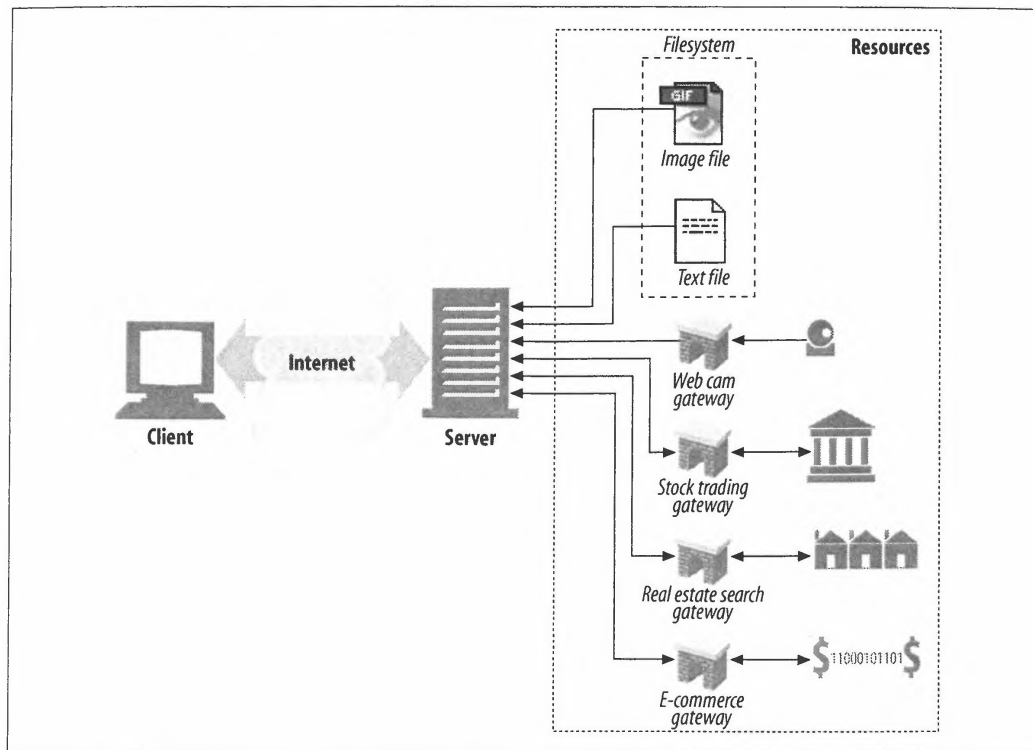


Figure 1-2. A web resource is anything that provides web content

In summary, a resource is any kind of content source. A file containing your company's sales forecast spreadsheet is a resource. A web gateway to scan your local public library's shelves is a resource. An Internet search engine is a resource.

Media Types

Because the Internet hosts many thousands of different data types, HTTP carefully tags each object being transported through the Web with a data format label called a *MIME type*. MIME (Multipurpose Internet Mail Extensions) was originally designed to solve problems encountered in moving messages between different electronic mail systems. MIME worked so well for email that HTTP adopted it to describe and label its own multimedia content.

Web servers attach a MIME type to all HTTP object data (see Figure 1-3). When a web browser gets an object back from a server, it looks at the associated MIME type to see if it knows how to handle the object. Most browsers can handle hundreds of popular object types: displaying image files, parsing and formatting HTML files, playing audio files through the computer's speakers, or launching external plug-in software to handle special formats.

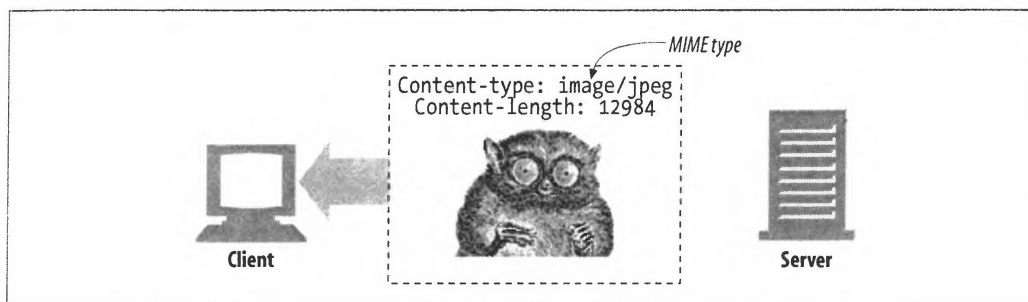


Figure 1-3. MIME types are sent back with the data content

A MIME type is a textual label, represented as a primary object type and a specific subtype, separated by a slash. For example:

- An HTML-formatted text document would be labeled with type `text/html`.
- A plain ASCII text document would be labeled with type `text/plain`.
- A JPEG version of an image would be `image/jpeg`.
- A GIF-format image would be `image/gif`.
- An Apple QuickTime movie would be `video/quicktime`.
- A Microsoft PowerPoint presentation would be `application/vnd.ms-powerpoint`.

There are hundreds of popular MIME types, and many more experimental or limited-use types. A very thorough MIME type list is provided in Appendix D.

URIs

Each web server resource has a name, so clients can point out what resources they are interested in. The server resource name is called a *uniform resource identifier*, or URI. URIs are like the postal addresses of the Internet, uniquely identifying and locating information resources around the world.

Here's a URI for an image resource on Joe's Hardware store's web server:

`http://www.joes-hardware.com/specials/saw-blade.gif`

Figure 1-4 shows how the URI specifies the HTTP protocol to access the saw-blade GIF resource on Joe's store's server. Given the URI, HTTP can retrieve the object. URIs come in two flavors, called URLs and URNs. Let's take a peek at each of these types of resource identifiers now.

URLs

The *uniform resource locator* (URL) is the most common form of resource identifier. URLs describe the specific location of a resource on a particular server. They tell you exactly how to fetch a resource from a precise, fixed location. Figure 1-4 shows how a URL tells precisely where a resource is located and how to access it. Table 1-1 shows a few examples of URLs.

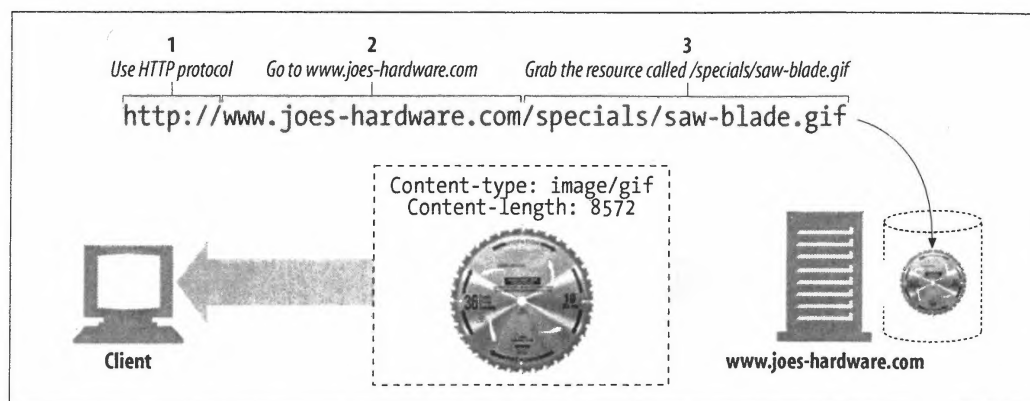


Figure 1-4. URLs specify protocol, server, and local resource

Table 1-1. Example URLs

URL	Description
<code>http://www.oreilly.com/index.html</code>	The home URL for O'Reilly & Associates, Inc.
<code>http://www.yahoo.com/images/logo.gif</code>	The URL for the Yahoo! web site's logo
<code>http://www.joes-hardware.com/inventory-check.cgi?item=12731</code>	The URL for a program that checks if inventory item #12731 is in stock
<code>ftp://joe:tools4u@ftp.joes-hardware.com/locking-pliers.gif</code>	The URL for the <i>locking-pliers.gif</i> image file, using password-protected FTP as the access protocol

Most URLs follow a standardized format of three main parts:

- The first part of the URL is called the *scheme*, and it describes the protocol used to access the resource. This is usually the HTTP protocol (`http://`).
- The second part gives the server Internet address (e.g., `www.joes-hardware.com`).
- The rest names a resource on the web server (e.g., `/specials/saw-blade.gif`).

Today, almost every URI is a URL.

URNs

The second flavor of URI is the *uniform resource name*, or URN. A URN serves as a unique name for a particular piece of content, independent of where the resource currently resides. These location-independent URNs allow resources to move from place to place. URNs also allow resources to be accessed by multiple network access protocols while maintaining the same name.

For example, the following URN might be used to name the Internet standards document "RFC 2141" regardless of where it resides (it may even be copied in several places):

`urn:ietf:rfc:2141`

URNs are still experimental and not yet widely adopted. To work effectively, URNs need a supporting infrastructure to resolve resource locations; the lack of such an infrastructure has also slowed their adoption. But URNs do hold some exciting promise for the future. We'll discuss URNs in a bit more detail in Chapter 2, but most of the remainder of this book focuses almost exclusively on URLs.

Unless stated otherwise, we adopt the conventional terminology and use URI and URL interchangeably for the remainder of this book.

Transactions

Let's look in more detail how clients use HTTP to transact with web servers and their resources. An HTTP transaction consists of a request command (sent from client to server), and a response result (sent from the server back to the client). This communication happens with formatted blocks of data called *HTTP messages*, as illustrated in Figure 1-5.

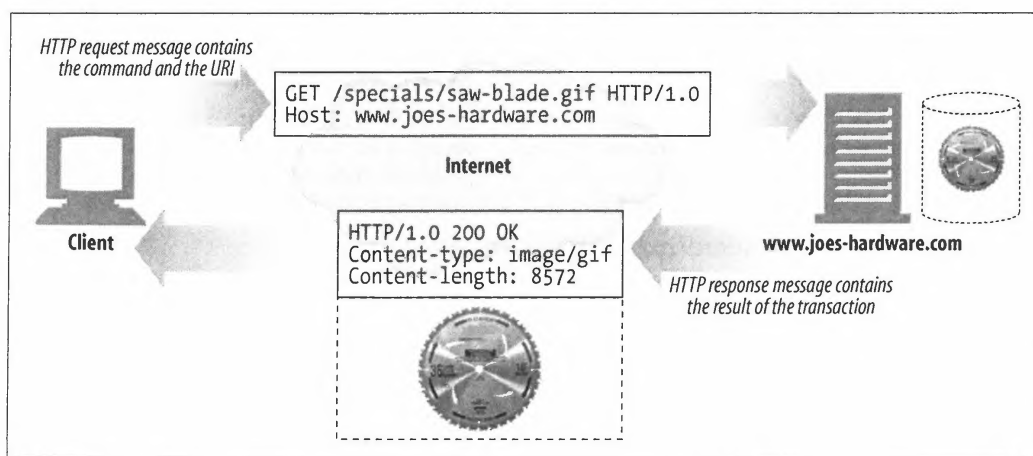


Figure 1-5. HTTP transactions consist of request and response messages

Methods

HTTP supports several different request commands, called *HTTP methods*. Every HTTP request message has a method. The method tells the server what action to perform (fetch a web page, run a gateway program, delete a file, etc.). Table 1-2 lists five common HTTP methods.

Table 1-2. Some common HTTP methods

HTTP method	Description
GET	Send named resource from the server to the client.
PUT	Store data from client into a named server resource.

Table 1-2. Some common HTTP methods (continued)

HTTP method	Description
DELETE	Delete the named resource from a server.
POST	Send client data into a server gateway application.
HEAD	Send just the HTTP headers from the response for the named resource.

We'll discuss HTTP methods in detail in Chapter 3.

Status Codes

Every HTTP response message comes back with a status code. The status code is a three-digit numeric code that tells the client if the request succeeded, or if other actions are required. A few common status codes are shown in Table 1-3.

Table 1-3. Some common HTTP status codes

HTTP status code	Description
200	OK. Document returned correctly.
302	Redirect. Go someplace else to get the resource.
404	Not Found. Can't find this resource.

HTTP also sends an explanatory textual “reason phrase” with each numeric status code (see the response message in Figure 1-5). The textual phrase is included only for descriptive purposes; the numeric code is used for all processing.

The following status codes and reason phrases are treated identically by HTTP software:

- 200 OK
- 200 Document attached
- 200 Success
- 200 All's cool, dude

HTTP status codes are explained in detail in Chapter 3.

Web Pages Can Consist of Multiple Objects

An application often issues multiple HTTP transactions to accomplish a task. For example, a web browser issues a cascade of HTTP transactions to fetch and display a graphics-rich web page. The browser performs one transaction to fetch the HTML “skeleton” that describes the page layout, then issues additional HTTP transactions for each embedded image, graphics pane, Java applet, etc. These embedded resources might even reside on different servers, as shown in Figure 1-6. Thus, a “web page” often is a collection of resources, not a single resource.

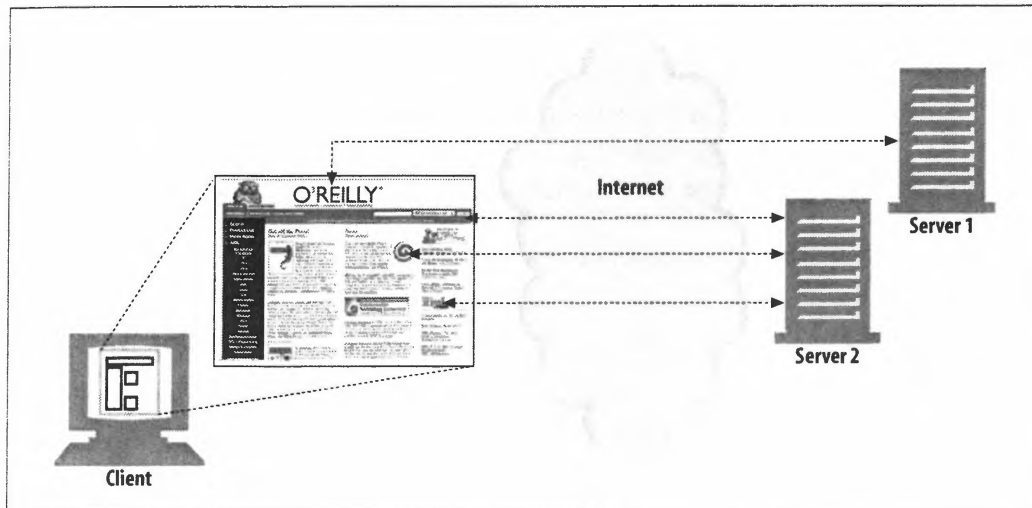


Figure 1-6. Composite web pages require separate HTTP transactions for each embedded resource

Messages

Now let's take a quick look at the structure of HTTP request and response messages. We'll study HTTP messages in exquisite detail in Chapter 3.

HTTP messages are simple, line-oriented sequences of characters. Because they are plain text, not binary, they are easy for humans to read and write.* Figure 1-7 shows the HTTP messages for a simple transaction.

(a) Request message		(b) Response message
GET /test/hi-there.txt HTTP/1.0	Start line	HTTP/1.0 200 OK
Accept: text/* Accept-Language: en,fr	Headers	Content-type: text/plain Content-length: 19
	Body	Hi! I'm a message!

Figure 1-7. HTTP messages have a simple, line-oriented text structure

HTTP messages sent from web clients to web servers are called *request messages*. Messages from servers to clients are called *response messages*. There are no other kinds of HTTP messages. The formats of HTTP request and response messages are very similar.

* Some programmers complain about the difficulty of HTTP parsing, which can be tricky and error-prone, especially when designing high-speed software. A binary format or a more restricted text format might have been simpler to process, but most HTTP programmers appreciate HTTP's extensibility and debuggability.

HTTP messages consist of three parts:

Start line

The first line of the message is the start line, indicating what to do for a request or what happened for a response.

Header fields

Zero or more header fields follow the start line. Each header field consists of a name and a value, separated by a colon (:) for easy parsing. The headers end with a blank line. Adding a header field is as easy as adding another line.

Body

After the blank line is an optional message body containing any kind of data. Request bodies carry data to the web server; response bodies carry data back to the client. Unlike the start lines and headers, which are textual and structured, the body can contain arbitrary binary data (e.g., images, videos, audio tracks, software applications). Of course, the body can also contain text.

Simple Message Example

Figure 1-8 shows the HTTP messages that might be sent as part of a simple transaction. The browser requests the resource `http://www.joes-hardware.com/tools.html`.

In Figure 1-8, the browser sends an HTTP request message. The request has a GET method in the start line, and the local resource is `/tools.html`. The request indicates it is speaking Version 1.0 of the HTTP protocol. The request message has no body, because no request data is needed to GET a simple document from a server.

The server sends back an HTTP response message. The response contains the HTTP version number (HTTP/1.0), a success status code (200), a descriptive reason phrase (OK), and a block of response header fields, all followed by the response body containing the requested document. The response body length is noted in the Content-Length header, and the document's MIME type is noted in the Content-Type header.

Connections

Now that we've sketched what HTTP's messages look like, let's talk for a moment about how messages move from place to place, across Transmission Control Protocol (TCP) connections.

TCP/IP

HTTP is an application layer protocol. HTTP doesn't worry about the nitty-gritty details of network communication; instead, it leaves the details of networking to TCP/IP, the popular reliable Internet transport protocol.

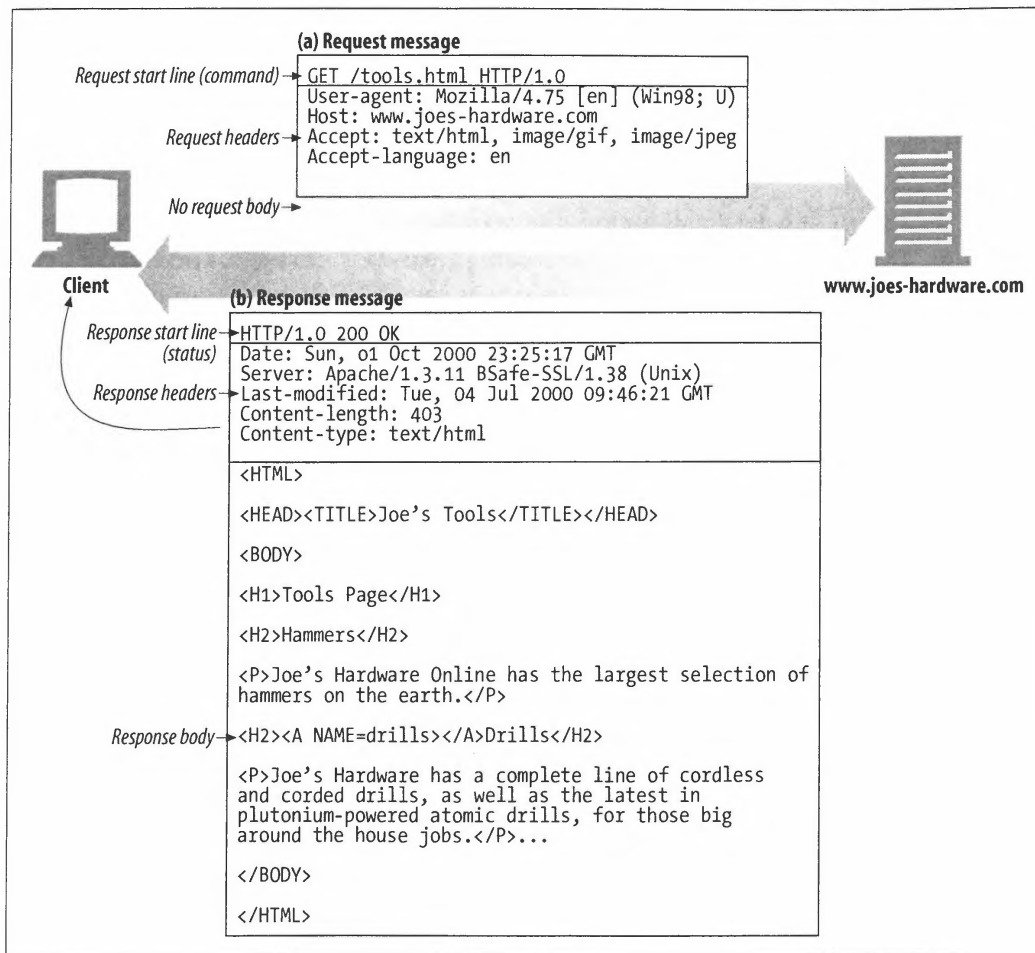


Figure 1-8. Example GET transaction for `http://www.joes-hardware.com/tools.html`

TCP provides:

- Error-free data transportation
- In-order delivery (data will always arrive in the order in which it was sent)
- Unsegmented data stream (can dribble out data in any size at any time)

The Internet itself is based on TCP/IP, a popular layered set of packet-switched network protocols spoken by computers and network devices around the world. TCP/IP hides the peculiarities and foibles of individual networks and hardware, letting computers and networks of any type talk together reliably.

Once a TCP connection is established, messages exchanged between the client and server computers will never be lost, damaged, or received out of order.

In networking terms, the HTTP protocol is *layered* over TCP. HTTP uses TCP to transport its message data. Likewise, TCP is layered over IP (see Figure 1-9).

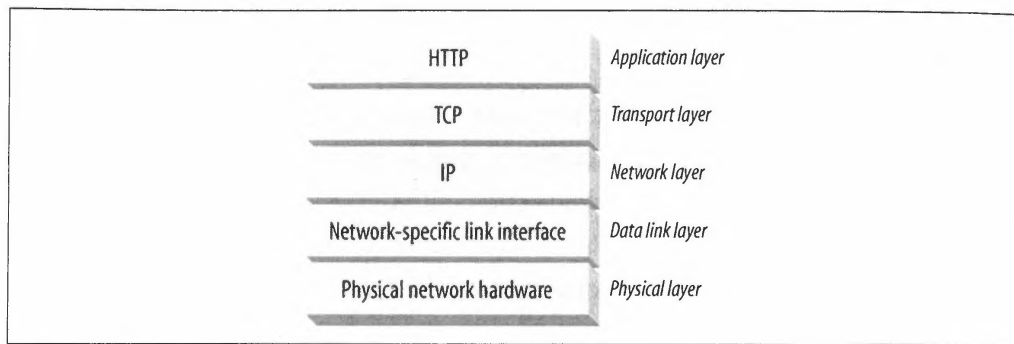


Figure 1-9. HTTP network protocol stack

Connections, IP Addresses, and Port Numbers

Before an HTTP client can send a message to a server, it needs to establish a TCP/IP connection between the client and server using Internet protocol (IP) addresses and port numbers.

Setting up a TCP connection is sort of like calling someone at a corporate office. First, you dial the company's phone number. This gets you to the right organization. Then, you dial the specific extension of the person you're trying to reach.

In TCP, you need the IP address of the server computer and the TCP port number associated with the specific software program running on the server.

This is all well and good, but how do you get the IP address and port number of the HTTP server in the first place? Why, the URL, of course! We mentioned before that URLs are the addresses for resources, so naturally enough they can provide us with the IP address for the machine that has the resource. Let's take a look at a few URLs:

```
http://207.200.83.29:80/index.html  
http://www.netscape.com:80/index.html  
http://www.netscape.com/index.html
```

The first URL has the machine's IP address, "207.200.83.29", and port number, "80".

The second URL doesn't have a numeric IP address; it has a textual domain name, or *hostname* ("www.netscape.com"). The hostname is just a human-friendly alias for an IP address. Hostnames can easily be converted into IP addresses through a facility called the Domain Name Service (DNS), so we're all set here, too. We will talk much more about DNS and URLs in Chapter 2.

The final URL has no port number. When the port number is missing from an HTTP URL, you can assume the default value of port 80.

With the IP address and port number, a client can easily communicate via TCP/IP. Figure 1-10 shows how a browser uses HTTP to display a simple HTML resource that resides on a distant server.

Here are the steps:

- (a) The browser extracts the server's hostname from the URL.
- (b) The browser converts the server's hostname into the server's IP address.
- (c) The browser extracts the port number (if any) from the URL.
- (d) The browser establishes a TCP connection with the web server.
- (e) The browser sends an HTTP request message to the server.
- (f) The server sends an HTTP response back to the browser.
- (g) The connection is closed, and the browser displays the document.

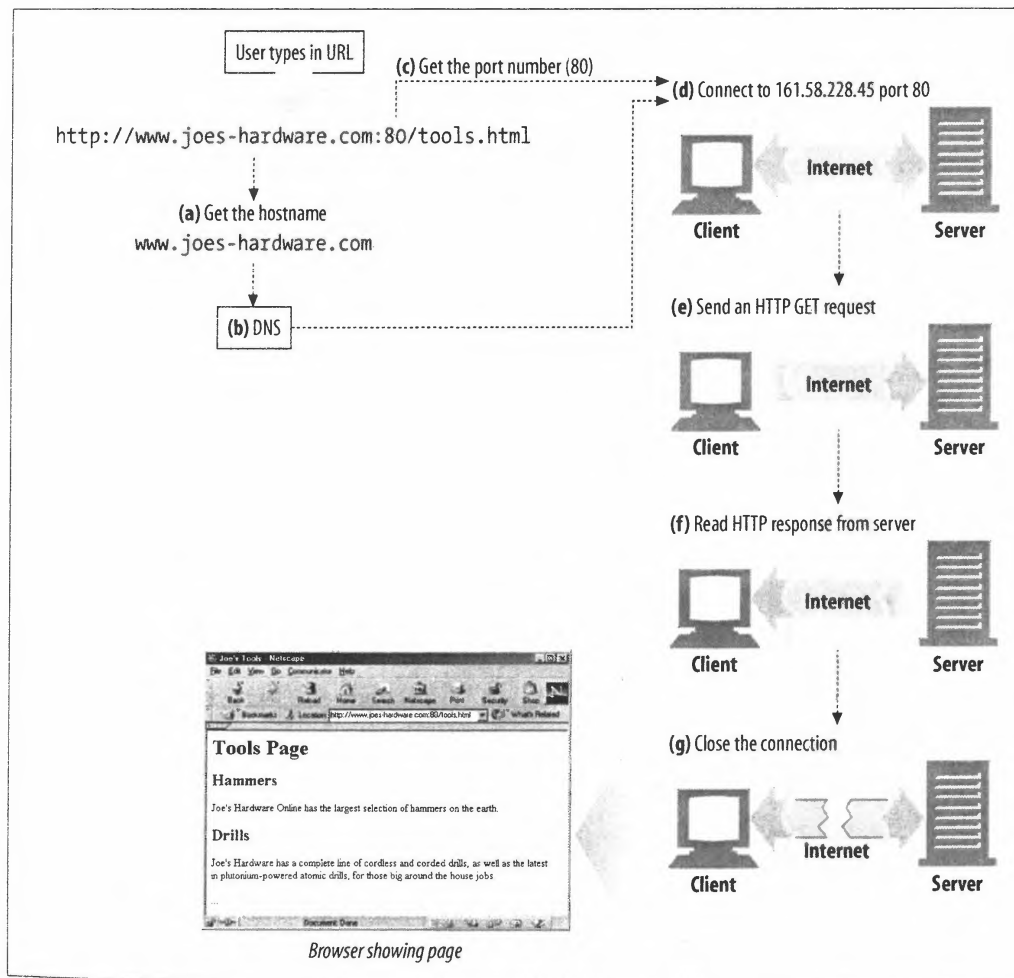


Figure 1-10. Basic browser connection process

A Real Example Using Telnet

Because HTTP uses TCP/IP, and is text-based, as opposed to using some obscure binary format, it is simple to talk directly to a web server.

The Telnet utility connects your keyboard to a destination TCP port and connects the TCP port output back to your display screen. Telnet is commonly used for remote terminal sessions, but it can generally connect to any TCP server, including HTTP servers.

You can use the Telnet utility to talk directly to web servers. Telnet lets you open a TCP connection to a port on a machine and type characters directly into the port. The web server treats you as a web client, and any data sent back on the TCP connection is displayed onscreen.

Let's use Telnet to interact with a real web server. We will use Telnet to fetch the document pointed to by the URL *http://www.joes-hardware.com:80/tools.html* (you can try this example yourself).

Let's review what should happen:

- First, we need to look up the IP address of *www.joes-hardware.com* and open a TCP connection to port 80 on that machine. Telnet does this legwork for us.
- Once the TCP connection is open, we need to type in the HTTP request.
- When the request is complete (indicated by a blank line), the server should send back the content in an HTTP response and close the connection.

Our example HTTP request for *http://www.joes-hardware.com:80/tools.html* is shown in Example 1-1. What we typed is shown in boldface.

Example 1-1. An HTTP transaction using telnet

```
% telnet www.joes-hardware.com 80
Trying 161.58.228.45...
Connected to joes-hardware.com.
Escape character is '^]'.
GET /tools.html HTTP/1.1
Host: www.joes-hardware.com

HTTP/1.1 200 OK
Date: Sun, 01 Oct 2000 23:25:17 GMT
Server: Apache/1.3.11 BSafe-SSL/1.38 (Unix) FrontPage/4.0.4.3
Last-Modified: Tue, 04 Jul 2000 09:46:21 GMT
ETag: "373979-193-3961b26d"
Accept-Ranges: bytes
Content-Length: 403
Connection: close
Content-Type: text/html
```

Example 1-1. An HTTP transaction using telnet (continued)

```
<HTML>
<HEAD><TITLE>Joe's Tools</TITLE></HEAD>
<BODY>
<H1>Tools Page</H1>
<H2>Hammers</H2>
<P>Joe's Hardware Online has the largest selection of hammers on the earth.</P>
<H2><A NAME=drills></A>Drills</H2>
<P>Joe's Hardware has a complete line of cordless and corded drills, as well as the latest
in plutonium-powered atomic drills, for those big around the house jobs.</P> ...
</BODY>
</HTML>
Connection closed by foreign host.
```

Telnet looks up the hostname and opens a connection to the *www.joes-hardware.com* web server, which is listening on port 80. The three lines after the command are output from Telnet, telling us it has established a connection.

We then type in our basic request command, “GET /tools.html HTTP/1.1”, and send a Host header providing the original hostname, followed by a blank line, asking the server to GET us the resource “/tools.html” from the server *www.joes-hardware.com*. After that, the server responds with a response line, several response headers, a blank line, and finally the body of the HTML document.

Beware that Telnet mimics HTTP clients well but doesn't work well as a server. And automated Telnet scripting is no fun at all. For a more flexible tool, you might want to check out nc (netcat). The nc tool lets you easily manipulate and script UDP- and TCP-based traffic, including HTTP. See <http://netcat.sourceforge.net> for details.

Protocol Versions

There are several versions of the HTTP protocol in use today. HTTP applications need to work hard to robustly handle different variations of the HTTP protocol. The versions in use are:

HTTP/0.9

The 1991 prototype version of HTTP is known as HTTP/0.9. This protocol contains many serious design flaws and should be used only to interoperate with legacy clients. HTTP/0.9 supports only the GET method, and it does not support MIME typing of multimedia content, HTTP headers, or version numbers. HTTP/0.9 was originally defined to fetch simple HTML objects. It was soon replaced with HTTP/1.0.

HTTP/1.0

1.0 was the first version of HTTP that was widely deployed. HTTP/1.0 added version numbers, HTTP headers, additional methods, and multimedia object handling. HTTP/1.0 made it practical to support graphically appealing web

pages and interactive forms, which helped promote the wide-scale adoption of the World Wide Web. This specification was never well specified. It represented a collection of best practices in a time of rapid commercial and academic evolution of the protocol.

HTTP/1.0+

Many popular web clients and servers rapidly added features to HTTP in the mid-1990s to meet the demands of a rapidly expanding, commercially successful World Wide Web. Many of these features, including long-lasting “keep-alive” connections, virtual hosting support, and proxy connection support, were added to HTTP and became unofficial, de facto standards. This informal, extended version of HTTP is often referred to as HTTP/1.0+.

HTTP/1.1

HTTP/1.1 focused on correcting architectural flaws in the design of HTTP, specifying semantics, introducing significant performance optimizations, and removing mis-features. HTTP/1.1 also included support for the more sophisticated web applications and deployments that were under way in the late 1990s. HTTP/1.1 is the current version of HTTP.

HTTP-NG (a.k.a. HTTP/2.0)

HTTP-NG is a prototype proposal for an architectural successor to HTTP/1.1 that focuses on significant performance optimizations and a more powerful framework for remote execution of server logic. The HTTP-NG research effort concluded in 1998, and at the time of this writing, there are no plans to advance this proposal as a replacement for HTTP/1.1. See Chapter 10 for more information.

Architectural Components of the Web

In this overview chapter, we’ve focused on how two web applications (web browsers and web servers) send messages back and forth to implement basic transactions. There are many other web applications that you interact with on the Internet. In this section, we’ll outline several other important applications, including:

Proxies

HTTP intermediaries that sit between clients and servers

Caches

HTTP storehouses that keep copies of popular web pages close to clients

Gateways

Special web servers that connect to other applications

Tunnels

Special proxies that blindly forward HTTP communications

Agents

Semi-intelligent web clients that make automated HTTP requests

Proxies

Let's start by looking at HTTP *proxy servers*, important building blocks for web security, application integration, and performance optimization.

As shown in Figure 1-11, a proxy sits between a client and a server, receiving all of the client's HTTP requests and relaying the requests to the server (perhaps after modifying the requests). These applications act as a proxy for the user, accessing the server on the user's behalf.

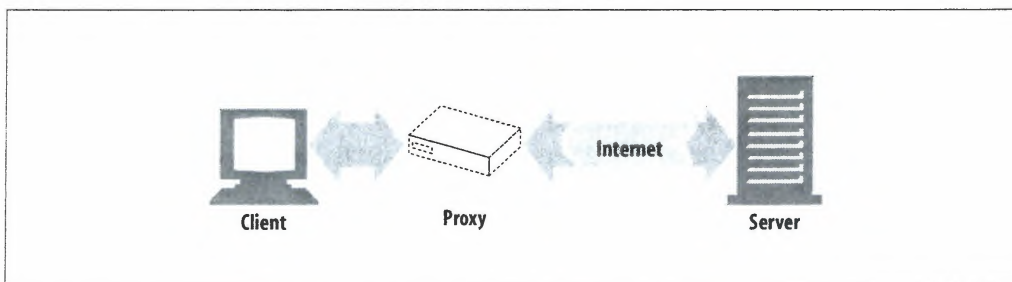


Figure 1-11. Proxies relay traffic between client and server

Proxies are often used for security, acting as trusted intermediaries through which all web traffic flows. Proxies can also filter requests and responses; for example, to detect application viruses in corporate downloads or to filter adult content away from elementary-school students. We'll talk about proxies in detail in Chapter 6.

Caches

A *web cache* or *caching proxy* is a special type of HTTP proxy server that keeps copies of popular documents that pass through the proxy. The next client requesting the same document can be served from the cache's personal copy (see Figure 1-12).

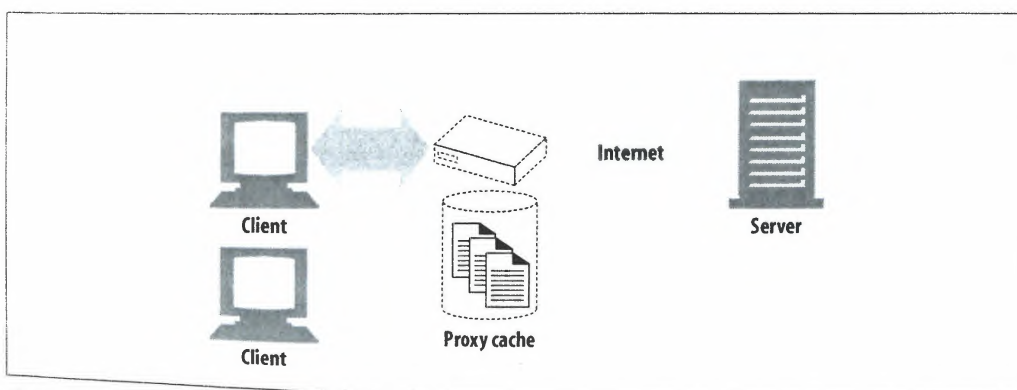


Figure 1-12. Caching proxies keep local copies of popular documents to improve performance

A client may be able to download a document much more quickly from a nearby cache than from a distant web server. HTTP defines many facilities to make caching more effective and to regulate the freshness and privacy of cached content. We cover caching technology in Chapter 7.

Gateways

Gateways are special servers that act as intermediaries for other servers. They are often used to convert HTTP traffic to another protocol. A gateway always receives requests as if it was the origin server for the resource. The client may not be aware it is communicating with a gateway.

For example, an HTTP/FTP gateway receives requests for FTP URIs via HTTP requests but fetches the documents using the FTP protocol (see Figure 1-13). The resulting document is packed into an HTTP message and sent to the client. We discuss gateways in Chapter 8.

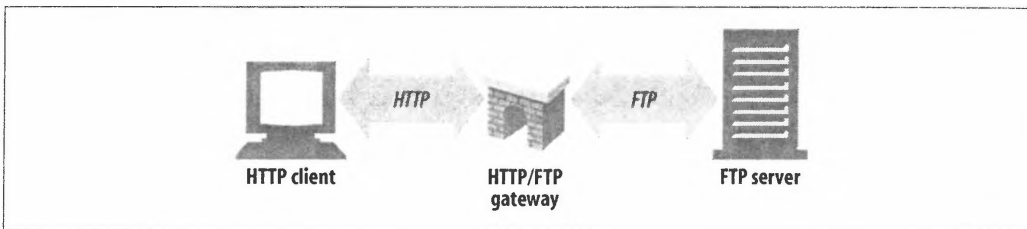


Figure 1-13. HTTP/FTP gateway

Tunnels

Tunnels are HTTP applications that, after setup, blindly relay raw data between two connections. HTTP tunnels are often used to transport non-HTTP data over one or more HTTP connections, without looking at the data.

One popular use of HTTP tunnels is to carry encrypted Secure Sockets Layer (SSL) traffic through an HTTP connection, allowing SSL traffic through corporate firewalls that permit only web traffic. As sketched in Figure 1-14, an HTTP/SSL tunnel receives an HTTP request to establish an outgoing connection to a destination address and port, then proceeds to tunnel the encrypted SSL traffic over the HTTP channel so that it can be blindly relayed to the destination server.

Agents

User agents (or just *agents*) are client programs that make HTTP requests on the user's behalf. Any application that issues web requests is an HTTP agent. So far, we've talked about only one kind of HTTP agent: web browsers. But there are many other kinds of user agents.

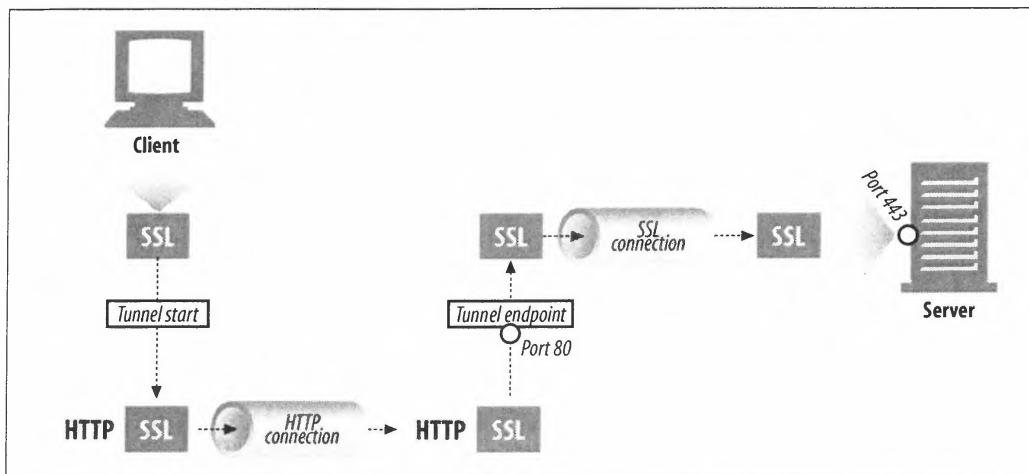


Figure 1-14. Tunnels forward data across non-HTTP networks (HTTP/SSL tunnel shown)

For example, there are machine-automated user agents that autonomously wander the Web, issuing HTTP transactions and fetching content, without human supervision. These automated agents often have colorful names, such as “spiders” or “web robots” (see Figure 1-15). Spiders wander the Web to build useful archives of web content, such as a search engine’s database or a product catalog for a comparison-shopping robot. See Chapter 9 for more information.

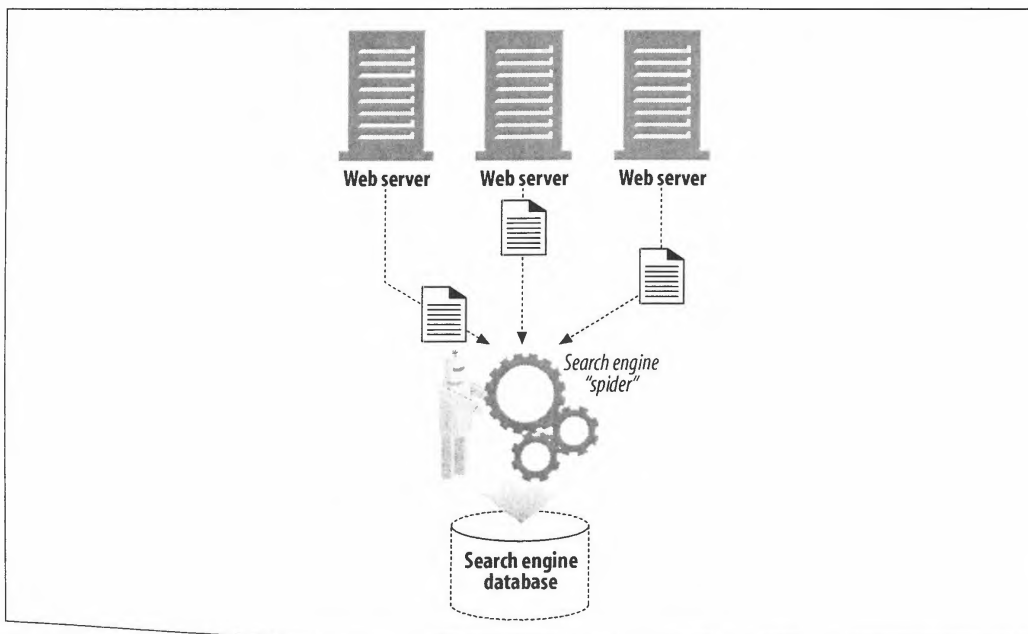


Figure 1-15. Automated search engine “spiders” are agents, fetching web pages around the world

The End of the Beginning

That's it for our quick introduction to HTTP. In this chapter, we highlighted HTTP's role as a multimedia transport protocol. We outlined how HTTP uses URIs to name multimedia resources on remote servers, we sketched how HTTP request and response messages are used to manipulate multimedia resources on remote servers, and we finished by surveying a few of the web applications that use HTTP.

The remaining chapters explain the technical machinery of the HTTP protocol, applications, and resources in much more detail.

For More Information

Later chapters of this book will explore HTTP in much more detail, but you might find that some of the following sources contain useful background about particular topics we covered in this chapter.

HTTP Protocol Information

HTTP Pocket Reference

Clinton Wong, O'Reilly & Associates, Inc. This little book provides a concise introduction to HTTP and a quick reference to each of the headers and status codes that compose HTTP transactions.

<http://www.w3.org/Protocols/>

This W3C web page contains many great links about the HTTP protocol.

<http://www.ietf.org/rfc/rfc2616.txt>

RFC 2616, "Hypertext Transfer Protocol—HTTP/1.1," is the official specification for HTTP/1.1, the current version of the HTTP protocol. The specification is a well-written, well-organized, detailed reference for HTTP, but it isn't ideal for readers who want to learn the underlying concepts and motivations of HTTP or the differences between theory and practice. We hope that this book fills in the underlying concepts, so you can make better use of the specification.

<http://www.ietf.org/rfc/rfc1945.txt>

RFC 1945, "Hypertext Transfer Protocol—HTTP/1.0," is an informational RFC that describes the modern foundation for HTTP. It details the officially sanctioned and "best-practice" behavior of web applications at the time the specification was written. It also contains some useful descriptions about behavior that is deprecated in HTTP/1.1 but still widely implemented by legacy applications.

<http://www.w3.org/Protocols/HTTP/AsImplemented.html>

This web page contains a description of the 1991 HTTP/0.9 protocol, which implements only GET requests and has no content typing.

Historical Perspective

<http://www.w3.org/Protocols/WhyHTTP.html>

This brief web page from 1991, from the author of HTTP, highlights some of the original, minimalist goals of HTTP.

<http://www.w3.org/History.html>

“A Little History of the World Wide Web” gives a short but interesting perspective on some of the early goals and foundations of the World Wide Web and HTTP.

<http://www.w3.org/DesignIssues/Architecture.html>

“Web Architecture from 50,000 Feet” paints a broad, ambitious view of the World Wide Web and the design principles that affect HTTP and related web technologies.

Other World Wide Web Information

<http://www.w3.org>

The World Wide Web Consortium (W3C) is the technology steering team for the Web. The W3C develops interoperable technologies (specifications, guidelines, software, and tools) for the evolving Web. The W3C site is a treasure trove of introductory and detailed documentation about web technologies.

<http://www.ietf.org/rfc/rfc2396.txt>

RFC 2396, “Uniform Resource Identifiers (URI): Generic Syntax,” is the detailed reference for URIs and URLs.

<http://www.ietf.org/rfc/rfc2141.txt>

RFC 2141, “URN Syntax,” is a 1997 specification describing URN syntax.

<http://www.ietf.org/rfc/rfc2046.txt>

RFC 2046, “MIME Part 2: Media Types,” is the second in a suite of five Internet specifications defining the Multipurpose Internet Mail Extensions standard for multimedia content management.

<http://www.wrec.org/Drafts/draft-ietf-wrec-taxonomy-06.txt>

This Internet draft, “Internet Web Replication and Caching Taxonomy,” specifies standard terminology for web architectural components.

URLs and Resources

Think of the Internet as a giant, expanding city, full of places to see and things to do. You and the other residents and tourists of this booming community would use standard naming conventions for the city's vast attractions and services. You'd use street addresses for museums, restaurants, and people's homes. You'd use phone numbers for the fire department, the boss's secretary, and your mother, who says you don't call enough.

Everything has a standardized name, to help sort out the city's resources. Books have ISBN numbers, buses have route numbers, bank accounts have account numbers, and people have social security numbers. Tomorrow you will meet your business partners at gate 31 of the airport. Every morning you take a Red-line train and exit at Kendall Square station.

And because everyone agreed on standards for these different names, we can easily share the city's treasures with each other. You can tell the cab driver to take you to 246 McAllister Street, and he'll know what you mean (even if he takes the long way).

Uniform resource locators (URLs) are the standardized names for the Internet's resources. URLs point to pieces of electronic information, telling you where they are located and how to interact with them.

In this chapter, we'll cover:

- URL syntax and what the various URL components mean and do
- URL shortcuts that many web clients support, including relative URLs and expandomatic URLs
- URL encoding and character rules
- Common URL schemes that support a variety of Internet information systems
- The future of URLs, including uniform resource names (URNs)—a framework to support objects that move from place to place while retaining stable names

Navigating the Internet's Resources

URLs are the resource locations that your browser needs to find information. They let people and applications find, use, and share the billions of data resources on the Internet. URLs are the usual human access point to HTTP and other protocols: a person points a browser at a URL and, behind the scenes, the browser sends the appropriate protocol messages to get the resource that the person wants.

URLs actually are a subset of a more general class of resource identifier called a uniform resource identifier, or URI. URIs are a general concept comprised of two main subsets, URLs and URNs. URLs identify resources by describing where resources are located, whereas URNs (which we'll cover later in this chapter) identify resources by name, regardless of where they currently reside.

The HTTP specification uses the more general concept of URIs as its resource identifiers; in practice, however, HTTP applications deal only with the URL subset of URIs. Throughout this book, we'll sometimes refer to URIs and URLs interchangeably, but we're almost always talking about URLs.

Say you want to fetch the URL `http://www.joes-hardware.com/seasonal/index-fall.html`:

- The first part of the URL (`http`) is the *URL scheme*. The scheme tells a web client *how* to access the resource. In this case, the URL says to use the HTTP protocol.
- The second part of the URL (`www.joes-hardware.com`) is the server location. This tells the web client *where* the resource is hosted.
- The third part of the URL (`/seasonal/index-fall.html`) is the resource path. The path tells *what* particular local resource on the server is being requested.

See Figure 2-1 for an illustration.

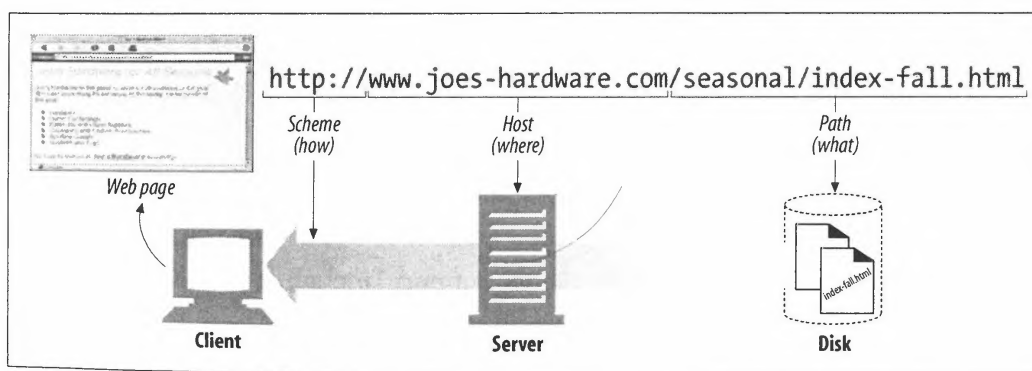


Figure 2-1. How URLs relate to browser, machine, server, and location on the server's filesystem

URLs can direct you to resources available through protocols other than HTTP. They can point you to any resource on the Internet, from a person's email account:

`mailto:president@whitehouse.gov`

to files that are available through other protocols, such as the File Transfer Protocol (FTP):

```
ftp://ftp.lots-o-books.com/pub/complete-price-list.xls
```

to movies hosted off of streaming video servers:

```
rtsp://www.joes-hardware.com:554/interview/cto_video
```

URLs provide a way to uniformly name resources. Most URLs have the same “scheme://server location/path” structure. So, for every resource out there and every way to get those resources, you have a single way to name each resource so that anyone can use that name to find it. However, this wasn’t always the case.

The Dark Days Before URLs

Before the Web and URLs, people relied on a rag-tag assortment of applications to access data distributed throughout the Net. Most people were not lucky enough to have all the right applications or were not savvy and patient enough to use them.

Before URLs came along, if you wanted to share the *complete-catalog.xls* file with a friend, you would have had to say something like this: “Use FTP to connect to *ftp.joes-hardware.com*. Log in as *anonymous*. Then type your username as the password. Change to the *pub* directory. Switch to binary mode. Now download the file named *complete-catalog.xls* to your local filesystem and view it there.”

Today, browsers such as Netscape Navigator and Microsoft Internet Explorer bundle much of this functionality into one convenient package. Using URLs, these applications are able to access many resources in a uniform way, through one interface. Instead of the complicated instructions above, you could just say “Point your browser at *ftp://ftp.lots-o-books.com/pub/complete-catalog.xls*.”

URLs have provided a means for applications to be aware of how to access a resource. In fact, many users are probably unaware of the protocols and access methods their browsers use to get the resources they are requesting.

With web browsers, you no longer need a news reader to read Internet news or an FTP client to access files on FTP servers. You don’t need an electronic mail program to send and receive email messages. URLs have helped to simplify the online world, by allowing the browser to be smart about how to access and handle resources.* Applications can use URLs to simplify access to information.

URLs give you and your browser all you need to find a piece of information. They define the particular resource you want, where it is located, and how to get it.

* Browsers often use other applications to handle specific resources. For example, Internet Explorer launches an email application to handle URLs that identify email resources.

URL Syntax

URLs provide a means of locating any resource on the Internet, but these resources can be accessed by different schemes (e.g., HTTP, FTP, SMTP), and URL syntax varies from scheme to scheme.

Does this mean that each different URL scheme has a radically different syntax? In practice, no. Most URLs adhere to a general URL syntax, and there is significant overlap in the style and syntax between different URL schemes.

Most URL schemes base their URL syntax on this nine-part general format:

```
<scheme>://<user>:<password>@<host>:<port>/<path>;<params>?<query>#<frag>
```

Almost no URLs contain all these components. The three most important parts of a URL are the *scheme*, the *host*, and the *path*. Table 2-1 summarizes the various components.

Table 2-1. General URL components

Component	Description	Default value
scheme	Which protocol to use when accessing a server to get a resource.	None
user	The username some schemes require to access a resource.	anonymous
password	The password that may be included after the username, separated by a colon (:).	<Email address>
host	The hostname or dotted IP address of the server hosting the resource.	None
port	The port number on which the server hosting the resource is listening. Many schemes have default port numbers (the default port number for HTTP is 80).	Scheme-specific
path	The local name for the resource on the server, separated from the previous URL components by a slash (/). The syntax of the path component is server- and scheme-specific. (We will see later in this chapter that a URL's path can be divided into segments, and each segment can have its own components specific to that segment.)	None
params	Used by some schemes to specify input parameters. Params are name/value pairs. A URL can contain multiple params fields, separated from themselves and the rest of the path by semicolons (;).	None
query	Used by some schemes to pass parameters to active applications (such as databases, bulletin boards, search engines, and other Internet gateways). There is no common format for the contents of the query component. It is separated from the rest of the URL by the "?" character.	None
frag	A name for a piece or part of the resource. The frag field is not passed to the server when referencing the object; it is used internally by the client. It is separated from the rest of the URL by the "#" character.	None

For example, consider the URL `http://www.joes-hardware.com:80/index.html`. The scheme is "http", the host is "www.joes-hardware.com", the port is "80", and the path is "/index.html".

Schemes: What Protocol to Use

The *scheme* is really the main identifier of how to access a given resource; it tells the application interpreting the URL what protocol it needs to speak. In our simple HTTP URL, the scheme is simply “http”.

The scheme component must start with an alphabetic character, and it is separated from the rest of the URL by the first “:” character. Scheme names are case-insensitive, so the URLs “http://www.joes-hardware.com” and “HTTP://www.joes-hardware.com” are equivalent.

Hosts and Ports

To find a resource on the Internet, an application needs to know what machine is hosting the resource and where on that machine it can find the server that has access to the desired resource. The *host* and *port* components of the URL provide these two pieces of information.

The host component identifies the host machine on the Internet that has access to the resource. The name can be provided as a hostname, as above (“www.joes-hardware.com”) or as an IP address. For example, the following two URLs point to the same resource—the first refers to the server by its hostname and the second by its IP address:

```
http://www.joes-hardware.com:80/index.html
http://161.58.228.45:80/index.html
```

The port component identifies the network port on which the server is listening. For HTTP, which uses the underlying TCP protocol, the default port is 80.

Usernames and Passwords

More interesting components are the *user* and *password* components. Many servers require a username and password before you can access data through them. FTP servers are a common example of this. Here are a few examples:

```
ftp://ftp.prep.ai.mit.edu/pub/gnu
ftp://anonymous@ftp.prep.ai.mit.edu/pub/gnu
ftp://anonymous:my_passwd@ftp.prep.ai.mit.edu/pub/gnu
http://joe:joespasswd@www.joes-hardware.com/sales_info.txt
```

The first example has no user or password component, just our standard scheme, host, and path. If an application is using a URL scheme that requires a username and password, such as FTP, it generally will insert a default username and password if they aren’t supplied. For example, if you hand your browser an FTP URL without specifying a username and password, it will insert “anonymous” for your username and send a default password (Internet Explorer sends “IEUser”, while Netscape Navigator sends “mozilla”).

The second example shows a username being specified as “anonymous”. This username, combined with the host component, looks just like an email address. The “@” character separates the user and password components from the rest of the URL.

In the third example, both a username (“anonymous”) and password (“my_passwd”) are specified, separated by the “:” character.

Paths

The *path* component of the URL specifies where on the server machine the resource lives. The path often resembles a hierarchical filesystem path. For example:

```
http://www.joes-hardware.com:80/seasonal/index-fall.html
```

The path in this URL is “/seasonal/index-fall.html”, which resembles a filesystem path on a Unix filesystem. The path is the information that the server needs to locate the resource.* The path component for HTTP URLs can be divided into *path segments* separated by “/” characters (again, as in a file path on a Unix filesystem). Each path segment can have its own *params* component.

Parameters

For many schemes, a simple host and path to the object just aren’t enough. Aside from what port the server is listening to and even whether or not you have access to the resource with a username and password, many protocols require more information to work.

Applications interpreting URLs need these protocol parameters to access the resource. Otherwise, the server on the other side might not service the request or, worse yet, might service it wrong. For example, take a protocol like FTP, which has two modes of transfer, binary and text. You wouldn’t want your binary image transferred in text mode, because the binary image could be scrambled.

To give applications the input parameters they need in order to talk to the server correctly, URLs have a *params* component. This component is just a list of name/value pairs in the URL, separated from the rest of the URL (and from each other) by “;” characters. They provide applications with any additional information that they need to access the resource. For example:

```
ftp://prep.ai.mit.edu/pub/gnu?type=d
```

In this example, there is one param, type=d, where the name of the param is “type” and its value is “d”.

* This is a bit of a simplification. In “Virtual Hosting” in Chapter 18, we will see that the path is not always enough information to locate a resource. Sometimes a server needs additional information.

As we mentioned earlier, the path component for HTTP URLs can be broken into path segments. Each segment can have its own params. For example:

```
http://www.joes-hardware.com/hammers;sale=false/index.html;graphics=true
```

In this example there are two path segments, `hammers` and `index.html`. The `hammers` path segment has the param `sale`, and its value is `false`. The `index.html` segment has the param `graphics`, and its value is `true`.

Query Strings

Some resources, such as database services, can be asked questions or queries to narrow down the type of resource being requested.

Let's say Joe's Hardware store maintains a list of unsold inventory in a database and allows the inventory to be queried, to see whether products are in stock. The following URL might be used to query a web database gateway to see if item number 12731 is available:

```
http://www.joes-hardware.com/inventory-check.cgi?item=12731
```

For the most part, this resembles the other URLs we have looked at. What is new is everything to the right of the question mark (`?`). This is called the *query* component. The query component of the URL is passed along to a gateway resource, with the path component of the URL identifying the gateway resource. Basically, gateways can be thought of as access points to other applications (we discuss gateways in detail in Chapter 8).

Figure 2-2 shows an example of a query component being passed to a server that is acting as a gateway to Joe's Hardware's inventory-checking application. The query is checking whether a particular item, 12731, is in inventory in size large and color blue.

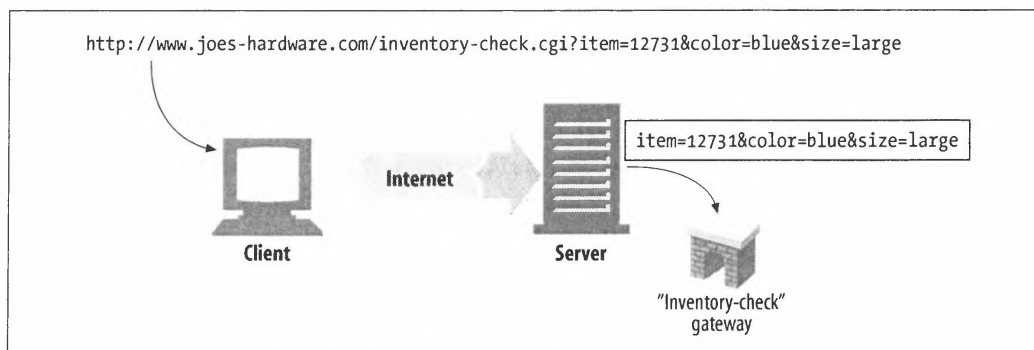


Figure 2-2. The URL query component is sent along to the gateway application

There is no requirement for the format of the query component, except that some characters are illegal, as we'll see later in this chapter. By convention, many gateways

expect the query string to be formatted as a series of “name=value” pairs, separated by “&” characters:

```
http://www.joes-hardware.com/inventory-check.cgi?item=12731&color=blue
```

In this example, there are two name/value pairs in the query component: `item=12731` and `color=blue`.

Fragments

Some resource types, such as HTML, can be divided further than just the resource level. For example, for a single, large text document with sections in it, the URL for the resource would point to the entire text document, but ideally you could specify the sections within the resource.

To allow referencing of parts or fragments of a resource, URLs support a *frag* component to identify pieces within a resource. For example, a URL could point to a particular image or section within an HTML document.

A fragment dangles off the right-hand side of a URL, preceded by a # character. For example:

```
http://www.joes-hardware.com/tools.html#drills
```

In this example, the fragment `drills` references a portion of the `/tools.html` web page located on the Joe’s Hardware web server. The portion is named “drills”.

Because HTTP servers generally deal only with entire objects,* not with fragments of objects, clients don’t pass fragments along to servers (see Figure 2-3). After your browser gets the *entire* resource from the server, it then uses the fragment to display the part of the resource in which you are interested.

URL Shortcuts

Web clients understand and use a few URL shortcuts. Relative URLs are a convenient shorthand for specifying a resource within a resource. Many browsers also support “automatic expansion” of URLs, where the user can type in a key (memorable) part of a URL, and the browser fills in the rest. This is explained in “Expandomatic URLs.”

Relative URLs

URLs come in two flavors: *absolute* and *relative*. So far, we have looked only at absolute URLs. With an absolute URL, you have all the information you need to access a resource.

* In “Range Requests” in Chapter 15, we will see that HTTP agents may request byte ranges of objects. However, in the context of URL fragments, the server sends the entire object and the agent applies the fragment identifier to the resource.

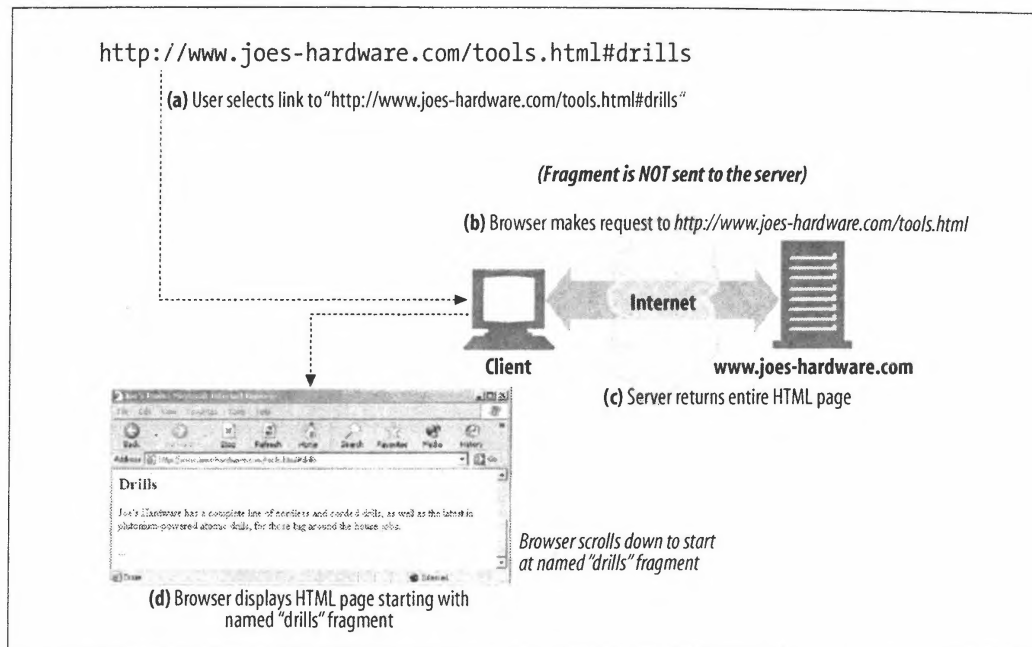


Figure 2-3. The URL fragment is used only by the client, because the server deals with entire objects

On the other hand, relative URLs are incomplete. To get all the information needed to access a resource from a relative URL, you must interpret it relative to another URL, called its *base*.

Relative URLs are a convenient shorthand notation for URLs. If you have ever written HTML by hand, you have probably found them to be a great shortcut. Example 2-1 contains an example HTML document with an embedded relative URL.

Example 2-1. HTML snippet with relative URLs

```
<HTML>
<HEAD><TITLE>Joe's Tools</TITLE></HEAD>
<BODY>
<H1> Tools Page </H1>
<H2> Hammers <H2>
<P> Joe's Hardware Online has the largest selection of <A HREF="./hammers.html">hammers
</A> on earth.
</BODY>
</HTML>
```

In Example 2-1, we have an HTML document for the resource:

<http://www.joes-hardware.com/tools.html>

In the HTML document, there is a hyperlink containing the URL *./hammers.html*. This URL seems incomplete, but it is a legal relative URL. It can be interpreted relative to the URL of the document in which it is found; in this case, relative to the resource */tools.html* on the Joe's Hardware web server.

The abbreviated relative URL syntax lets HTML authors omit from URLs the scheme, host, and other components. These components can be inferred by the *base* URL of the resource they are in. URLs for other resources also can be specified in this shorthand.

In Example 2-1, our base URL is:

```
http://www.joes-hardware.com/tools.html
```

Using this URL as a base, we can infer the missing information. We know the resource is *./hammers.html*, but we don't know the scheme or host. Using the base URL, we can infer that the scheme is *http* and the host is *www.joes-hardware.com*. Figure 2-4 illustrates this.

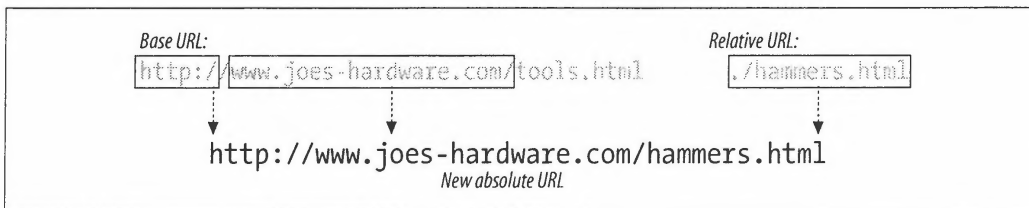


Figure 2-4. Using a base URL

Relative URLs are only fragments or pieces of URLs. Applications that process URLs (such as your browser) need to be able to convert between relative and absolute URLs.

It is also worth noting that relative URLs provide a convenient way to keep a set of resources (such as HTML pages) portable. If you use relative URLs, you can move a set of documents around and still have their links work, because they will be interpreted relative to the new base. This allows for things like mirroring content on other servers.

Base URLs

The first step in the conversion process is to find a base URL. The base URL serves as a point of reference for the relative URL. It can come from a few places:

Explicitly provided in the resource

Some resources explicitly specify the base URL. An HTML document, for example, may include a `<BASE>` HTML tag defining the base URL by which to convert all relative URLs in that HTML document.

Base URL of the encapsulating resource

If a relative URL is found in a resource that does not explicitly specify a base URL, as in Example 2-1, it can use the URL of the resource in which it is embedded as a base (as we did in our example).

No base URL

In some instances, there is no base URL. This often means that you have an absolute URL; however, sometimes you may just have an incomplete or broken URL.

Resolving relative references

Previously, we showed the basic components and syntax of URLs. The next step in converting a relative URL into an absolute one is to break up both the relative and base URLs into their component pieces.

In effect, you are just parsing the URL, but this is often called *decomposing* the URL, because you are breaking it up into its components. Once you have broken the base and relative URLs into their components, you can then apply the algorithm pictured in Figure 2-5 to finish the conversion.

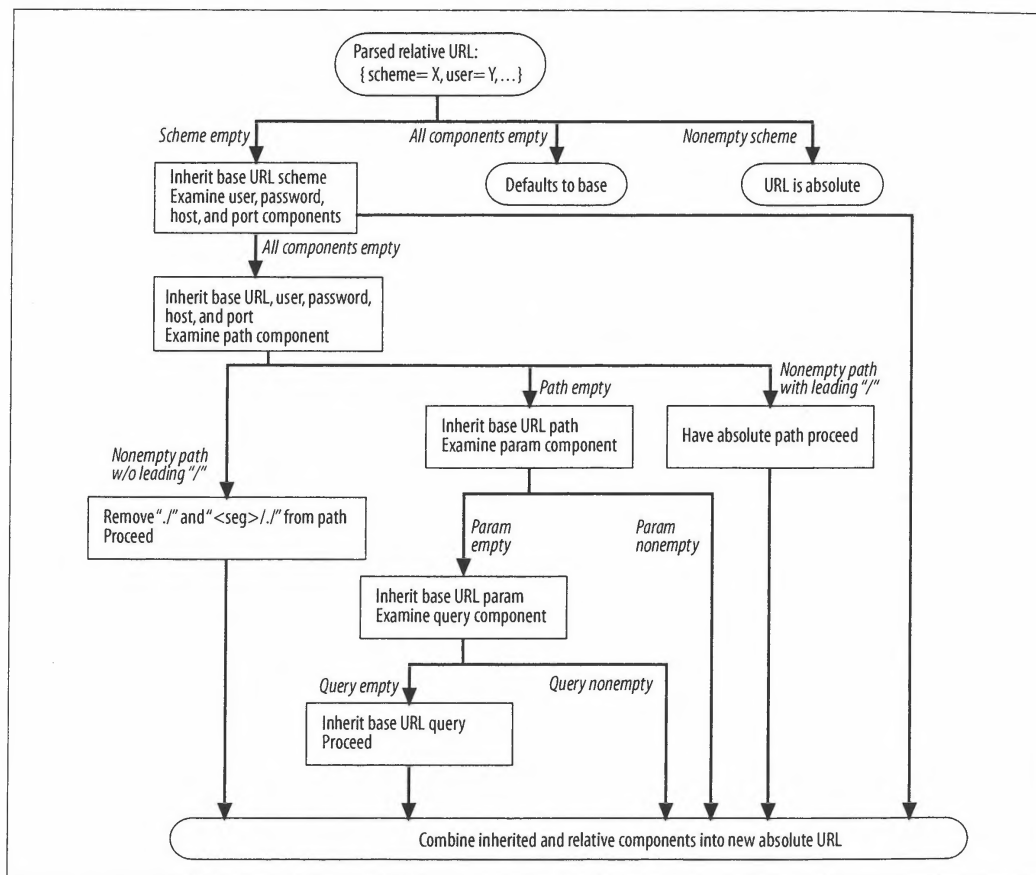


Figure 2-5. Converting relative to absolute URLs

This algorithm converts a relative URL to its absolute form, which can then be used to reference the resource. This algorithm was originally specified in RFC 1808 and later incorporated into RFC 2396.

With our *./hammers.html* example from Example 2-1, we can apply the algorithm depicted in Figure 2-5:

1. Path is *./hammers.html*; base URL is *http://www.joes-hardware.com/tools.html*.
2. Scheme is empty; proceed down left half of chart and inherit the base URL scheme (HTTP).
3. At least one component is non-empty; proceed to bottom, inheriting host and port components.
4. Combining the components we have from the relative URL (path: *./hammers.html*) with what we have inherited (scheme: *http*, host: *www.joes-hardware.com*, port: 80), we get our new absolute URL: *http://www.joes-hardware.com/hammers.html*.

Expandomatic URLs

Some browsers try to expand URLs automatically, either after you submit the URL or while you're typing. This provides users with a shortcut: they don't have to type in the complete URL, because it automatically expands itself.

These "expandomatic" features come in two flavors:

Hostname expansion

In hostname expansion, the browser can often expand the hostname you type in into the full hostname without your help, just by using some simple heuristics.

For example if you type "yahoo" in the address box, your browser can automatically insert "www." and ".com" onto the hostname, creating "www.yahoo.com". Some browsers will try this if they are unable to find a site that matches "yahoo", trying a few expansions before giving up. Browsers apply these simple tricks to save you some time and frustration.

However, these expansion tricks on hostnames can cause problems for other HTTP applications, such as proxies. In Chapter 6, we will discuss these problems in more detail.

History expansion

Another technique that browsers use to save you time typing URLs is to store a history of the URLs that you have visited in the past. As you type in the URL, they can offer you completed choices to select from by matching what you type to the prefixes of the URLs in your history. So, if you were typing in the start of a URL that you had visited previously, such as *http://www.joes-*, your browser could suggest *http://www.joes-hardware.com*. You could then select that instead of typing out the complete URL.

Be aware that URL auto-expansion may behave differently when used with proxies. We discuss this further in “URI Client Auto-Expansion and Hostname Resolution” in Chapter 6.

Shady Characters

URLs were designed to be *portable*. They were also designed to uniformly name all the resources on the Internet, which means that they will be transmitted through various protocols. Because all of these protocols have different mechanisms for transmitting their data, it was important for URLs to be designed so that they could be transmitted safely through any Internet protocol.

Safe transmission means that URLs can be transmitted without the risk of losing information. Some protocols, such as the Simple Mail Transfer Protocol (SMTP) for electronic mail, use transmission methods that can strip off certain characters.* To get around this, URLs are permitted to contain only characters from a relatively small, universally safe alphabet.

In addition to wanting URLs to be transportable by all Internet protocols, designers wanted them to be *readable* by people. So invisible, nonprinting characters also are prohibited in URLs, even though these characters may pass through mailers and otherwise be portable.†

To complicate matters further, URLs also need to be *complete*. URL designers realized there would be times when people would want URLs to contain binary data or characters outside of the universally safe alphabet. So, an escape mechanism was added, allowing unsafe characters to be encoded into safe characters for transport.

This section summarizes the universal alphabet and encoding rules for URLs.

The URL Character Set

Default computer system character sets often have an Anglocentric bias. Historically, many computer applications have used the US-ASCII character set. US-ASCII uses 7 bits to represent most keys available on an English typewriter and a few nonprinting control characters for text formatting and hardware signalling.

US-ASCII is very portable, due to its long legacy. But while it’s convenient to citizens of the U.S., it doesn’t support the inflected characters common in European languages or the hundreds of non-Romanic languages read by billions of people around the world.

* This is caused by the use of a 7-bit encoding for messages; this can strip off information if the source is encoded in 8 bits or more.

† Nonprinting characters include whitespace (note that RFC 2396 recommends that applications ignore whitespace).

Furthermore, some URLs may need to contain arbitrary binary data. Recognizing the need for completeness, the URL designers have incorporated *escape sequences*. Escape sequences allow the encoding of arbitrary character values or data using a restricted subset of the US-ASCII character set, yielding portability and completeness.

Encoding Mechanisms

To get around the limitations of a safe character set representation, an encoding scheme was devised to represent characters in a URL that are not safe. The encoding simply represents the unsafe character by an “escape” notation, consisting of a percent sign (%) followed by two hexadecimal digits that represent the ASCII code of the character.

Table 2-2 shows a few examples.

Table 2-2. Some encoded character examples

Character	ASCII code	Example URL
~	126 (0x7E)	http://www.joes-hardware.com/%7Ejoe
SPACE	32 (0x20)	http://www.joes-hardware.com/more%20tools.html
%	37 (0x25)	http://www.joes-hardware.com/100%25satisfaction.html

Character Restrictions

Several characters have been reserved to have special meaning inside of a URL. Others are not in the defined US-ASCII printable set. And still others are known to confuse some Internet gateways and protocols, so their use is discouraged.

Table 2-3 lists characters that should be encoded in a URL before you use them for anything other than their reserved purposes.

Table 2-3. Reserved and restricted characters

Character	Reservation/Restriction
%	Reserved as escape token for encoded characters
/	Reserved for delimiting splitting up path segments in the path component
.	Reserved in the path component
..	Reserved in the path component
#	Reserved as the fragment delimiter
?	Reserved as the query-string delimiter
;	Reserved as the params delimiter
:	Reserved to delimit the scheme, user/password, and host/port components
\$_+.	Reserved
@&=	Reserved because they have special meaning in the context of some schemes

Table 2-3. Reserved and restricted characters (continued)

Character	Reservation/Restriction
{ } \ ^ ~ [] ' < > "	Restricted because of unsafe handling by various transport agents, such as gateways
0x00–0x1F, 0x7F	Unsafe; should be encoded because these characters often have meaning outside the scope of the URL, such as delimiting the URL itself in a document (e.g., "http://www.joes-hardware.com")
> 0x7F	Restricted; characters within these hex ranges fall within the nonprintable section of the US-ASCII character set
	Restricted; characters whose hex values fall within this range do not fall within the 7-bit range of the US-ASCII character set

A Bit More

You might be wondering why nothing bad has happened when you have used characters that are unsafe. For instance, you can visit Joe’s home page at:

<http://www.joes-hardware.com/~joe>

and not encode the “~” character. For some transport protocols this is not an issue, but it is still unwise for application developers not to encode unsafe characters.

Applications need to walk a fine line. It is best for client applications to convert any unsafe or restricted characters before sending any URL to any other application.* Once all the unsafe characters have been encoded, the URL is in a *canonical form* that can be shared between applications; there is no need to worry about the other application getting confused by any of the characters’ special meanings.

The original application that gets the URL from the user is best fit to determine which characters need to be encoded. Because each component of the URL may have its own safe/unsafe characters, and which characters are safe/unsafe is scheme-dependent, only the application receiving the URL from the user really is in a position to determine what needs to be encoded.

Of course, the other extreme is for the application to encode all characters. While this is not recommended, there is no hard and fast rule against encoding characters that are considered safe already; however, in practice this can lead to odd and broken behavior, because some applications may assume that safe characters will not be encoded.

Sometimes, malicious folks encode extra characters in an attempt to get around applications that are doing pattern matching on URLs—for example, web filtering applications. Encoding safe URL components can cause pattern-matching applications to fail to recognize the patterns for which they are searching. In general, applications interpreting URLs must decode the URLs before processing them.

* Here we are specifically talking about client applications, not other HTTP intermediaries, like proxies. In “In-Flight URI Modification” in Chapter 6, we discuss some of the problems that can arise when proxies or other intermediary HTTP applications attempt to change (e.g., encode) URLs on the behalf of a client.

Some URL components, such as the scheme, need to be recognized readily and are required to start with an alphabetic character. Refer back to “URL Syntax” for more guidelines on the use of reserved and unsafe characters within different URL components.*

A Sea of Schemes

In this section, we’ll take a look at the more common scheme formats on the Web. Appendix A gives a fairly exhaustive list of schemes and references to their individual documentation.

Table 2-4 summarizes some of the most popular schemes. Reviewing “URL Syntax” will make the syntax portion of the table a little more familiar.

Table 2-4. Common scheme formats

Scheme	Description
http	The Hypertext Transfer Protocol scheme conforms to the general URL format, except that there is no username or password. The port defaults to 80 if omitted. Basic form: <i>http://<host>:<port>/<path>?<query>#<frag></i> Examples: <i>http://www.joes-hardware.com/index.html</i> <i>http://www.joes-hardware.com:80/index.html</i>
https	The https scheme is a twin to the http scheme. The only difference is that the https scheme uses Netscape’s Secure Sockets Layer (SSL), which provides end-to-end encryption of HTTP connections. Its syntax is identical to that of HTTP, with a default port of 443. Basic form: <i>https://<host>:<port>/<path>?<query>#<frag></i> Example: <i>https://www.joes-hardware.com/secure.html</i>
mailto	Mailto URLs refer to email addresses. Because email behaves differently from other schemes (it does not refer to objects that can be accessed directly), the format of a mailto URL differs from that of the standard URL. The syntax for Internet email addresses is documented in Internet RFC 822. Basic form: <i>mailto:<RFC-822-addr-spec></i> Example: <i>mailto:joe@joes-hardware.com</i>

* Table 2-3 lists reserved characters for the various URL components. In general, encoding should be limited to those characters that are unsafe for transport.

Table 2-4. Common scheme formats (continued)

Scheme	Description
ftp	File Transfer Protocol URLs can be used to download and upload files on an FTP server and to obtain listings of the contents of a directory structure on an FTP server. FTP has been around since before the advent of the Web and URLs. Web applications have assimilated FTP as a data-access scheme. The URL syntax follows the general form. Basic form: <pre>ftp://<user>:<password>@<host>:<port>/<path>;<params></pre> Example: <pre>ftp://anonymous:joe%40joes-hardware.com@prep.ai.mit.edu:21/pub/gnu/</pre>
rtsp, rtspu	RTSP URLs are identifiers for audio and video media resources that can be retrieved through the Real Time Streaming Protocol. The “u” in the rtspu scheme denotes that the UDP protocol is used to retrieve the resource. Basic forms: <pre>rtsp://<user>:<password>@<host>:<port>/<path></pre> <pre>rtspu://<user>:<password>@<host>:<port>/<path></pre> Example: <pre>rtsp://www.joes-hardware.com:554/interview/cto_video</pre>
file	The file scheme denotes files directly accessible on a given host machine (by local disk, a network filesystem, or some other file-sharing system). The fields follow the general form. If the host is omitted, it defaults to the local host from which the URL is being used. Basic form: <pre>file://<host>/<path></pre> Example: <pre>file://OFFICE-FS/policies/casual-fridays.doc</pre>
news	The news scheme is used to access specific articles or newsgroups, as defined by RFC 1036. It has the unusual property that a news URL in itself does not contain sufficient information to locate the resource. The news URL is missing information about where to acquire the resource—no hostname or machine name is supplied. It is the interpreting application’s job to acquire this information from the user. For example, in your Netscape browser, under the Options menu, you can specify your NNTP (news) server. This tells your browser what server to use when it has a news URL. News resources can be accessed from multiple servers. They are said to be location-independent, as they are not dependent on any one source for access. The “@” character is reserved within a news URL and is used to distinguish between news URLs that refer to newsgroups and news URLs that refer to specific news articles. Basic forms: <pre>news:<newsgroup></pre> <pre>news:<news-article-id></pre> Example: <pre>news:rec.arts.startrek</pre>

Table 2-4. Common scheme formats (continued)

Scheme	Description
telnet	The telnet scheme is used to access interactive services. It does not represent an object per se, but an interactive application (resource) accessible via the telnet protocol. Basic form: <code>telnet://<user>:<password>@<host>:<port>/</code> Example: <code>telnet://slurp:webhound@joes-hardware.com:23/</code>

The Future

URLs are a powerful tool. Their design allows them to name all existing objects and easily encompass new formats. They provide a uniform naming mechanism that can be shared between Internet protocols.

However, they are not perfect. URLs are really addresses, not true names. This means that a URL tells you where something is located, for the moment. It provides you with the name of a specific server on a specific port, where you can find the resource. The downfall of this scheme is that if the resource is moved, the URL is no longer valid. And at that point, it provides no way to locate the object.

What would be ideal is if you had the real name of an object, which you could use to look up that object regardless of its location. As with a person, given the name of the resource and a few other facts, you could track down that resource, regardless of where it moved.

The Internet Engineering Task Force (IETF) has been working on a new standard, uniform resource names (URNs), for some time now, to address just this issue. URNs provide a stable name for an object, regardless of where that object moves (either inside a web server or across web servers).

Persistent uniform resource locators (PURLs) are an example of how URN functionality can be achieved using URLs. The concept is to introduce another level of indirection in looking up a resource, using an intermediary *resource locator* server that catalogues and tracks the actual URL of a resource. A client can request a persistent URL from the locator, which can then respond with a resource that redirects the client to the actual and current URL for the resource (see Figure 2-6). For more information on PURLs, visit <http://purl.oclc.org>.

If Not Now, When?

The ideas behind URNs have been around for some time. Indeed, if you look at the publication dates for some of their specifications, you might ask yourself why they have yet to be adopted.

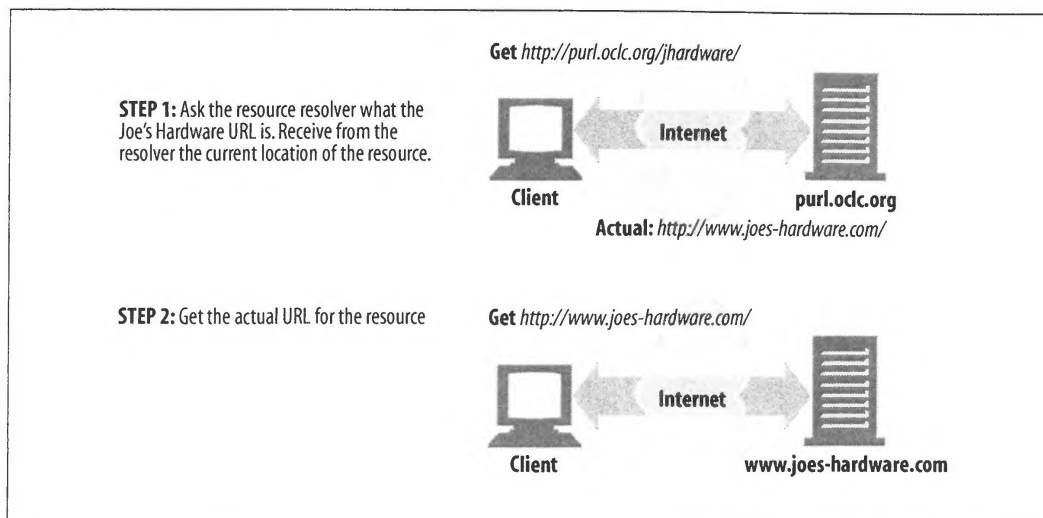


Figure 2-6. PURLs use a resource locator server to name the current location of a resource

The change from URLs to URNs is an enormous task. Standardization is a slow process, often for good reason. Support for URNs will require many changes—consensus from the standards bodies, modifications to various HTTP applications, etc. A tremendous amount of critical mass is required to make such changes, and unfortunately (or perhaps fortunately), there is so much momentum behind URLs that it will be some time before all the stars align to make such a conversion possible.

Throughout the explosive growth of the Web, Internet users—everyone from computer scientists to the average Internet user—have been taught to use URLs. While they suffer from clumsy syntax (for the novice) and persistence problems, people have learned how to use them and how to deal with their drawbacks. URLs have some limitations, but they're not the web development community's most pressing problem.

Currently, and for the foreseeable future, URLs are the way to name resources on the Internet. They are everywhere, and they have proven to be a very important part of the Web's success. It will be a while before any other naming scheme unseats URLs. However, URLs do have their limitations, and it is likely that new standards (possibly URNs) will emerge and be deployed to address some of these limitations.

For More Information

For more information on URLs, refer to:

<http://www.w3.org/Addressing/>

The W3C web page about naming and addressing URIs and URLs.

<http://www.ietf.org/rfc/rfc1738>

RFC 1738, "Uniform Resource Locators (URL)," by T. Berners-Lee, L. Masinter, and M. McCahill.

<http://www.ietf.org/rfc/rfc2396.txt>

RFC 2396, "Uniform Resource Identifiers (URI): Generic Syntax," by T. Berners-Lee, R. Fielding, and L. Masinter.

<http://www.ietf.org/rfc/rfc2141.txt>

RFC 2141, "URN Syntax," by R. Moats.

<http://purl.oclc.org>

The persistent uniform resource locator web site.

<http://www.ietf.org/rfc/rfc1808.txt>

RFC 1808, "Relative Uniform Resource Locators," by R. Fielding.

CHAPTER 5

Web Servers

Web servers dish out billions of web pages a day. They tell you the weather, load up your online shopping carts, and let you find long-lost high-school buddies. Web servers are the workhorses of the World Wide Web. In this chapter, we:

- Survey the many different types of software and hardware web servers.
- Describe how to write a simple diagnostic web server in Perl.
- Explain how web servers process HTTP transactions, step by step.

Where it helps to make things concrete, our examples use the Apache web server and its configuration options.

Web Servers Come in All Shapes and Sizes

A web server processes HTTP requests and serves responses. The term “web server” can refer either to web server software or to the particular device or computer dedicated to serving the web pages.

Web servers come in all flavors, shapes, and sizes. There are trivial 10-line Perl script web servers, 50-MB secure commerce engines, and tiny servers-on-a-card. But whatever the functional differences, all web servers receive HTTP requests for resources and serve content back to the clients (look back to Figure 1-5).

Web Server Implementations

Web servers implement HTTP and the related TCP connection handling. They also manage the resources served by the web server and provide administrative features to configure, control, and enhance the web server.

The web server logic implements the HTTP protocol, manages web resources, and provides web server administrative capabilities. The web server logic shares responsibilities for managing TCP connections with the operating system. The underlying

operating system manages the hardware details of the underlying computer system and provides TCP/IP network support, filesystems to hold web resources, and process management to control current computing activities.

Web servers are available in many forms:

- You can install and run general-purpose software web servers on standard computer systems.
- If you don't want the hassle of installing software, you can purchase a web server appliance, in which the software comes preinstalled and preconfigured on a computer, often in a snazzy-looking chassis.
- Given the miracles of microprocessors, some companies even offer embedded web servers implemented in a small number of computer chips, making them perfect administration consoles for consumer devices.

Let's look at each of those types of implementations.

General-Purpose Software Web Servers

General-purpose software web servers run on standard, network-enabled computer systems. You can choose open source software (such as Apache or W3C's Jigsaw) or commercial software (such as Microsoft's and iPlanet's web servers). Web server software is available for just about every computer and operating system.

While there are tens of thousands of different kinds of web server programs (including custom-crafted, special-purpose web servers), most web server software comes from a small number of organizations.

In February 2002, the Netcraft survey (<http://www.netcraft.com/survey/>) showed three vendors dominating the public Internet web server market (see Figure 5-1):

- The free Apache software powers nearly 60% of all Internet web servers.
- Microsoft web server makes up another 30%.
- Sun iPlanet servers comprise another 3%.

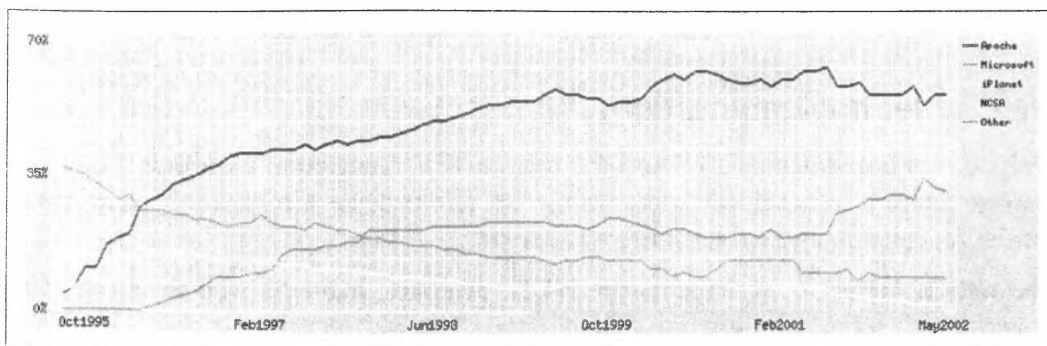


Figure 5-1. Web server market share as estimated by Netcraft's automated survey

Take these numbers with a few grains of salt, however, as the Netcraft survey is commonly believed to exaggerate the dominance of Apache software. First, the survey counts servers independent of server popularity. Proxy server access studies from large ISPs suggest that the amount of pages served from Apache servers is much less than 60% but still exceeds Microsoft and Sun iPlanet. Additionally, it is anecdotally believed that Microsoft and iPlanet servers are more popular than Apache inside corporate enterprises.

Web Server Appliances

Web server appliances are prepackaged software/hardware solutions. The vendor preinstalls a software server onto a vendor-chosen computer platform and preconfigures the software. Some examples of web server appliances include:

- Sun/Cobalt RaQ web appliances (<http://www.cobalt.com>)
- Toshiba Magnia SG10 (<http://www.toshiba.com>)
- IBM Whistle web server appliance (<http://www.whistle.com>)

Appliance solutions remove the need to install and configure software and often greatly simplify administration. However, the web server often is less flexible and feature-rich, and the server hardware is not easily repurposeable or upgradable.

Embedded Web Servers

Embedded servers are tiny web servers intended to be embedded into consumer products (e.g., printers or home appliances). Embedded web servers allow users to administer their consumer devices using a convenient web browser interface.

Some embedded web servers can even be implemented in less than one square inch, but they usually offer a minimal feature set. Two examples of very small embedded web servers are:

- IPic match-head sized web server (<http://www-ccs.cs.umass.edu/~shri/iPic.html>)
- NetMedia SitePlayer SP1 Ethernet Web Server (<http://www.siteplayer.com>)

A Minimal Perl Web Server

If you want to build a full-featured HTTP server, you have some work to do. The core of the Apache web server has over 50,000 lines of code, and optional processing modules make that number much bigger.

All this software is needed to support HTTP/1.1 features: rich resource support, virtual hosting, access control, logging, configuration, monitoring, and performance features. That said, you can create a minimally functional HTTP server in under 30 lines of Perl. Let's take a look.

Example 5-1 shows a tiny Perl program called *type-o-serve*. This program is a useful diagnostic tool for testing interactions with clients and proxies. Like any web server, *type-o-serve* waits for an HTTP connection. As soon as *type-o-serve* gets the request message, it prints the message on the screen; then it waits for you to type (or paste) in a response message, which is sent back to the client. This way, *type-o-serve* pretends to be a web server, records the exact HTTP request messages, and allows you to send back any HTTP response message.

This simple *type-o-serve* utility doesn't implement most HTTP functionality, but it is a useful tool to generate server response messages the same way you can use Telnet to generate client request messages (refer back to Example 5-1). You can download the *type-o-serve* program from <http://www.http-guide.com/tools/type-o-serve.pl>.

Example 5-1. type-o-serve—a minimal Perl web server used for HTTP debugging

```
#!/usr/bin/perl

use Socket;
use Carp;
use FileHandle;

# (1) use port 8080 by default, unless overridden on command line
$port = (@ARGV ? $ARGV[0] : 8080);

# (2) create local TCP socket and set it to listen for connections
$proto = getprotobyname('tcp');
socket(S, PF_INET, SOCK_STREAM, $proto) || die;
setsockopt(S, SOL_SOCKET, SO_REUSEADDR, pack("l", 1)) || die;
bind(S, sockaddr_in($port, INADDR_ANY)) || die;
listen(S, SOMAXCONN) || die;

# (3) print a startup message
printf("    <<<Type-O-Serve Accepting on Port %d>>>\n\n", $port);

while (1)
{
    # (4) wait for a connection C
    $cport_caddr = accept(C, S);
    ($cport, $caddr) = sockaddr_in($cport_caddr);
    C->autoflush(1);

    # (5) print who the connection is from
    $cname = gethostbyaddr($caddr, AF_INET);
    printf("    <<<Request From '%s'>>>\n", $cname);

    # (6) read request msg until blank line, and print on screen
    while ($line = <C>)
    {
        print $line;
        if ($line =~ /\^\r/) { last; }
    }
}
```

Example 5-1. *type-o-serve*—a minimal Perl web server used for HTTP debugging (continued)

```
# (7) prompt for response message, and input response lines,
#     sending response lines to client, until solitary "."
printf("    <<<Type Response Followed by '.'>>>\n");

while ($line = <STDIN>)
{
    $line =~ s/\r//;
    $line =~ s/\n//;
    if ($line =~ /^\.\/) { last; }
    print C $line . "\r\n";
}
close(C);
}
```

Figure 5-2 shows how the administrator of Joe's Hardware store might use *type-o-serve* to test HTTP communication:

- First, the administrator starts the *type-o-serve* diagnostic server, listening on a particular port. Because Joe's Hardware store already has a production web server listening on port 80, the administrator starts the *type-o-serve* server on port 8080 (you can pick any unused port) with this command line:

```
% type-o-serve.pl 8080
```
- Once *type-o-serve* is running, you can point a browser to this web server. In Figure 5-2, we browse to <http://www.joes-hardware.com:8080/foo/bar/blah.txt>.
- The *type-o-serve* program receives the HTTP request message from the browser and prints the contents of the HTTP request message on screen. The *type-o-serve* diagnostic tool then waits for the user to type in a simple response message, followed by a period on a blank line.
- *type-o-serve* sends the HTTP response message back to the browser, and the browser displays the body of the response message.

What Real Web Servers Do

The Perl server we showed in Example 5-1 is a trivial example web server. State-of-the-art commercial web servers are much more complicated, but they do perform several common tasks, as shown in Figure 5-3:

1. Set up connection—accept a client connection, or close if the client is unwanted.
2. Receive request—read an HTTP request message from the network.
3. Process request—interpret the request message and take action.
4. Access resource—access the resource specified in the message.
5. Construct response—create the HTTP response message with the right headers.
6. Send response—send the response back to the client.
7. Log transaction—place notes about the completed transaction in a log file.

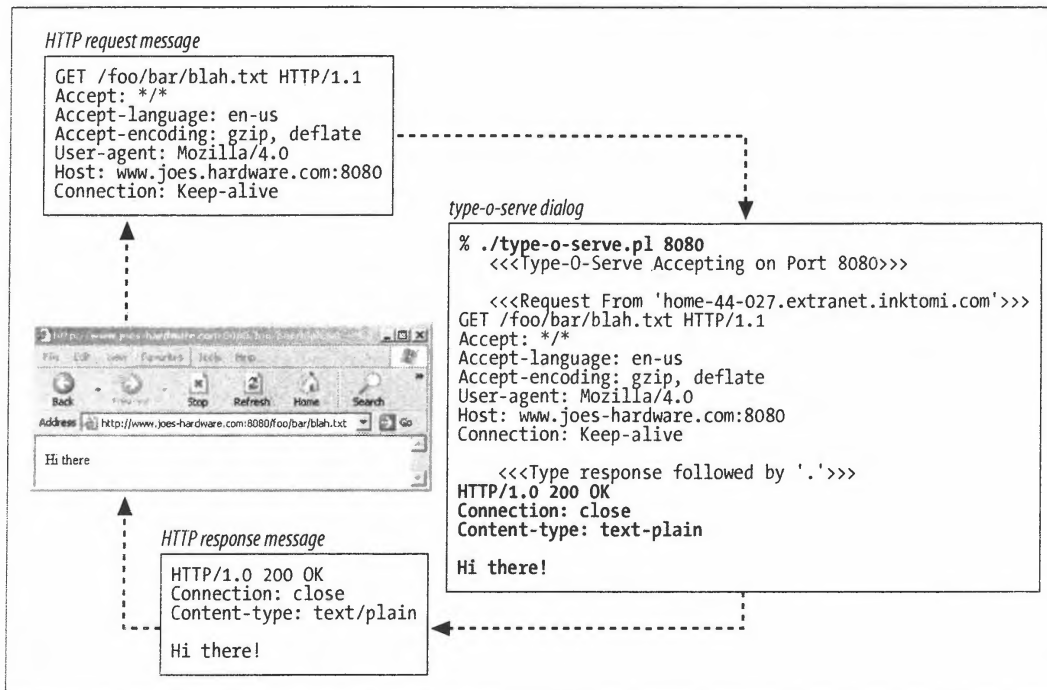


Figure 5-2. The type-o-serve utility lets you type in server responses to send back to clients

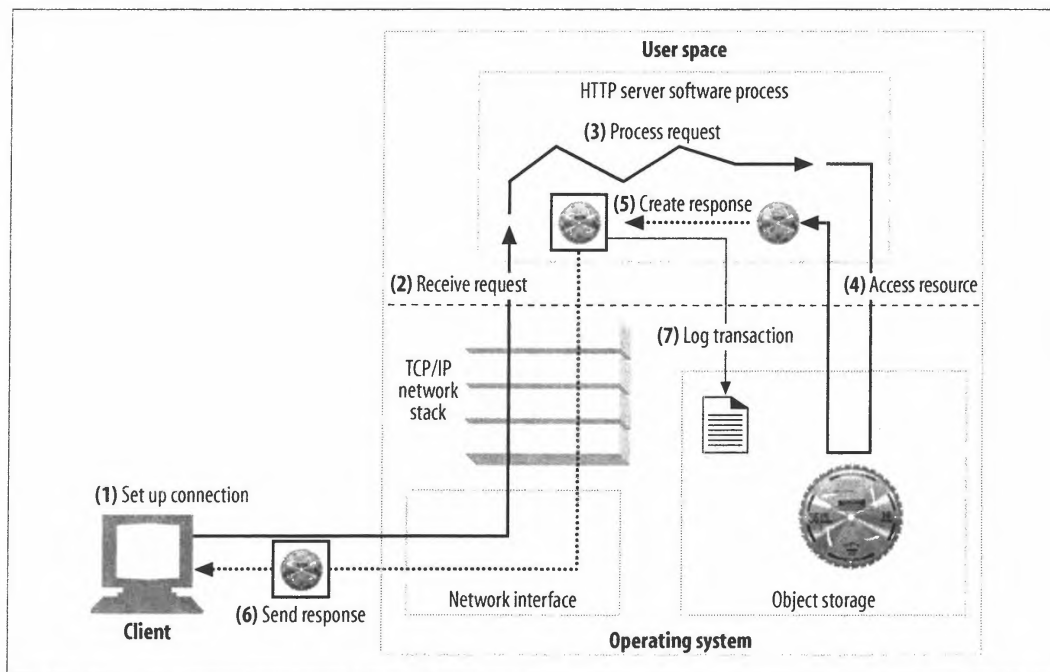


Figure 5-3. Steps of a basic web server request

The next seven sections highlight how web servers perform these basic tasks.

Step 3: Processing Requests

Once the web server has received a request, it can process the request using the method, resource, headers, and optional body.

Some methods (e.g., POST) require entity body data in the request message. Other methods (e.g., OPTIONS) allow a request body but don't require one. A few methods (e.g., GET) forbid entity body data in request messages.

We won't talk about request processing here, because it's the subject of most of the chapters in the rest of this book!

Step 4: Mapping and Accessing Resources

Web servers are resource servers. They deliver precreated content, such as HTML pages or JPEG images, as well as dynamic content from resource-generating applications running on the servers.

Before the web server can deliver content to the client, it needs to identify the source of the content, by mapping the URI from the request message to the proper content or content generator on the web server.

Docroots

Web servers support different kinds of resource mapping, but the simplest form of resource mapping uses the request URI to name a file in the web server's filesystem. Typically, a special folder in the web server filesystem is reserved for web content. This folder is called the *document root*, or *docroot*. The web server takes the URI from the request message and appends it to the document root.

In Figure 5-8, a request arrives for */specials/saw-blade.gif*. The web server in this example has document root */usr/local/httpd/files*. The web server returns the file */usr/local/httpd/files/specials/saw-blade.gif*.

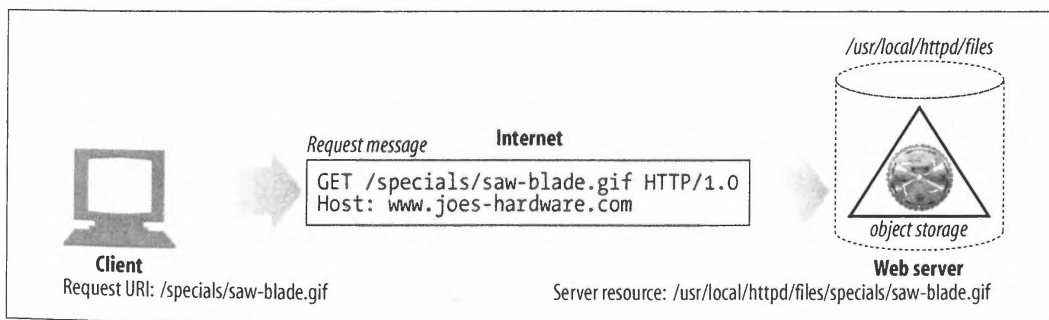


Figure 5-8. Mapping request URI to local web server resource

To set the document root for an Apache web server, add a *DocumentRoot* line to the *httpd.conf* configuration file:

```
DocumentRoot /usr/local/httpd/files
```

Servers are careful not to let relative URLs back up out of a docroot and expose other parts of the filesystem. For example, most mature web servers will not permit this URI to see files above the Joe's Hardware document root:

```
http://www.joes-hardware.com/../../
```

Virtually hosted docroots

Virtually hosted web servers host multiple web sites on the same web server, giving each site its own distinct document root on the server. A virtually hosted web server identifies the correct document root to use from the IP address or hostname in the URI or the Host header. This way, two web sites hosted on the same web server can have completely distinct content, even if the request URIs are identical.

In Figure 5-9, the server hosts two sites: *www.joes-hardware.com* and *www.marys-antiques.com*. The server can distinguish the web sites using the HTTP Host header, or from distinct IP addresses.

- When request A arrives, the server fetches the file for */docs/joe/index.html*.
- When request B arrives, the server fetches the file for */docs/mary/index.html*.

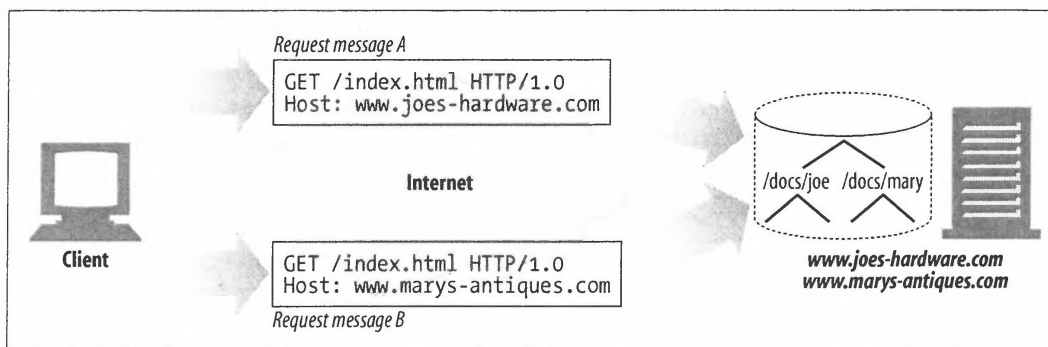


Figure 5-9. Different docroots for virtually hosted requests

Configuring virtually hosted docroots is simple for most web servers. For the popular Apache web server, you need to configure a *VirtualHost* block for each virtual web site, and include the *DocumentRoot* for each virtual server (Example 5-3).

Example 5-3. Apache web server virtual host docroot configuration

```
<VirtualHost www.joes-hardware.com>  
  ServerName www.joes-hardware.com  
  DocumentRoot /docs/joe
```

Example 5-3. Apache web server virtual host docroot configuration (continued)

```
TransferLog /logs/joe.access_log
ErrorLog /logs/joe.error_log
</VirtualHost>

<VirtualHost www.marys-antiques.com>
  ServerName www.marys-antiques.com
  DocumentRoot /docs/mary
  TransferLog /logs/mary.access_log
  ErrorLog /logs/mary.error_log
</VirtualHost>

...
```

Look forward to “Virtual Hosting” in Chapter 18 for much more detail about virtual hosting.

User home directory docroots

Another common use of docroots gives people private web sites on a web server. A typical convention maps URIs whose paths begin with a slash and tilde (/~) followed by a username to a private document root for that user. The private docroot is often the folder called *public_html* inside that user’s home directory, but it can be configured differently (Figure 5-10).

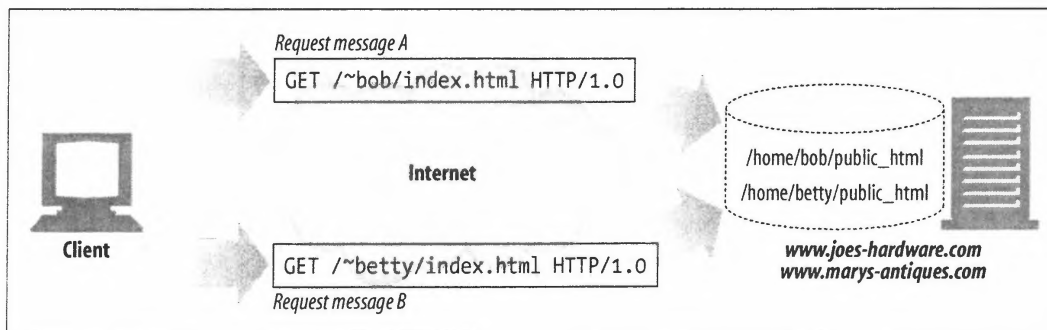


Figure 5-10. Different docroots for different users

Directory Listings

A web server can receive requests for directory URLs, where the path resolves to a directory, not a file. Most web servers can be configured to take a few different actions when a client requests a directory URL:

- Return an error.
- Return a special, default, “index file” instead of the directory.
- Scan the directory, and return an HTML page containing the contents.

Most web servers look for a file named *index.html* or *index.htm* inside a directory to represent that directory. If a user requests a URL for a directory and the directory

contains a file named *index.html* (or *index.htm*), the server will return the contents of that file.

In the Apache web server, you can configure the set of filenames that will be interpreted as default directory files using the *DirectoryIndex* configuration directive. The *DirectoryIndex* directive lists all filenames that serve as directory index files, in preferred order. The following configuration line causes Apache to search a directory for any of the listed files in response to a directory URL request:

```
DirectoryIndex index.html index.htm home.html home.htm index.cgi
```

If no default index file is present when a user requests a directory URI, and if directory indexes are not disabled, many web servers automatically return an HTML file listing the files in that directory, and the sizes and modification dates of each file, including URI links to each file. This file listing can be convenient, but it also allows nosy people to find files on a web server that they might not normally find.

You can disable the automatic generation of directory index files with the Apache directive:

```
Options -Indexes
```

Dynamic Content Resource Mapping

Web servers also can map URIs to dynamic resources—that is, to programs that generate content on demand (Figure 5-11). In fact, a whole class of web servers called *application servers* connect web servers to sophisticated backend applications. The web server needs to be able to tell when a resource is a dynamic resource, where the dynamic content generator program is located, and how to run the program. Most web servers provide basic mechanisms to identify and map dynamic resources.

Apache lets you map URI pathname components into executable program directories. When a server receives a request for a URI with an executable path component, it attempts to execute a program in a corresponding server directory. For example, the following Apache configuration directive specifies that all URIs whose paths begin with */cgi-bin/* should execute corresponding programs found in the directory */usr/local/etc/httpd/cgi-programs/*:

```
ScriptAlias /cgi-bin/ /usr/local/etc/httpd/cgi-programs/
```

Apache also lets you mark executable files with a special file extension. This way, executable scripts can be placed in any directory. The following Apache configuration directive specifies that all web resources ending in *.cgi* should be executed:

```
AddHandler cgi-script .cgi
```

CGI is an early, simple, and popular interface for executing server-side applications. Modern application servers have more powerful and efficient server-side dynamic content support, including Microsoft's Active Server Pages and Java servlets.

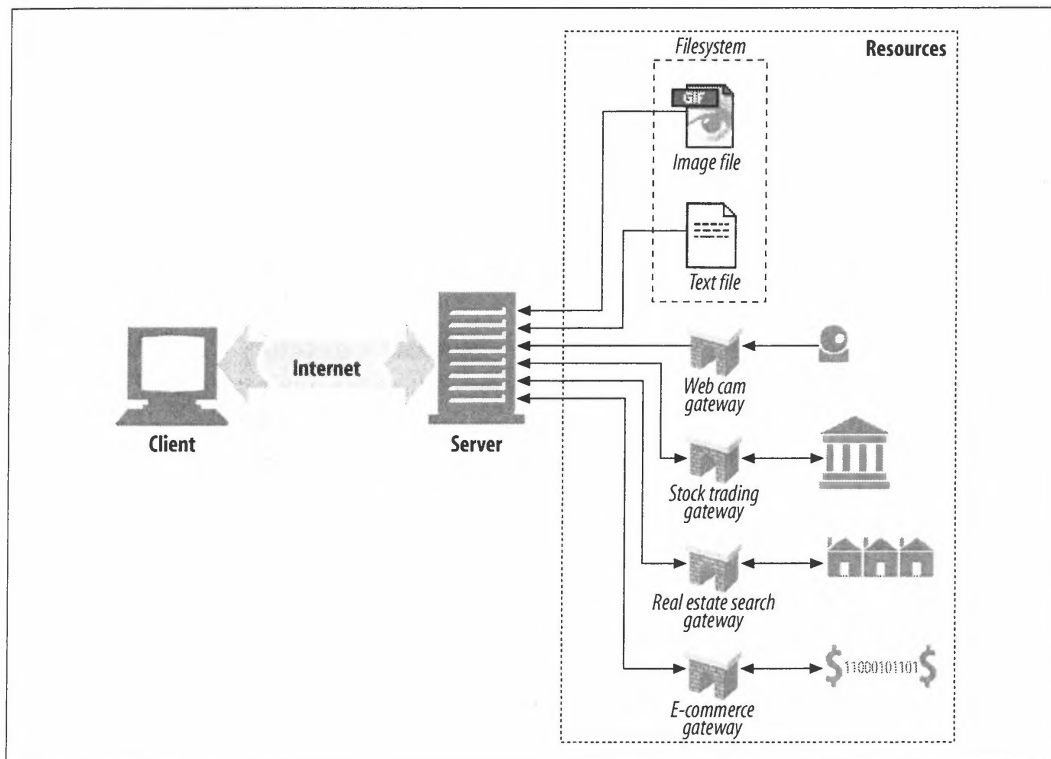


Figure 5-11. A web server can serve static resources as well as dynamic resources

Server-Side Includes (SSI)

Many web servers also provide support for server-side includes. If a resource is flagged as containing server-side includes, the server processes the resource contents before sending them to the client.

The contents are scanned for certain special patterns (often contained inside special HTML comments), which can be variable names or embedded scripts. The special patterns are replaced with the values of variables or the output of executable scripts. This is an easy way to create dynamic content.

Access Controls

Web servers also can assign access controls to particular resources. When a request arrives for an access-controlled resource, the web server can control access based on the IP address of the client, or it can issue a password challenge to get access to the resource.

Refer to Chapter 12 for more information about HTTP authentication.

Step 5: Building Responses

Once the web server has identified the resource, it performs the action described in the request method and returns the response message. The response message contains a response status code, response headers, and a response body if one was generated. HTTP response codes were detailed in “Status Codes” in Chapter 3.

Response Entities

If the transaction generated a response body, the content is sent back with the response message. If there was a body, the response message usually contains:

- A Content-Type header, describing the MIME type of the response body
- A Content-Length header, describing the size of the response body
- The actual message body content

MIME Typing

The web server is responsible for determining the MIME type of the response body. There are many ways to configure servers to associate MIME types with resources:

mime.types

The web server can use the extension of the filename to indicate MIME type. The web server scans a file containing MIME types for each extension to compute the MIME type for each resource. This extension-based type association is the most common; it is illustrated in Figure 5-12.

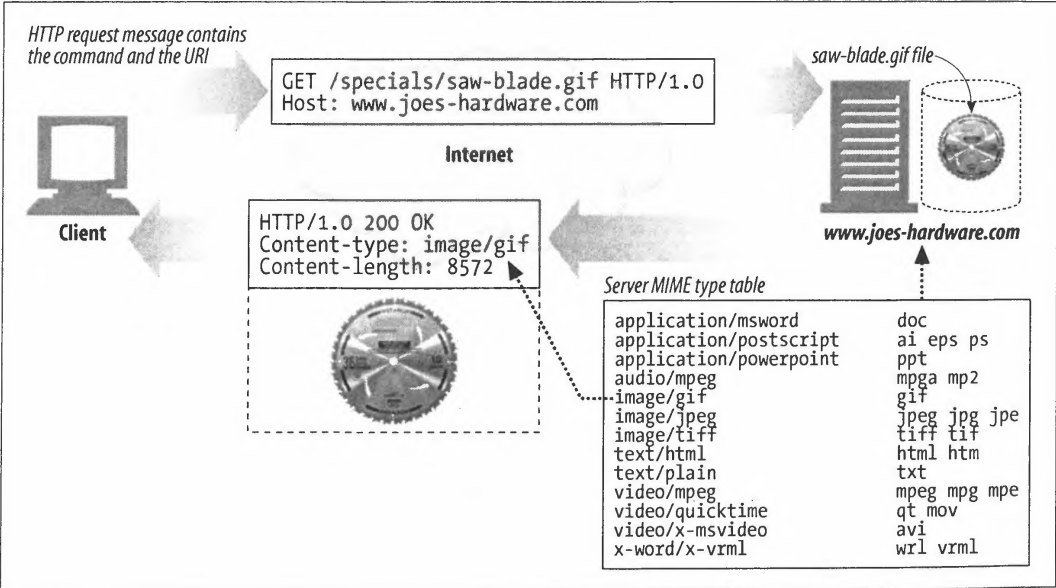


Figure 5-12. A web server uses MIME types file to set outgoing Content-Type of resources

Magic typing

The Apache web server can scan the contents of each resource and pattern-match the content against a table of known patterns (called the *magic* file) to determine the MIME type for each file. This can be slow, but it is convenient, especially if the files are named without standard extensions.

Explicit typing

Web servers can be configured to force particular files or directory contents to have a MIME type, regardless of the file extension or contents.

Type negotiation

Some web servers can be configured to store a resource in multiple document formats. In this case, the web server can be configured to determine the “best” format to use (and the associated MIME type) by a negotiation process with the user. We’ll discuss this in Chapter 17.

Web servers also can be configured to associate particular files with MIME types.

Redirection

Web servers sometimes return redirection responses instead of success messages. A web server can redirect the browser to go elsewhere to perform the request. A redirection response is indicated by a 3XX return code. The Location response header contains a URI for the new or preferred location of the content. Redirects are useful for:

Permanently moved resources

A resource might have been moved to a new location, or otherwise renamed, giving it a new URL. The web server can tell the client that the resource has been renamed, and the client can update any bookmarks, etc. before fetching the resource from its new location. The status code 301 Moved Permanently is used for this kind of redirect.

Temporarily moved resources

If a resource is temporarily moved or renamed, the server may want to redirect the client to the new location. But, because the renaming is temporary, the server wants the client to come back with the old URL in the future and not to update any bookmarks. The status codes 303 See Other and 307 Temporary Redirect are used for this kind of redirect.

URL augmentation

Servers often use redirects to rewrite URLs, often to embed context. When the request arrives, the server generates a new URL containing embedded state information and redirects the user to this new URL.* The client follows the redirect, reissuing the request, but now including the full, state-augmented URL. This is a

* These extended, state-augmented URLs are sometimes called “fat URLs.”

useful way of maintaining state across transactions. The status codes 303 See Other and 307 Temporary Redirect are used for this kind of redirect.

Load balancing

If an overloaded server gets a request, the server can redirect the client to a less heavily loaded server. The status codes 303 See Other and 307 Temporary Redirect are used for this kind of redirect.

Server affinity

Web servers may have local information for certain users; a server can redirect the client to a server that contains information about the client. The status codes 303 See Other and 307 Temporary Redirect are used for this kind of redirect.

Canonicalizing directory names

When a client requests a URI for a directory name without a trailing slash, most web servers redirect the client to a URI with the slash added, so that relative links work correctly.

Step 6: Sending Responses

Web servers face similar issues sending data across connections as they do receiving. The server may have many connections to many clients, some idle, some sending data to the server, and some carrying response data back to the clients.

The server needs to keep track of the connection state and handle persistent connections with special care. For nonpersistent connections, the server is expected to close its side of the connection when the entire message is sent.

For persistent connections, the connection may stay open, in which case the server needs to be extra cautious to compute the Content-Length header correctly, or the client will have no way of knowing when a response ends (see Chapter 4).

Step 7: Logging

Finally, when a transaction is complete, the web server notes an entry into a log file, describing the transaction performed. Most web servers provide several configurable forms of logging. Refer to Chapter 21 for more details.

For More Information

For more information on Apache, Jigsaw, and *ident*, check out:

Apache: The Definitive Guide

Ben Laurie and Peter Laurie, O'Reilly & Associates, Inc.

Professional Apache

Peter Wainwright, Wrox Press.

<http://www.w3c.org/jigsaw/>

Jigsaw—W3C's Server W3C Consortium Web Site.

<http://www.ietf.org/rfc/rfc1413.txt>

RFC 1413, "Identification Protocol," by M. St. Johns.

Protocol Gateways

You can direct HTTP traffic to gateways the same way you direct traffic to proxies. Most commonly, you explicitly configure browsers to use gateways, intercept traffic transparently, or configure gateways as surrogates (reverse proxies).

Figure 8-3 shows the dialog boxes used to configure a browser to use server-side FTP gateways. In the configuration shown, the browser is configured to use *gw1.joes-hardware.com* as an HTTP/FTP gateway for all FTP URLs. Instead of sending FTP commands to an FTP server, the browser will send HTTP commands to the HTTP/FTP gateway *gw1.joes-hardware.com* on port 8080.

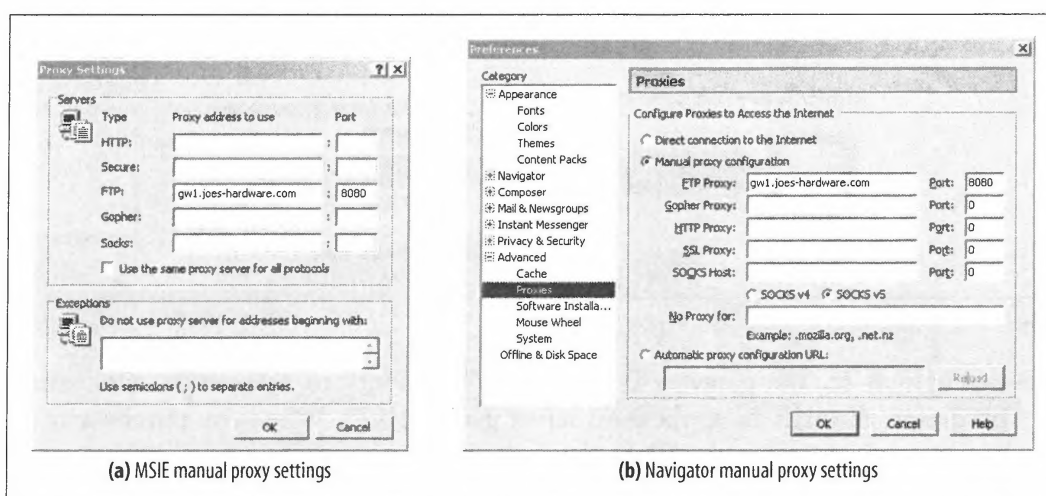


Figure 8-3. Configuring an HTTP/FTP gateway

The result of this gateway configuration is shown in Figure 8-4. Normal HTTP traffic is unaffected; it continues to flow directly to origin servers. But requests for FTP URLs are sent to the gateway *gw1.joes-hardware.com* within HTTP requests. The gateway performs the FTP transactions on the client's behalf and carries results back to the client by HTTP.

The following sections describe common kinds of gateways: server protocol converters, server-side security gateways, client-side security gateways, and application servers.

HTTP/*: Server-Side Web Gateways

Server-side web gateways convert client-side HTTP requests into a foreign protocol, as the requests travel inbound to the origin server (see Figure 8-5).

In Figure 8-5, the gateway receives an HTTP request for an FTP resource:

```
ftp://ftp.irs.gov/pub/00-index.txt
```

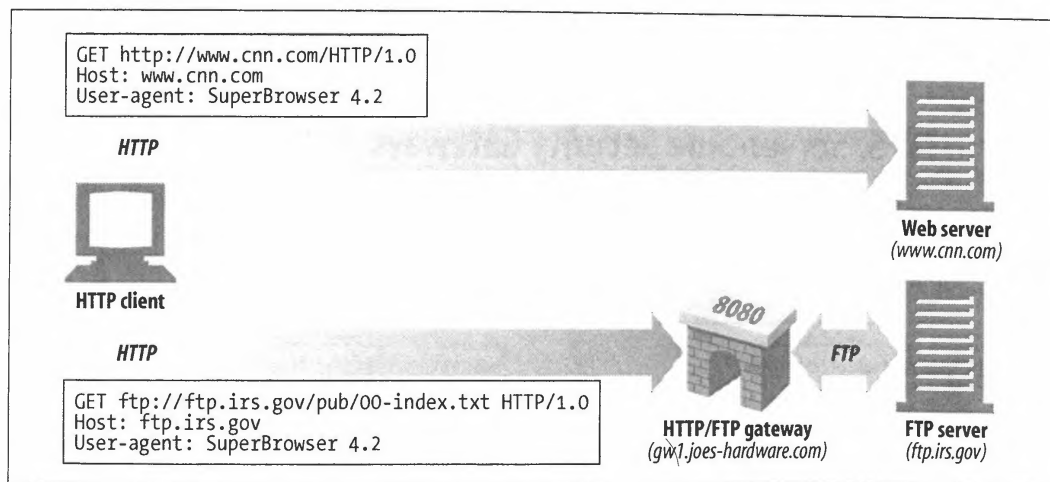


Figure 8-4. Browsers can configure particular protocols to use particular gateways

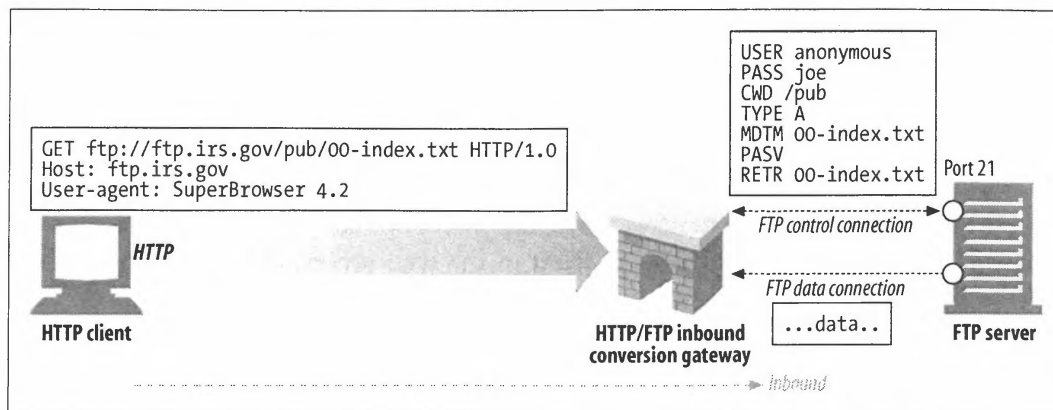


Figure 8-5. The HTTP/FTP gateway translates HTTP request into FTP requests

The gateway proceeds to open an FTP connection to the FTP port on the origin server (port 21) and speak the FTP protocol to fetch the object. The gateway does the following:

- Sends the USER and PASS commands to log in to the server
- Issues the CWD command to change to the proper directory on the server
- Sets the download type to ASCII
- Fetches the document's last modification time with MDTM
- Tells the server to expect a passive data retrieval request using PASV
- Requests the object retrieval using RETR
- Opens a data connection to the FTP server on a port returned on the control channel; as soon as the data channel is opened, the object content flows back to the gateway

URL is crawled, the corresponding presence bit is set. If the presence bit is already set, the crawler assumes the URL has already been crawled.*

Checkpoints

Be sure to save the list of visited URLs to disk, in case the robot program crashes.

Partitioning

As the Web grows, it may become impractical to complete a crawl with a single robot on a single computer. That computer may not have enough memory, disk space, computing power, or network bandwidth to complete a crawl.

Some large-scale web robots use “farms” of robots, each a separate computer, working in tandem. Each robot is assigned a particular “slice” of URLs, for which it is responsible. Together, the robots work to crawl the Web. The individual robots may need to communicate to pass URLs back and forth, to cover for malfunctioning peers, or to otherwise coordinate their efforts.

A good reference book for implementing huge data structures is *Managing Gigabytes: Compressing and Indexing Documents and Images*, by Witten, et. al (Morgan Kaufmann). This book is full of tricks and techniques for managing large amounts of data.

Aliases and Robot Cycles

Even with the right data structures, it is sometimes difficult to tell if you have visited a page before, because of URL “aliasing.” Two URLs are *aliases* if the URLs look different but really refer to the same resource.

Table 9-1 illustrates a few simple ways that different URLs can point to the same resource.

Table 9-1. Different URLs that alias to the same documents

	First URL	Second URL	When aliased
a	<code>http://www.foo.com/bar.html</code>	<code>http://www.foo.com:80/bar.html</code>	Port is 80 by default
b	<code>http://www.foo.com/~fred</code>	<code>http://www.foo.com/%7Ffred</code>	%7F is same as ~
c	<code>http://www.foo.com/x.html#early</code>	<code>http://www.foo.com/x.html#middle</code>	Tags don't change the page
d	<code>http://www.foo.com/readme.htm</code>	<code>http://www.foo.com/README.HTM</code>	Case-insensitive server
e	<code>http://www.foo.com/</code>	<code>http://www.foo.com/index.html</code>	Default page is <i>index.html</i>
f	<code>http://www.foo.com/index.html</code>	<code>http://209.231.87.45/index.html</code>	<i>www.foo.com</i> has this IP address

* Because there are a potentially infinite number of URLs and only a finite number of bits in the presence bit array, there is potential for collision—two URLs can map to the same presence bit. When this happens, the crawler mistakenly concludes that a page has been crawled when it hasn't. In practice, this situation can be made very unlikely by using a large number of presence bits. The penalty for collision is that a page will be omitted from a crawl.

Canonicalizing URLs

Most web robots try to eliminate the obvious aliases up front by “canonicalizing” URLs into a standard form. A robot might first convert every URL into a canonical form, by:

1. Adding “:80” to the hostname, if the port isn’t specified
2. Converting all %xx escaped characters into their character equivalents
3. Removing # tags

These steps can eliminate the aliasing problems shown in Table 9-1a–c. But, without knowing information about the particular web server, the robot doesn’t have any good way of avoiding the duplicates from Table 9-1d–f:

- The robot would need to know whether the web server was case-insensitive to avoid the alias in Table 9-1d.
- The robot would need to know the web server’s index-page configuration for this directory to know whether the URLs in Table 9-1e were aliases.
- The robot would need to know if the web server was configured to do virtual hosting (covered in Chapter 5) to know if the URLs in Table 9-1f were aliases, even if it knew the hostname and IP address referred to the same physical computer.

URL canonicalization can eliminate the basic syntactic aliases, but robots will encounter other URL aliases that can’t be eliminated through converting URLs to standard forms.

Filesystem Link Cycles

Symbolic links on a filesystem can cause a particularly insidious kind of cycle, because they can create an illusion of an infinitely deep directory hierarchy where none exists. Symbolic link cycles usually are the result of an unintentional error by the server administrator, but they also can be created by “evil webmasters” as a malicious trap for robots.

Figure 9-3 shows two filesystems. In Figure 9-3a, *subdir* is a normal directory. In Figure 9-3b, *subdir* is a symbolic link pointing back to /. In both figures, assume the file */index.html* contains a hyperlink to the file *subdir/index.html*.

Using Figure 9-3a’s filesystem, a web crawler may take the following actions:

1. GET `http://www.foo.com/index.html`
Get */index.html*, find link to *subdir/index.html*.
2. GET `http://www.foo.com/subdir/index.html`
Get *subdir/index.html*, find link to *subdir/logo.gif*.
3. GET `http://www.foo.com/subdir/logo.gif`
Get *subdir/logo.gif*, no more links, all done.

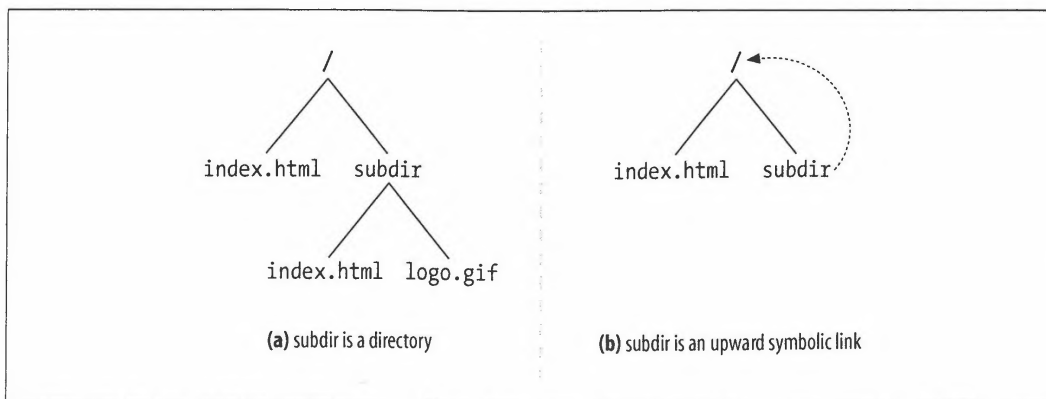


Figure 9-3. Symbolic link cycles

But in Figure 9-3b's filesystem, the following might happen:

1. GET `http://www.foo.com/index.html`
Get `/index.html`, find link to `subdir/index.html`.
2. GET `http://www.foo.com/subdir/index.html`
Get `subdir/index.html`, but get back same `index.html`.
3. GET `http://www.foo.com/subdir/subdir/index.html`
Get `subdir/subdir/index.html`.
4. GET `http://www.foo.com/subdir/subdir/subdir/index.html`
Get `subdir/subdir/subdir/index.html`.

The problem with Figure 9-3b is that `subdir/` is a cycle back to `/`, but because the URLs look different, the robot doesn't know from the URL alone that the documents are identical. The unsuspecting robot runs the risk of getting into a loop. Without some kind of loop detection, this cycle will continue, often until the length of the URL exceeds the robot's or the server's limits.

Dynamic Virtual Web Spaces

It's possible for malicious webmasters to intentionally create sophisticated crawler loops to trap innocent, unsuspecting robots. In particular, it's easy to publish a URL that looks like a normal file but really is a gateway application. This application can whip up HTML on the fly that contains links to imaginary URLs on the same server. When these imaginary URLs are requested, the nasty server fabricates a new HTML page with new imaginary URLs.

The malicious web server can take the poor robot on an Alice-in-Wonderland journey through an infinite virtual web space, even if the web server doesn't really contain any files. Even worse, it can make it very difficult for the robot to detect the cycle, because the URLs and HTML can look very different each time. Figure 9-4 shows an example of a malicious web server generating bogus content.

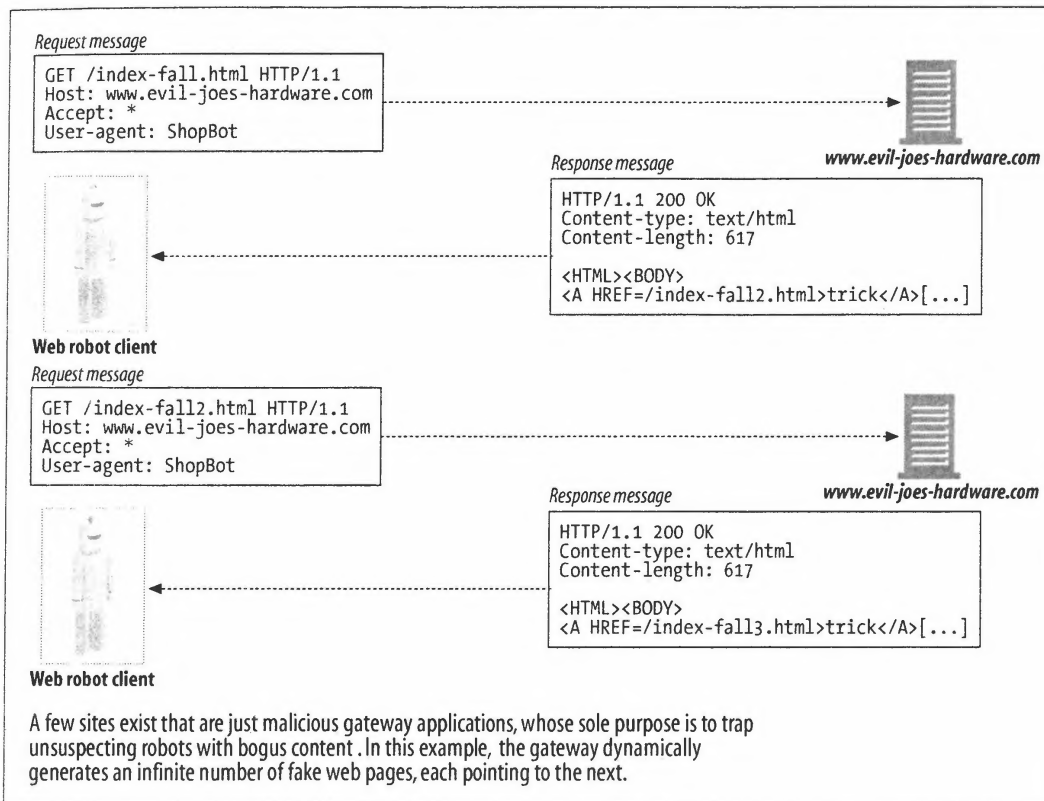


Figure 9-4. Malicious dynamic web space example

More commonly, well-intentioned webmasters may unwittingly create a crawler trap through symbolic links or dynamic content. For example, consider a CGI-based calendaring program that generates a monthly calendar and a link to the next month. A real user would not keep requesting the next-month link forever, but a robot that is unaware of the dynamic nature of the content might keep requesting these resources indefinitely.*

Avoiding Loops and Dups

There is no foolproof way to avoid all cycles. In practice, well-designed robots need to include a set of heuristics to try to avoid cycles.

Generally, the more autonomous a crawler is (less human oversight), the more likely it is to get into trouble. There is a bit of a trade-off that robot implementors need to make—these heuristics can help avoid problems, but they also are somewhat “lossy,” because you can end up skipping valid content that looks suspect.

* This is a real example mentioned on <http://www.searchtools.com/robots/robot-checklist.html> for the calendaring site at <http://cgi.umbc.edu/cgi-bin/WebEvent/webevent.cgi>. As a result of dynamic content like this, many robots refuse to crawl pages that have the substring “cgi” anywhere in the URL.

*Nosy robots**

Some robots may get URLs that point to private data and make that data easily accessible through Internet search engines and other applications. If the owner of the data didn't actively advertise the web pages, she may view the robotic publishing as a nuisance at best and an invasion of privacy at worst.

Usually this happens because a hyperlink to the "private" content that the robot followed already exists (i.e., the content isn't as secret as the owner thought it was, or the owner forgot to remove a preexisting hyperlink). Occasionally it happens when a robot is very zealous in trying to scavenge the documents on a site, perhaps by fetching the contents of a directory, even if no explicit hyperlink exists.

Robot implementors retrieving large amounts of data from the Web should be aware that their robots are likely to retrieve sensitive data at some point—data that the site implementor never intended to be accessible over the Internet. This sensitive data can include password files or even credit card information. Clearly, a mechanism to disregard content once this is pointed out (and remove it from any search index or archive) is important. Malicious search engine and archive users have been known to exploit the abilities of large-scale web crawlers to find content—some search engines, such as Google,[†] actually archive representations of the pages they have crawled, so even if content is removed, it can still be found and accessed for some time.

Dynamic gateway access

Robots don't always know what they are accessing. A robot may fetch a URL whose content comes from a gateway application. In this case, the data obtained may be special-purpose and may be expensive to compute. Many web site administrators don't like naïve robots requesting documents that come from gateways.

Excluding Robots

The robot community understood the problems that robotic web site access could cause. In 1994, a simple, voluntary technique was proposed to keep robots out of where they don't belong and provide webmasters with a mechanism to better control their behavior. The standard was named the "Robots Exclusion Standard" but is often just called *robots.txt*, after the file where the access-control information is stored.

The idea of *robots.txt* is simple. Any web server can provide an optional file named *robots.txt* in the document root of the server. This file contains information about what robots can access what parts of the server. If a robot follows this voluntary

* Generally, if a resource is available over the public Internet, it is likely referenced somewhere. Few resources are truly private, with the web of links that exists on the Internet.

† See search results at <http://www.google.com>. A cached link, which is a copy of the page that the Google crawler retrieved and indexed, is available on most results.

```
<a:ishidden b:dt="boolean">0</a:ishidden>
<a:iscollection b:dt="boolean">0</a:iscollection>
</a:prop>
</a:propstat>
</a:response></a:multistatus>
```

In this example, the server responds with a 207 Multi-Status code. WebDAV uses the 207 response for PROPFIND and a few other WebDAV methods that act simultaneously on multiple resources and potentially have different responses for each resource.

A few XML elements in the response need to be defined:

<multistatus>

A container for multiple responses.

<href>

Identifies the resource's URI.

<status>

Contains the HTTP status code for the particular request.

<propstat>

Groups one <status> element and one <prop> element. The <prop> element may contain one or more property name/value pairs for the given resource.

In the sample response listed above, the response is for one URI, *http://minstar/ch-publish.fm*. The <propstat> element embeds one <status> element and one <prop> element. For this URI, the server returned a 200 OK response, as defined by the <status> element. The <prop> element has several subelements; only some are listed in the example.

One instant application of PROPFIND is the support for directory listing. Given the expressability of a PROPFIND request, one single call can retrieve the entire hierarchy of the collection with all the properties of individual entities.

The PROPPATCH Method

The PROPPATCH method provides an atomic mechanism to set or remove multiple properties on a given resource. The atomicity will guarantee that either all of the requests are successful or none of them made it.

The base XML element for the PROPPATCH method is <propertyupdate>. It acts as a container for all the properties that need updating. The XML elements <set> and <remove> are used to specify the operation:

<set>

Specifies the property values to be set. The <set> contains one or more <prop> subelements, which in turn contains the name/value pairs of the properties to be set for the resource. If the property already exists, the value is replaced.

`<remove>`

Specifies the properties that are to be removed. Unlike with `<set>`, only the names of the properties are listed in the `<prop>` container.

This trivial example sets and removes the “owner” property:

```
<d:propertyupdate xmlns:d="DAV:" xmlns:o="http://name-space/scheme/">
  <d:set>
    <d:prop>
      <o:owner>Author A</o:owner>
    </d:prop>
  </d:set>

  <d:remove>
    <d:prop>
      <o:owner/>
    </d:prop>
  </d:remove>
</d:propertyupdate>
```

The response to PROPPATCH requests is very similar to that for PROPFIND requests. For more information, refer to RFC 2518.

Table 19-2 summarizes the status codes for the PROPFIND and PROPPATCH methods.

Table 19-2. Status codes for PROPFIND and PROPPATCH methods

Status code	Defined by	Methods	Effect
200 OK	HTTP	PROPFIND, PROPPATCH	Command success.
207 Multi-Status	WEBDAV	PROPFIND, PROPPATCH	When acting on one or more resources (or a collection), the status for each object is encapsulated into one 207 response. This is a typical success response.
401 Unauthorized	HTTP	PROPPATCH	Requires authorization to complete the property modification operation.
403 Forbidden	HTTP	PROPFIND, PROPPATCH	For PROPFIND, the client is not allowed to access the property. For PROPPATCH, the client may not change the property.
404 Not Found	HTTP	PROPFIND	No such property.
409 Conflict	HTTP	PROPPATCH	Conflict of update semantics—for example, trying to update a read-only property.
423 Locked	WebDAV	PROPPATCH	Destination resource is locked and there is no lock token or the lock token does not match.
507 Insufficient Storage	WebDAV	PROPPATCH	Not enough space for registering the modified property.

Collections and Namespace Management

A collection refers to a logical or physical grouping of resources in a predefined hierarchy. A classic example of a collection is a directory. Like directories in a filesystem,

collections act as containers of other resources, including other collections (equivalent to directories on the filesystem).

WebDAV uses the XML namespace mechanism. Unlike traditional namespaces, XML namespace partitions allow for precise structural control while preventing any namespace collisions.

WebDAV provides five methods for manipulating the namespace: DELETE, MKCOL, COPY, MOVE, and PROPFIND. PROPFIND was discussed previously in this chapter, but let's talk about the other methods.

The MKCOL Method

The MKCOL method allows clients to create a collection at the indicated URL on the server. At first sight, it may seem rather redundant to define an entire new method just for creating a collection. Overlaying on top of a PUT or POST method seems like a perfect alternative. The designers of the WebDAV protocol did consider these alternatives and still chose to define a new method. Some of the reasons behind that decision are:

- To have a PUT or a POST create a collection, the client needs to send some extra “semantic glue” along with the request. While this certainly is feasible, defining an ad hoc protocol may become tedious and error-prone.
- Most of the access-control mechanisms are based on the type of methods—only a few are allowed to create and delete resources in the repository. If we overload other methods, these access-control mechanisms will not work.

For example, a request might be:

```
MKCOL /publishing HTTP/1.1
Host: minstar
Content-Length: 0
Connection: Keep-Alive
```

And the response might be:

```
HTTP/1.1 201 Created
Server: Microsoft-IIS/5.0
Date: Fri, 10 May 2002 23:20:36 GMT
Location: http://minstar/publishing/
Content-Length: 0
```

Let us examine a few pathological cases:

- Suppose the collection already exists. If a MKCOL /colA request is made and colA already exists (i.e., namespace conflict), the request will fail with a 405 Method Not Allowed status code.
- If there are no write permissions, the MKCOL request will fail with a 403 Forbidden status code.

- If a request such as MKCOL /colA/colB is made and colA does not exist, the request will fail with a 409 Conflict status code.

Once the file or collection is created, we can delete it with the DELETE method.

The DELETE Method

We already saw the DELETE method in Chapter 3. WebDAV extends the semantics to cover collections.

If we need to delete a directory, the Depth header is needed. If the Depth header is not specified, the DELETE method assumes the Depth header to be set to infinity—that is, all the files in the directory and any subdirectories thereof are deleted. The response also has a Content-Location header identifying the collection that just got deleted. The request might read:

```
DELETE /publishing HTTP/1.0
Host: minstar
```

And the response might read:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 14 May 2002 16:41:44 GMT
Content-Location: http://minstar/publishing/
Content-Type: text/xml
Content-Length: 0
```

When removing collections, there always is a chance that a file in the collection is locked by someone else and can't be deleted. In such a case, the collection itself can't be deleted, and the server replies with a 207 Multi-Status status code. The request might read:

```
DELETE /publishing HTTP/1.0
Host: minstar
```

And the response might read:

```
HTTP/1.1 207 Multi-Status
Server: Microsoft-IIS/5.0
Content-Location: http://minstar/publishing/
.....
<?xml version="1.0"?>
<a:multistatus xmlns:a="DAV:">
<a:response>
<a:href>http://minstar/index3/ch-publish.fm</a:href>
<a:status> HTTP/1.1 423 Locked </a:status>
</a:response>
</a:multistatus>
```

In this transaction, the <status> XML element contains the status code 423 Locked, indicating that the resource *ch-publish.fm* is locked by another user.

Table D-3. "Application" MIME types (continued)

MIME type	Description	Extension	Contact and reference
application/x-ustar	The extended tar interchange format.	ustar	See the IEEE 1003.1(1990) specifications
application/x-wais-source	WAIS source structure.	src	
application/xml	Extensible Markup Language format file (use text/xml if you want the file treated as plain text by browsers, etc.).	xml, dtd	RFC 2376
application/zip	PKWARE zip archives.	zip	

audio/*

Table D-4 summarizes audio content types.

Table D-4. "Audio" MIME types

MIME type	Description	Extension	Contact and reference
audio/32kadpcm	8 kHz ADPCM audio encoding.		RFC 2421
audio/basic	Audio encoded with 8-kHz monaural 8-bit ISDN u-law PCM.	au, snd	RFC 1341
audio/G.722.1	G.722.1 compresses 50Hz–7kHz audio signals into 24 kbit/s or 32 kbit/s. It may be used for speech, music, and other types of audio.		RFC 3047
audio/L16	Audio/L16 is based on L16, described in RFC 1890. L16 denotes uncompressed audio data, using 16-bit signed representation.		RFC 2586
audio/MP4A-LATM	MPEG-4 audio.		RFC 3016
audio/midi	MIDI music files.	mid, midi, kar	
audio/mpeg	MPEG encoded audio files.	mpga, mp2, mp3	RFC 3003
audio/parityfec	Parity-based forward error correction for RTP audio.		RFC 3009
audio/prs.sid	Commodore 64 SID audio files.	sid, psid	http://www.geocities.com/SiliconValley/Lakes/5147/sidplay/docs.html#fileformats
audio/telephone-event	Logical telephone event.		RFC 2833
audio/tone	Telephonic sound pattern.		RFC 2833
audio/vnd.cns.anp1	Supports voice and unified messaging application features available on the Access NP network services platform from Comverse Network Systems.		Ann McLaughlin Comverse Network Systems amclaughlin@comversens.com

Table D-4. "Audio" MIME types (continued)

MIME type	Description	Extension	Contact and reference
audio/vnd.cns.inf1	Supports voice and unified messaging application features available on the TRILOGUE Infinity network services platform from Comverse Network Systems.		Same as above
audio/vnd.digital-winds	Digital Winds music is never-ending, reproducible, and interactive MIDI music in very small packages (<3K).	eol	Armands Strazds armands.strazds@medienhaus-bremen.de
audio/vnd.everad.plj	Proprietary EverAD audio encoding.	plj	Tomer Weisberg tom@everad.com
audio/vnd.lucent.voice	Voice messaging including Lucent Technologies' Intuity™ AUDIX® Multimedia Messaging System and the Lucent Voice Player.	lvp	Frederick Block rickblock@lucent.com http://www.lucent.com/lvp/
audio/vnd.nortel.vbk	Proprietary Nortel Networks Voice Block audio encoding.	vbk	Glenn Parsons Glenn.Parsons@NortelNetworks.com
audio/vnd.nuera.ecelp4800	Proprietary Nuera Communications audio and speech encoding, available in Nuera voice-over-IP gateways, terminals, application servers, and as a media service for various host platforms and OSs.	ecelp4800	Michael Fox mfox@nuera.com
audio/vnd.nuera.ecelp7470	Same as above.	ecelp7470	Same as above
audio/vnd.nuera.ecelp9600	Same as above.	ecelp9600	Same as above
audio/vnd.octel.sbc	Variable-rate encoding averaging 18 kbps used for voice messaging in Lucent Technologies' Sierra™, Overture™, and IMA™ platforms.		Jeff Bouis jbouis@lucent.com
audio/vnd.qcelp	Qualcomm audio encoding.	qcp	Andy Dejaco adejaco@qualcomm.com
audio/vnd.rhetorex.32kadpcm	32-kbps Rhetorex™ ADPCM audio encoding used in voice messaging products such as Lucent Technologies' CallPerformer™, Unified Messenger™, and other products.		Jeff Bouis jbouis@lucent.com
audio/vnd.vmx.cvsd	Audio encoding used in voice messaging products including Lucent Technologies' Overture200™, Overture 300™, and VMX 300™ product lines.		Same as above
audio/x-aiff	AIFF audio file format.	aif, aiff, aifc	
audio/x-pn-realaudio	RealAudio metafile format by Real Networks (formerly Progressive Networks).	ram, rm	
audio/x-pn-realaudio-plugin	From Apache <i>mime.types</i> .	rpm	

Table D-4. "Audio" MIME types (continued)

MIME type	Description	Extension	Contact and reference
audio/x-realaudio	RealAudio audio format by Real Networks (formerly Progressive Networks).	ra	
audio/x-wav	WAV audio files.	wav	

chemical/*

Much of the information in Table D-5 was obtained courtesy of the "Chemical MIME Home Page" (<http://www.ch.ic.ac.uk/chemime/>).

Table D-5. "Chemical" MIME types

MIME type	Description	Extension	Contact and reference
chemical/x-alchemy	Alchemy format	alc	http://www.camsoft.com
chemical/x-cache-csf		csf	
chemical/x-cactvs-binary	CACTVS binary format	cbn	http://cactvs.cit.nih.gov
chemical/x-cactvs-ascii	CACTVS ASCII format	cascii	http://cactvs.cit.nih.gov
chemical/x-cactvs-table	CACTVS table format	ctab	http://cactvs.cit.nih.gov
chemical/x-cdx	ChemDraw eXchange file	cdx	http://www.camsoft.com
chemical/x-cerius	MSI Cerius II format	cer	http://www.msi.com
chemical/x-chemdraw	ChemDraw file	chm	http://www.camsoft.com
chemical/x-cif	Crystallographic Interchange Format	cif	http://www.bernstein-plus-sons.com/software/rasmol/ http://ndbserver.rutgers.edu/NDB/mmcif/examples/index.html
chemical/x-mmCIF	MacroMolecular CIF	mcif	Same as above
chemical/x-chem3d	Chem3D format	c3d	http://www.camsoft.com
chemical/x-cmdf	CrystalMaker Data Format	cmdf	http://www.crystallmaker.co.uk
chemical/x-compass	Compass program of the Takahashi	cpa	
chemical/x-crossfire	Crossfire file	bsd	
chemical/x-cml	Chemical Markup Language	cml	http://www.xml-cml.org
chemical/x-csml	Chemical Style Markup Language	csml, csm	http://www.mdli.com
chemical/x-ctx	Gasteiger group CTX file format	ctx	
chemical/x-cxf		cxf	
chemical/x-daylight-smiles	Smiles format	smi	http://www.daylight.com/dayhtml/smiles/index.html
chemical/x-embl-dl-nucleotide	EMBL nucleotide format	emb	http://mercury.ebi.ac.uk
chemical/x-galactic-spc	SPC format for spectral and chromatographic data	spc	http://www.galactic.com/galactic/Data/spcvue.htm

Table D-10. "Text" MIME types (continued)

MIME type	Description	Extension	Contact and reference
text/vnd.wap.si	Service Indication (SI) objects contain a message describing an event and a URI describing where to load the corresponding service.	si, xml	WAP Forum Ltd http://www.wapforum.org
text/vnd.wap.sl	The Service Loading (SL) content type provides a means to convey a URI to a user agent in a mobile client. The client itself automatically loads the content indicated by that URI and executes it in the addressed user agent without user intervention when appropriate.	sl, xml	Same as above
text/vnd.wap.wml	Wireless Markup Language (WML) is a markup language, based on XML, that defines content and user interface for narrow-band devices, including cellular phones and pagers.	wml	Same as above
text/vnd.wap.wmlscript	WMLScript is an evolution of JavaScript for wireless devices.	wmls	Same as above
text/x-setext	From Apache <i>mime.types</i> .	etx	
text/xml	Extensible Markup Language format file (use application/xml if you want the browser to save to file when downloaded).	xml	RFC 2376

video/*

Table D-11 lists some popular video movie formats. Note that some video formats are classified as application types.

Table D-11. "Video" MIME types

MIME type	Description	Extension	Contact and reference
video/MP4V-ES	MPEG-4 video payload, as carried by RTP.		RFC 3016
video/mpeg	Video encoded per the ISO 11172 CD MPEG standard.	mpeg, mpg, mpe	RFC 1341
video/parityfec	Forward error correcting video format for data carried through RTP streams.		RFC 3009
video/pointer	Transporting pointer position information for presentations.		RFC 2862
video/quicktime	Apple Quicktime video format.	qt, mov	http://www.apple.com
video/vnd.fvt	Video format from FAST Search & Transfer.	fvt	Arild Fuldseth Arild.Fuldseth@fast.no

Table D-11. "Video" MIME types (continued)

MIME type	Description	Extension	Contact and reference
video/vnd.motorola.video video/vnd.motorola.videop	Proprietary formats used by products from Motorola ISG.		Tom McGinty Motorola ISG tmcginty@dma.isg.mot
video/vnd.mpegurl	This media type consists of a series of URLs of MPEG Video files.	mxu	Heiko Recktenwald uzs106@uni-bonn.de "Power and Responsibility: Conversations with Contributors," Guy van Belle, et al., LMJ 9 (1999), 127–133, 129 (MIT Press)
video/vnd.nokia.interleaved-multimedia	Used in Nokia 9210 Communicator video player and related tools.	nim	Petteri Kangaslampi petteri.kangaslampi@nokia.com
video/x-msvideo	Microsoft AVI movies.	avi	http://www.microsoft.com
video/x-sgi-movie	Silicon Graphics's movie format.	movie	http://www.sgi.com

Experimental Types

The set of primary types supports most content types. Table D-12 lists one experimental type, for conferencing software, that is configured in some web servers.

Table D-12. Extension MIME types

MIME type	Description	Extension	Contact and reference
x-conference/x-cooltalk	Collaboration tool from Netscape	ice	



HTTP: The Definitive Guide



Behind every successful web transaction lurks the Hypertext Transfer Protocol (HTTP), the language by which web clients and servers exchange documents and information. HTTP is commonly known as the workhorse behind the browsers we use every day to access our company intranets, locate out-of-print books, or research census information.

But HTTP is used for far more than browsing the Web: the simplicity and ubiquity of HTTP also have made it the choice protocol for many other networked applications, most notably through web services such as SOAP and XML-RPC.

As the title suggests, *HTTP: The Definitive Guide* explains the HTTP protocol: how it works and how to use it to develop web-based applications. However, this book is not just about HTTP; it's also about all the other core Internet technologies that HTTP depends on to work effectively. Although HTTP is at the center of the book, the essence of *HTTP: The Definitive Guide* is in understanding how the Web works and how to apply that knowledge to web programming and administration. The book explains the technical workings, motivations, performance considerations, and objectives of HTTP and the technologies around which it revolves.

HTTP: The Definitive Guide is the bible for the HTTP protocol and related web technologies. Topics covered include:

- HTTP methods, headers, and status codes
- Optimizing proxies and caches
- Strategies for designing web robots and crawlers
- Cookies, authentication, and Secure HTTP
- Internationalization and content negotiation
- Redirection and load-balancing strategies

Written by experts with years of practical experience, this book uses clear, concise language and a plethora of detailed illustrations to help readers visualize what goes on behind the scenes, providing a complete understanding of the story behind each query on the Web.

All web programmers, administrators, and application developers need to be familiar with HTTP in order to work effectively. There are many books that explain how to use the Web, but this is the book that explains how the Web works.

www.oreilly.com

US \$54.99

CAN \$63.99

ISBN: 978-1-56592-509-0

