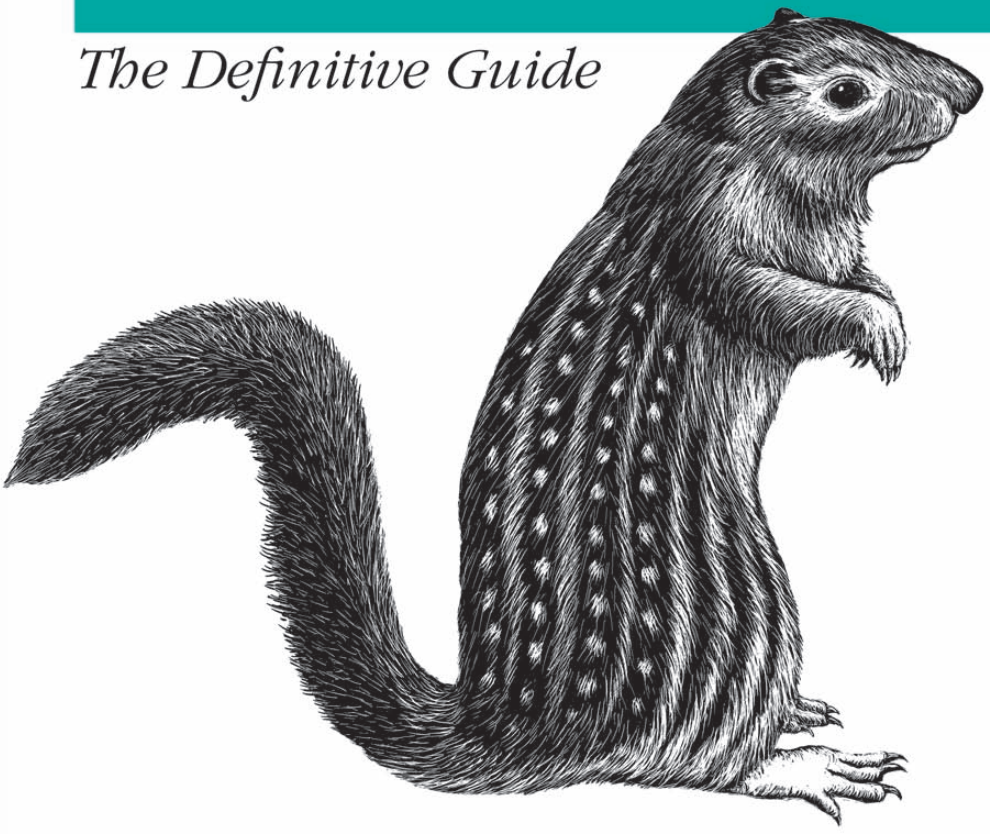
Understanding Web Internals

HTTP

The Definitive Guide



O'REILLY®

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

MICROSOFT-1019
Microsoft Corp. v. Sandpiper CDN, LLC
IPR2026-00180

Web Programming

O'REILLY®

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 $44.95 **CAN $69.95**

ISBN: 978-1-56592-509-0

HTTP

The Definitive Guide

HTTP

The Definitive Guide

David Gourley and Brian Totty

with Marjorie Sayer, Sailu Reddy, and Anshu Aggarwal

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • 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.

RepKover™ This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 1-56592-509-2
ISBN-13: 978-1-56592-509-0
[C] [01/08]

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

PART I

HTTP: The Web's Foundation

This section is an introduction to the HTTP protocol. The next four chapters describe the core technology of HTTP, the foundation of the Web:

- Chapter 1, *Overview of HTTP*, is a rapid-paced overview of HTTP.
- Chapter 2, *URLs and Resources*, details the formats of URLs and the various types of resources that URLs name across the Internet. We also outline the evolution to URNs.
- Chapter 3, *HTTP Messages*, details the HTTP messages that transport web content.
- Chapter 4, *Connection Management*, discusses the commonly misunderstood and poorly documented rules and behavior for managing TCP connections by HTTP.

CHAPTER 1

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.

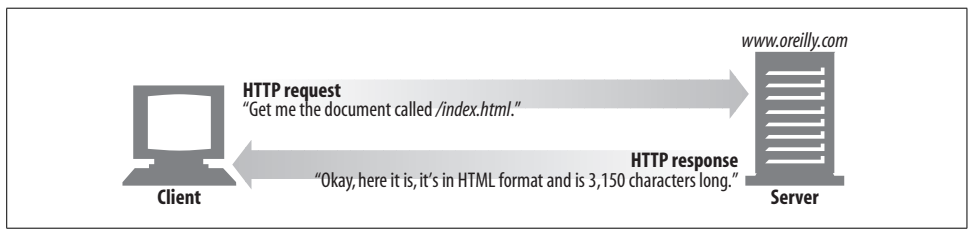


Figure 1-1. Web clients and servers

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).

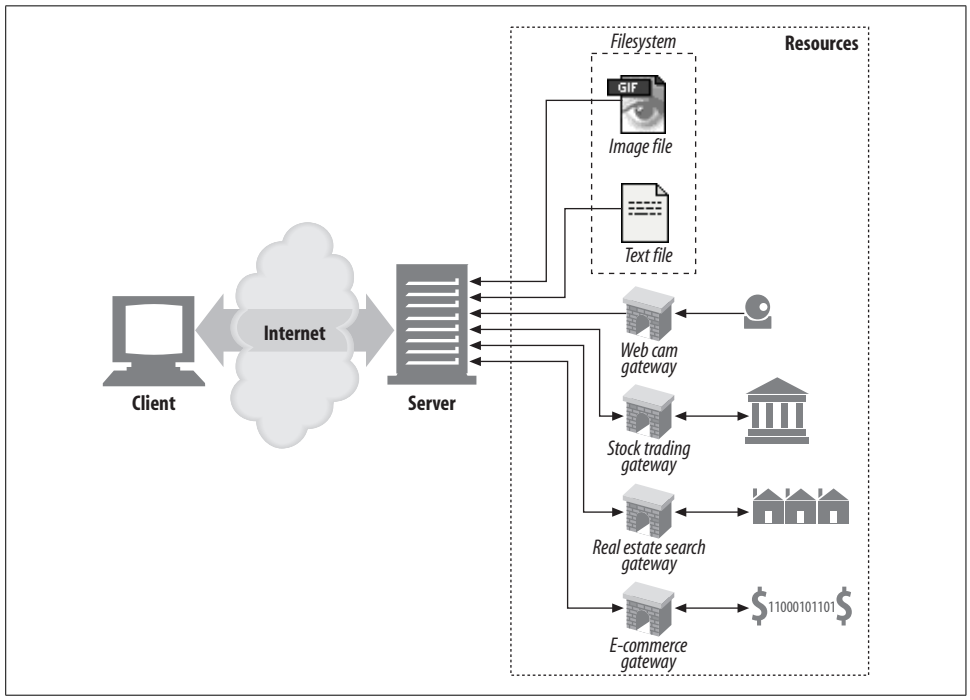


Figure 1-2. A web resource is anything that provides web content

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.

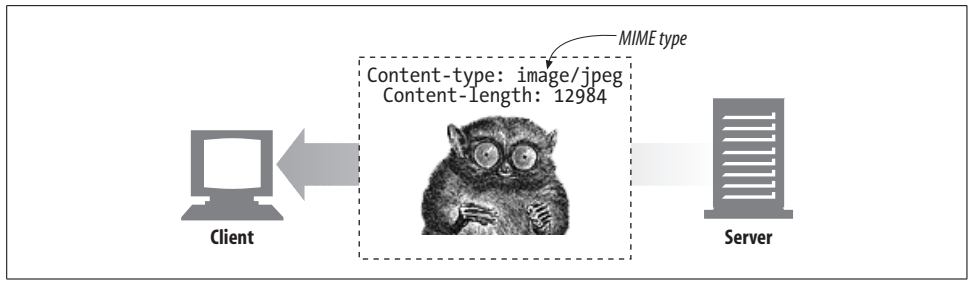


Figure 1-3. MIME types are sent back with the data content

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.

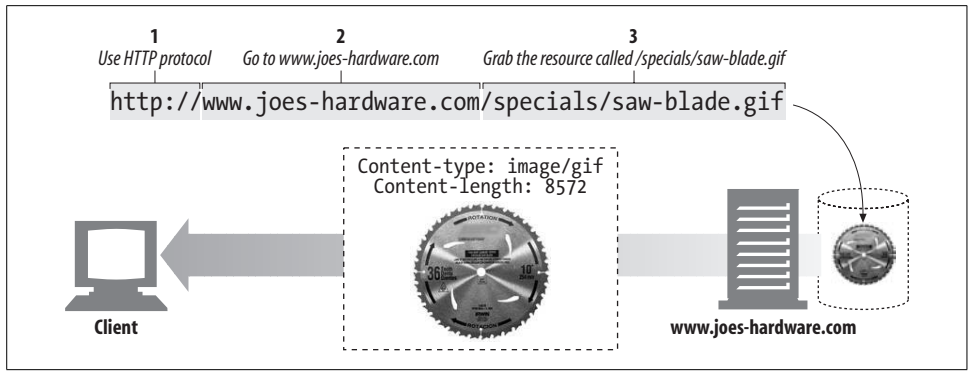


Figure 1-4. URLs specify protocol, server, and local resource

Table 1-1. Example URLs

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

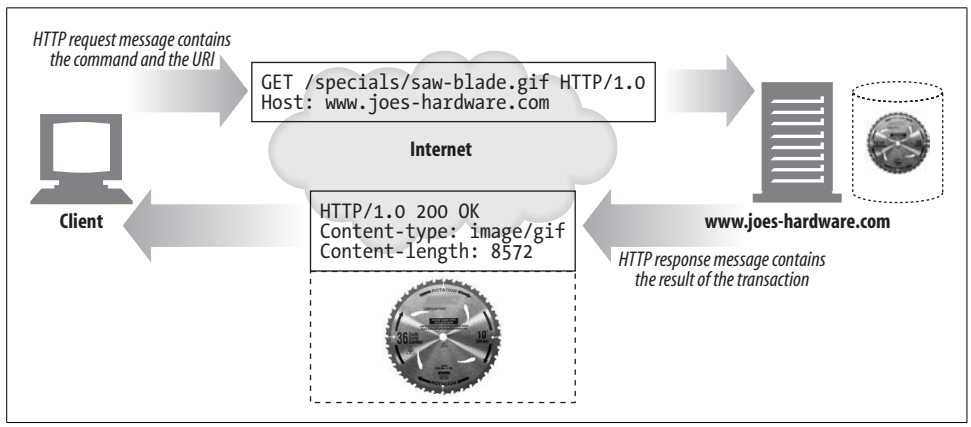


Figure 1-5. HTTP transactions consist of request and response messages

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.

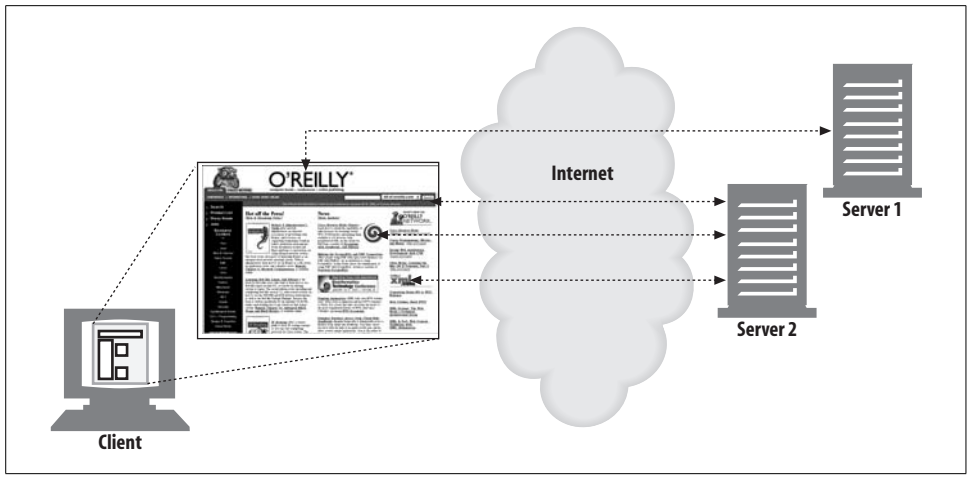


Figure 1-6. Composite web pages require separate HTTP transactions for each embedded resource

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.

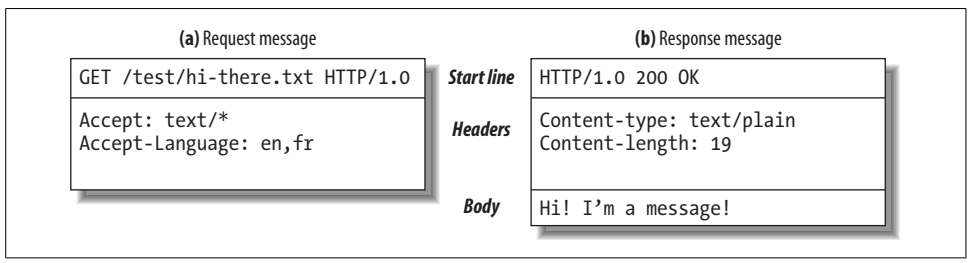


Figure 1-7. HTTP messages have a simple, line-oriented text structure

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.

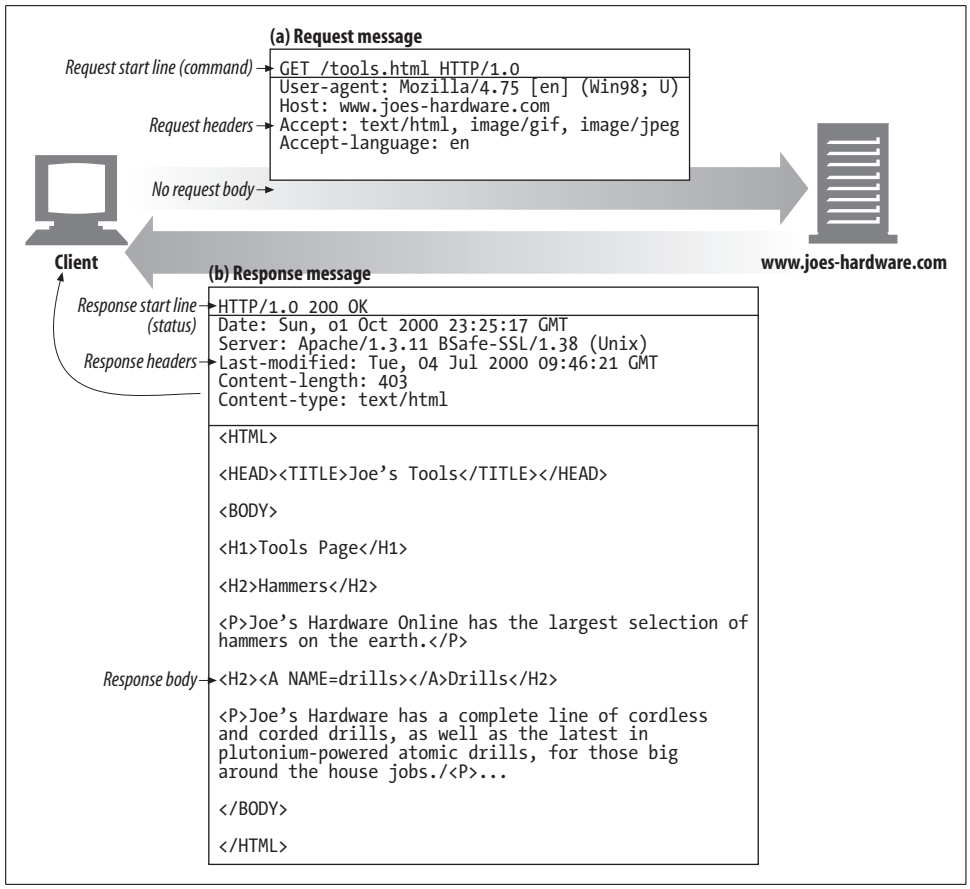


Figure 1-8. Example GET transaction for http://www.joes-hardware.com/tools.html

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).

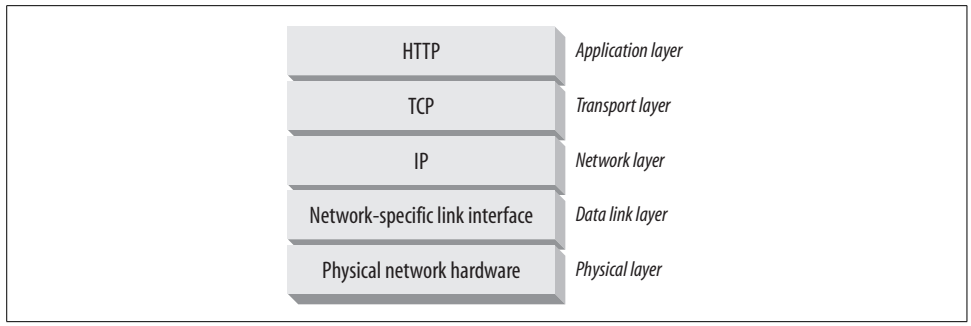


Figure 1-9. HTTP network protocol stack

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.

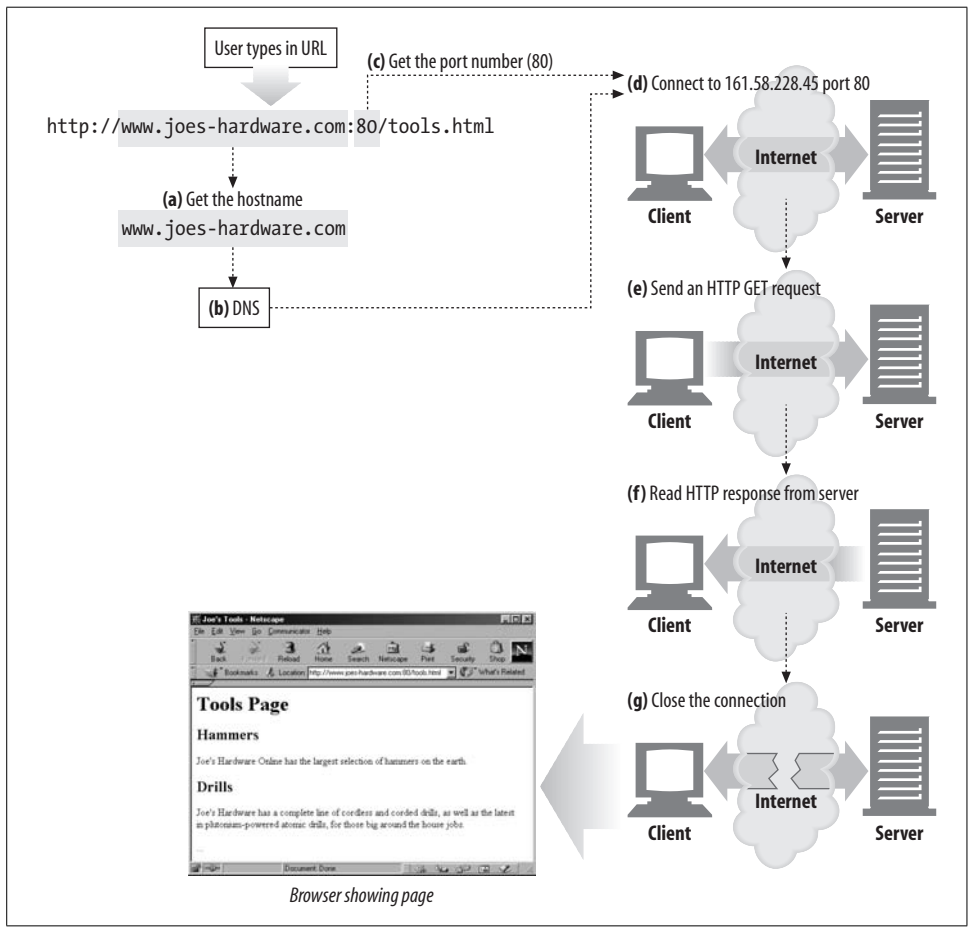


Figure 1-10. Basic browser connection process

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.

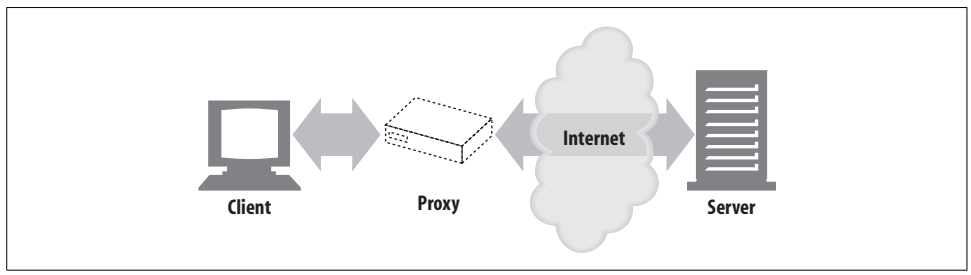


Figure 1-11. Proxies relay traffic between client and server

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).

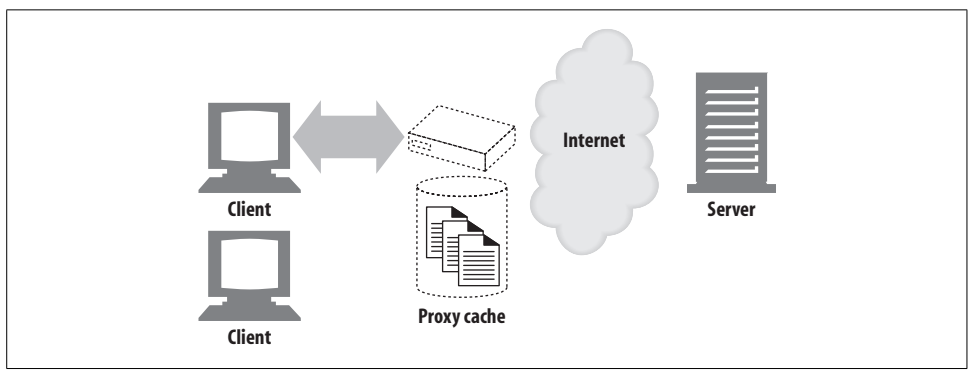


Figure 1-12. Caching proxies keep local copies of popular documents to improve performance

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.

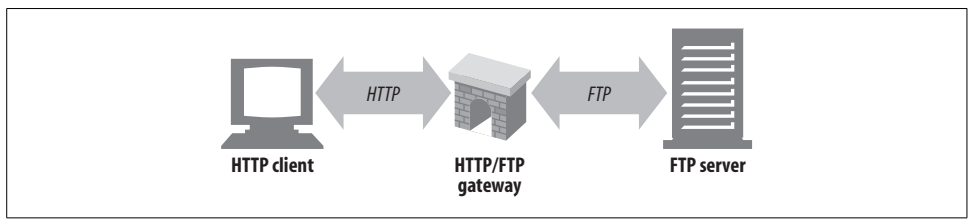


Figure 1-13. HTTP/FTP gateway

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.

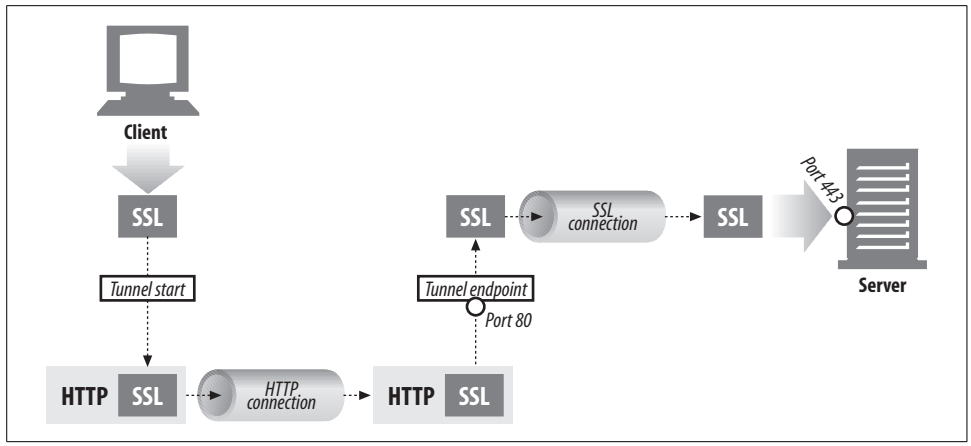


Figure 1-14. Tunnels forward data across non-HTTP networks (HTTP/SSL tunnel shown)

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.

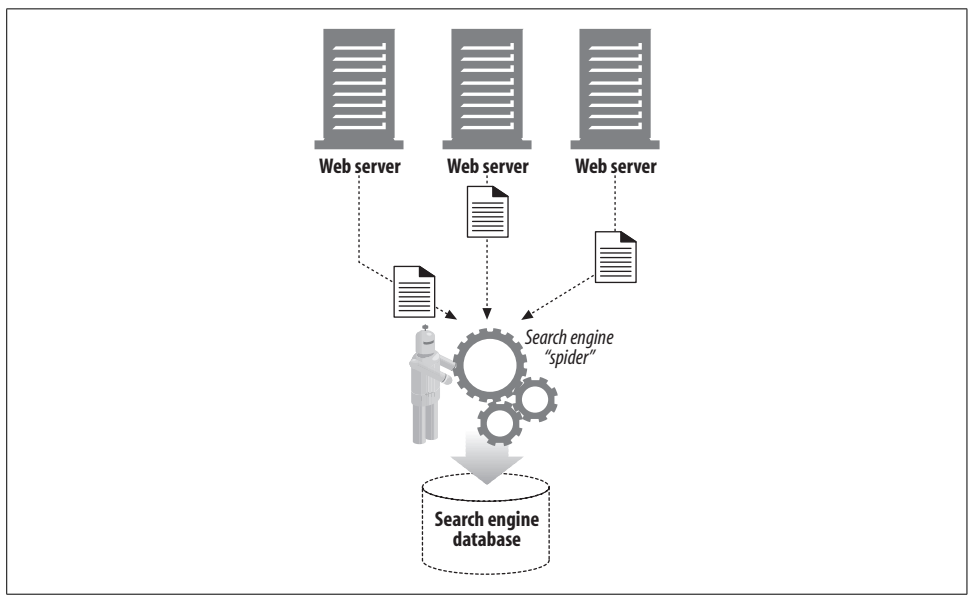


Figure 1-15. Automated search engine "spiders" are agents, fetching web pages around the world

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.

CHAPTER 2

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.

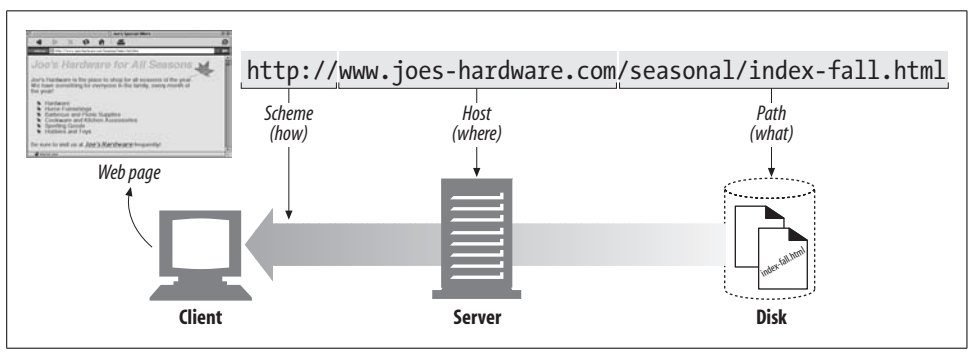


Figure 2-1. How URLs relate to browser, machine, server, and location on the server's filesystem

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`.

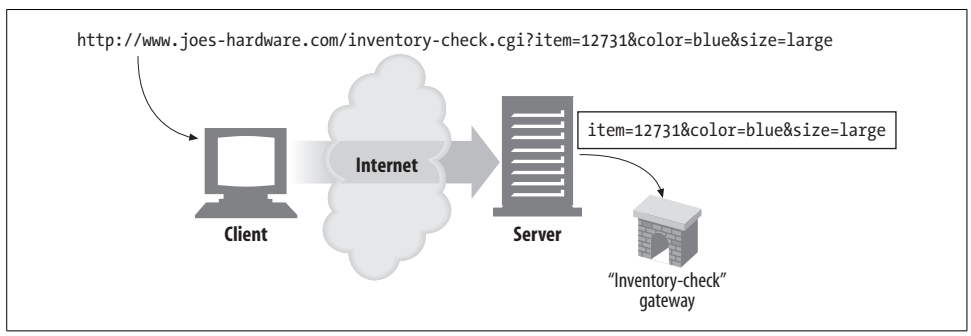


Figure 2-2. The URL query component is sent along to the gateway application

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