UNRAVELING THE MYSTERIES OF J2EE WEB APPLICATION COMMUNICATIONS —AN HTTP PRIMER

Peter Koletzke, Quovera

It is true that I may not find
an opportunity of transmitting it to the world,
but I will not fail to make the endeavor.
At the last moment I will enclose the MS. in a bottle
and cast it within the sea.

—Edgar Allan Poe (1809–1849),
MS. Found in a Bottle

In computer systems, *communications protocols* are guidelines for formatting the messages sent from one hardware or software component to another, for example, from a client's web browser to an application server. Since effective communications cannot occur without an agreed upon format, the communications protocol is key to successful communication in any interaction involving computer systems.

The most important communications protocol for all web applications, including those built using Oracle products, is *Hypertext Transfer Protocol* (HTTP). In fact, web application servers are HTTP servers, so HTTP is the heart of applications running both on the World Wide Web and also on organization-level application servers. The mechanics, features, and constraints of HTTP communications are usually a bit of a mystery to those accustomed to working with client/server and terminal mode applications. Even those who have worked in a web environment and have a concept of the HTTP communications path may not have studied HTTP in detail.

Knowing the components, features, capabilities, and communication paths that apply to HTTP is essential knowledge for developers and administrators who write and support web applications. This knowledge will prove useful when developing programs that need to read or write from global session or process areas. It will also help when interpreting error or status messages from the server and when debugging application code. This white paper demystifies HTTP by discussing HTTP basics including the request and response messages and methods. It then steps through a sample communications exchange, and explains some terms you will see when writing, debugging, and running web application code.

Note

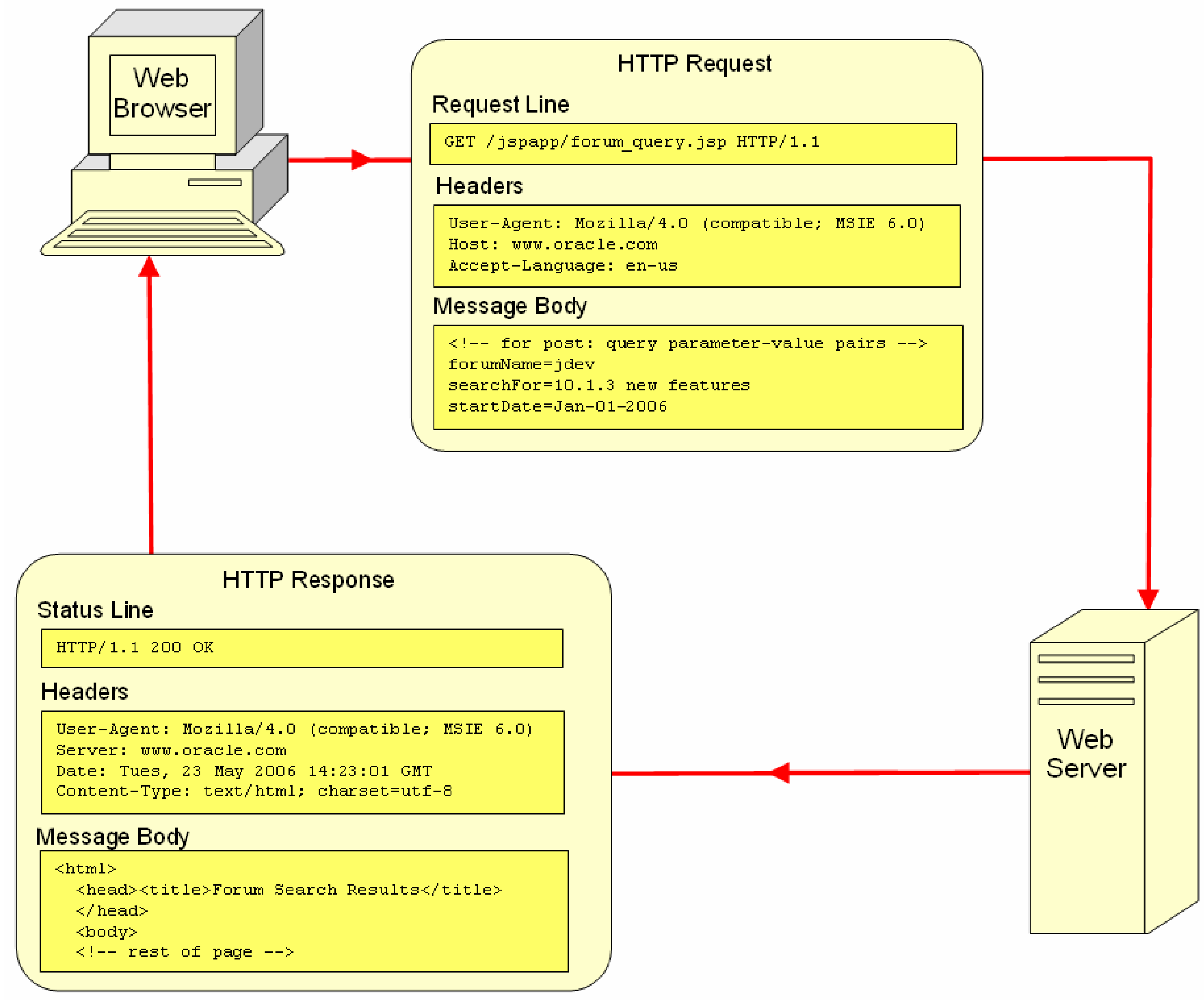
A web browser is capable of issuing requests in other protocols such as File Transfer Protocol (FTP), Lightweight Directory Access Protocol (LDAP), mailto, and Hypertext Transfer Protocol Secure (HTTPS). Web servers can only handle http and sometimes https so web applications are written for those protocols.

HTTP Basics

Web applications use HTTP for communications between the client's web browser and the application code running on an *application server*—a networked machine running application code as well as communication services code (also called a *web server* in this paper). As with all communications protocols, a roundtrip communication process consists of a request and a response message. The *request* is a message asking for resources (such as an HTML page or image file) or an action from another machine. A *response* is a return message from the machine to which a request was sent. These messages often include browser content such as HTML text or images. Figure 1 shows these two messages with some of their contents. Descriptions of the contents of the request and response messages, as well as methods and other features follow.

Note

HTTP uses TCP/IP (Transmission Control Protocol/Internet Protocol). TCP/IP is a lower-level protocol that defines how the hardware communicates. When you develop web applications, you interface with HTTP, not directly with TCP/IP.



```
GET /jspapp/forum_query.jsp HTTP/1.1
```

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0)
Host: www.oracle.com
Accept-Language: en-us
```

```
<!-- for post: query parameter-value pairs -->
forumName=jdev
searchFor=10.1.3 new features
startDate=Jan-01-2006
```

```
HTTP/1.1 200 OK
```

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0)
Server: www.oracle.com
Date: Tues, 23 May 2006 14:23:01 GMT
Content-Type: text/html; charset=utf-8
```

```
<html>
  <head><title>Forum Search Results</title>
  </head>
  <body>
  <!-- rest of page -->
```

Figure 1: An HTTP communication session

HTTP Request

The browser sends a request message when the user clicks a link or button on the page or enters an address in the browser's address (location) field. As shown in Figure 1, the request consists of a request line, headers, and message body.

Note

A detailed explanation of HTTP appears on the TCP/IP Guide website, www.tcpipguide.com.

Request Line

A sample request line follows:

```
GET /app/jobhist.jsp HTTP/1.1
```

This example consists of the following values:

- **Method** ”GET” signifies the method, a command to the server for a specific operation (described later).
- **URI** ”/app/jobhist.jsp” indicates the *Uniform Resource Identifier (URI),* which uniquely identifies a resource on the Web, in this case a JavaServer Pages (JSP) file. URIs contain components that uniquely identify a file available on the Web. A *Uniform Resource Locator* (URL), a subset of the URI standard, is used to find files using HTTP. URLs are described further in the sidebar “About the Uniform Resource Locator (URL).”
- **HTTP version** “HTTP/1.1” refers to HTTP version 1.1. The version of HTTP used is meaningful to both client and server because different versions offer different features. For example, HTTP version 1.1—the most recent and most popular version—allows multiple requests to be served in the same connection session.

About the Uniform Resource Locator (URL)

As mentioned, the Web uses the URL format, a subset of the URI format, to uniquely locate a web resource (such as a file) in HTTP communications. URLs contain the protocol (HTTP), host name and port of the web server, the path (directory structure) in which the resource may be found and the resource name, *query parameters* (name and value pairs used by the server application), and an optional bookmark name (called a *named anchor* in HTML) that scrolls the browser to a specific location on the page. Here is an example using the Oracle website’s domain and some fictitious details. (The following two lines represent a single URL line.)

```
http://www.oracle.com:8080/jspapp/forum_query.jsp?forum_name=jdev&
   searchFor=10.1.3%20new%20features&startDate=Jan-01-2006
```

This URL identifies the following components:

- **Host** (www.oracle.com). The host name uses dot separators between hierarchical components
- **Web server listener** (port 8080). No port identifier declares that the default port, usually 80, will be used.
- **Context root** (/jspapp), also called the *application directory* (or *virtual directory*), the top level directory for a web application. This part of the URL may be mapped to a physical directory on the application server. It alternatively may be translated to a servlet or other service through an entry in the web deployment descriptor, web.xml (described later in the sidebar “About server.xml and web.xml”).
- **File name** (forum_query.jsp). This file is processed by the Web Tier container (the J2EE architectural tier which runs code that sends HTML to the browser). If this had been a static HTML file, you could add an anchor name (such as “#response3”) to cause the browser to scroll to the point on the page where the anchor (“<a>” tag) with that name is coded.
- **Query parameters and values** for the JSP file (forum_name with a value of “jdev,” searchFor with a value of “10.1.3 new features” (“%20” is translated to a space character), and startDate with a value of “Jan-01-2006).

Headers

Headers identify the requestor and indicate how the content will be obtained. Headers consist of a series of header fields. Each header field consists of the name of the header entry followed by either a value or a directive, for example:

```
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Host: otn.oracle.com
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
```

These headers hold the following information:

- **Accept** This header field indicates media types the client will receive. Media types are defined by another standard, *Multipurpose Internet Mail Extensions* (MIME). For example, image/jpeg and plain/text are MIME types that a browser can accept. (Refer to the sidebar “About MIME Types” for more information.) The “*/*” in the Accept example before means that all types of content will be accepted.

- **Accept-Language** This header field defines the language that will be accepted in the request (in this example, U.S. English).
- **Accept-Encoding** This line indicates the compression algorithms that are acceptable. A server uses *compression* methods to reduce the response message size. *Deflate* and *gzip* are two common compression algorithms. The order indicates that gzip is preferred over deflate.
- **Connection** A *persistent HTTP connection* allows a TCP protocol connection established by a browser to the web server to be reused for additional request/response roundtrips. For HTTP 1.0, the Connection field value “Keep-Alive” specifies a persistent connection. Persistent connections are a default in HTTP 1.1; in both versions, they save the time and resources required to open a new connection for each request. They allow the server to send multiple responses (for example, image files displayed in an HTML page) within the same connection session. The session is closed when the request has been completely fulfilled; that is, when all files requested as part of the request have been sent as responses.
- **Host** This header defines the machine to which the request is sent. The host header is the only required header line for a request.
- **User-Agent** The user-agent header field declares the software used by the web browser including the name and version. The server can use this for statistics and for modifying the content to take advantage of a featured offered by a specific type of browser.

About MIME Types

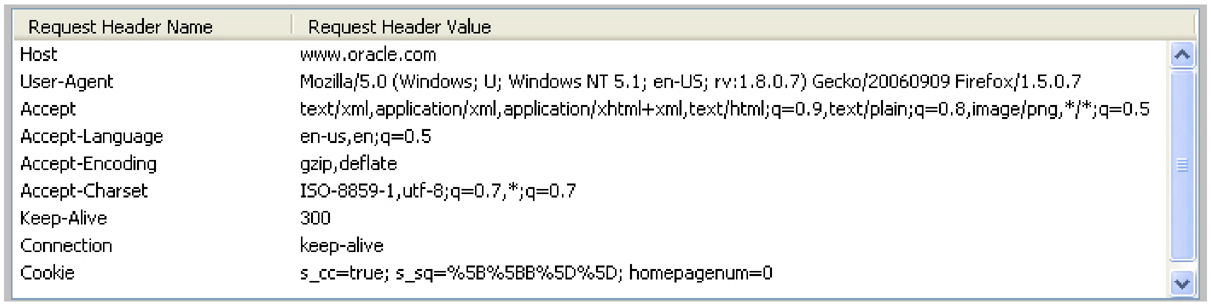
As mentioned, *Multipurpose Internet Mail Extensions* (MIME) declares the format (encoding scheme) of a file. The MIME identifier consists of a type and subtype (with optional parameters). Here are some examples:

```
image/gif
image/jpg
text/html
text/plain
application/msword
application/pdf
```

These identify types of image, text, and application, each of which has two subtypes. Both client and server can process the content depending upon its type. For example, if the type is “text,” the browser renders it using HTML processing (text/html) or as unformatted text (text/plain). If the subtype is “image,” the browser displays the image file using the appropriate image renderer for the page (GIF and JPG are the two most common image formats). If the type is “application,” the browser opens up a helper program to assist in the display of the file (in these examples, MS Word for “msword” or Adobe Reader for “pdf”).

ANOTHER SAMPLE REQUEST HEADER

The following display shows the request header for a call to the URL www.oracle.com as displayed in the Firefox Tamper Data utility (http://addons.mozilla.org/firefox/966/):



Request Header Name	Request Header Value
Host	www.oracle.com
User-Agent	Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) Gecko/20060909 Firefox/1.5.0.7
Accept	text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language	en-us,en;q=0.5
Accept-Encoding	gzip,deflate
Accept-Charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive	300
Connection	keep-alive
Cookie	s_cc=true; s_sq=%5B%5BB%5D%5D; homepagenum=0

Message Body

The request can also send a message body to the server. This is typically used for a POST request (described later) to supply parameter values to the server application. It is not usually used with a GET method because the information in the URL provides parameter values to the server application.

HTTP Response

When the web server receives the HTTP request, it gathers content and sends it back to the browser as an HTTP response. As shown in Figure 1, the response is made of the status line, headers, and content.

Status Line

The status line contains codes that indicate success or failure of the request. The following shows an example:

```
HTTP/1.1 200 OK
```

This status line contains the following parts:

- **HTTP version** "HTTP/1.1" indicates the HTTP version. This helps the browser interpret the content in the request.
- **Status code** "200" is the status code for success in handling the request. Codes in the 100s indicate that the server is still processing the request; 200s indicate that the request was processed; 300s indicate a redirection problem; 400s indicate an error with the client request (such as authorization failure); and 500s indicate a server error.
- **Reason phrase** "OK" is a message unique to the status code. It repeats the information of the code in a friendlier format.

Note

You can find a listing of all status codes at www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

Headers

The response headers are formatted in the same way as the request headers. A sample header follows.

```
Cache-Control: no-cache
Content-Length: 2748
Content-Type: image/gif
Date: Tues, 20 Dec 2005 12:00:00 GMT
Expires: -1
Server: Microsoft-IIS/5.1
set-cookie: <cookie information>
```

These headers provide the following information:

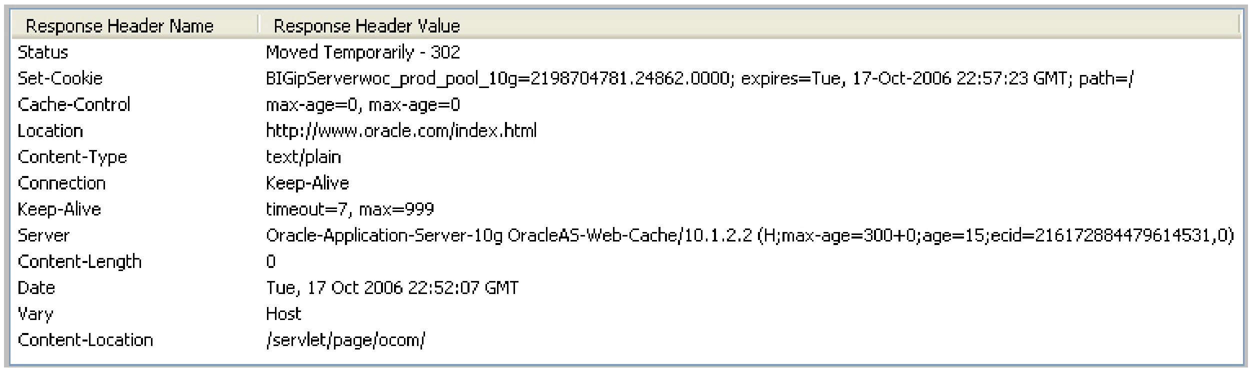
- **Cache-Control** This header field signifies whether the browser or other means can *cache* the content; that is, the client stores the content so that a subsequent request can be filled from the cache and not from the server response; the "no-cache" directive disables caching. Caching is also affected by the headers Expires and Last-Modified.
- **Content-Length** This header field designates the length in bytes of the message body sent after the headers.
- **Content-Type** This header field defines the MIME format of the content. In this example, the file is an image file.
- **Date** This header field indicates the date and time on the web server.
- **Expires** This header field defines the date and time when the content should be considered out of date. If the cache is enabled for the content, the browser can obtain the content from the cache before the date indicated in this header. "-1" means that the content expires immediately.
- **Server** This header is the type and version of the web server.
- **set-cookie** This header writes a cookie to the client machine so that the server can determine later what requests client previously sent. A later section, "State Management and Cookies," explains cookies in more detail.

ANOTHER SAMPLE RESPONSE HEADER

The display in Figure 2 shows the response header for a response from the www.oracle.com server (also displayed using the Firefox Tamper Header utility):

Message Body

After the headers, the HTTP response includes the actual content (message body) requested by the client. This is usually the file (also called an *entity*) requested by the client.



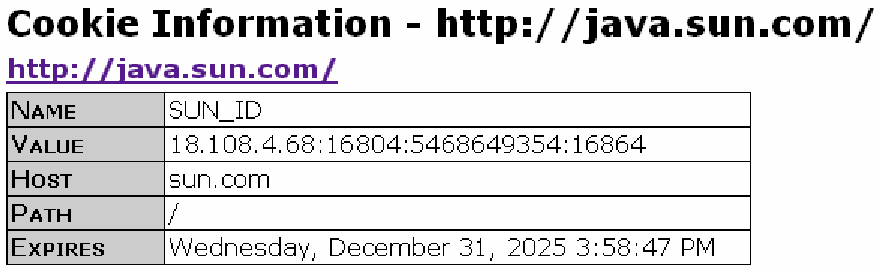
Response Header Name	Response Header Value
Status	Moved Temporarily - 302
Set-Cookie	BIGipServerwoc_prod_pool_10g=2198704781.24862.0000; expires=Tue, 17-Oct-2006 22:57:23 GMT; path=/
Cache-Control	max-age=0, max-age=0
Location	http://www.oracle.com/index.html
Content-Type	text/plain
Connection	Keep-Alive
Keep-Alive	timeout=7, max=999
Server	Oracle-Application-Server-10g OracleAS-Web-Cache/10.1.2.2 (H;max-age=300+0;age=15;ecid=216172884479614531,0)
Content-Length	0
Date	Tue, 17 Oct 2006 22:52:07 GMT
Vary	Host
Content-Location	/servlet/page/ocom/

Figure 2. Response header

State Management and Cookies

HTTP defines a *stateless connection* consisting of a standalone request and response. A stateless connection is one where the server does not know that a particular request was issued by a client that issued a previous request. This interaction works for a situation that only requires requesting and returning a file, but it is unsuited for e-commerce and database transaction situations where the server needs to tie a number of requests together.

HTTP was not originally designed to manage state for the request-response communication. However, a feature called *cookies* allows the server to store information about the client session in memory and, optionally (for *persistent cookies*), in a file on the client's machine (in a directory maintained by the browser, for example, C:\Documents and Settings\<username>\Local Settings\Temporary Internet Files). Then, when another request is issued to the same server, this information will be sent in the request header. The cookie contains the host name and path, a cookie name, value, and expiration date as shown next in the cookie information after a request from the Sun Microsystem’s website (displayed using the View Cookie Information add-on to Firefox available at http://addons.mozilla.org/firefox/60):



Cookie Information - http://java.sun.com/

http://java.sun.com/

Name	SUN_ID
Value	18.108.4.68:16804:5468649354:16864
Host	sun.com
Path	/
Expires	Wednesday, December 31, 2025 3:58:47 PM

As an example, you could cause the application to store write a cookie that stores the last page visited. Then, when the same user issues a subsequent request, the application code can process the request based on the page the user last viewed.

Another example of cookie usage is the session ID. The session ID ties one request from a particular browser to a subsequent request. This is useful for many database transactions whose life cycle spans multiple HTTP requests. For example, if the user issued one request to update a record, another to insert a record, and yet another to commit the update and insert. The application can write a cookie (in a persistent file or in client memory) that stores the session ID retrieved from the application server, and this information can be sent with each request so the application server can tie the requests to a particular database session.

However, users can disable cookies and break the mechanism that stores the session ID. If this could be a problem in an application, the application can use a technique called *URL rewriting*, to circumvent this problem. URL rewriting consists of writing the session ID directly into the HTTP response message body. Then the browser can send this session ID back to the application server in subsequent requests. URL rewriting is a service of most modern controller frameworks such as Struts or the JSF controller. If your application does not use one of these frameworks, you will need to implement URL rewriting if you need to maintain the session ID.

URL rewriting is different from the HTTP persistent connection concept mentioned before; persistent connections describe that multiple files can be sent in one request-response communication session. Preserving state using cookies is used to preserve state between connection sessions.

The J2EE environment has added an API called the *HTTP Session* that allows developers to maintain state. HTTP Session consists of a Java class containing methods that can be used to create and read information from the application user session.

Methods

An HTTP method specified in the HTTP request commands the server to perform a particular task. The most frequently-used methods are GET, POST, and HEAD although several other methods are available.

GET

The *GET method* retrieves content from the server based on the URL. It can only supply parameters coded into the URL (the query parameters mentioned before); the URL is usually specified using a link or entered using the URL field of the browser. The URL can also be constructed from an HTML form submission. Large amounts of data cannot be passed to the server this way because the *URL size limitation* is 2,083 characters for most browsers.

Another limitation of the GET method is that the query parameters are visible to the user in the URL (address or location) field of the browser. Users could potentially figure out the calling syntax for a server action and make an unintended request by sending different parameter values.

Get is used for requests that can be repeated safely without side effects—usually, just retrieving a file. The request could be resubmitted without causing a change of data. For example, when ordering books online, viewing your shopping cart has no effect on your order. If you refresh the page and another request is sent to the server to display the shopping cart, your order will not change.

POST

POST sends information to the server. The parameters are coded into the request's message body as depicted in Figure 1; this hides the calling mechanism from the user and is a bit more secure. A POST request is most often sent from a button or image click after filling out fields in an HTML form (described later).

POST is best for requests that cannot be repeated safely. For example, if you were ordering one copy of the *Oracle JDeveloper 10*g *Handbook*, you would navigate to the description page for that book, fill in the quantity of "1" and click the submit button. This adds one book to your order. If that page were to use a GET request, and that page were refreshed, the quantity of books would increase. This is an undesirable side effect.

HEAD

The HEAD method works the same way as the GET method but it requests the server to not send the message body, only the headers. This allows the client to determine the existence or size of a resource before requesting that the server send the resource.

Other Methods

The following HTTP methods are less often used:

- **Options** This method requests the server to send information about a resource or the server.
- **Put** The put method is used when you want to allow the user to copy a file to the web server.
- **Delete** This method removes an object such as a file from the server. Obviously, this command must be carefully used, but is handy if you allow users to maintain files on the website.
- **Trace** This method requests the server to send the entire request back to the browser. Normally, the server processes the request but does not allow the browser to see it. Trace is useful for debugging a problematic request.

Other HTTP Features

You may run into several other features of HTTP as you work with web applications:

- **Redirection** Servers can send a request to another location. This will return a status code in the 300s to the client. One use of redirection is to instruct the client to show a page from its cache.
- **HTTPS** HTTP Secure (HTTPS) is used in an encrypted Secure Sockets Layer (SSL) session. The request and response messages are encrypted and considered secure because only a key shared between client and server will allow the messages to be read. It works the same way as HTTP otherwise.

The Steps in a Web Application Roundtrip

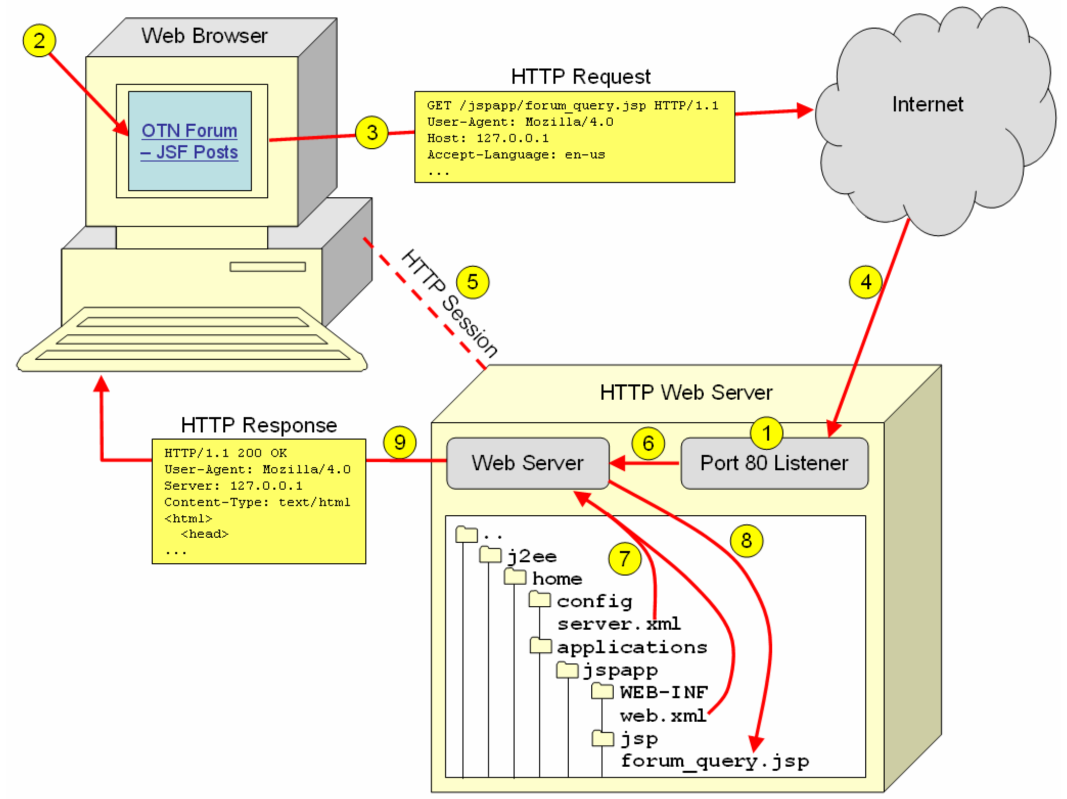
Using the concepts from the preceding overview of how HTTP works, we can now examine the steps an application goes through in a request-response communication roundtrip as shown in Figure 2. This discussion assumes that the host machine domain has been assigned a domain name on the client machine or on a server as described in the sidebar "About Domain Name System and Domains." It also assumes the HTTP request is sent through the Internet instead of being handled by the client or local network.

About Domain Name System and Domains

HTTP supports locating a web server by using an IP address (such as 141.146.8.66). Numbers, such as those that comprise an IP address, are not very user-friendly. In addition, they are subject to change based on hardware architecture. Therefore, an important part of a typical roundtrip is resolution of the domain name.

The client machine may contain a list of domains and IP addresses; on a Windows machine, it is usually in the C:\Windows\drivers\system32\drivers\etc\hosts file. If this file does not contain the IP address, the address must be supplied by a *Domain Name System (DNS) server*—a network server that translates a domain name (such as www.oracle.com) to the Internet Protocol (IP) address that represents an actual host machine. This DNS (sometimes expanded to *Domain Name Service*) server could be located on the local network or on the Web. The name resolution process is not part of HTTP although it plays an integral part in the HTTP roundtrip.

A domain name on the Web is assigned a unique IP address by a "domain registration service," which has access to the means to copy a domain/IP pair to DNS servers on the Internet.



```
GET /jspapp/forum_query.jsp HTTP/1.1
User-Agent: Mozilla/4.0
Host: 127.0.0.1
Accept-Language: en-us
...
```

```
HTTP/1.1 200 OK
User-Agent: Mozilla/4.0
Server: 127.0.0.1
Content-Type: text/html
<html>
  <head>
...
```

Figure 2: HTTP request-response roundtrip

1. A process on the web server—called the *HTTP daemon (HTTPD)*, listens for a request from the network on a specific port (by default, port 80).
2. The user clicks a link on an HTML page that contains the following reference:
   ```
   http://www.oracle.com/jspapp/forum_query.jsp?forum_name=jdev&
      searchFor=10.1.3%20new%20features&startDate=Jan-01-2006
   ```
3. The browser assembles an HTTP request and sends it to the network (the Internet in this example).

4. A DNS server on the network translates the domain name to an IP address and sends the message to the web server.
5. The port 80 listener accepts the request and allows the client to set up a connection session so that data communication can occur.
6. The web server program takes control from the listener.
7. The web server parses the request and determines if the request is for static content that can be retrieved from the file system or dynamic content that requires another process to build the content. The *context root* (in this example, jspapp) is associated with the location of the static content or the process that will supply the content. Some examples follow:
   - **For a static HTML file**, the server locates the file within the physical directory mapped to the context root. No additional program (other than retrieving the file) is needed.
   - **For a J2EE application file** such as forum_query.jsp in this example, the file requested in the URL is found in the physical directory that maps to the context root. The sidebar "About server.xml and web.xml" describes the mapping mechanism in more detail.
   - **For a J2EE application without a file name**, such as http://www.oracle.com/jspapp, the web server finds the welcome file for the application and returns its content. The sidebar "About server.xml and web.xml" provides more detail about how the web server determines the welcome file.
8. The web server runs the code associated with the file (if it is a J2EE program such as a servlet or JSP page) or opens the file (if it is an HTML or other type of non-program file).
9. The web server constructs a response comprised of the status line, headers, and message body (containing the requested content) and sends that response to the browser. The browser then renders the content and closes the connection.

About server.xml and web.xml

J2EE specifies standards used to write and place descriptor files the web server uses for fulfilling requests. When the web server needs to find a J2EE web application file to satisfy an HTTP request, it parses the context root from the URL. It then looks in the *server.xml* file (an XML configuration file located in the web server's ../j2ee/home/config directory) for an entry such as the following:

```
<application name="jspapp" path="../applications/jspapp" auto-start="true" />
```

This entry identifies the application (the *name* attribute) and associates it with the physical directory (the *path* attribute). The file mentioned in the URL will be located in a subdirectory (for example, /jsp) of the context root directory. Therefore, the URL http://www.oracle.com/jspapp/ forum_query.jsp may point to the forum_query.jsp file in the Oracle Application Server 10*g* (web server) directory, O:\Oracle\Product\mtier10g\j2ee\home\applications\jspapp\jsp.

If the URL contains no file name, the application server determines which file to open based on an entry in *web.xml*—the *web module deployment descriptor*. Web.xml is another XML file, which is located in the context root's WEB-INF directory. Web.xml contains, among other entries, an entry that defines the startup page for the application, for example:

```
<welcome-file-list>
  <welcome-file>forum_query.jsp</welcome-file>
</welcome-file-list>
```

The web server determines from this entry that a URL containing only the application context root will start the forum_query.jsp file.

Note

Tim Berners-Lee, the inventor of the World Wide Web, tells the story of how web communications work to "kids of various ages (6–96)" at his W3C website page, www.w3.org/People/Berners-Lee/Kids.html.

Note

You can find information about how Oracle Forms runs on the Web in Chapter 3 of the “Oracle Application Server Forms Services Deployment Guide 10g Release 2 (10.1.2)” online documentation available at otn.oracle.com (for example, at download-east.oracle.com/docs/cd/B25016_04/doc/dl/web.htm).

Conclusion

This paper has described various aspects of the basic components and features of HTTP communications. Although you will likely never write programs at the communications level, knowledge of what is happening behind the scenes between the web browser and the application server will serve you well in developing and debugging web applications.

About the Author

Peter Koletzke is a technical director and principal instructor for the Enterprise e-Commerce Solutions practice at Quovera, in Mountain View, California, and has worked in the database industry since 1984. Peter has presented at various Oracle users group conferences more than 170 times and has won awards such as Pinnacle Publishing's Technical Achievement, Oracle Development Tools Users Group (ODTUG) Editor's Choice, ECO/SEOUC Oracle Designer Award, the ODTUG Volunteer of the Year, and NYOUG Editor’s Choice. He is an Oracle Fusion Middleware Regional Director and Oracle Certified Master. Peter is coauthor of the Oracle Press books: *Oracle JDeveloper 10g for Forms & PL/SQL Developers* (with Duncan Mills from which some of the material in this white paper is taken); *Oracle JDeveloper 10*g *Handbook* and *Oracle9*i *JDeveloper Handbook* (with Dr. Paul Dorsey and Avrom Roy-Faderman); *Oracle JDeveloper 3 Handbook*, *Oracle Developer Advanced Forms and Reports*, *Oracle Designer Handbook, 2nd Edition*, and *Oracle Designer/2000 Handbook* (with Dr; Paul Dorsey).

Quovera is a business consulting and technology integration firm that specializes in delivering solutions to the high technology, telecommunications, semiconductor, manufacturing, software and services, public sector, and financial services industries. Quovera deploys solutions that deliver optimized business processes quickly and economically, driving increased productivity and improved operational efficiency. Founded in 1995, the company has a track record of delivering hundreds of strategy, design, and implementation projects to over 250 Fortune 2000 and high growth middle market companies. Quovera's client list includes notable companies such as Cisco Systems, ON Semiconductor, New York State, Sun Microsystems, Seagate, Toyota, Fujitsu, Visa, and Cendant. www.quovera.com.