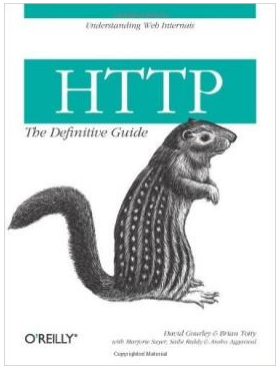
CSE 686 Internet Programming

Week 6: HTTP

Edmund Yu, PhD
Associate Teaching Professor
esyu@syr.edu

February 19 & 21, 2018



References:
Java Network Programming, Elliotte Harold, 2013, Chapter 6

HTTP

- The **Hypertext Transfer Protocol** (HTTP) is a standard that defines:
  - How a web client (a.k.a. browser) talks to a web server
  - How data is transferred from the server back to the client
- Although HTTP is usually thought of as a means of transferring HTML files and the pictures embedded in them, HTTP is **data format agnostic**:
  - It can be used to transfer image files, Microsoft Word documents, Windows *.exe* files, query results, or anything else that can be represented in bytes on the World Wide Web - collectively called **resources**. (See next slide)
- Together with HTML forms the foundation of the Web

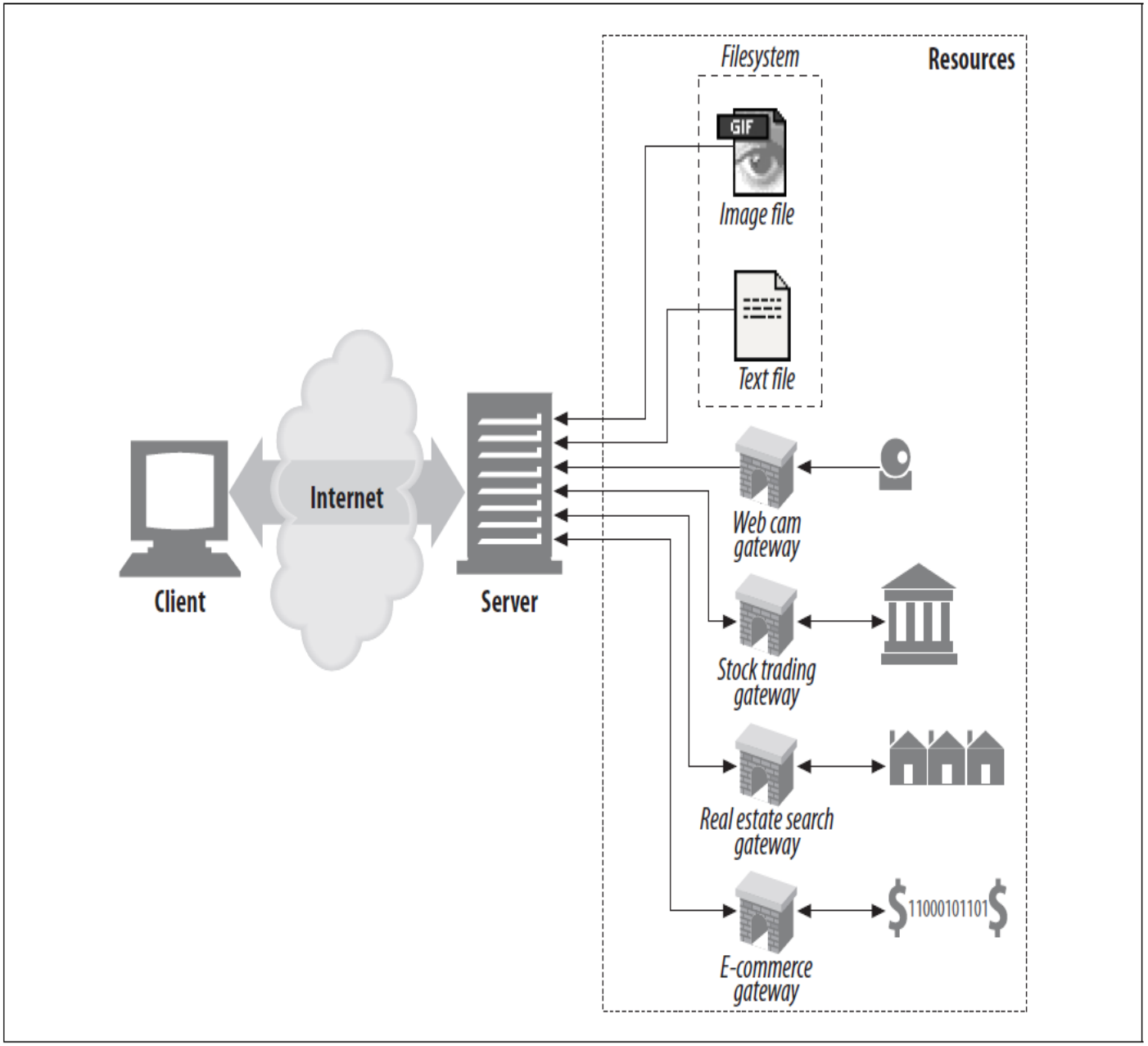


Figure 1-2. A web resource is anything that provides web content

HTTP, cont.

- Was standardized by IETF (RFC 2616, HTTP/1.1) in 1999
  - In June 2014, RFC 2616 was retired and HTTP/1.1 was redefined by RFCs 7230, 7231, 7232, 7233, 7234, and 7235.
- HTTP/2 was published as RFC 7540 in May 2015.
  - According to W3Techs, as of February 2018, 24.1% of the top 10 million websites supported HTTP/2

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Čeština
Deutsch
Español
Français
한국어
Italiano
עברית
Magyar
日本語
Polski
Русский

HTTP/2

From Wikipedia, the free encyclopedia

HTTP/2 (originally named **HTTP/2.0**) is a major revision of the HTTP network protocol used by the World Wide Web. It was derived from the earlier experimental SPDY protocol, originally developed by Google.[1] HTTP/2[2] was developed by the Hypertext Transfer Protocol working group httpbis (where bis means "second") of the Internet Engineering Task Force.[3] HTTP/2 is the first new version of HTTP since HTTP 1.1, which was standardized in RFC 2068 in 1997. The Working Group presented HTTP/2 to IESG for consideration as a Proposed Standard in December 2014,[4][5] and IESG approved it to publish as Proposed Standard on February 17, 2015.[6][7] The HTTP/2 specification was published as RFC 7540 in May 2015.[8]

The standardization effort was supported by Chrome, Opera, Firefox,[9] Internet Explorer 11, Safari, Amazon Silk, and Edge browsers.[10] Most major browsers had added HTTP/2 support by the end of 2015.[11]

According to W3Techs, as of February 2018, 24.1% of the top 10 million websites supported HTTP/2.[12]

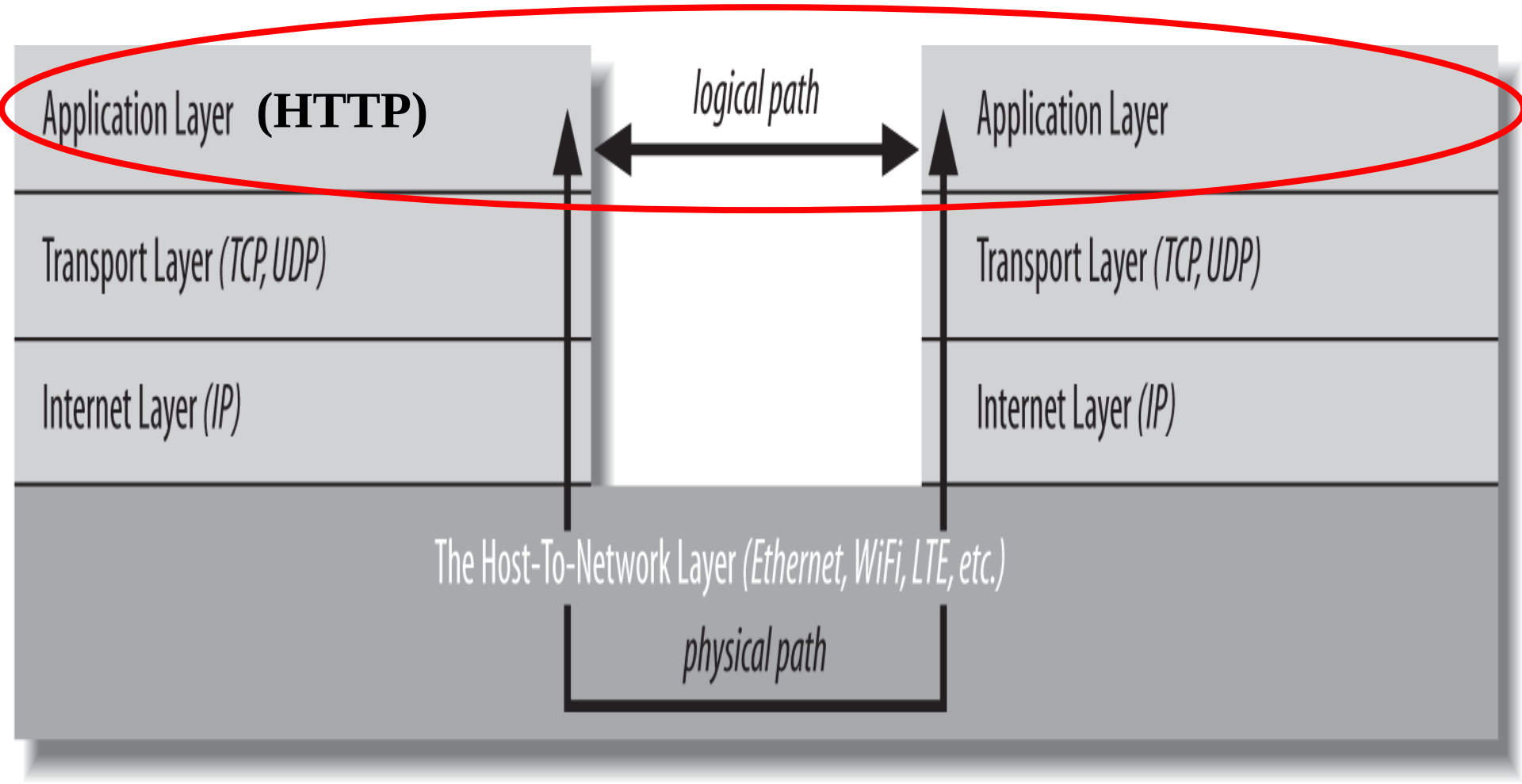
Contents [hide]

Goals [edit]

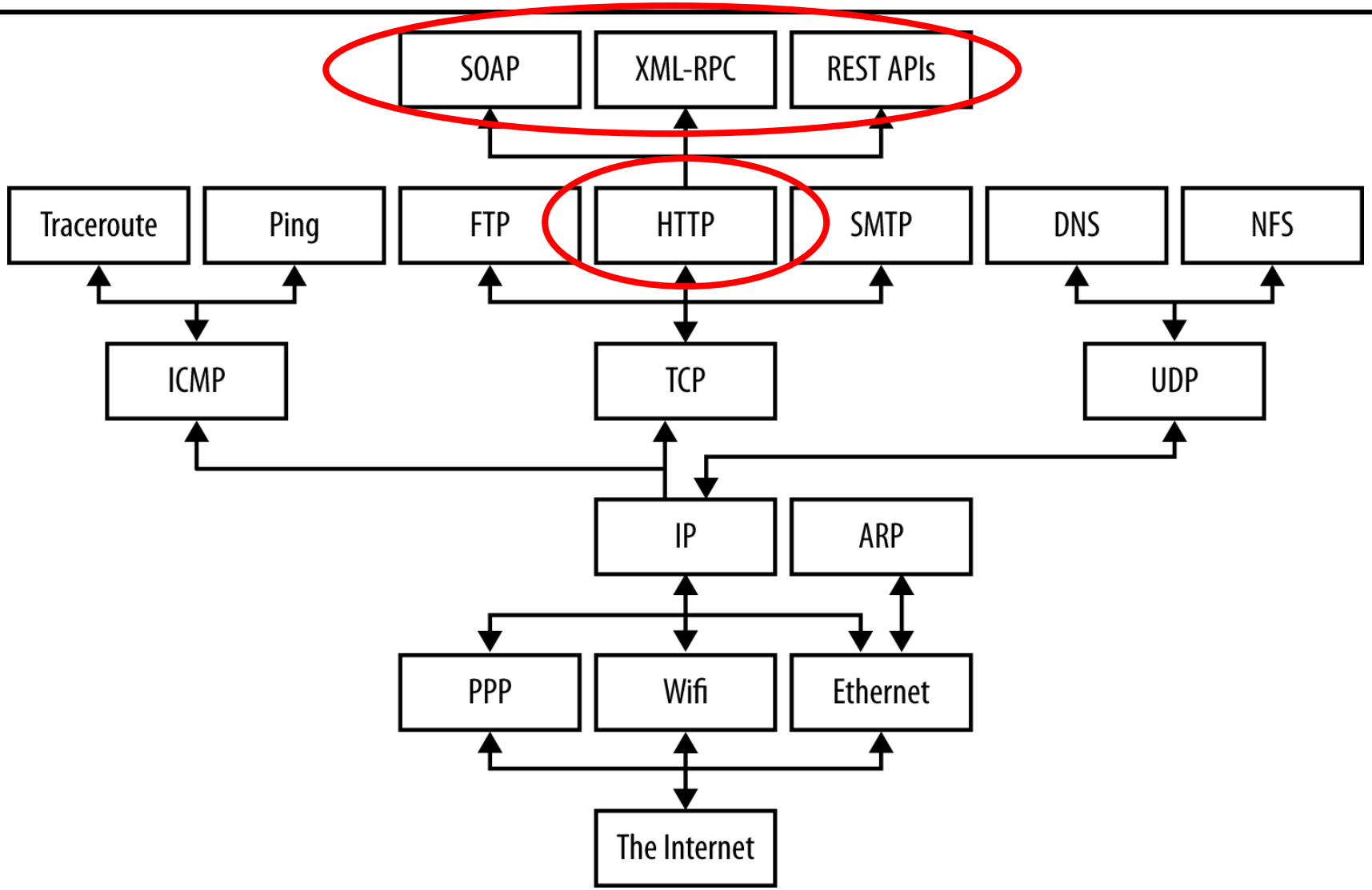
The working group charter mentions several goals and issues of concern:[3]

- Create a negotiation mechanism that allows clients and servers to elect to use HTTP 1.1, 2.0, or potentially other non-HTTP protocols.
- Maintain high-level compatibility with HTTP 1.1 (for example with methods, status codes, URIs, and most header fields).
- Decrease latency to improve page load speed in web browsers by considering:
  - Data compression of HTTP headers

The TCP/IP 4-Layer Model (A Reminder)



Protocols (A Reminder)



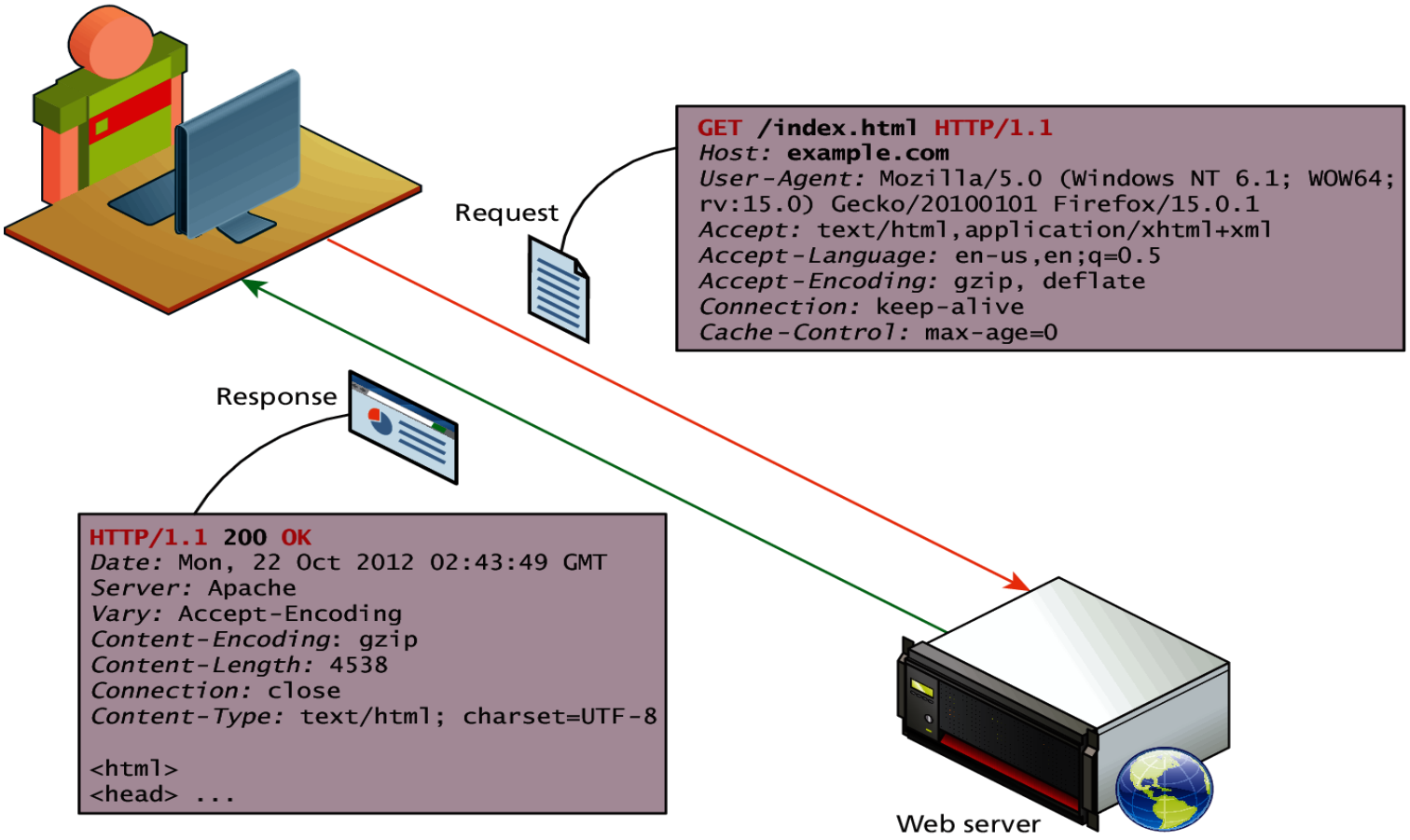
Core Features of the Web (Revisited)

- Shortly after that initial proposal Berners-Lee developed the main features of the web:

1. A URL to uniquely identify a resource on the WWW.
2. **The HTTP protocol to describe how requests and responses operate.**
3. A software program (later called web server software) that can respond to HTTP requests.
4. HTML to publish documents.
5. A program (later called a browser) to make HTTP requests from URLs and that can display the HTML it receives.

HTTP as a Requests-Responses Protocol

- HTTP is a relatively simple **request–response protocol** used for communication between an active client and a passive server.



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.

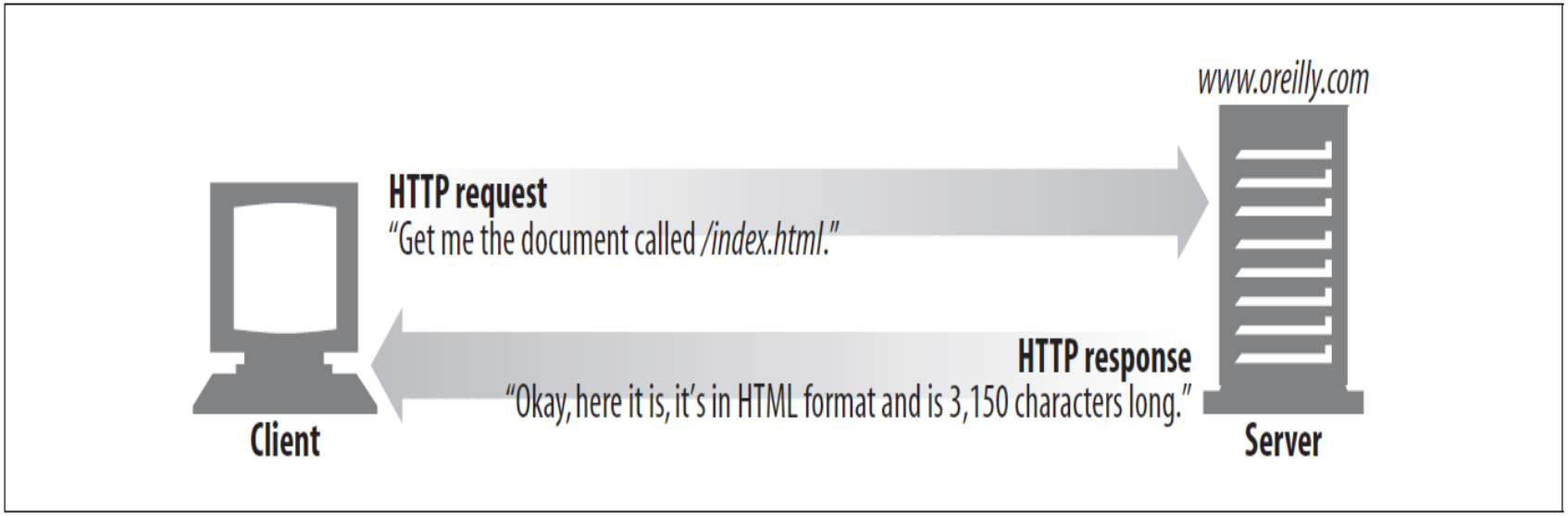


Figure 1-1. Web clients and servers

HTTP Requests

- A request has the form:

  Request =

  Request-Line

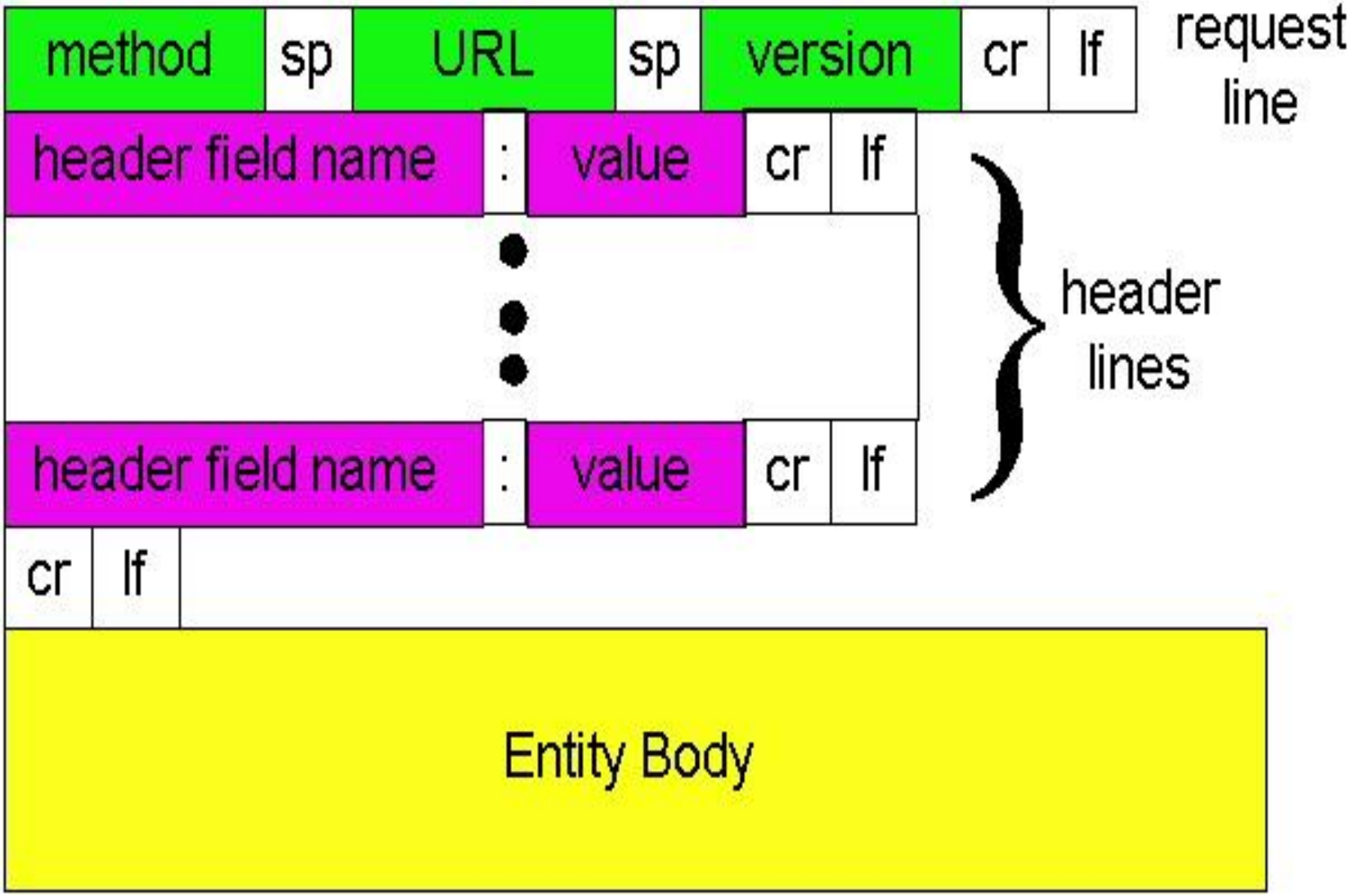
  *((general-header | request-header| entity-header) **CRLF**)

  CRLF

  [message-body, aka entity]

  - The first line of a request specifies the **request method** (‘GET’, ‘POST, etc.), the ‘resource’ and the protocol being used (HTTP/2.0 or HTTP/1.1).
  - The next few lines are headers.
    - These lines have a keyword that identifies the type of general header or request header, or entity header, and some data.
  - There will be a **blank line** following these header data, and then possibly a message body (‘entity’).

HTTP Request Message Format



Requests: An Example

GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:20.0) Gecko/20100101 Firefox/20.0
Host: en.wikipedia.org
Connection: keep-alive
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
[blank line]

- GET requests like this one do not contain a message body, so the request ends with a blank line.

The Request Line

- The request line specifies the **method** (request type), the **resource** and the **HTTP version**.

  GET /index.html HTTP/1.1

- The resource **URI** should either be an absolute URI:

  GET http://www.syr.edu/index.html HTTP/1.1

- Or an relative path, in which case there should also be a Host header:

  GET /index.html HTTP/1.1

  HOST: www.syr.edu

The GET Request

- GET is the most common HTTP method.

- A ‘GET’ request asks for a resource:
  - A **static resource** - fixed web page or image file
  - A **dynamically generated resource** - the URI names the script or program that will generate the resource.

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.

A static resource

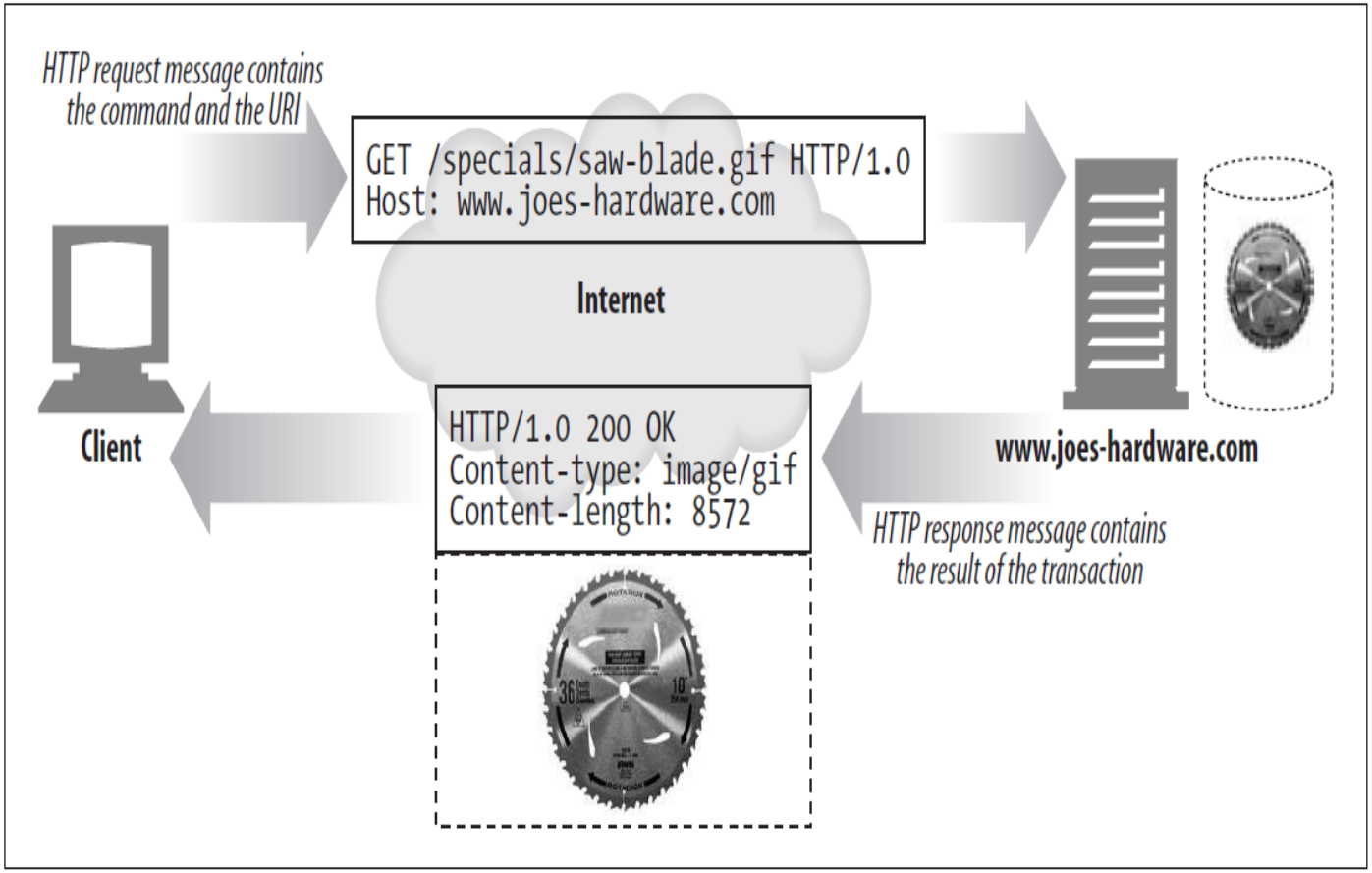


Figure 1-5. HTTP transactions consist of request and response messages

Figure 8-1 depicts a kind of resource gateway. Here, the Joe's Hardware server is acting as a gateway to database content—note that the client is simply asking for a resource through HTTP, and the Joe's Hardware server is interfacing with a gateway to get at the resource.

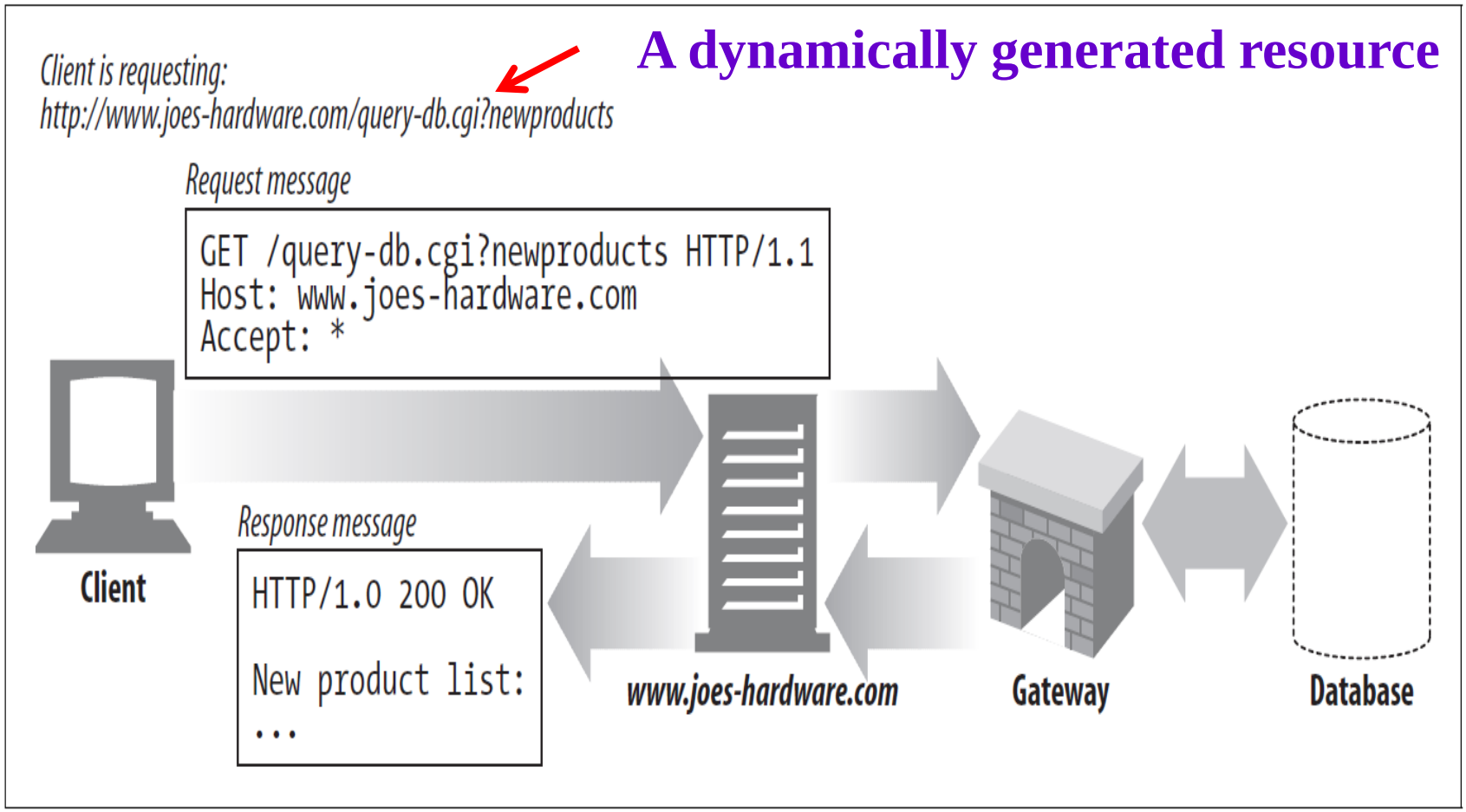


Figure 8-1. Gateway magic

CGI (Common Gateway Interface)

- CGI's contribution was to standardize the relationship/interface between the web server and a program (a child process of the Web server program) that is responsible for generating the dynamic content.
  - The CGI specification defined the data that would get passed to the program, and the formats for these data.
  - It also defined how the web server would take the output from the program, add a **HTTP header** and return these data as the response sent back to a client.
- The programs responsible for handling the form and generate the dynamic content are known as **CGI scripts,** which are usually written in a **scripting language**.

The GET Request

- The **URI** may have an appended query string.
  - This supplies parameter data for the system that is generating a resource.
    - These data take the form of *name=value* substrings that are separated by ampersand (&) characters.

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`.

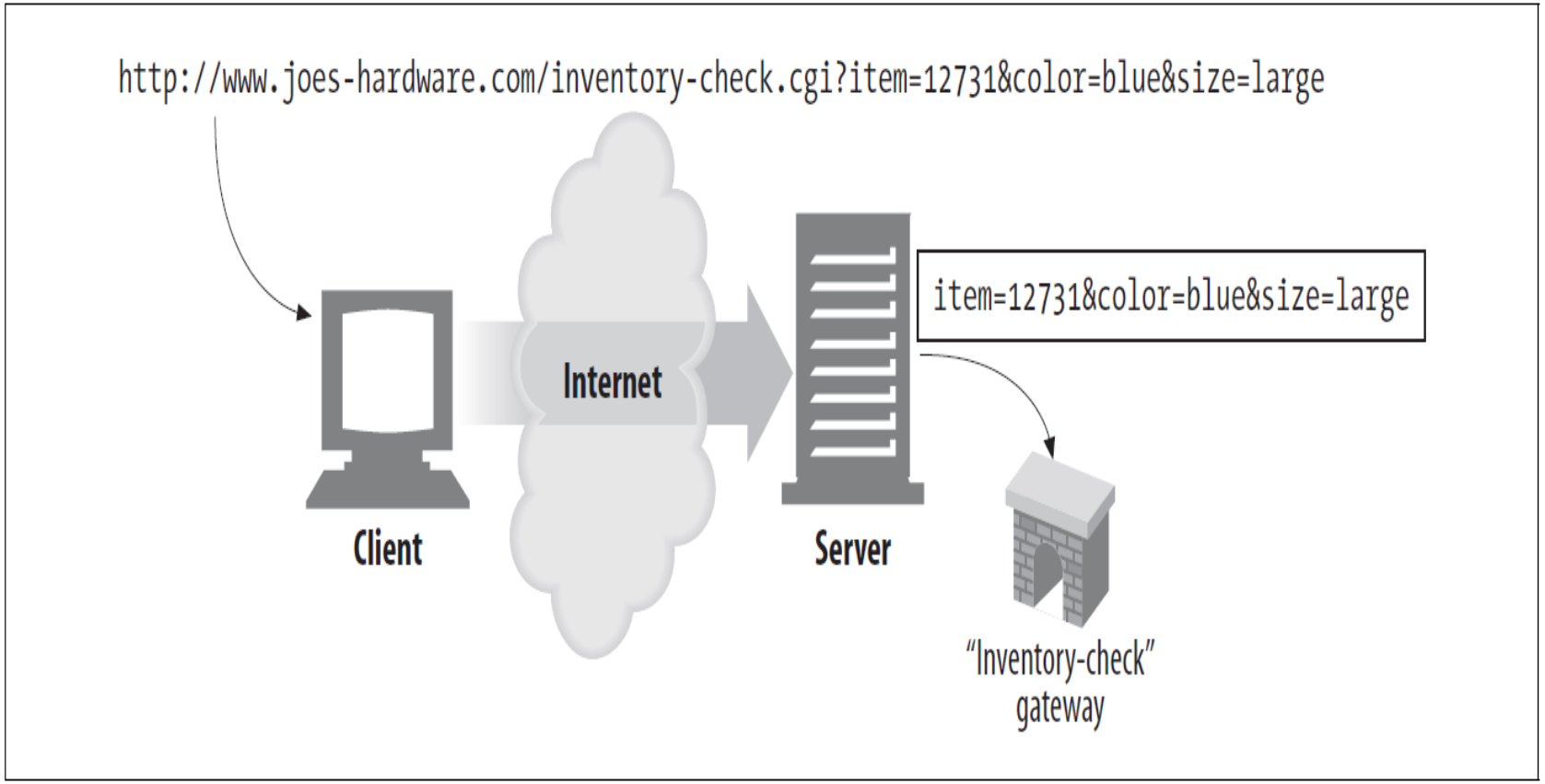


Figure 2-2. The URL query component is sent along to the gateway application

URI

- A Uniform Resource Identifier (URI) is a string of characters in a particular syntax that identifies a resource.
  - The resource identified may be a file on a server, but it may also be:
    - An email address
    - A news message
    - A book
    - A person's name
    - An Internet host
    - The current stock price of Apple…
- The syntax of a URI is composed of a scheme and a scheme-specific part, separated by a colon:

scheme:scheme-specific-part

URI

scheme:scheme-specific-part

- The syntax of the scheme-specific part depends on the scheme being used. Current schemes include:
  - *data* - Base64-encoded data included directly in a link; see RFC 2397
  - *file* - A file on a local disk
  - *ftp* - An FTP server
  - ***http*** - A World Wide Web server using the Hypertext Transfer Protocol
  - *mailto* - An email address
  - *magnet* - A resource available for download via peer-to-peer networks such as BitTorrent
  - *telnet* - A connection to a Telnet-based service
  - ***urn* - A Uniform Resource Name**
  - In addition, Java makes heavy use of nonstandard custom schemes such as *rmi*, *jar*…

URI

scheme:scheme-specific-part

- There is no specific syntax that applies to the scheme-specific parts of all URIs. However, many have a hierarchical form, like this:

  //authority/path?query

- The **authority** part of the URI names the authority responsible for resolving the rest of the URI:

  http://www.ietf.org/rfc/rfc3986.txt

  - The scheme: *http*
  - The authority: *www.ietf.org*
  - The path: */rfc/rfc3986.txt* (initial slash included)

URI

scheme:scheme-specific-part

- For the URI:

  http://www.powells.com/cgibin/biblio?inkey=62-1565928709-0

  - The scheme: *http*
  - The authority: *www.powells.com*
  - The path: */cgi-bin/biblio*
  - The query inkey=62-1565928709-0.

- The URI:

  urn:isbn:156592870

  - The scheme is **urn** but it doesn't follow the hierarchical form for scheme-specific parts.

URL

- A Uniform Resource Locator (URL) is a URI that, in addition to identifying a resource, provides a specific network location for the resource that a client can use to retrieve a representation of that resource.
  - A URI may tell you what a resource is, but not actually tell you where or how to get that resource.
    - In the physical world, it's the difference between the title "Harry Potter and The Deathly Hallows" and the library location "Room 312, Row 28, Shelf 7".
    - In Java, the difference between the **java.net.URI** and **java.net.URL** classes is: the former only identifies resources, while the latter can both identify and retrieve resources (more useful for this course).

URL

- The syntax of a URL is:

  protocol://*userInfo@host*:*port*/*path*?*query#fragment*

- Here the protocol is another word for what was called the scheme of the URI. It can be: *file*, *ftp*, *http*, *https*, *magnet*, *telnet*, or others, but not urn.
- The **host** part of a URL is the name of the server that provides the resource you want
  - It could be a host name, such as w*ww.oreilly.com* or an IP address, such as 204.148.40.9
- The **userInfo** is optional login information for the server.
  - If present, it contains a user name and, rarely, a password.
- The **port** number is also optional.
  - It's not necessary if the service is running on its default port (port 80 for HTTP servers).
- Together, the userInfo, host, and port constitute the **authority**.

URL

- The syntax of a URL is:

  protocol://*userInfo@host*:*port*/*path*?*query#fragment*

- The **path** points to a particular resource on the specified server.
  - It often looks like a file system path such as */forum/index.php*.
    - However, it may or may not actually map to a filesystem on the server.
    - If it does map to a file system, the path is relative to the **document root** of the server, not necessarily to the root of the file system on the server.
- The **query** string provides additional arguments for the server.
  - It's commonly used only in *http* URLs, where it contains **form data** for input to programs running on the server.

URL

- The syntax of a URL is:

  protocol://*userInfo@host*:*port*/*path*?*query#fragment*

- The **fragment** references a particular part of the remote resource.
  - If the remote resource is HTML, the fragment identifier names an **anchor** in the HTML document. (See next slide)
  - If the remote resource is XML, the fragment identifier is an **XPointer**.
  - Some sources refer to the fragment part of the URL as a **section**.
  - Java rather refers to the fragment identifier as a **Ref** .
- Fragment identifier targets are created in an HTML document with an *id* attribute: **<h3 id="xtocid1902914">Comments</h3>**
  - To refer to this point:

    http://www.cafeaulait.org/javafaq.html#xtocid1902914

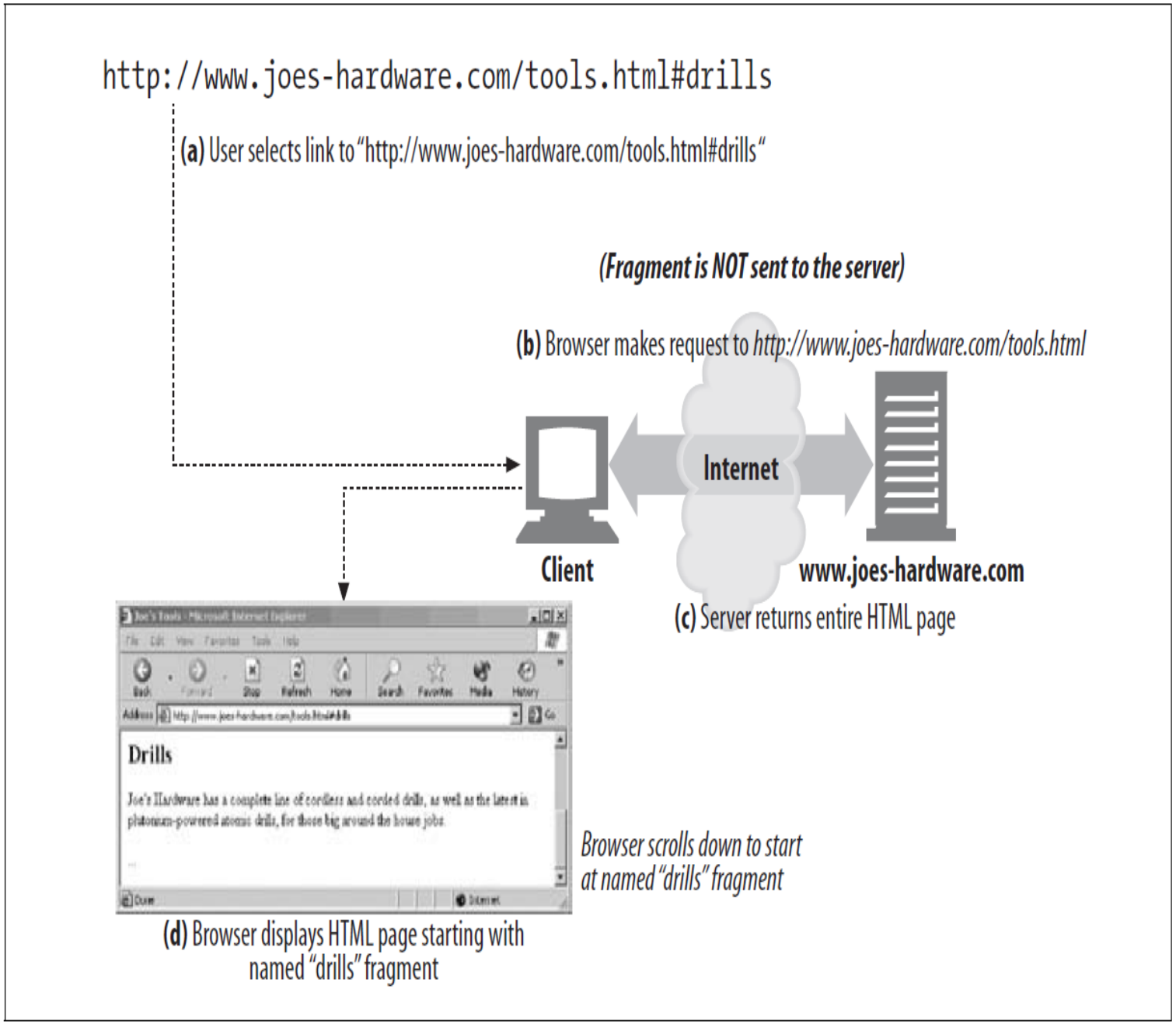


Figure 2-3. The URL fragment is used only by the client, because the server deals with entire objects

The URL Class in Java

- A URL object has two main components:
  - **Protocol identifier** (aka **scheme**):
    - E.g. file, ftp, http, mailto, telnet,…
    - For the URL http://example.com, the protocol identifier is http
  - **Resource name** (aka the scheme specific part):
    - For the URL http://example.com, the resource name is example.com

Creating a URL in Java - 1

- The easiest way to create a URL object is from a String that represents the human-readable form of the URL address.
  - In your Java program, you can use a String containing this text to create a URL object:

    URL myURL = new URL("http://example.com/");

  - The URL object created above represents an **absolute URL**.
    - An absolute URL contains all of the information necessary to reach the resource in question.

Constructor Summary

Constructors

Constructor and Description
`URL(String spec)` Creates a URL object from the String representation.
`URL(String protocol, String host, int port, String file)` Creates a URL object from the specified protocol, host, port number, and file.
`URL(String protocol, String host, int port, String file, URLStreamHandler handler)` Creates a URL object from the specified protocol, host, port number, file, and handler.
`URL(String protocol, String host, String file)` Creates a URL from the specified protocol name, host name, and file name.
`URL(URL context, String spec)` Creates a URL by parsing the given spec within a specified context.
`URL(URL context, String spec, URLStreamHandler handler)` Creates a URL by parsing the given spec with the specified handler within a specified context.

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
boolean	`equals(Object obj)` Compares this URL for equality with another object.

Creating a URL in Java - 2

- You can also create URL objects from a **relative URL** address.
- You can create URL objects for these pages relative to their common base URL: **http://example.com/pages/** like this:

```
URL myURL = new URL("http://example.com/pages/");
URL page1URL = new URL(myURL, "page1.html");
URL page2URL = new URL(myURL, "page2.html");
```

Creating a Relative URL in Java

- This code snippet uses the URL constructor that lets you create a URL object from another URL object (the base) and a relative URL specification.
  - The general form of this constructor is:

    URL(URL *baseURL*, String *relativeURL*)
  - If *baseURL* is null, then this constructor treats *relativeURL* like an absolute URL.
  - Conversely, if *relativeURL* is an absolute URL, then the constructor ignores *baseURL*.

Constructor Summary

Constructors

Constructor and Description
`URL(String spec)` Creates a URL object from the String representation.
`URL(String protocol, String host, int port, String file)` Creates a URL object from the specified protocol, host, port number, and file.
`URL(String protocol, String host, int port, String file, URLStreamHandler handler)` Creates a URL object from the specified protocol, host, port number, file, and handler.
`URL(String protocol, String host, String file)` Creates a URL from the specified protocol name, host name, and file name.
`URL(URL context, String spec)` Creates a URL by parsing the given spec within a specified context.
`URL(URL context, String spec, URLStreamHandler handler)` Creates a URL by parsing the given spec with the specified handler within a specified context.

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
boolean	`equals(Object obj)` Compares this URL for equality with another object.

URL with Special Characters

- Some URL addresses contain special characters, for example the space character:

  http://example.com/hello world/

- To make these characters legal they need to be encoded before passing them to the URL constructor:

```
URL url = new URL("http://example.com/hello%20world");
```

www.AsciiTable.com

ASCII Table and Description

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort. ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose. Below is the ASCII character table and this includes descriptions of the first 32 non-printing characters. ASCII was actually designed for use with teletypes and so the descriptions are somewhat obscure. If someone says they want your CV however in ASCII format, all this means is they want 'plain' text with no formatting such as tabs, bold or underscoring - the raw format that any computer can understand. This is usually so they can easily import the file into their own applications without issues. Notepad.exe creates ASCII text, or in MS Word you can save a file as 'text only'

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t

Malformed URL Exception

- All URL constructors throws a **MalformedURLException** if the arguments to the constructor refer to a null or unknown protocol.
  - Typically, you want to catch and handle this exception by embedding your URL constructor statements in a try/catch pair, like this:

```
try {
    URL myURL = new URL(...);
}
catch (MalformedURLException e) {
    // exception handler code here // ...
}
```

Parsing an URL

- The URL class provides several methods that let you query URL objects:
  - **getProtocol** Returns the protocol identifier component of the URL.
  - **getAuthority** Returns the authority component of the URL.
  - **getHost** Returns the host name component of the URL.
  - **getPort** Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.
  - **getPath** Returns the path component of this URL.
  - **getQuery** Returns the query component of this URL.
  - **getFile** Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.
  - **getRef** Returns the reference component of the URL.

Parsing an URL: An Example

```
import java.net.*;
import java.io.*;
public class ParseURL {
  public static void main(String[] args) throws Exception {
    URL aURL = new URL("http://www.example.com:80/docs/books/tutorial"
+ "/index.html?name=networking#DOWNLOADING");
    System.out.println("protocol = " + aURL.getProtocol());
    System.out.println("authority = " + aURL.getAuthority());
    System.out.println("host = " + aURL.getHost());
    System.out.println("port = " + aURL.getPort());
    System.out.println("path = " + aURL.getPath());
    System.out.println("query = " + aURL.getQuery());
    System.out.println("filename = " + aURL.getFile());
    System.out.println("ref = " + aURL.getRef());
  }
}
```

Reading Directly from a URL

- After you've successfully created a URL, you can call the URL's **openStream()** method to get an input stream from which you can read the contents of the URL.
  - The openStream() method returns a **java.io.InputStream** object, so reading from a URL is as easy as reading from an input stream.

```
import java.net.*;
import java.io.*;
public class URLReader {
    public static void main(String[] args) throws Exception {
        URL syr = new URL("http://www.syr.edu/");
        BufferedReader in = new BufferedReader( new
            InputStreamReader(syr.openStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
                System.out.println(inputLine);
        in.close();
    }
}
```

openStream

```
public final InputStream openStream()
                           throws IOException
```

Opens a connection to this URL and returns an InputStream for reading from that connection. This method is a shorthand for:

```
openConnection().getInputStream()
```

Returns:

an input stream for reading from the URL connection.

Throws:

IOException - if an I/O exception occurs.

See Also:

openConnection(), URLConnection.getInputStream()

getContent

```
public final Object getContent()
                       throws IOException
```

Gets the contents of this URL. This method is a shorthand for:

```
openConnection().getContent()
```

Returns:

the contents of this URL.

Throws:

java.io

Class InputStreamReader

java.lang.Object
  java.io.Reader
    java.io.InputStreamReader

All Implemented Interfaces:

Closeable, AutoCloseable, Readable

Direct Known Subclasses:

FileReader

```
public class InputStreamReader
extends Reader
```

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified `charset`. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

Each invocation of one of an InputStreamReader's read() methods may cause one or more bytes to be read from the underlying byte-input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation.

For top efficiency, consider wrapping an InputStreamReader within a BufferedReader. For example:

```
BufferedReader in
  = new BufferedReader(new InputStreamReader(System.in));
```

java.io

Class BufferedReader

java.lang.Object
  java.io.Reader
    java.io.BufferedReader

All Implemented Interfaces:

Closeable, AutoCloseable, Readable

Direct Known Subclasses:

LineNumberReader

```
public class BufferedReader
extends Reader
```

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

```
BufferedReader in
  = new BufferedReader(new FileReader("foo.in"));
```

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient

Connecting to a URL

- After you've successfully created a URL object, you can call the URL object's **openConnection** method to get a **URLConnection** object
  - You can use this URLConnection object to setup parameters and request properties that you may need before connecting.
  - Connection to the remote object represented by the URL is only initiated when the **URLConnection.connect**() method is called.
  - When you do this you are initializing a communication link between your Java program and the URL over the network:

```
try {
    URL myURL = new URL("http://example.com/");
    URLConnection myURLConnection = myURL.openConnection();
    myURLConnection.connect();
}
catch (MalformedURLException e) { // new URL() failed // ... }
catch (IOException e) { // openConnection() failed // ... }
```

Reading from a URLConnection

- The following program performs the same function as the **URLReader** program shown earlier. But it explicitly retrieves a URLConnection object and gets an input stream from the connection, instead of getting an input stream directly from the URL.
- The connection is opened implicitly by calling getInputStream:

```
import java.net.*;
import java.io.*;
public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL u = new URL("http://www.syr.edu/");
        URLConnection uc = u.openConnection();
        BufferedReader in = new BufferedReader(new InputStreamReader(
                                                    uc.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) System.out.println(inputLine);
        in.close();
    }
}
```

openConnection

```
public URLConnection openConnection()
                           throws IOException
```

Returns a URLConnection instance that represents a connection to the remote object referred to by the URL.

A new instance of URLConnection is created every time when invoking the URLStreamHandler.openConnection(URL) method of the protocol handler for this URL.

It should be noted that a URLConnection instance does not establish the actual network connection on creation. This will happen only when calling URLConnection.connect().

If for the URL's protocol (such as HTTP or JAR), there exists a public, specialized URLConnection subclass belonging to one of the following packages or one of their subpackages: java.lang, java.io, java.util, java.net, the connection returned will be of that subclass. For example, for HTTP an HttpURLConnection will be returned, and for JAR a JarURLConnection will be returned.

Returns:

a URLConnection linking to the URL.

Throws:

IOException - if an I/O exception occurs.

See Also:

URL(java.lang.String, java.lang.String, int, java.lang.String)

openConnection

```
public URLConnection openConnection(Proxy proxy)
                           throws IOException
```

Same as openConnection(), except that the connection will be made through the specified proxy; Protocol handlers that do not support proxing will ignore king this method preempts the system's default ProxySelector settings.

Reading from a HttpURLConnection

```
import java.net.*;
import java.io.*;

public class HttpURLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL u = new URL("http://www.syr.edu/");
        HttpURLConnection httpCon = (HttpURLConnection) u.openConnection();
        System.out.println(httpCon.getResponseCode());
        System.out.println(httpCon.getResponseMessage());

        for (int i=0; i< 200; i++) // get 200 headers, which can be changed
                        if (httpCon.getHeaderFieldKey(i) != null)
                            System.out.println(httpCon.getHeaderFieldKey(i) + ": " +
                                                httpCon.getHeaderField(i));
```

Reading from a HttpURLConnection

```
        /System.out.println(httpCon.getContent());
        // Read content/HTML Document
        BufferedReader in = new BufferedReader(new
                                InputStreamReader(httpCon.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) System.out.println(inputLine);

        httpCon.disconnect();
    }
}
```

Writing to a URLConnection

- Sometimes you need to write data to a **URLConnection**, for example, when you submit a form to a web server using **POST** or upload a file using **PUT**. (To be introduced later.)
  - The getOutputStream() method returns an OutputStream on which you can write data for transmission to a server:

```
public OutputStream getOutputStream()
```

- A URLConnection doesn't allow output by default, so you have to call **setDoOutput(true)** before asking for an output stream.
  - When you set doOutput to true for a URL, the request method is changed from GET to POST.

Back to The Request Line

- The request line specifies the **method** (request type), the **resource** and the **HTTP version**.

  GET /index.html HTTP/1.1

- The resource URI should either be an absolute URI:

  GET http://www.syr.edu/index.html HTTP/1.1

- Or an relative path, in which case there should also be a Host header:

  GET /index.html HTTP/1.1

  HOST: www.syr.edu

Table 3-1. Common HTTP methods

Method	Description	Message body?
GET	Get a document from the server.	No
HEAD	Get just the headers for a document from the server.	No
POST	Send data to the server for processing.	Yes
PUT	Store the body of the request on the server.	Yes
TRACE	Trace the message through proxy servers to the server.	No
OPTIONS	Determine what methods can operate on a server.	No
DELETE	Remove a document from the server.	No

HTTP Requests Revisited

- A request has the form:

  Request =

  Request-Line

  *((general-header | request-header| entity-header) **CRLF**)

  CRLF

  [message-body]

  - The first line of a request specifies the **request method** (‘GET’, ‘POST, etc.), the ‘resource’ and the protocol being used (HTTP/1.0 or HTTP/1.1).
  - The next few lines are headers.
    - These lines have a keyword that identifies the type of general header or request header, or entity header, and some data.
  - There will be a blank line following these header data, and then possibly a message body (‘entity’).

Table 3-1. Common HTTP methods

Method	Description	Message body?
GET	Get a document from the server.	No
HEAD	Get just the headers for a document from the server.	No
POST	Send data to the server for processing.	Yes
PUT	Store the body of the request on the server.	Yes
TRACE	Trace the message through proxy servers to the server.	No
OPTIONS	Determine what methods can operate on a server.	No
DELETE	Remove a document from the server.	No

The HEAD Request

- The HEAD method behaves exactly like the GET method, but the server returns only the **headers** (sometimes called **metadata**) in the response. (We will discuss common HTTP headers after we are done with HTTP methods.)
- No message body is ever returned. This allows a client to inspect the headers for a resource without having to actually get the resource.
- Using HEAD, you can:
  - Find out about a resource (e.g. determine its type) without getting it.
  - See if an object exists, by looking at the status code of the response.
  - Test if the resource has been modified, by looking at the headers.
  - Server developers (i.e. you) must ensure that the headers returned are exactly those that a GET request would return.
  - Both the GET and HEAD methods are required for HTTP compliance.
  - Next slide shows the HEAD method in action.

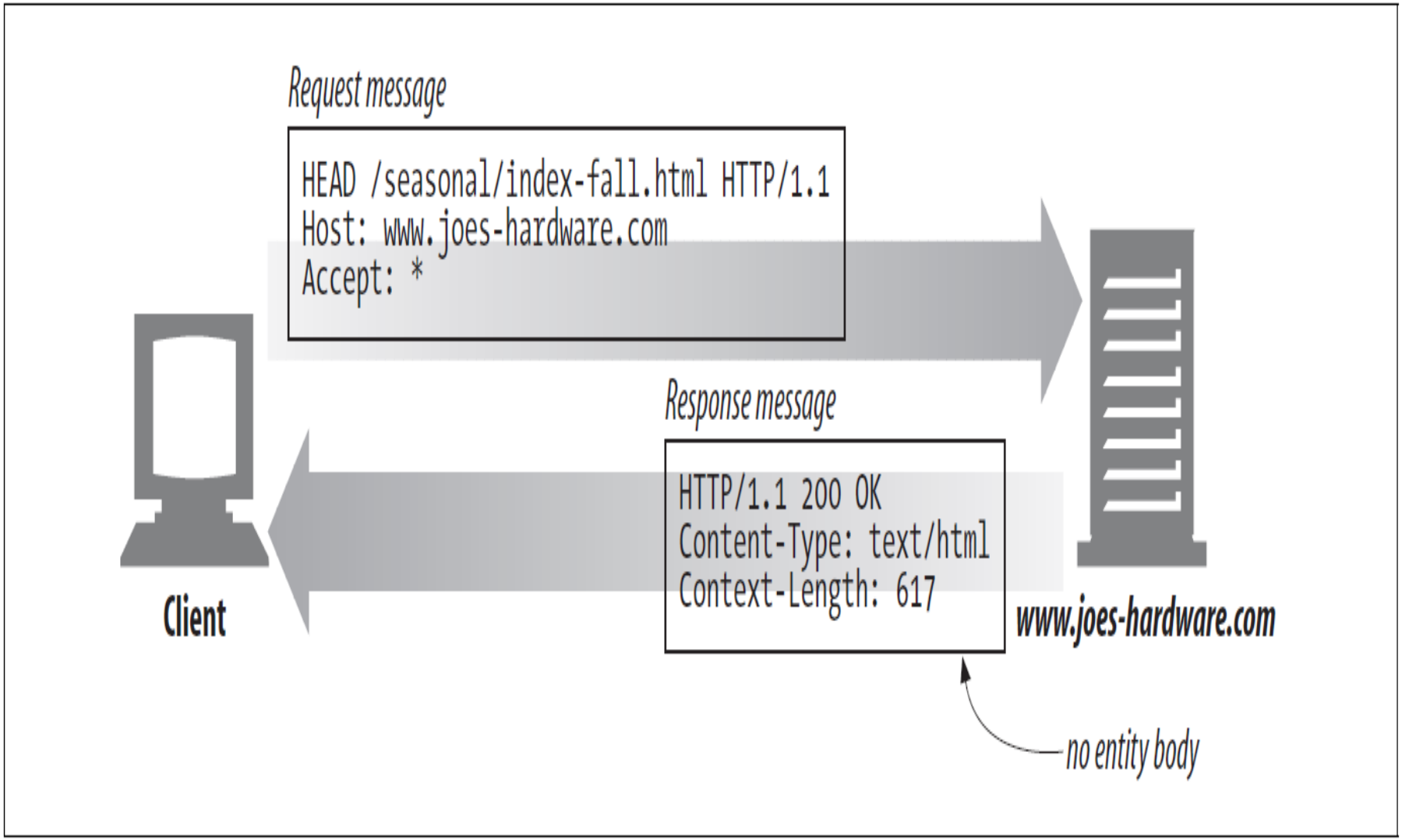


Figure 3-8. HEAD example

Table 3-1. Common HTTP methods

Method	Description	Message body?
GET	Get a document from the server.	No
HEAD	Get just the headers for a document from the server.	No
POST	Send data to the server for processing.	Yes
PUT	Store the body of the request on the server.	Yes
TRACE	Trace the message through proxy servers to the server.	No
OPTIONS	Determine what methods can operate on a server.	No
DELETE	Remove a document from the server.	No

The POST Request

- The POST method was designed to send input data to the server.
- In practice, it is often used to support HTML forms.
  - The data from a filled-in form typically is sent to the server, which then forwards it to the processing script or program (e.g., a CGI program), which then processes it.
  - The data form the body of the message ('**entity**').
  - The URI identifies the processing script or program.
  - Typically, all or part of the submitted data are used to update database tables, or are added to a bulletin board, or are placed in a file.
  - The server will usually acknowledge a change due to a POST request (**200 OK**, or **204 No data**)
    - It may also provide a more explicit acknowledgment (such as 201 Created – if a new file was created – or 202 accepted).

Forms

- The ‘form’ extensions to HTML allowed browsers to display pages very much like the **input dialog boxes** already familiar to Windows and Macintosh users.
  - These pages could contain input text fields, and controls allowing for selection amongst predefined options (radio buttons, checkboxes, menu lists, etc.)
  - An ‘**submit**’ button in a form allowed the user to submit the entered data for processing on the server.
  - The browser would take the data from the various **input fields**, encode them for transmission, and return the encoded data to the **URL** defined as an attribute in the now standardized ‘form’ tag.
- See next slide for an example

Forms: An Example

```
<form action="http://www.dontdeliverpizzas.com/order.cgi"
   name=pizzaform
   method=POST >
Pizza size<br>
Regular <input type=radio name=p_size value=reg><br>
Family <input type=radio name=p_size value=fam><br>
Popular <input type=radio checked name=p_size value=pop><br>
<p>
Toppings
<select name=tops size=3 multiple>
   <option>Cheese
   <option>Pepperoni
   ...
<option>Sun dried tomatoes
</select>
Extras
<select name=xtra size=1 multiple>
   <option selected>Coke
   <option>Ice cream
   . ..
```

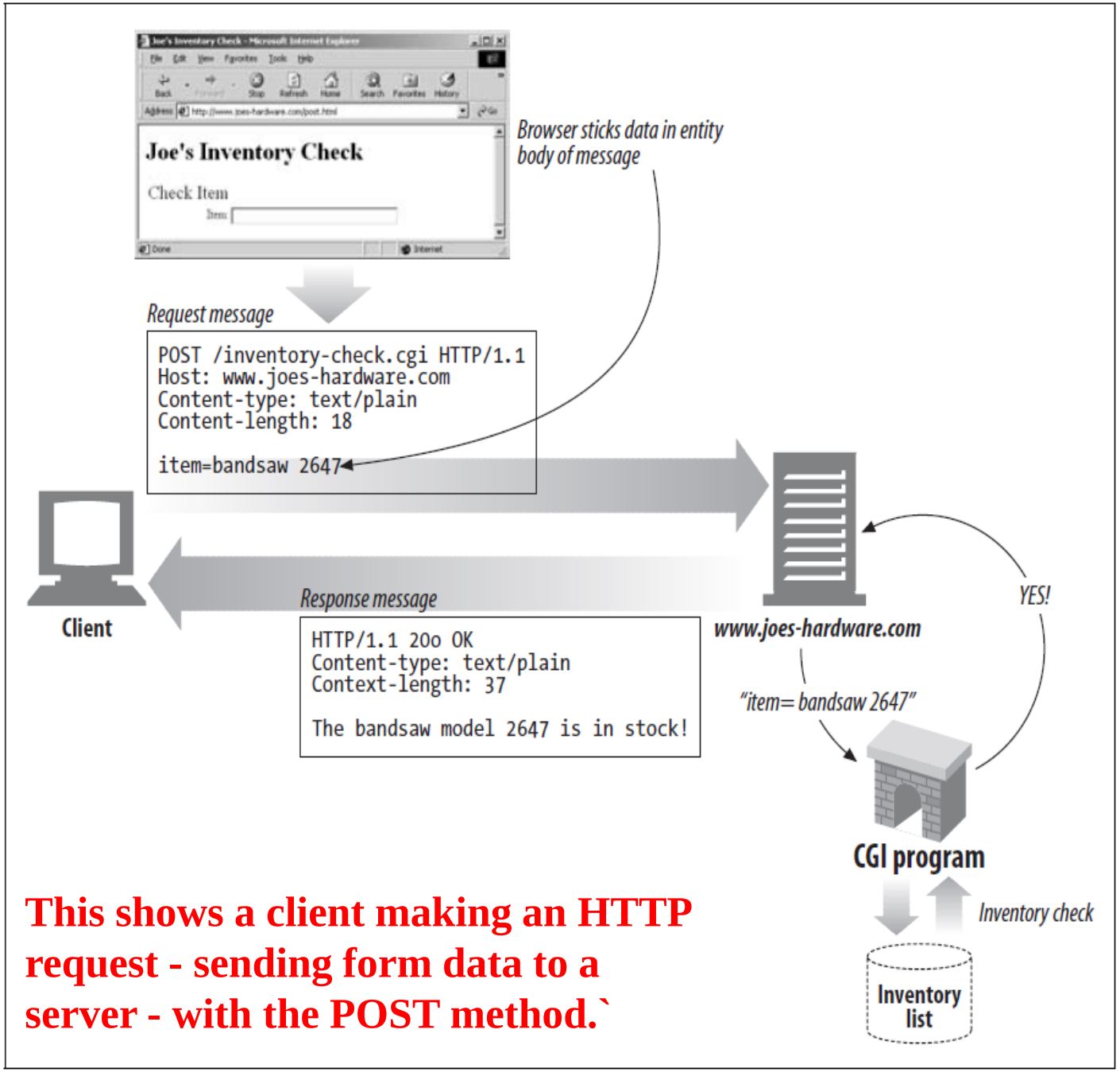


Figure 3-10. POST example

The POST Request: An Example

- This POST request sends form data to a server:

  POST /cgi-bin/register.pl HTTP 1.0
  Date: Sun, 27 Apr 2013 12:32:36
  Host: www.cafeaulait.org
  Content-type: application/x-www-form-urlencoded
  Content-length: 54
  [Blank Line]
  username=Elliotte+Harold&email=elharo%40ibiblio.org

- The body contains an *application/x-www-form-urlencoded* data. The HTTP header should include two fields that specify the nature of the body.
  - A Content-length field that specifies how many bytes are in the body (54 in thepreceding example)
  - A Content-type field that specifies the **MIME media type**. (later)

Table 3-1. Common HTTP methods

Method	Description	Message body?
GET	Get a document from the server.	No
HEAD	Get just the headers for a document from the server.	No
POST	Send data to the server for processing.	Yes
PUT	Store the body of the request on the server.	Yes
TRACE	Trace the message through proxy servers to the server.	No
OPTIONS	Determine what methods can operate on a server.	No
DELETE	Remove a document from the server.	No

The PUT Request

- The PUT method writes documents to a server, in the inverse of the way that GET reads documents from a server.
  - In comparison, POST is used to send data to a server, while PUT is used to deposit data into a resource on the server (e.g., a file).

- Some publishing systems let you create web pages and install them directly on a web server using PUT. (see next slide)

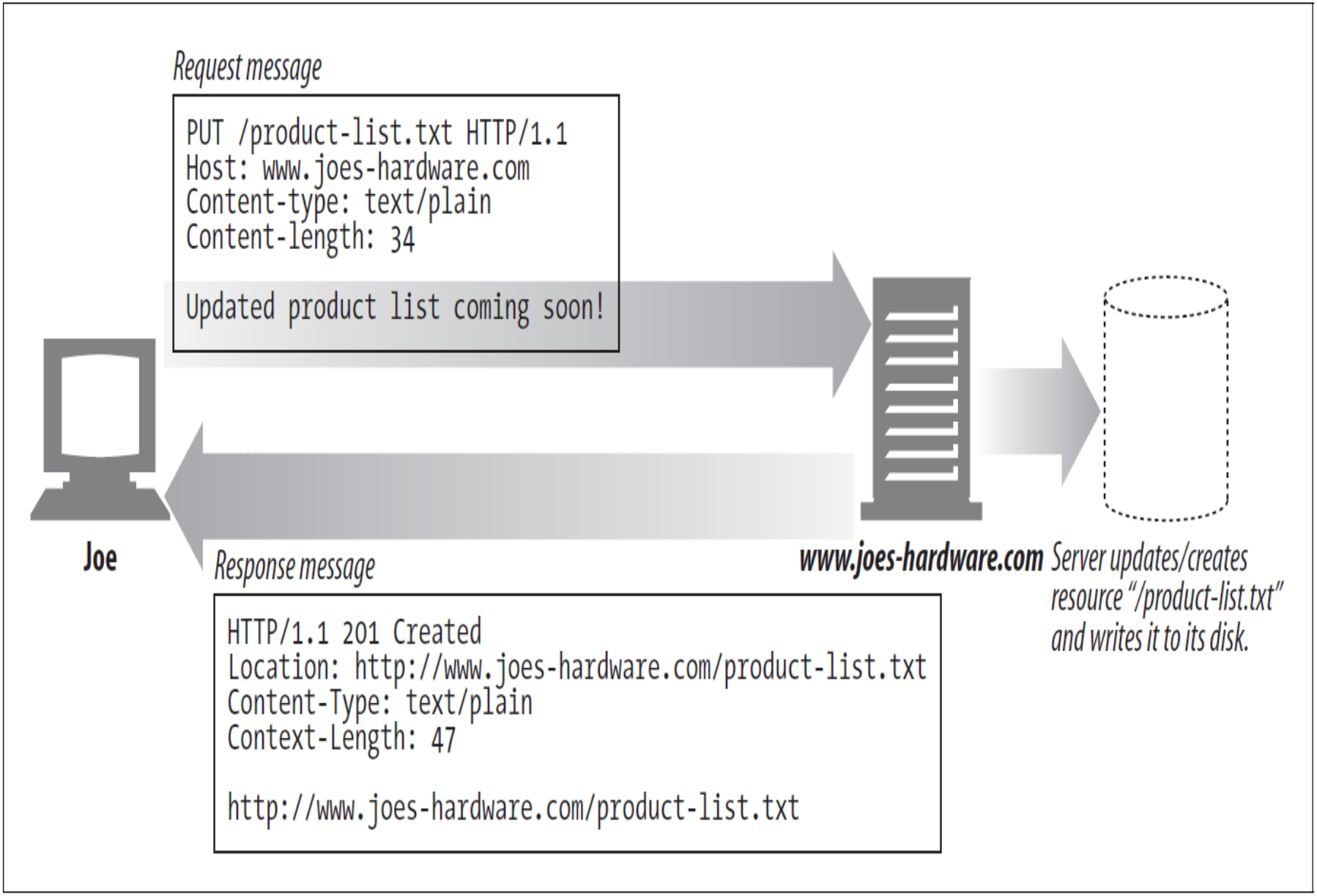


Figure 3-9. PUT example

The PUT Request, cont.

- The semantics of the PUT method are for the server to take the body of the request and either use it to create a new document/file named by the requested URL or, if that URL already exists, use the body to replace it.

- Because PUT allows you to change content, many web servers require you to log in with a password before you can perform a PUT. (We will discuss Authentication/Authorization later)

- The ‘PUT’ method is usually not supported.
  - Files are normally ‘**posted**’ to a processing program (as form data).
  - The processing program that handles these posted data can then select where the data are to be saved.

Table 3-1. Common HTTP methods

Method	Description	Message body?
GET	Get a document from the server.	No
HEAD	Get just the headers for a document from the server.	No
POST	Send data to the server for processing.	Yes
PUT	Store the body of the request on the server.	Yes
TRACE	Trace the message through proxy servers to the server.	No
OPTIONS	Determine what methods can operate on a server.	No
DELETE	Remove a document from the server.	No

The TRACE Request

- When a client makes a request, that request may have to travel through firewalls, proxies, gateways, or other applications.
  - Each of these has the opportunity to modify the original HTTP request.
  - The TRACE method allows clients to see how its request looks when it finally makes it to the server – **good for debugging**.
- A TRACE request initiates a “loopback” diagnostic at the destination server.
  - The server at the final leg of the trip bounces back a TRACE response, with the original request message it received in the body of its response.
  - A client can then see how its original message was modified. (see next slide)

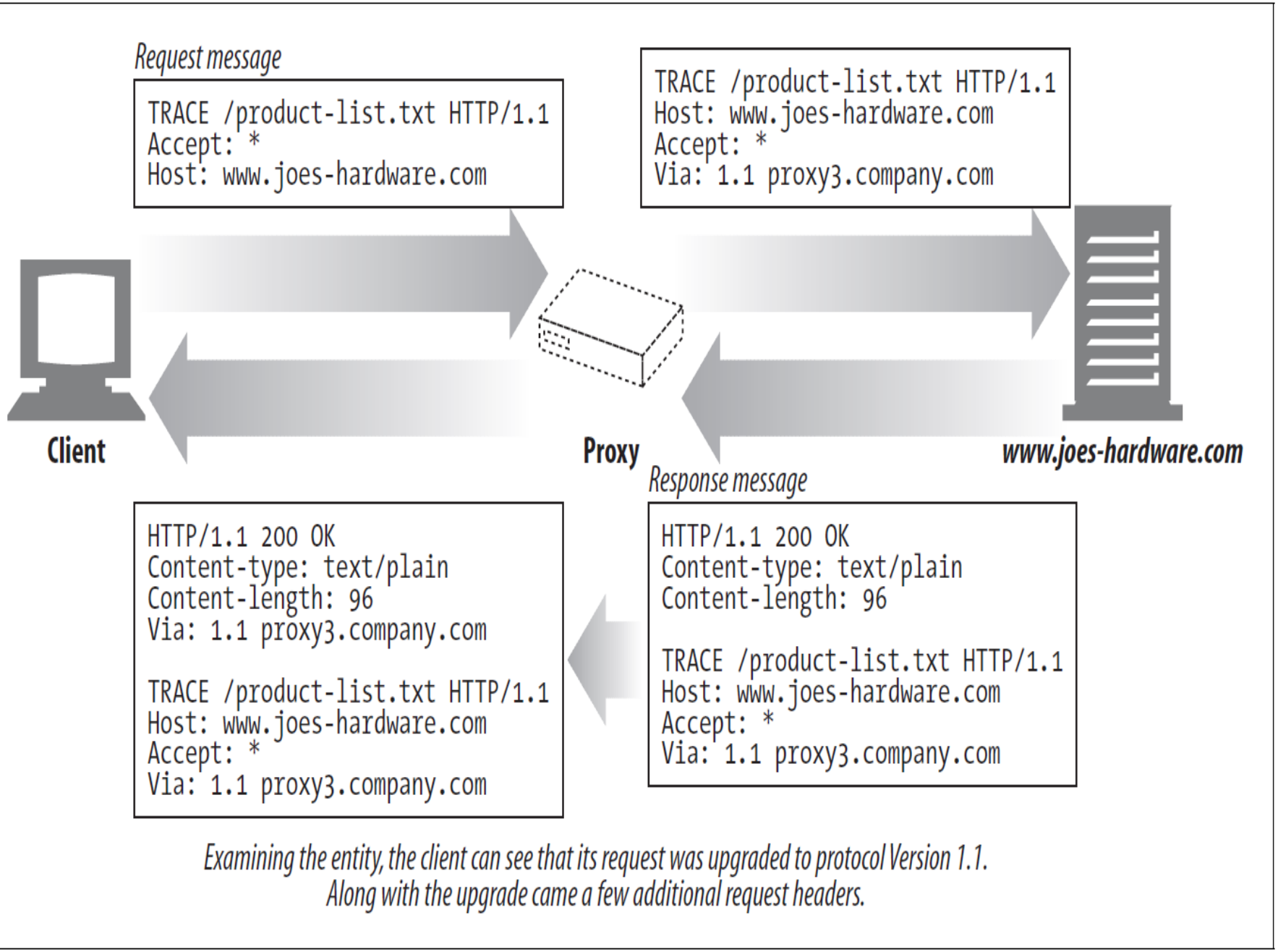


Figure 3-11. TRACE example

Table 3-1. Common HTTP methods

Method	Description	Message body?
GET	Get a document from the server.	No
HEAD	Get just the headers for a document from the server.	No
POST	Send data to the server for processing.	Yes
PUT	Store the body of the request on the server.	Yes
TRACE	Trace the message through proxy servers to the server.	No
OPTIONS	Determine what methods can operate on a server.	No
DELETE	Remove a document from the server.	No

The OPTIONS Request

- The OPTIONS method asks the server to tell us about the various supported capabilities of the web server, without initiating a resource retrieval or action.
  - You can ask a server about what methods it supports in general or for particular resources. (Some servers may support particular operations only on particular kinds of objects).
    - For example, a client might use such a request to determine whether a server could switch to secure communications.
  - This provides a means for client applications to determine how best to access various resources without actually having to access them.
- Next slide shows a request scenario using the OPTIONS method

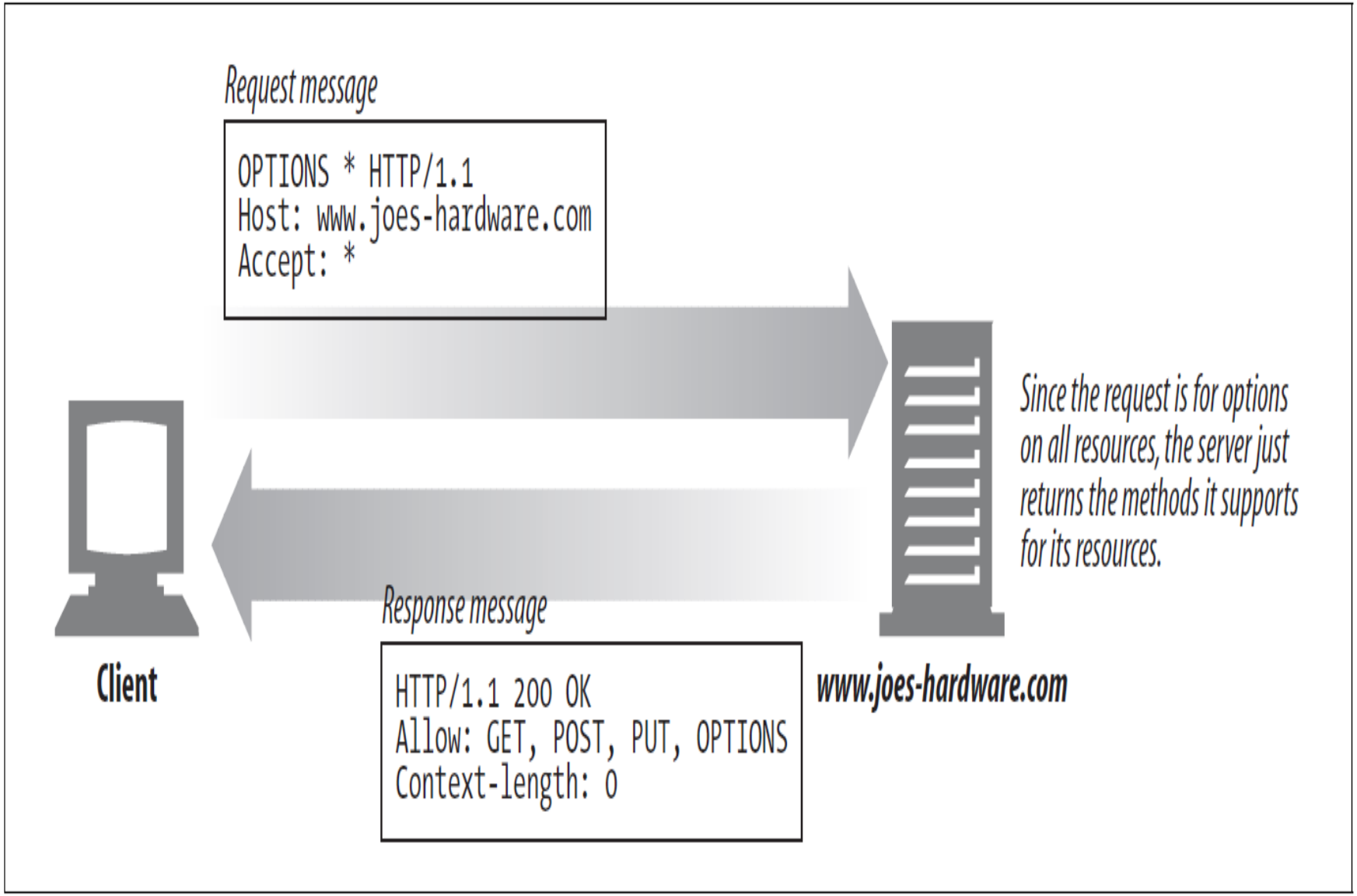


Figure 3-12. OPTIONS example

Table 3-1. Common HTTP methods

Method	Description	Message body?
GET	Get a document from the server.	No
HEAD	Get just the headers for a document from the server.	No
POST	Send data to the server for processing.	Yes
PUT	Store the body of the request on the server.	Yes
TRACE	Trace the message through proxy servers to the server.	No
OPTIONS	Determine what methods can operate on a server.	No
DELETE	Remove a document from the server.	No

The DELETE Request

- The DELETE method does just what you would think—it asks the server to delete the resources specified by the request URL.
  - However, the client application is not guaranteed that the delete is carried out.
  - This is because the HTTP specification allows the server to override the request without telling the client.
- If you aren't sure whether a DELETE request succeeded, because the socket got disconnected after you sent the request but before you received a response, you could just send the request again.
- The DELETE method is rarely supported.
  - You might want to support this method in a specialized context on a local intranet.
- Next slide shows an example of the DELETE method.

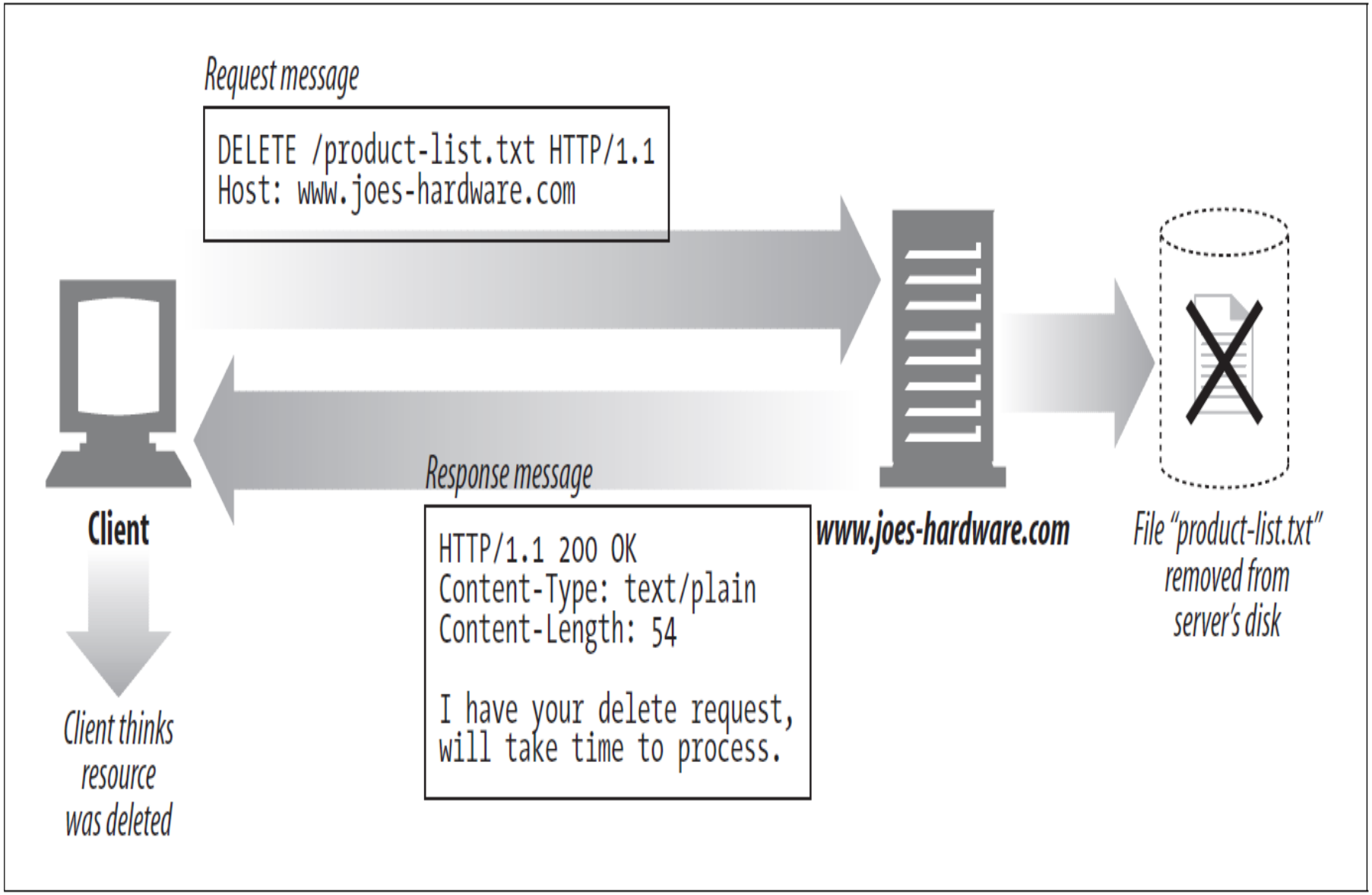


Figure 3-13. DELETE example

HTTP Requests Revisited

- A request has the form:

  Request =

  Request-Line

  *((**general-header** | request-header| entity-header) **CRLF**)

  CRLF

  [message-body]

  - The first line of a request specifies the **request method** ('GET', 'POST, etc.), the 'resource' and the protocol being used (HTTP/1.0 or HTTP/1.1).
  - The next few lines are headers.
    - These lines have a keyword that identifies the type of general header or request header, or entity header, and some data.
  - There will be a blank line following these header data, and then possibly a message body ('entity').

General Headers

- **General headers**
  - These are generic headers used by both clients and servers.
  - They serve general purposes that are useful for clients, servers, and other applications to supply to one another.
    - For example, the **Date** header is a general-purpose header that allows both sides to indicate the time and date at which the message was constructed:

Date: Wed, 11 November 2015 02:30:00 GMT

Table 3-11 lists the general informational headers.

Table 3-11. General informational headers

Header	Description
Connection	Allows clients and servers to specify options about the request/response connection
Date[a]	Provides a date and time stamp telling when the message was created
MIME-Version	Gives the version of MIME that the sender is using
Trailer	Lists the set of headers that are in the trailer of a message encoded with the chunked transfer encoding[b]
Transfer-Encoding	Tells the receiver what encoding was performed on the message in order for it to be transported safely
Upgrade	Gives a new version or protocol that the sender would like to "upgrade" to using
Via	Shows what intermediaries (proxies, gateways) the message has gone through

[a] Appendix C lists the acceptable date formats for the Date header.

[b] Chunked transfer codings are discussed further in "Chunking and persistent connections" in Chapter 15.

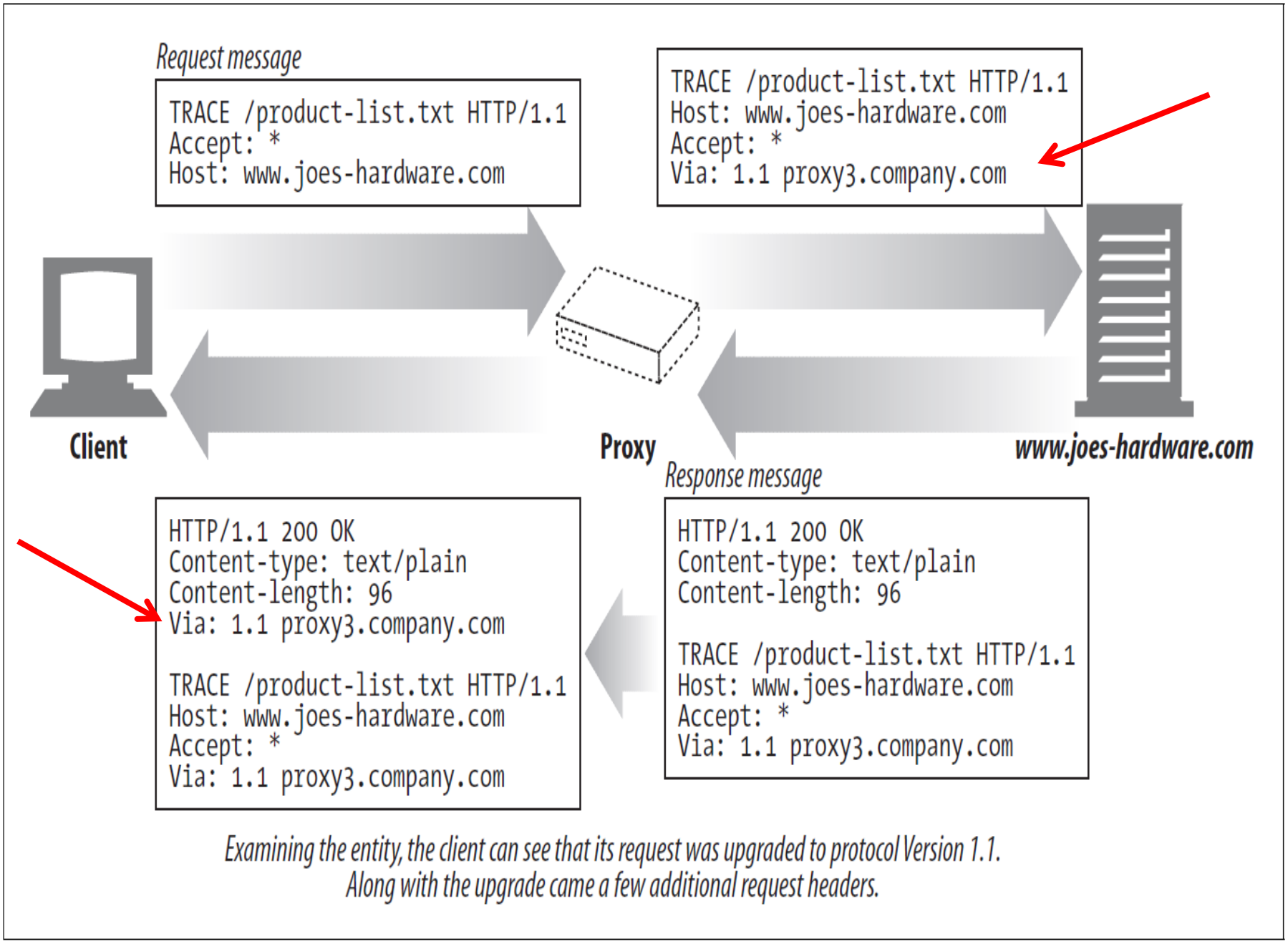


Figure 3-11. TRACE example

Table 3-11 lists the general informational headers.

Table 3-11. General informational headers

Header	Description
Connection	Allows clients and servers to specify options about the request/response connection
Date[a]	Provides a date and time stamp telling when the message was created
MIME-Version	Gives the version of MIME that the sender is using
Trailer	Lists the set of headers that are in the trailer of a message encoded with the chunked transfer encoding[b]
Transfer-Encoding	Tells the receiver what encoding was performed on the message in order for it to be transported safely
Upgrade	Gives a new version or protocol that the sender would like to "upgrade" to using
Via	Shows what intermediaries (proxies, gateways) the message has gone through

[a] Appendix C lists the acceptable date formats for the Date header.

[b] Chunked transfer codings are discussed further in "Chunking and persistent connections" in Chapter 15.

Requests [edit]

Field name	Description	Example
Accept	Content-Types that are acceptable for the response	`Accept: text/plain`
Accept-Charset	Character sets that are acceptable	`Accept-Charset: utf-8`
Accept-Encoding	Acceptable encodings. See HTTP compression.	`Accept-Encoding: gzip, deflate`
Accept-Language	Acceptable human languages for response	`Accept-Language: en-US`
Accept-Datetime	Acceptable version in time	`Accept-Datetime: Thu, 31 May 2007 20:35:00 GMT`
Authorization	Authentication credentials for HTTP authentication	`Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`
Cache-Control	Used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain	`Cache-Control: no-cache`
Connection	What type of connection the user-agent would prefer	`Connection: keep-alive`
Cookie	an HTTP cookie previously sent by the server with `Set-Cookie` (below)	`Cookie: $Version=1; Skin=new;`
Content-Length	The length of the request body in octets (8-bit bytes)	`Content-Length: 348`
Content-MD5	A Base64-encoded binary MD5 sum of the	

Client

Keep Alive

- In HTTP/1.0, the server closes its end of a TCP/IP connection once it has sent its response. This may result in quite inefficient processing of requests.
  - A typical HTML page contains links to images and to stylesheet files.
  - The client browser that parses the HTML will find these links and must download these files before it can display the page to the user.
  - Consequently, the browser must make new connections to the server for each of the files.
  - The opening and closing of a TCP/IP connection has a significant cost and takes time.

Keep Alive

- Reuse of a connection is a standard feature in HTTP/1.1.
  - A client can specify in a request header that it wants its connection to **keep alive** after the first response
  - A server can indicate in a response header that it will be closing the connection once it has completed its current response.

Responses [edit]

Field name	Description	Example
Access-Control-Allow-Origin	Specifying which web sites can participate in cross-origin resource sharing	`Access-Control-Allow-Origin: *`
Accept-Ranges	What partial content range types this server supports	`Accept-Ranges: bytes`
Age	The age the object has been in a proxy cache in seconds	`Age: 12`
Allow	Valid actions for a specified resource. To be used for a *405 Method not allowed*	`Allow: GET, HEAD`
Cache-Control	Tells all caching mechanisms from server to client whether they may cache this object. It is measured in seconds	`Cache-Control: max-age=3600`
Connection	Options that are desired for the connection[21]	`Connection: close`
Content-Encoding	The type of encoding used on the data. See HTTP compression.	`Content-Encoding: gzip`
Content-Language	The language the content is in	`Content-Language: da`
	The length of the response body in	

Server

Keep-Alive

- **Keep-Alive**
  - If no data has been exchanged for a while (default: 2 hours), each endpoint may be wondering if the other is still around.
  - TCP provides a keep-alive mechanism where, after a certain time of inactivity, a probe message is sent to the other endpoint.
  - If the endpoint is alive and well, it sends an acknowledgment.
  - After a few retries (default: 9 times, every 75 seconds) without acknowledgment, the probe sender gives up and closes the socket, causing an exception on the next attempted I/O operation.
  - Note that the application only sees keep-alive working if the probes **fail**.

setKeepAlive

```
public void setKeepAlive(boolean on)
                  throws SocketException
```

For Socket

Enable/disable SO_KEEPALIVE.

Parameters:

`on` - whether or not to have socket keep alive turned on.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

1.3

See Also:

`getKeepAlive()`

getKeepAlive

```
public boolean getKeepAlive()
                    throws SocketException
```

Tests if SO_KEEPALIVE is enabled.

Returns:

a `boolean` indicating whether or not SO_KEEPALIVE is enabled.

getKeepAlive

For Socket

```
public boolean getKeepAlive()
                    throws SocketException
```

Tests if SO_KEEPALIVE is enabled.

Returns:

a `boolean` indicating whether or not SO_KEEPALIVE is enabled.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

1.3

See Also:

`setKeepAlive(boolean)`

setTrafficClass

```
public void setTrafficClass(int tc)
                    throws SocketException
```

Sets traffic class or type-of-service octet in the IP header for packets sent from this Socket. As the underlying network implementation may ignore this value applications should consider it a hint.

The tc **must** be in the range `0 <= tc <= 255` or an IllegalArgumentException will be thrown.

Request Headers

- **Request headers**
  - As the name implies, request headers are specific to request messages.
  - They provide extra information to servers, such as what type of data the client is willing to receive.
  - For example, the following **Accept** header tells the server that the client will accept any media type that matches its request:

    Accept: */*

Internet media type

From Wikipedia, the free encyclopedia

An **Internet media type**[1] is a two-part identifier for file formats on the Internet. The identifiers were originally defined in RFC 2046 for use in email sent through SMTP, but their use has expanded to other protocols such as HTTP, RTP and SIP. These types were called **MIME types**, and are sometimes referred to as **Content-types**, after the name of a header in several protocols whose value is such a type. The original name **MIME type** referred to usage to identify non-ASCII parts of email messages composed using the MIME (Multipurpose Internet Mail Extensions) specification. Without MIME types, email clients would not be able to understand if an attachment file were a graphics file or a spreadsheet etc. and would not be able to handle the attachment appropriately.

A media type is composed of two or more parts: A *type*, a *subtype*, and zero or more optional parameters. For example, subtypes of `text` have an optional `charset` parameter that can be included to indicate the character encoding (e.g. `text/html; charset=UTF-8`), and subtypes of `multipart` type often define a `boundary` between parts. Allowed `charset` values are defined in the list of IANA character sets.

Prior to RFC 6648,[2] experimental or non-standard[3] media types were prefixed with `x-`, but this practice was deprecated due to incompatibility problems when the experimental types were standardized. Subtypes that begin with `vnd.` are vendor-specific;[4] subtypes that begin with `prs.` are in the personal or vanity tree.[5] New media types can be created with the procedures outlined in RFC 4288.

In addition to email clients, web browsers also support various media types. This enables the browser to display or output files

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export

Languages
Česky

MIME

- MIME types are classified at two levels: a **type** and a **subtype**.
  - The type shows very generally what kind of data is contained: is it a picture, text, or movie?
  - The subtype identifies the specific type of data: GIF image, JPEG image, TIFF image.
    - For example, HTML's content type is **text/html**; the type is text, and the subtype is html.
    - The content type for a JPEG image is **image/jpeg**; the type is image, and the subtype is jpeg.

MIME

- Eight top-level types have been defined:
  - text/* : for human-readable words
  - image/* : for pictures
  - model/* : for 3D models such as VRML files
  - audio/* : for sound
  - video/* : for moving pictures, possibly including sound
  - application/* : for binary data
  - message/* : for protocol-specific envelopes such as email messages and HTTP responses
  - multipart/* : for containers of multiple documents and resources
- Each of these has many different subtypes.

Type `audio` [edit]

For Audio.

- `audio/basic`: mulaw audio at 8 kHz, 1 channel; Defined in RFC 2046
- `audio/L24`: 24bit Linear PCM audio at 8–48 kHz, 1-N channels; Defined in RFC 3190
- `audio/mp4`: MP4 audio
- `audio/mpeg`: MP3 or other MPEG audio; Defined in RFC 3003
- `audio/ogg`: Ogg Vorbis, Speex, Flac and other audio; Defined in RFC 5334
- `audio/vorbis`: Vorbis encoded audio; Defined in RFC 5215
- `audio/vnd.rn-realaudio`: RealAudio; Documented in RealPlayer Help[13]
- `audio/vnd.wave`: WAV audio; Defined in RFC 2361
- `audio/webm`: WebM open media format

Type `image` [edit]

- `image/gif`: GIF image; Defined in RFC 2045 and RFC 2046
- `image/jpeg`: JPEG JFIF image; Defined in RFC 2045 and RFC 2046
- `image/pjpeg`: JPEG JFIF image; Associated with Internet Explorer; Listed in ms775147(v=vs.85) - Progressive JPEG, initiated before global browser support for progressive JPEGs (Microsoft and Firefox).
- `image/png`: Portable Network Graphics; Registered,[14] Defined in RFC 2083
- `image/svg+xml`: SVG vector image; Defined in SVG Tiny 1.2 Specification Appendix M
- `image/tiff`: Tag Image File Format (only for Baseline TIFF); Defined in RFC 3302
- `image/vnd.microsoft.icon`: ICO image; Registered[15]

Type `message` [edit]

- `message/http`: Defined in RFC 2616

MIME

- The most current list of registered MIME types is available from:

  http://www.iana.org/assignments/media-types/

- In addition, nonstandard custom types and subtypes can be freely defined as long as they begin with *x-*.
  - For example, **Flash** files are commonly assigned the type:

    application/x-shockwave-flash

Header: Accept

- The Accept request-header field can be used to specify certain media types which are acceptable for the response.

  Accept = "**Accept**" ":"
  #(media-range [accept-params])
  media-range = ("*/*"
  | (type "/" "*")
  | (type "/" subtype)
) *(";" parameter)
  accept-params = ";" "**q**" "=" qvalue *(accept-extension)
  accept-extension = ";" token ["=" (token | quoted-string)]

- An example: **Accept:** audio/*; q=0.2, audio/basic
  - SHOULD be interpreted as "I prefer audio/basic, but send me any audio type if it is the best available after an 80% mark-down in quality." (The default value is q=1.)

Header: Accept-Charset

- The Accept-Charset request-header field can be used to indicate what character sets are acceptable for the response.
  - This field allows clients capable of understanding more comprehensive or special-purpose character sets to signal that capability to a server which is capable of representing documents in those character sets.

Accept-Charset = "Accept-Charset" ":"
#((charset | "*")[";" "q" "=" qvalue])

- Each charset MAY be given an associated quality value which represents the user's preference for that charset.
  - The default value is q=1.
- An example:

Accept-Charset: iso-8859-5, unicode-1-1;q=0.8

Header: Accept-Encoding

- The Accept-Encoding request-header field is similar to Accept, but restricts the content **codings** instead of media types.

  Accept-Encoding = "Accept-Encoding" ":"
  1# (codings [";" "q" "=" qvalue])

  codings = (content-coding | "*")

- Examples of its use are:
  - Accept-Encoding: compress, gzip
  - Accept-Encoding: *
  - Accept-Encoding: compress;q=0.5, gzip;q=1.0
  - Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0

- If an Accept-Encoding field is present, but the server cannot send an acceptable response, then the server SHOULD send an error response with the **406 (Not Acceptable)** status code.

Header: Host

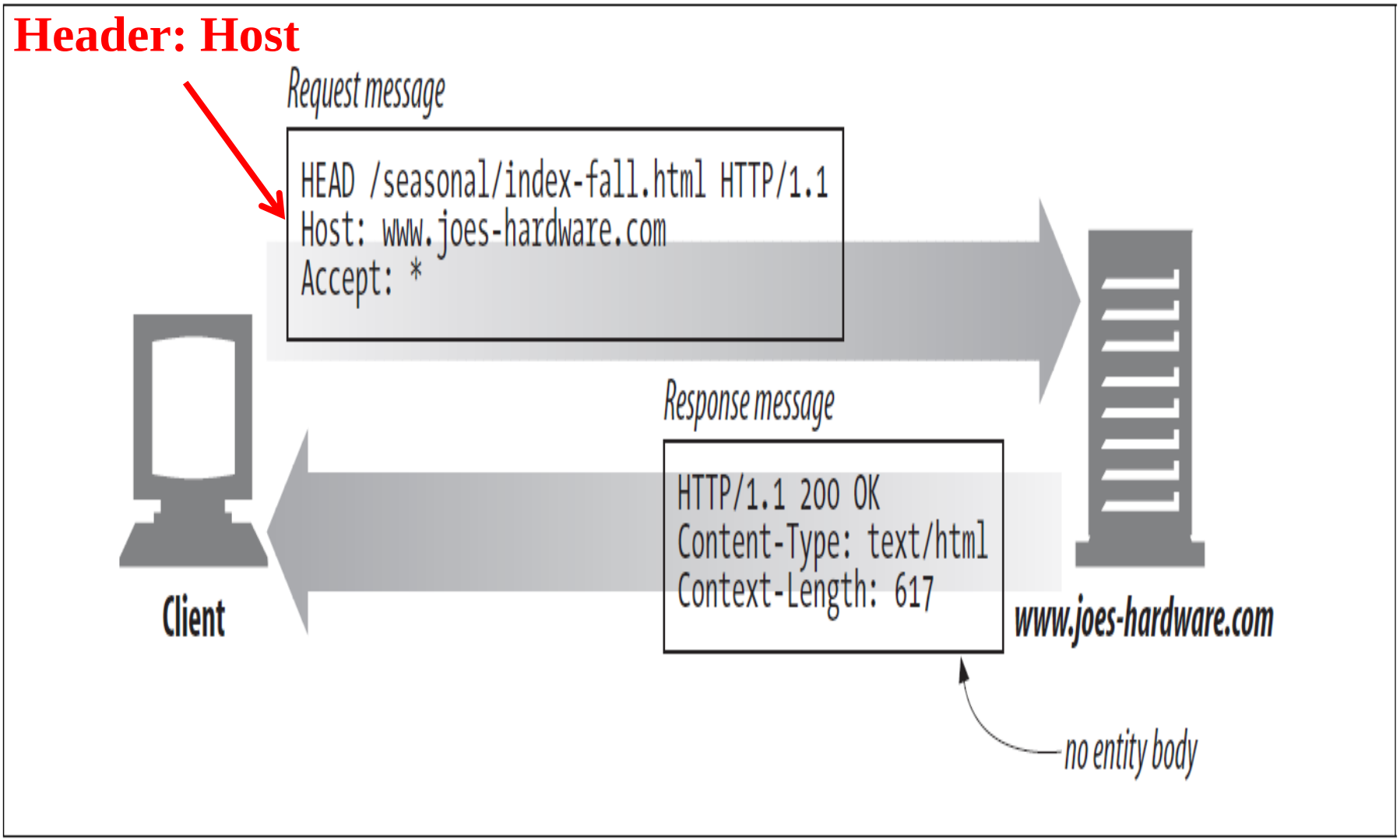
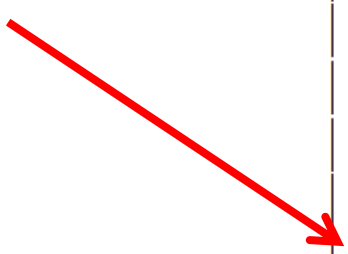


Figure 3-8. HEAD example

5.3 Request Header Fields

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method invocation.

```
request-header = Accept                   ; Section 14.1
               | Accept-Charset           ; Section 14.2
               | Accept-Encoding          ; Section 14.3
               | Accept-Language          ; Section 14.4
               | Authorization            ; Section 14.8
               | Expect                   ; Section 14.20
               | From                     ; Section 14.22
               | Host                     ; Section 14.23
               | If-Match                 ; Section 14.24
               | If-Modified-Since        ; Section 14.25
               | If-None-Match            ; Section 14.26
               | If-Range                 ; Section 14.27
               | If-Unmodified-Since      ; Section 14.28
               | Max-Forwards             ; Section 14.31
               | Proxy-Authorization      ; Section 14.34
               | Range                    ; Section 14.35
               | Referer                  ; Section 14.36
               | TE                       ; Section 14.39
               | User-Agent               ; Section 14.43
```



Header: User-Agent

- A user agent acts as a client in a network protocol used in communications within a client–server computing system.
  - In particular, the HTTP identifies the client software originating the request, using a **"User-Agent"** header, even when the client is not operated by a user.
  - Example:

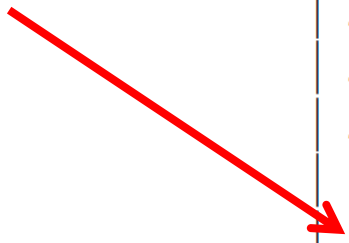
    User-Agent: Mozilla/5.0 (iPad; U; CPU OS 3_2_1 like Mac OS X; en-us) AppleWebKit/531.21.10 (KHTML, like Gecko) Mobile/7B405

Header: User-Agent

- The components of this string are as follows:
  - *Mozilla/5.0*: The client software (browser)
  - *(iPad; U; CPU OS 3_2_1 like Mac OS X; en-us)*: Details of the system in which the browser is running
  - *AppleWebKit/531.21.10*: The platform the browser uses
  - *(KHTML, like Gecko)*: Browser platform details
  - *Mobile/7B405*: This is used by the browser to indicate specific enhancements that are available directly in the browser or through third parties.
    - An example of this is Microsoft Live Meeting which registers an extension so that the Live Meeting service knows if the software is already installed, which means it can provide a streamlined experience to joining meetings.

5.3 Request Header Fields

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method invocation.



```
request-header = Accept                   ; Section 14.1
               | Accept-Charset           ; Section 14.2
               | Accept-Encoding          ; Section 14.3
               | Accept-Language          ; Section 14.4
               | Authorization            ; Section 14.8
               | Expect                   ; Section 14.20
               | From                     ; Section 14.22
               | Host                     ; Section 14.23
               | If-Match                 ; Section 14.24
               | If-Modified-Since        ; Section 14.25
               | If-None-Match            ; Section 14.26
               | If-Range                 ; Section 14.27
               | If-Unmodified-Since      ; Section 14.28
               | Max-Forwards             ; Section 14.31
               | Proxy-Authorization      ; Section 14.34
               | Range                    ; Section 14.35
               | Referer                  ; Section 14.36
               | TE                       ; Section 14.39
               | User-Agent               ; Section 14.43
```

Header: From

- The From request-header field, if given, SHOULD contain an Internet e-mail address for the human user who controls the requesting user agent.
  - The address SHOULD be machine-usable, as defined by "mailbox" in RFC 822 as updated by RFC 1123:

    From = "From" ":" mailbox

  - An example:

    From: webmaster@w3.org

Header: If-Modified-Since

- The If-Modified-Since request-header field is used with a method to make it conditional:
  - If the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server.
    - Instead, a **304 (not modified)** response will be returned without any message-body.

If-Modified-Since = "If-Modified-Since" ":" HTTP-date

- An example of the field is:

If-Modified-Since: Sun, 13 Nov 2016 00:00:00 GMT

Entity Headers

- **Entity Headers**
  - Entity headers refer to headers that deal with the message-body (aka Entity).
  - For example, entity headers can tell the type of the data in the entity body.
    - The following **Content-Type** header lets the application know that the data is an HTML document in the iso-latin-1 character set:

      Content-Type: text/html; charset=iso-latin-1

- Note: Latin alphabet No. 1 is part of the ISO 8859 series of ASCII-based standard character encodings, first edition published in 1987. It is generally intended for Western European languages.

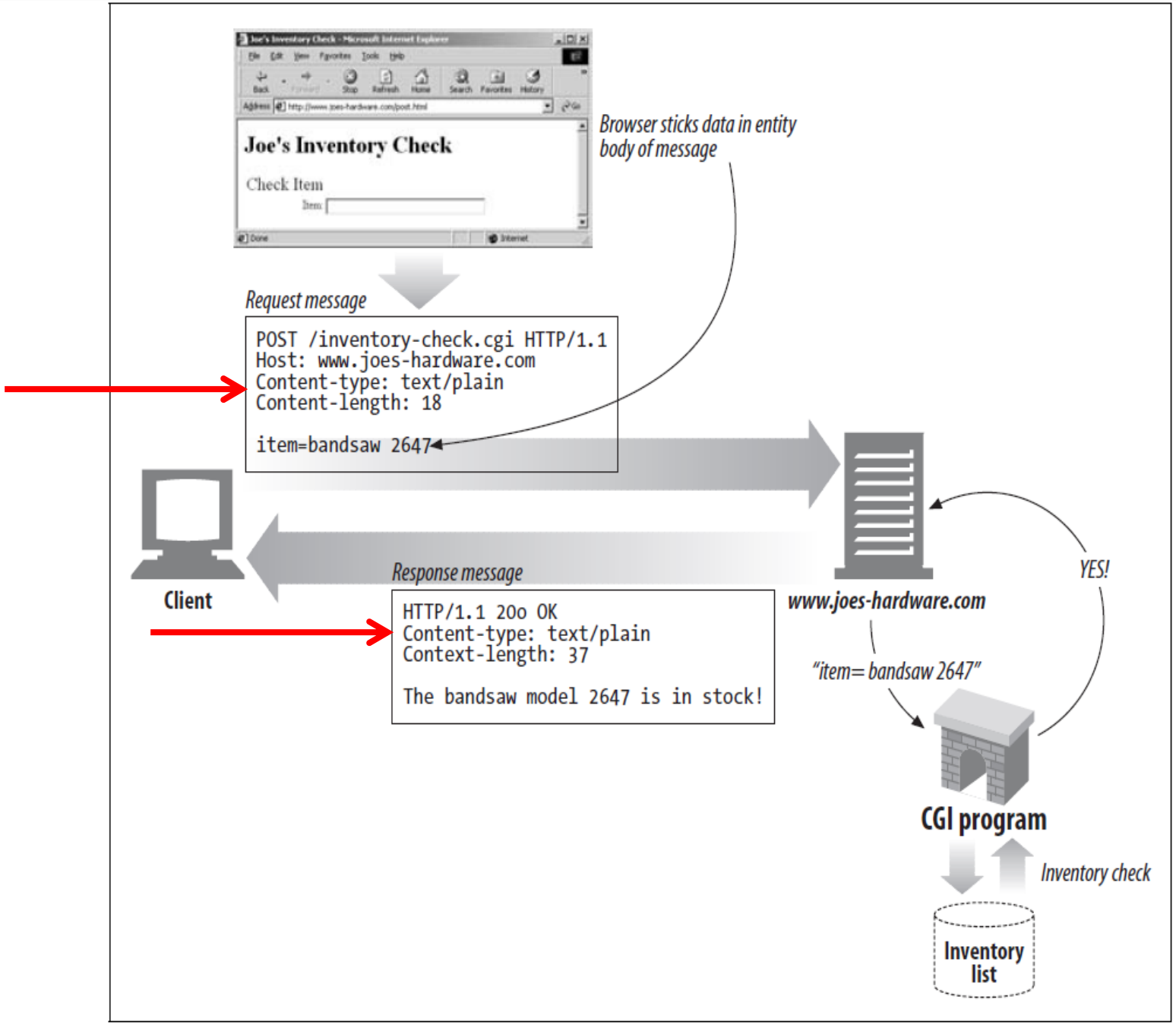


Figure 3-10. POST example

List of HTTP header fields

From Wikipedia, the free encyclopedia

HTTP header fields are components of the message header of requests and responses in the Hypertext Transfer Protocol (HTTP). They define the operating parameters of an HTTP transaction.

The header fields are transmitted after the request or response line, the first line of a message. Header fields are colon-separated name-value pairs in clear-text string format, terminated by a carriage return (CR) and line feed (LF) character sequence. The end of the header fields is indicated by an empty field, resulting in the transmission of two consecutive CR-LF pairs. Long lines can be folded into multiple lines; continuation lines are indicated by presence of space (SP) or

HTTP

Persistence · Compression · HTTPS

Request methods

OPTIONS · GET · HEAD · POST · PUT · DELETE · TRACE · CONNECT · PATCH

Header fields

Cookie · ETag · Location · HTTP referer · DNT · X-Forwarded-For

Status codes

301 Moved permanently · 302 Found ·

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export

Languages
Deutsch

Responses

- HTTP response messages have the form:

  Response = Status-Line

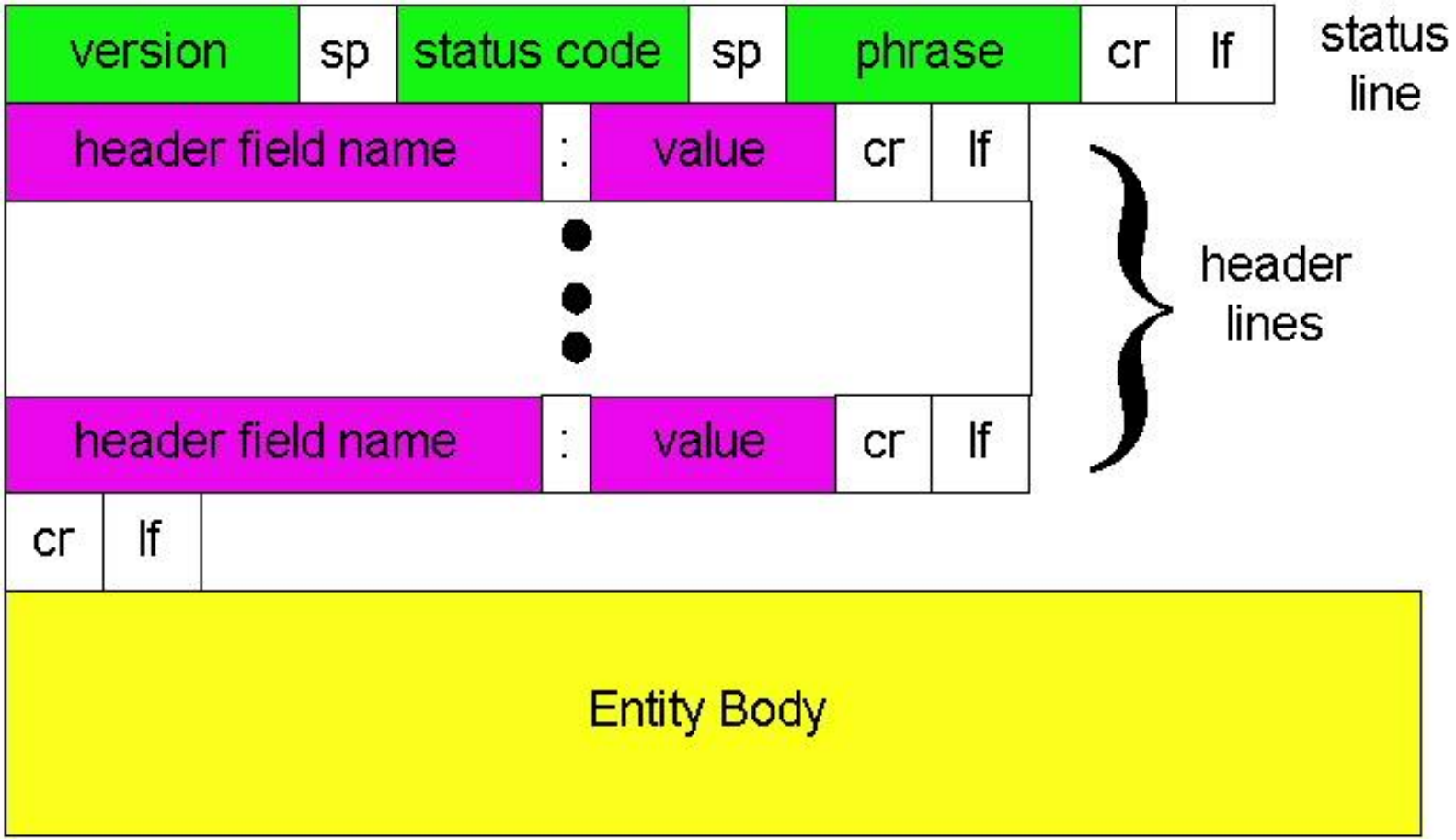
  *((general-header | response-header | entity-header) **CRLF**)

  CRLF

  [message-body]

- This says that a response is a status line, followed by some number of headers, a blank line and an optional message body.
- The status line specifies the **HTTP protocol version**, **status code** and **phrase**.
- There are five groups of status codes:

HTTP Response Message Format



Responses: An Example

- The following is an example of an HTTP response:

```
HTTP/1.1 200 OK
Date: Mon, 4 Feb 2013 13:13:33 GMT
Content-Type: text/html
Last-Modified: Mon, 11 Feb 1998 13:23:42 GMT
Content-Length: 112
CRLF
<html>
<head>
<title>HTTP Response Example</title></head>
<body> Welcome to Brainy Software </body>
</html>
```

Status Codes

- HTTP status codes are classified into five broad categories:
  - 1xx (**Informational**)
  - 2xx (**Success**)
  - 3xx (**Redirection**)
  - 4xx (**Client Error**)
  - 5xx (**Server Error**)

Status Codes – 1xx

- 1xx (**Informational**)
  - This class of status code indicates a provisional response, consisting only of the Status-Line and optional headers, and is terminated by an empty line.
  - 100 Continue - This means that the server has received the request headers, and that the client should proceed to send the request body (in the case of a request for which a body needs to be sent; for example, a POST request).

Status Codes – 2xx

- 2xx (**Success**)
  - This class of status code indicates that the client's request was successfully received, understood, and accepted.

Status code	Reason phrase	Meaning
200	OK	Request is okay, entity body contains requested resource.
201	Created	For requests that create server objects (e.g., PUT). The entity body of the response should contain the various URLs for referencing the created resource, with the Location header containing the most specific reference. See Table 3-21 for more on the Location header. The server must have created the object prior to sending this status code.
202	Accepted	The request was accepted, but the server has not yet performed any action with it. There are no guarantees that the server will complete the request; this just means that the request looked valid when accepted. The server should include an entity body with a description indicating the status of the request and possibly an estimate for when it will be completed (or a pointer to where this information can be obtained).
203	Non-Authoritative Information	The information contained in the entity headers (see "Entity Headers" for more information on entity headers) came not from the origin server but from a copy of the resource. This could happen if an intermediary had a copy of a resource but could not or did not validate the meta-information (headers) it sent about the resource. This response code is not required to be used; it is an option for applications that have a response that would be a 200 status if the entity headers had come from the origin server.
204	No Content	The response message contains headers and a status line, but no entity body. Primarily used to update browsers without having them move to a new document (e.g., refreshing a form page).
205	Reset Content	Another code primarily for browsers. Tells the browser to clear any HTML form elements on the current page.
206	Partial Content	A partial or *range* request was successful. Later, we will see that clients can request part or a range of a document by using special headers—this status code indicates that the range request was successful. See "Range Requests" in Chapter 15 for more on the Range header. A 206 response must include a Content-Range, Date, and either ETag or Content-Location header.

Status Codes – An Example

- The following example of the headers for a HTTP response message is taken from the HTTP/1.1 specification (RFC 2616):

  HTTP/1.1 206 Partial content

  Date: Wed, 15 Nov 1995 06:25:24 GMT

  Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT

  Content-Range: bytes 21010-47021/47022

  Content-Length: 26012

  Content-Type: image/gif

Status Codes – 3xx

- 3xx (**Redirection**)
  - This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request.
    - The action required MAY be carried out by the user agent without interaction with the user if and only if the method used in the second request is GET or HEAD.
      - Example: a resource being temporarily or permanently moved to a new URI. The response should then include a **Location** header that contains this URI. Ideally, a browser will interpret this and proceed by submitting a new request to the specified location.

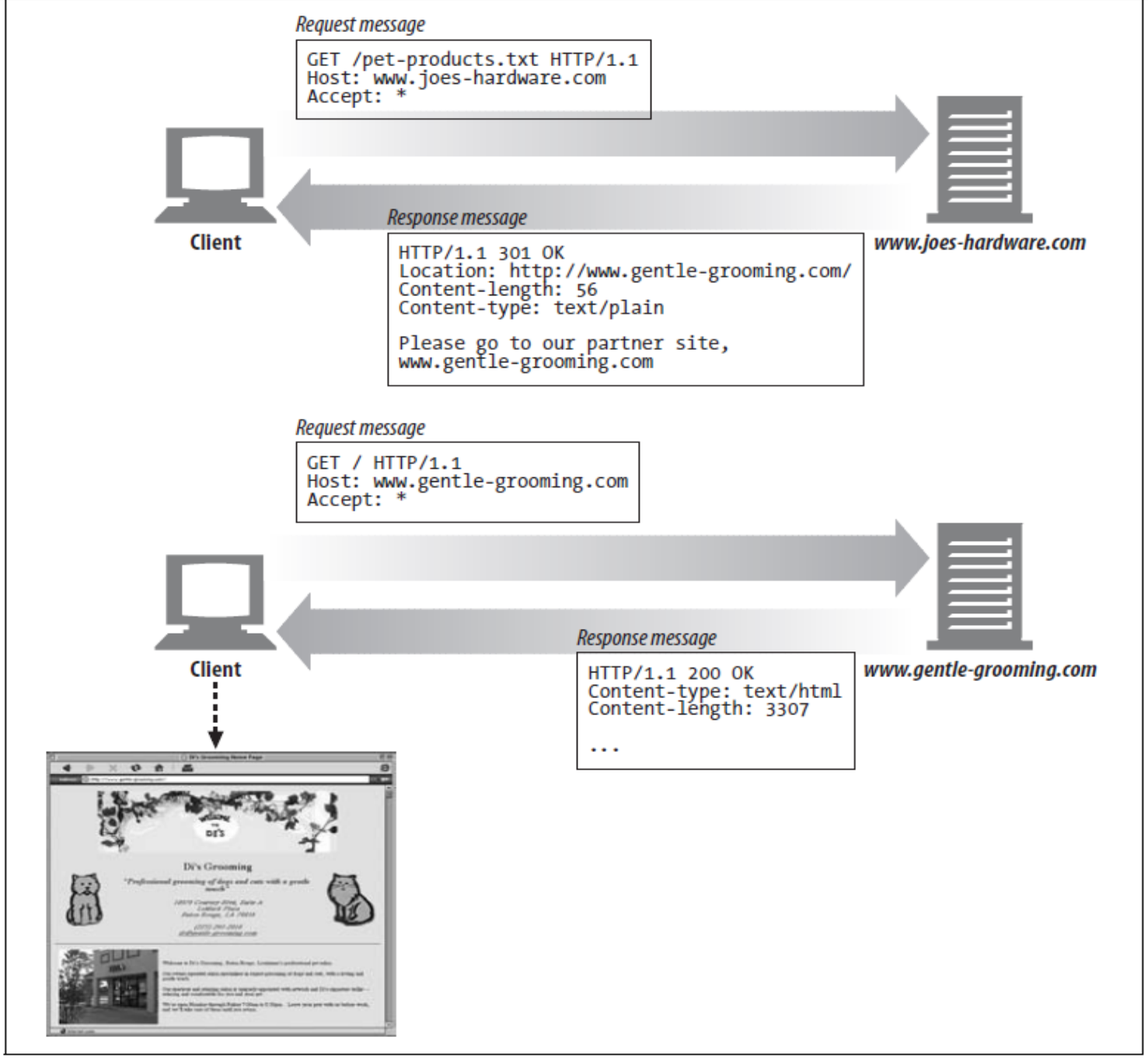


Figure 3-14. Redirected request to new location

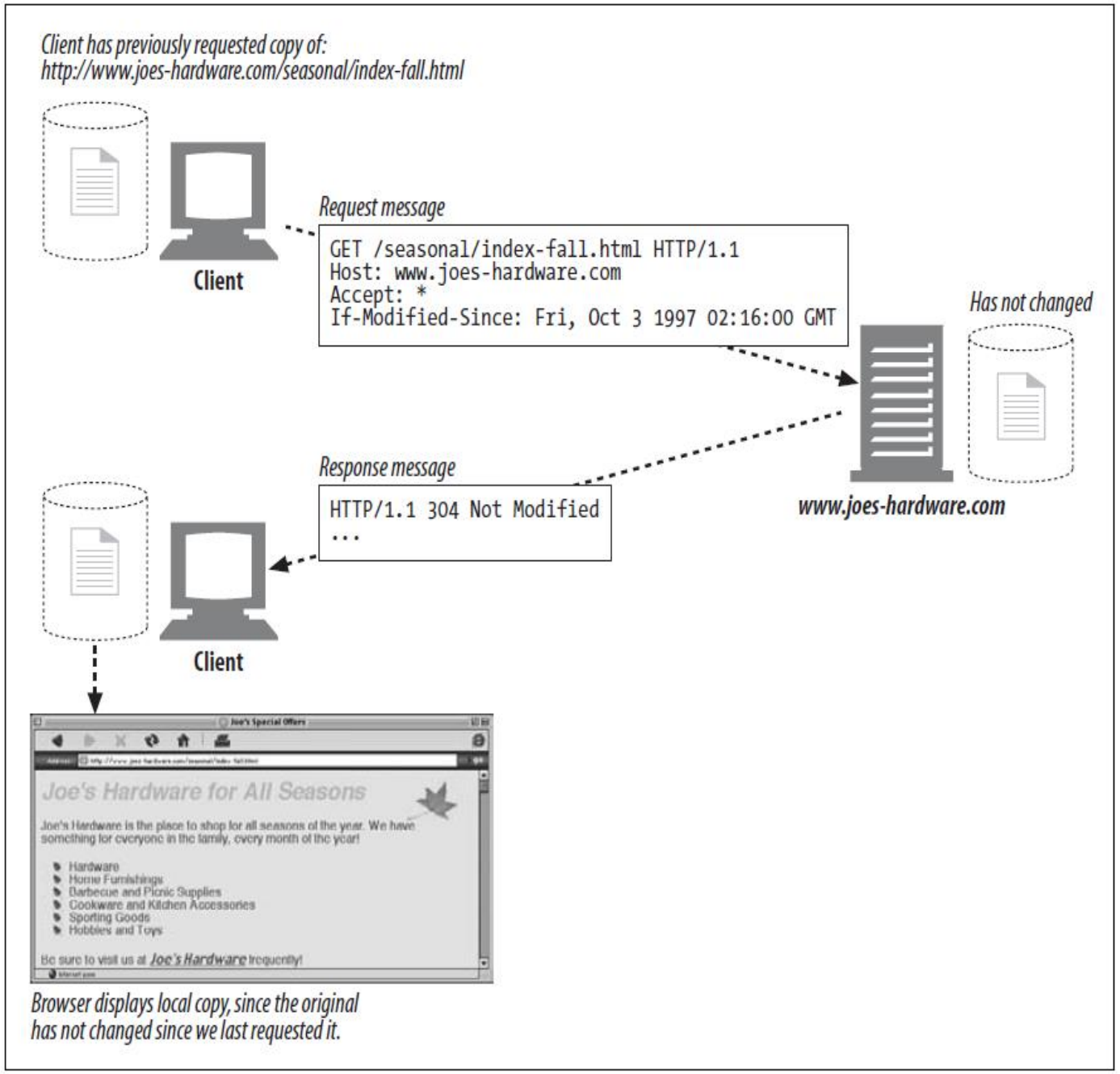


Figure 3-15. Request redirected to use local copy

Status Codes - 4xx

- 4xx (**Client Error**)
  - This class of status code is intended for cases in which the client seems to have erred.
  - Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition.
  - User agents SHOULD display any included entity to the user.
  - Examples:
    - A resource is not available
    - Client submitting a request for a non-existent byte range from a resource
    - To enforce a requirement like the need to submit name and password data.

Authorization & Status Code 403

- The server applies any IP/domain restrictions as its first step in handling a request.
- If the client does not satisfy the restrictions, the server will respond with a ‘forbidden’ message like the following:

<HTML><HEAD>
<TITLE>**403 Forbidden**</TITLE>
</HEAD><BODY>
<H1>**Forbidden**</H1>
You don't have permission to access *resource* on this server.<P>
<HR>
<ADDRESS>Apache/1.3.17 Server at ... 8080</ADDRESS>
</BODY></HTML>

Authorization & Status Code 401

- When a client submits a request for a resource in a password-controlled realm, the server responds with an ‘unauthorized’ message like the following:

  HTTP/1.1 **401 Authorization Required**

  Date: Thu, 24 May 2001 08:09:45 GMT

  Server: Apache/1.3.17 (Unix)

  WWW-Authenticate: Basic realm="Controlled space"

  Transfer-Encoding: chunked

  Content-Type: text/html; charset=iso-8859-1

  …

- The server administrator will have chosen the ‘realm’ name (the name is ‘**Controlled space**’ in the example).

Authorization & Status Code 401

- After receiving the '**401 Authorization Required**' the response from the server, a client browser will:
  - Either display the default web page that is included as the body of the message (not shown on the previous slide)
  - Or handle this challenge properly:
    - The browser should put up a dialog box that identifies the realm and has input fields where the user can enter a username and password.
    - Once these data are entered, they can be sent in an **Authorization header** (next slide) of a follow-up request.
    - The server can then check the name–password combination and return the resource if the password is validated.

Authorization Header

- The **Authorization header** is constructed as follows:
  - Username and password are combined into a string "username:password"
  - The resulting string is then encoded using Base64
  - ‘Authorization’ and a space and "Basic " is then put before the encoded string.
    - For example, if the user agent uses '**Aladdin**' as the username and '**open sesame**' as the password then the header is formed as follows:

Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

Status Codes - 5xx

- 5xx (**Server Error**)
  - This class of status code indicates cases in which the server is aware that it has erred or is incapable of performing the request.
  - Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition.
  - User agents SHOULD display any included entity to the user.

List of HTTP status codes

From Wikipedia, the free encyclopedia

The following is a list of **Hypertext Transfer Protocol (HTTP) response status codes**. This includes codes from IETF internet standards as well as other IETF RFCs, other specifications and some additional commonly used codes. The first digit of the status code specifies one of five classes of response; the bare minimum for an HTTP client is that it recognises these five classes. The phrases used are the standard examples, but any human-readable alternative can be provided. Unless otherwise stated, the status code is part of the HTTP/1.1 standard (RFC 2616).

The Internet Assigned Numbers Authority (IANA) maintains the official registry of HTTP status codes.

Microsoft IIS sometimes uses additional decimal sub-codes to provide more specific information,[1] but these are not listed here.

Contents [hide]

HTTP

Persistence · Compression · HTTPS

Request methods

OPTIONS · GET · HEAD · POST · PUT · DELETE · TRACE · CONNECT · PATCH

Header fields

Cookie · ETag · Location · HTTP referer · DNT · X-Forwarded-For

Status codes

301 Moved permanently · 302 Found · 303 See Other · 403 Forbidden · 404 Not Found

V · T · E

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export

Languages
Česky

Responses

- HTTP response messages have the form:

  Response = Status-Line

  *((general-header | **response-header** | entity-header) **CRLF**)

  CRLF

  [message-body]

- This says that a response is a status line, followed by some number of headers, a blank line and an optional message body.
- The status line specifies the **HTTP protocol version**, **status code** and **phrase**.

Response Headers

- Response messages have their own set of response headers.
- Response headers provide clients with extra information, such as who is sending the response, the capabilities of the responder, or even special instructions regarding the response.
- These headers help the client deal with the response and make better requests in the future.
- Next slide lists common response headers.

6.2 Response Header Fields

The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

```
response-header = Accept-Ranges           ; Section 14.5
                | Age                     ; Section 14.6
                | ETag                    ; Section 14.19
                | Location                ; Section 14.30
                | Proxy-Authenticate      ; Section 14.33
                | Retry-After             ; Section 14.37
                | Server                  ; Section 14.38
                | Vary                    ; Section 14.44
                | WWW-Authenticate        ; Section 14.47
```

Response-header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields MAY be given the semantics of response-header fields if all parties in the communication recognize them to be response-header fields. Unrecognized header fields are treated as entity-header fields.

7 Entity

Request and Response messages MAY transfer an entity if not otherwise restricted by the request method or response status code. An entity consists of entity-header fields and an entity-body, although some responses will only

Response fields [edit]

Field name	Description	Example	Status
Access-Control-Allow-Origin	Specifying which web sites can participate in cross-origin resource sharing	`Access-Control-Allow-Origin: *`	Provisional
Accept-Patch[31]	Specifies which patch document formats this server supports	`Accept-Patch: text/example;charset=utf-8`	Permanent
Accept-Ranges	What partial content range types this server supports via byte serving	`Accept-Ranges: bytes`	Permanent
Age	The age the object has been in a proxy cache in seconds	`Age: 12`	Permanent
Allow	Valid actions for a specified resource. To be used for a *405 Method not allowed*	`Allow: GET, HEAD`	Permanent
Alt-Svc[32]	A server uses "Alt-Svc" header (meaning Alternative Services) to indicate that its resources can also be accessed at a different network location (host or port) or using a different protocol	`Alt-Svc: h2="http2.example.com:443"; ma=7200`	Permanent
Cache-Control	Tells all caching mechanisms from server to client whether they may cache this object. It is measured in seconds	`Cache-Control: max-age=3600`	Permanent
Connection	Control options for the current connection and list of hop-by-hop response fields[8]	`Connection: close`	Permanent
Content-Disposition[33]	An opportunity to raise a "File Download" dialogue box for a known MIME type with binary format or suggest a filename for dynamic content. Quotes are necessary with special characters.	`Content-Disposition: attachment; filename="fname.ext"`	Permanent
Content-Encoding	The type of encoding used on the data. See HTTP compression.	`Content-Encoding: gzip`	Permanent
Content-Language	The natural language or languages of the intended audience for the enclosed content[34]	`Content-Language: da`	Permanent
Content-Length	The length of the response body in octets (8-bit bytes)	`Content-Length: 348`	Permanent
Content-Location	An alternate location for the returned data	`Content-Location: /index.htm`	Permanent
Content-MD5	A Base64-encoded binary MD5 sum of the content of the response	`Content-MD5: Q2hlY2sgSW50ZWdyaXR5IQ==`	Obsolete[9]
Content-Range	Where in a full body message this partial message belongs	`Content-Range: bytes 21010-47021/47022`	Permanent
Content-Type	The MIME type of this content	`Content-Type: text/html; charset=utf-8`	Permanent
Date	The date and time that the message was sent (in "HTTP-date" format as defined by RFC 7231) [35]	`Date: Tue, 15 Nov 1994 08:12:31 GMT`	Permanent

Cookies
(Optional)

Cookies

- Cookies are the best way to **identify users** and allow persistent sessions.
- Cookies were first developed by **Netscape** but now are supported by all major browsers.
- Because cookies are important, and they define new HTTP headers, we're going to explore them in a little more detail.

How Cookies Work

- The first time the user visits a web site, the web server doesn't know anything about the user, but the web server expects that this same user will return again, so it wants to "slap" a unique cookie onto the user so it can identify this user in the future. (See next slide, part a)

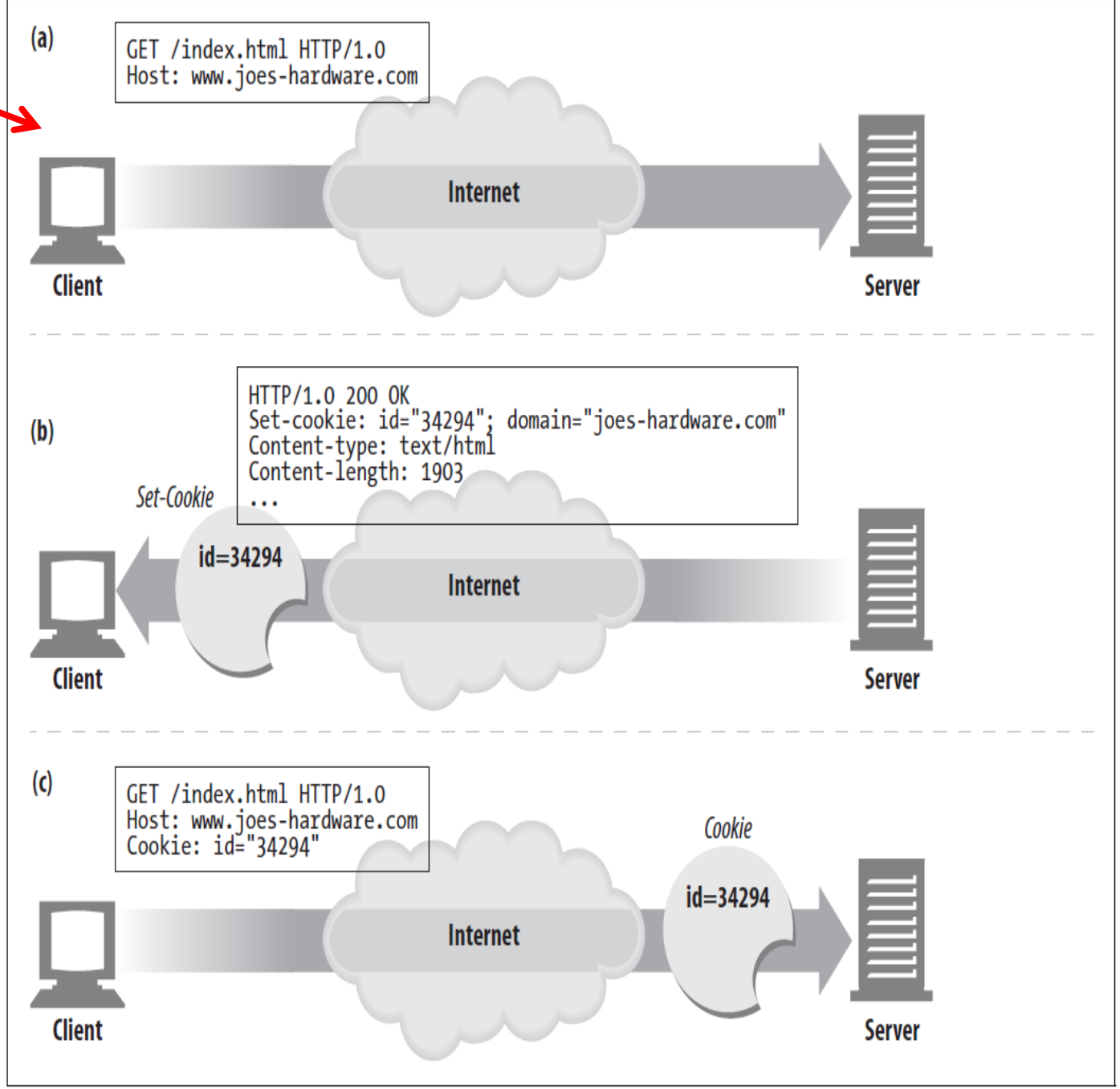


Figure 11-3. Slapping a cookie onto a user

How Cookies Work

- The cookie contains an arbitrary list of ***name=value*** information, and it is attached to the user using the **Set-Cookie** response header. (See Figure 11-3b)
- Cookies can contain any information, but they often contain just a unique **identification number**, generated by the server for tracking purposes.
  - For example, in Figure 11-3b, the server slaps onto the user a cookie that says **id="34294"**.
- The server can use this number to look up database information that the server accumulates for its visitors (purchase history, address information, etc.).

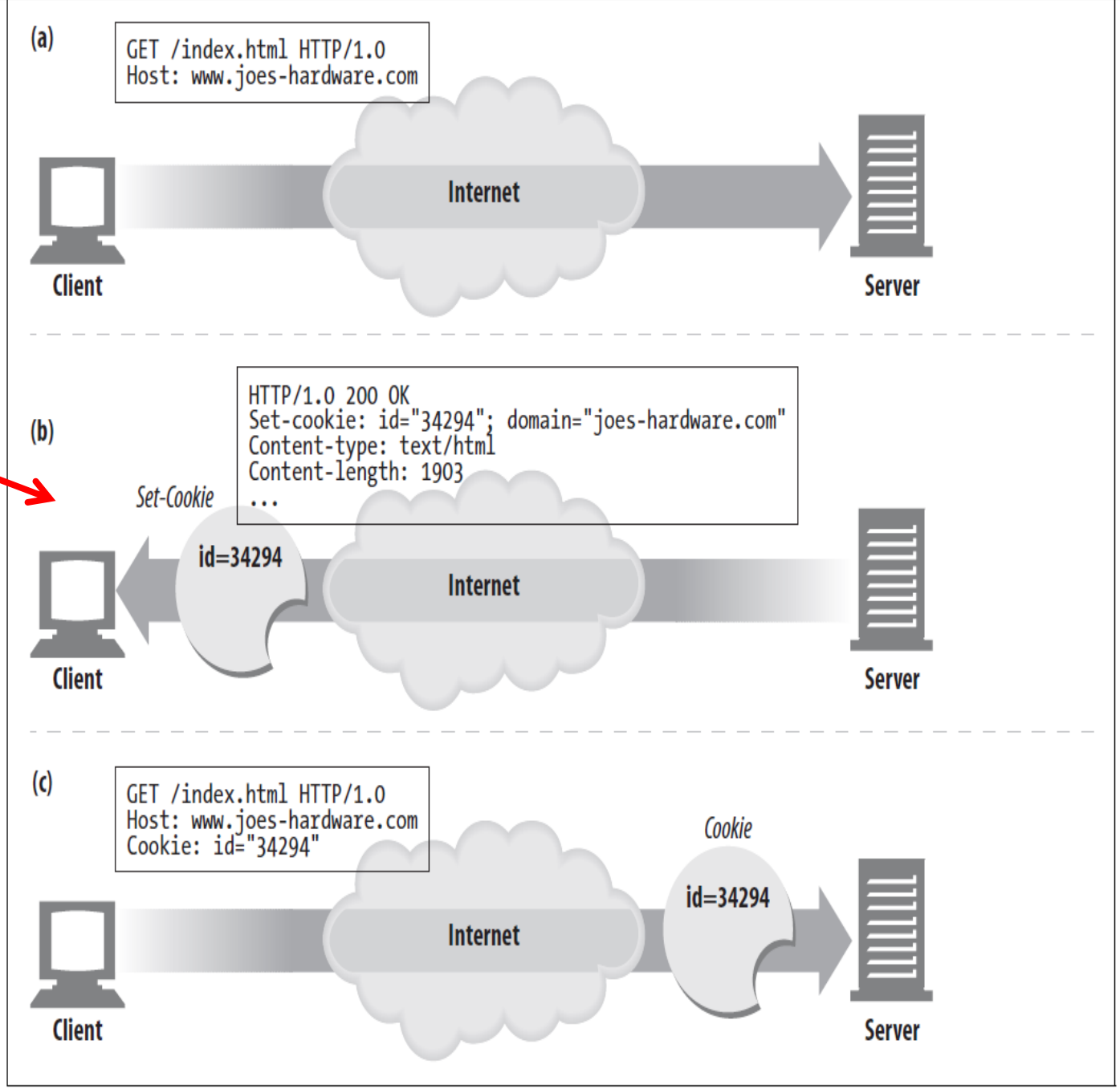


Figure 11-3. Slapping a cookie onto a user

How Cookies Work

- Cookies are not restricted to just ID numbers. Many web servers choose to keep information directly in the cookies.
  - For example:

    Cookie: name="Brian Totty"; phone="555-1212“

- The browser remembers the cookie contents sent back from the server in **Set-Cookie** header, storing the set of cookies in a browser cookie database –
  - Think of it like a suitcase with stickers from various countries on it).
- When the user returns to the same site in the future (Figure 11-3c), the browser will select those cookies slapped onto the user by that server and pass them back in a **Cookie** request header.

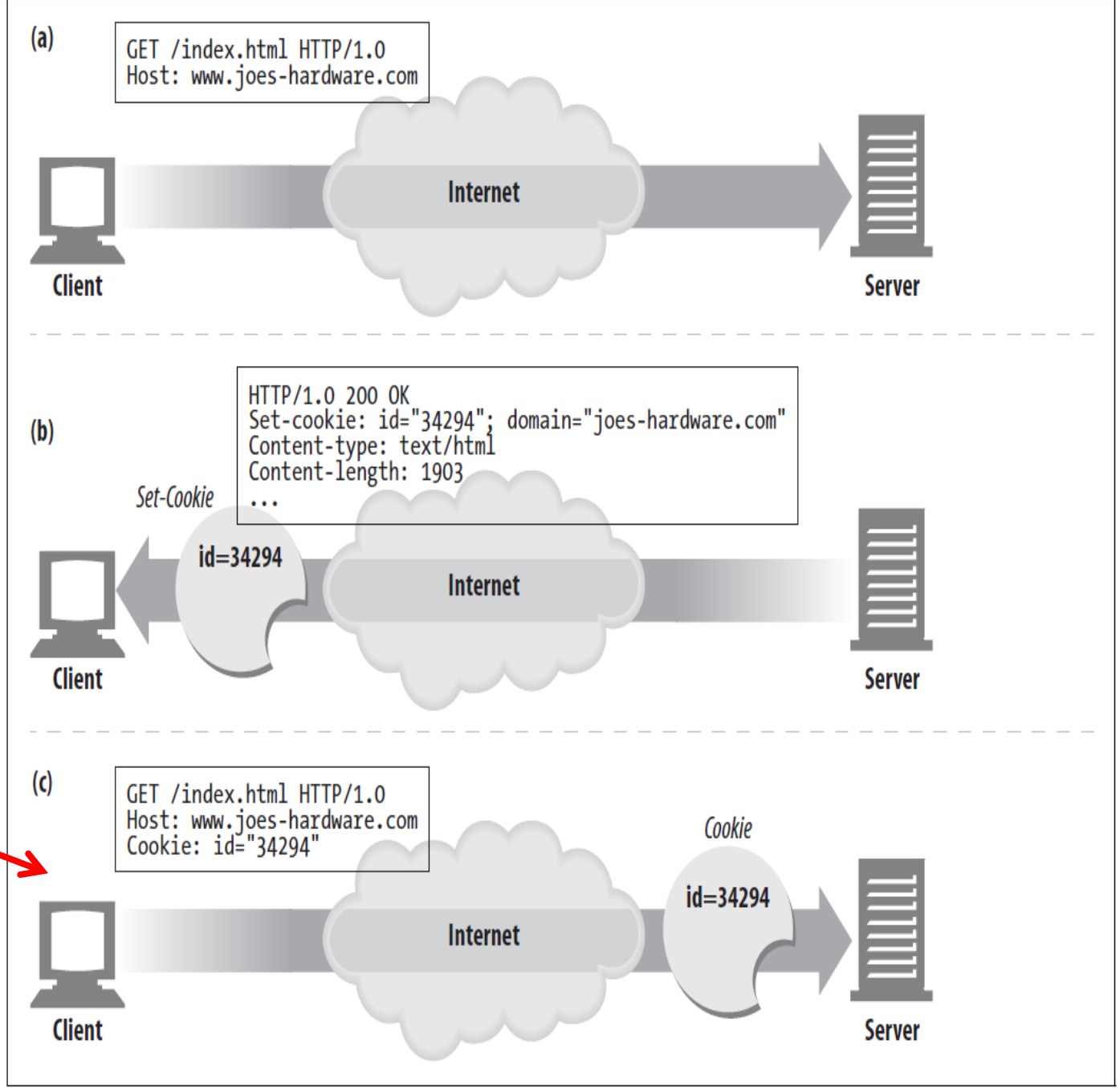


Figure 11-3. Slapping a cookie onto a user

Cookies: A Closer Look

- Many websites use small strings of text as **cookies** to store persistent client-side state between connections.
  - Cookies are passed from server to client and back again in the HTTP headers of requests and responses.
  - Cookies can be used by a server to indicate session IDs, shopping cart contents, login credentials, user preferences, etc.:

    ISBN=0802099912&price=$34.95

    - A cookie set by an online bookstore to specify a book that I've put in my shopping cart.
  - However, the value might be a meaningless string such as ATVPDKIKX0DER, which identifies a particular record in a database of some kind where the real information is kept.
    - Usually the cookie values do not contain the data but merely point to it on the server.

Cookies: A Closer Look

- Cookies are limited to non-whitespace ASCII text, and may not contain commas or semicolons.
  - To set a cookie in a browser, the server includes a **Set-Cookie** header line in the HTTP header.
    - The following HTTP header sets the cookie "cart" to the value "ATVPDKIKX0DER":

      HTTP/1.1 200 OK

      Content-type: text/html

      Set-Cookie: cart=ATVPDKIKX0DER

  - If a browser makes a second request to the same server, it will send the cookie back in a Cookie line in the HTTP request header like so:

    GET /index.html HTTP/1.1

    Host: www.example.org

    Cookie: cart=ATVPDKIKX0DER

    Accept: text/html

Cookies: A Closer Look

- As long as the server doesn't reuse cookies, this enables it to track individual users and sessions across multiple, otherwise stateless, HTTP connections.
- Servers can set more than one cookie.
  - Amazon may feed us five cookies for a request we make:

```
Set-Cookie:skin=noskin
Set-Cookie:ubid-main=176-5578236-9590213
Set-Cookie:session-token=Zg6afPNqbaMv2WmYFOv57zCU1O6Ktr
Set-Cookie:session-id-time=2082787201l
Set-Cookie:session-id=187-4969589-3049309
```

Cookies: A Closer Look

- In addition to a simple **name=value** pair, cookies can have several attributes that control their scope including expiration date, path, domain, port, version, and security options.
  - By default, a cookie applies to the server it came from.
    - If a cookie is originally set by *www.foo.example.com*, the browser will only send the cookie back to *www.foo.example.com*.
    - However, a site can also indicate that a cookie applies within an entire subdomain, not just at the original server.
  - The following request sets a user cookie for the entire *foo.example.com* domain:

    Set-Cookie: user=elharo;Domain**=.foo.example.com**

Cookies: A Closer Look

- Cookies are also scoped by path, so they are returned for some directories on the server, but not all.
  - The default scope is the original URL and any subdirectories.
    - If a cookie is set for the URL:

      http://www.cafeconleche.org/XOM/
    - The cookie also applies in:

      http://www.cafeconleche.org/XOM/apidocs/,
    - but not in:

      http://www.cafeconleche.org/slides/ or
      http://www.cafeconleche.org/

Cookies: A Closer Look

- However, the default scope can be changed using a Path attribute in the cookie.
  - The following response sends the browser a cookie with the name “user” and the value “elharo” that applies only within the server’s */restricted* subtree, not on the rest of the site:

    Set-Cookie: user=elharo; Path=/restricted

- When requesting a document in the subtree */restricted* from the same server, the client echoes that cookie back.
  - It does not use the cookie in other directories on the site.

Cookies: A Closer Look

- A cookie can include both a domain and a path.
  - The following cookie applies in the */restricted* path on any servers within the *example.com* domain:

Set-Cookie: user=elharo;Path=/restricted;Domain=.example.com

- The order of the different cookie attributes doesn't matter, as long as they're all separated by semicolons and the cookie's own name and value come first.
  - Unless the client is sending the cookie back to the server.
    - In this case, the path must precede the domain:

Cookie: user=elharo; Path=/restricted;Domain=.foo.example.com

Cookies: A Closer Look

- A cookie can be set to expire at a certain point in time by setting the expires attribute to a date in the form:

  Wdy, DD-Mon-YYYY HH:MM:SS GMT

- Weekday and month are given as three-letter abbreviations. The rest are numeric, padded with initial zeros if necessary.

- In the pattern language used by **java.text.SimpleDateFormat**, this is:

  E, dd-MMM-yyyy H:m:s z

  Set-Cookie: user=elharo; **expires**=Wed, 21-Dec-2015 15:23:00 GMT

- The browser should remove this cookie from its cache after that date has passed.

Cookies: Max-Age

- The **Max-Age** attribute that sets the cookie to expire after a certain number of seconds have passed instead of at a specific moment.

- The following cookie expires one hour (3,600 seconds) after it's first set:

  Set-Cookie: user="elharo"; **Max-Age**=3600

- The browser should delete this cookie after this amount of time has elapsed.

Cookies: secure & httponly

- Because cookies can contain sensitive information such as passwords and session keys, some cookie transactions should be secure.
  - Most of the time this means using HTTPS instead of HTTP
  - But cookie can have a **secure** attribute with no value:

    Set-Cookie: key=etrogl7*;Domain=.foo.example.com; **secure**

  - Browsers are supposed to refuse to send such cookies over insecure channels.
- For additional security against cookie-stealing attacks, cookies can set the **HttpOnly** attribute.
  - This tells the browser to only return the cookie via HTTP and HTTPS and specifically *not* by JavaScript:

Set-Cookie: key=etrogl7*;Domain=.foo.example.com; secure; **httponly**

Cookies by Amazon

Set-Cookie: skin=noskin; path=/; domain=.amazon.com; ***expires***=Fri, 03-May-2013 21:46:43 GMT

Set-Cookie: ubid-main=176-5578236-9590213; path=/; domain=.amazon.com; ***expires***=Tue, 01-Jan-2036 08:00:01 GMT

Set-Cookie: session-token=Zg6afPNqbaMv2WmYFOv57zCU1O6KtrMMdskcmllbZ

cY4q6t0PrMywqO82PR6AgtfIJhtBABhomNUW2dITwuLfOZuhXILp7Toya+

AvWaYJxpfY1lj4ci4cnJxiuUZTev1WV31p5bcwzRM1Cmn3QOCezNNqenhzZD8TZ UnOL/9Ya; path=/; domain=.amazon.com; ***expires***=Thu, 28-Apr-2033 21:46:43 GMT

Set-Cookie: session-id-time=2082787201l; path=/; domain=.amazon.com;

expires=Tue, 01-Jan-2036 08:00:01 GMT

Set-Cookie: session-id=187-4969589-3049309; path=/; domain=.amazon.com;

expires=Tue, 01-Jan-2036 08:00:01 GMT

- Of course, browsers are free to ignore all these requests, and users can delete or block cookies at any time.

Cookies in Java (Optional)

- Java, since Java 6, has a concrete **java.net.CookieManager** class, a subclass of **CookieHandler**, that you can use. However, it is not turned on by default.
- Before Java will store and return cookies, you need to enable it:

```
CookieManager manager = new CookieManager();
CookieHandler.setDefault(manager);
```

- If all you want is to receive cookies from sites and send them back to those sites, you're done.
- After installing a CookieManager with those two lines of code, Java will store any cookies sent by HTTP servers you connect to with the **URL** class, and will send the stored cookies back to those same servers in subsequent requests.

Cookies in Java

- However, you may wish to be a bit more careful about whose cookies you accept. You can do this by specifying a CookiePolicy:
  1. **CookiePolicy.ACCEPT_ALL** (All cookies allowed)
  2. **CookiePolicy.ACCEPT_NONE** (No cookies allowed)
  3. **CookiePolicy.ACCEPT_ORIGINAL_SERVER** (Only first party cookies allowed)

- The following code fragment tells Java to block third-party cookies but accept first party cookies:

```
CookieManager manager = new CookieManager();
manager.setCookiePolicy(CookiePolicy.ACCEPT_ORIGINAL_SERVER);
CookieHandler.setDefault(manager);
```

  - It will only accept cookies for the server that you're talking to, not for any server on the Internet.

Cookies in Java

- If you want more fine-grained control, for instance to allow cookies from some known domains but not others, you can implement the **CookiePolicy** interface yourself and override the **shouldAccept()** method:

```
public boolean shouldAccept(URI uri, HttpCookie cookie)
```

- The following example (Example 6-1 in Textbook #2) shows a simple **CookiePolicy** that blocks cookies from *.gov* domains, but allows others.

Cookies in Java

```
import java.net.*;
public class NoGovernmentCookies implements CookiePolicy {
        @Override
        public boolean shouldAccept(URI uri, HttpCookie cookie) {
                if (uri.getAuthority().toLowerCase().endsWith(".gov") ||
                  cookie.getDomain().toLowerCase().endsWith(".gov")) {
                        return false;
                }
                return true;
        }
}
```

Cookie Store

- It is sometimes necessary to put and get cookies locally.
  - Example: When an application quits, it can save the cookie store to disk and load those cookies again when it next starts up.

- You can retrieve the store in which the CookieManager saves its cookies with the **getCookieStore()** method:

```
CookieStore store = manager.getCookieStore();
```

Cookie Store

- The **CookieStore** class allows you to add, remove, and list cookies so you can control the cookies that are sent outside the normal flow of HTTP requests and responses:

```
public void add(URI uri, HttpCookie cookie)
public List<HttpCookie> get(URI uri)
public List<HttpCookie> getCookies()
public List<URI> getURIs()
public boolean remove(URI uri, HttpCookie cookie)
public boolean removeAll()
```

HttpCookie

- Each cookie in the store is encapsulated in an **HttpCookie** object, which provides methods for inspecting the attributes of the cookie:

```
public class HttpCookie implements Cloneable {
    public HttpCookie(String name, String value)
    public boolean hasExpired()
    public void setDiscard(boolean discard)
    public boolean getDiscard()
    public void setPortlist(String ports)
    public String getPortlist()
    public void setDomain(String domain)
    public String getDomain()
    public void setMaxAge(long expiry)
    public long getMaxAge()
    …
```

- For the complete list, see:

https://docs.oracle.com/javase/8/docs/api/java/net/HttpCookie.html