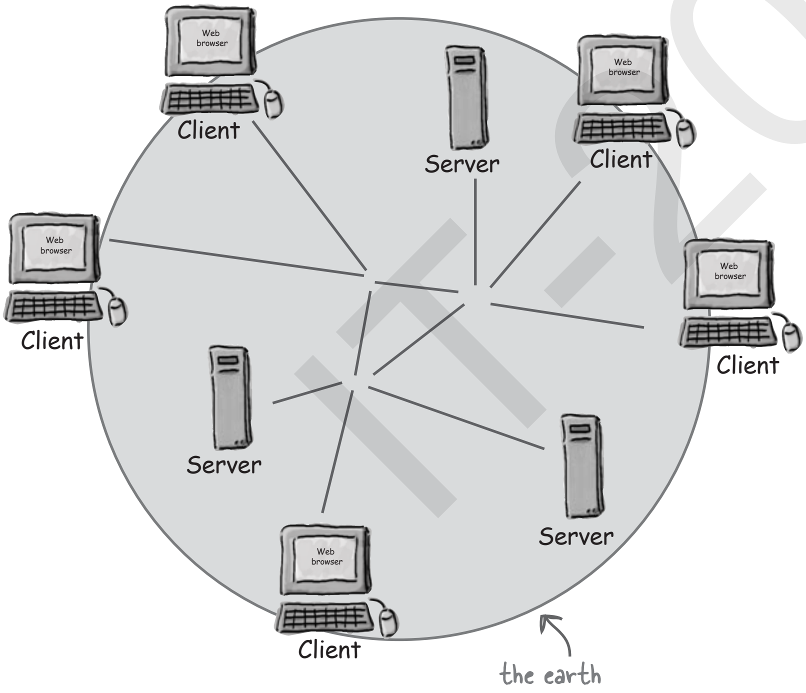
Everybody wants a web site

You have a killer idea for a web site. To destroy the competition, you need a flexible, scalable architecture. You need servlets and JSPs.

Before we start building, let's take a look at the World Wide Web from about 40k feet. What we care most about in this chapter are how web clients and web servers talk to one another.

These next several pages are probably all review for you, especially if you're already a web application developer, but it'll give us a chance to expose some of the terminology we use throughout the book.



The web consists of gazillions of clients (using browsers like Mozilla or Safari) and servers (using web server apps like Apache) connected through wires and wireless networks. Our goal is to build a web application that clients around the globe can access. And to become obscenely rich.

What does your web server do?

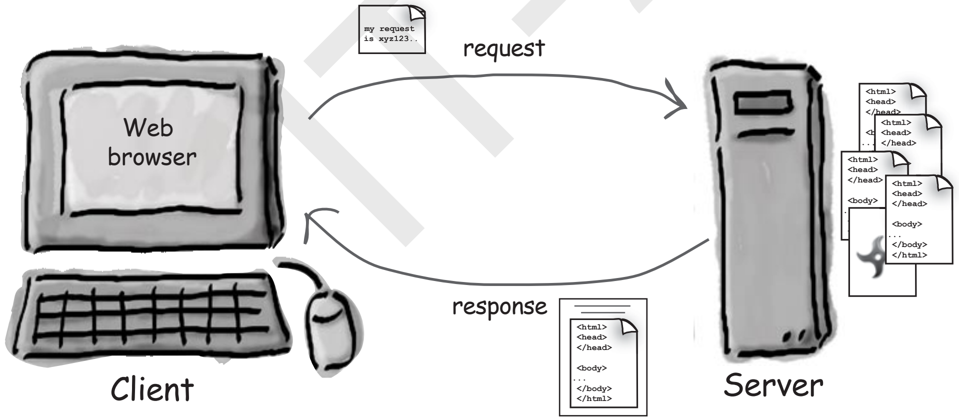
A web server takes a client request and gives something back to the client.

A web *browser* lets a user request a *resource*. The web *server* gets the request, finds the resource, and returns something to the user. Sometimes that resource is an *HTML page*. Sometimes it's a *picture*. Or a *sound* file. Or even a *PDF* document. Doesn't matter—the client asks for the thing (resource) and the server sends it back.

Unless the thing isn't there. Or at least it's not where the server is expecting it to be. You're of course quite familiar with the "404 Not Found" error—the response you get when the server can't find what it thinks you asked for.

When we say "server", we mean *either* the physical machine (hardware) or the web server application (software). Throughout the book, if the difference between server hardware and software matters, we'll explicitly say which one (hardware or software) we're talking about.

The client's request contains the name and address (the URL), of the thing the client is looking for.



The server usually has lots of "content" that it can send to clients. That content can be web pages, JPEGs, and other resources.

The server's response contains the actual document that the client requested (or an error code if the request could not be processed).

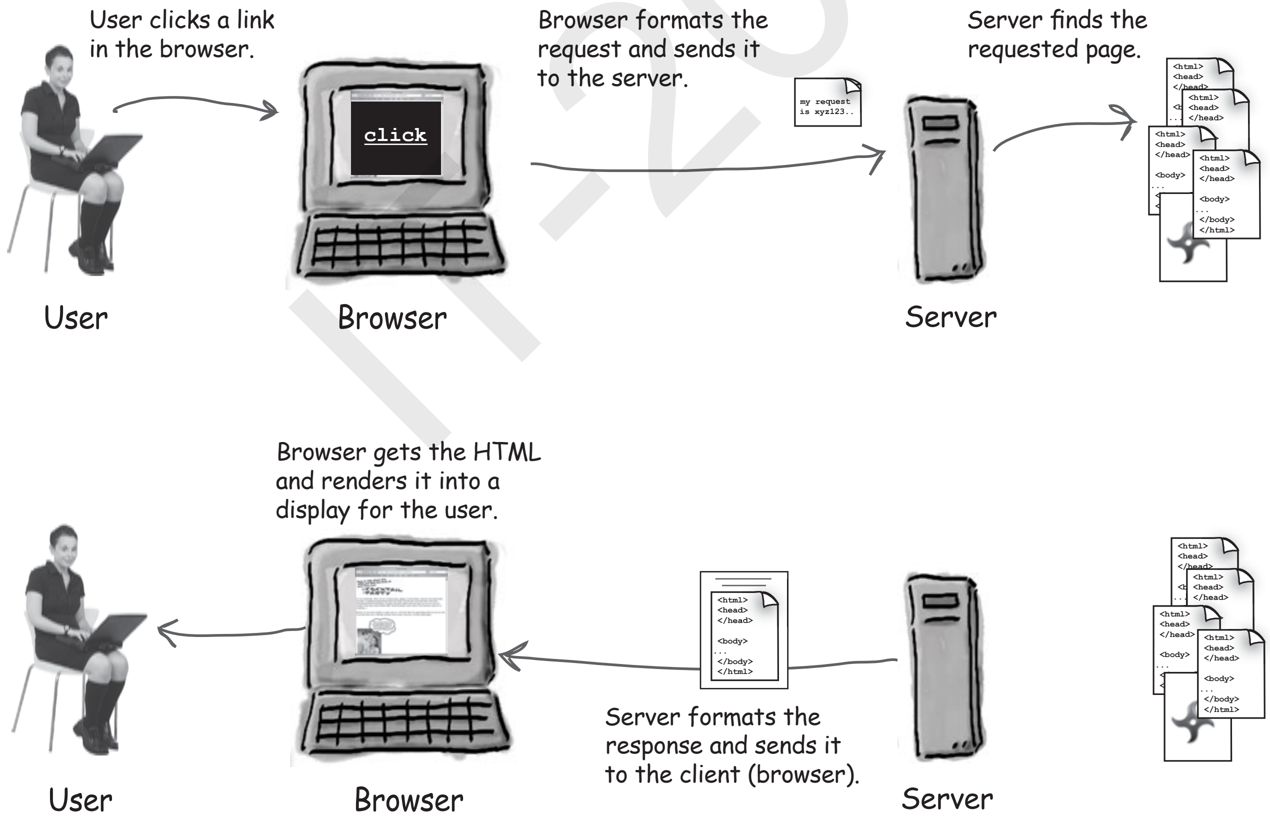
What does a web client do?

A web client lets the user request something on the server, and shows the user the result of the request.

When we talk about *clients*, though, we usually mean both (or either) the *human* user and the browser *application*.

The *browser* is the piece of software (like Netscape or Mozilla) that knows how to communicate with the server. The browser's other big job is interpreting the HTML code and *rendering* the web page for the user.

So from now on, when we use the term *client*, we usually won't care whether we're talking about the human user *or* the browser app. In other words, the *client* is the *browser app doing what the user asked it to do*.


User clicks a link in the browser.
click
Browser formats the request and sends it to the server.
my request is xyz123..
Server finds the requested page.
<html>
<head>
</head>
<body>
...
</body>
</html>
User
Browser
Server
Browser gets the HTML and renders it into a display for the user.
Server formats the response and sends it to the client (browser).
User
Browser
Server

Clients and servers know HTML and HTTP

HTML

When a server answers a request, the server usually sends some type of content to the browser so that the browser can display it. Servers often send the browser a set of instructions written in HTML, the HyperText Markup Language. The HTML tells the browser how to present the content to the user.

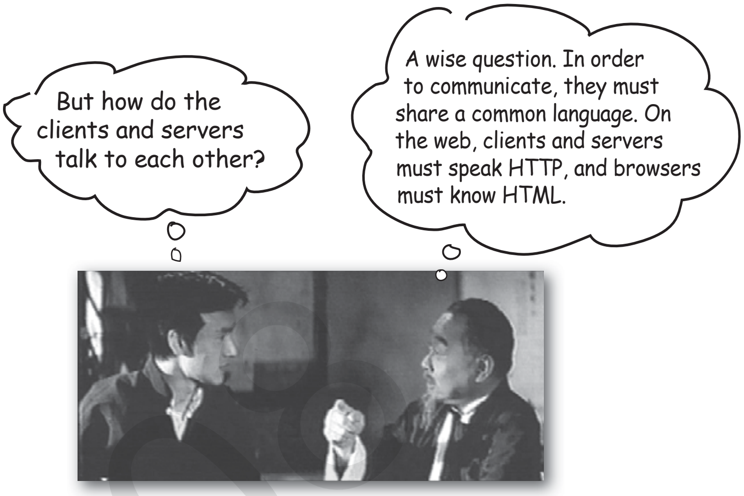
All web browsers know what to do with HTML, although sometimes an *older* browser might not understand parts of a page that was written using *newer* versions of HTML.

HTTP

Most of the conversations held on the web between clients and servers are held using the HTTP protocol, which allows for simple request and response conversations. The client sends an HTTP request, and the server answers with an HTTP response. Bottom line: ***if you're a web server, you speak HTTP.***

When a web server sends an HTML page to the client, it sends it using HTTP. (You'll see the details on how all this works in the next few pages.)

(FYI: HTTP stands for HyperText Transfer Protocol.)


But how do the clients and servers talk to each other?
A wise question. In order to communicate, they must share a common language. On the web, clients and servers must speak HTTP, and browsers must know HTML.

HTML tells the browser how to display the content to the user.

HTTP is the protocol clients and servers use on the web to communicate.

The server uses HTTP to send HTML to the client.

Two-minute HTML guide

When you develop a web page, you use HTML to describe what the page should look like and how it should behave.

HTML has dozens of ***tags*** and hundreds of tag ***attributes***. The goal of HTML is to take a text document and add tags that tell the browser how to format the text. Below are the tags we use in the next several chapters. If you need a more complete understanding of HTML, we recommend the book *HTML & XHTML The Definitive Guide* (O'Reilly).

Tag	Description
`<!--   -->`	where you put your *comments*
`<a>`	*anchor* - usually for putting in a hyperlink
`<align>`	*align* the contents left, right, centered, or justified
`<body>`	define the boundaries of the document's *body*
`<br>`	a *line break*
`<center>`	*center* the contents
`<form>`	define a *form* (which usually provides input fields)
`<h1>`	the first level *heading*
`<head>`	define the boundaries of the document's *header*
`<html>`	define the boundaries of the HTML *document*
`<input type>`	defines an *input widget* to a form
`<p>`	a new *paragraph*
`<title>`	the HTML document's *title*

(Technically, the <center> and <align> tags have been deprecated in HTML 4.0, but we're using them in some of our examples because it's simpler to read than the alternative, and you're not here to learn HTML anyway.)

What you write...

(the HTML)

Imagine you’re creating a login page. The simple HTML might look something like this:

```
    <html>
    <!-- Some sample HTML -->
    <head>
(A)   <title>A Login Page</title>
    </head>
    <body>
(B) <h1 align=”center”>Skyler’s Login Page</h1>

    <p align=”right”>
(C)   <img src=”SKYLER2.jpg” width=”130” height=”150”/>
    </p>

    <form action=”date2”>
(D)        Name: <input type=”text” name=”param1”/><br/>
      Password: <input type=”text” name=”param2”/><br/><br/><br/>

      <center>
        <input type=”SUBMIT”/>
(E)   </center>
    </form>

    </body>
    </html>
```

An HTML comment

The <img> tag is nested inside a paragraph <align> tag in order to place the image roughly where we want it. (Remember, <align> is deprecated, but we’re using it because it’s simple to read.)

The servlet to send the request to.

We’ll talk more about forms later, but briefly, the browser can collect the user’s input and return it to the server.

The <br> tags cause line breaks.

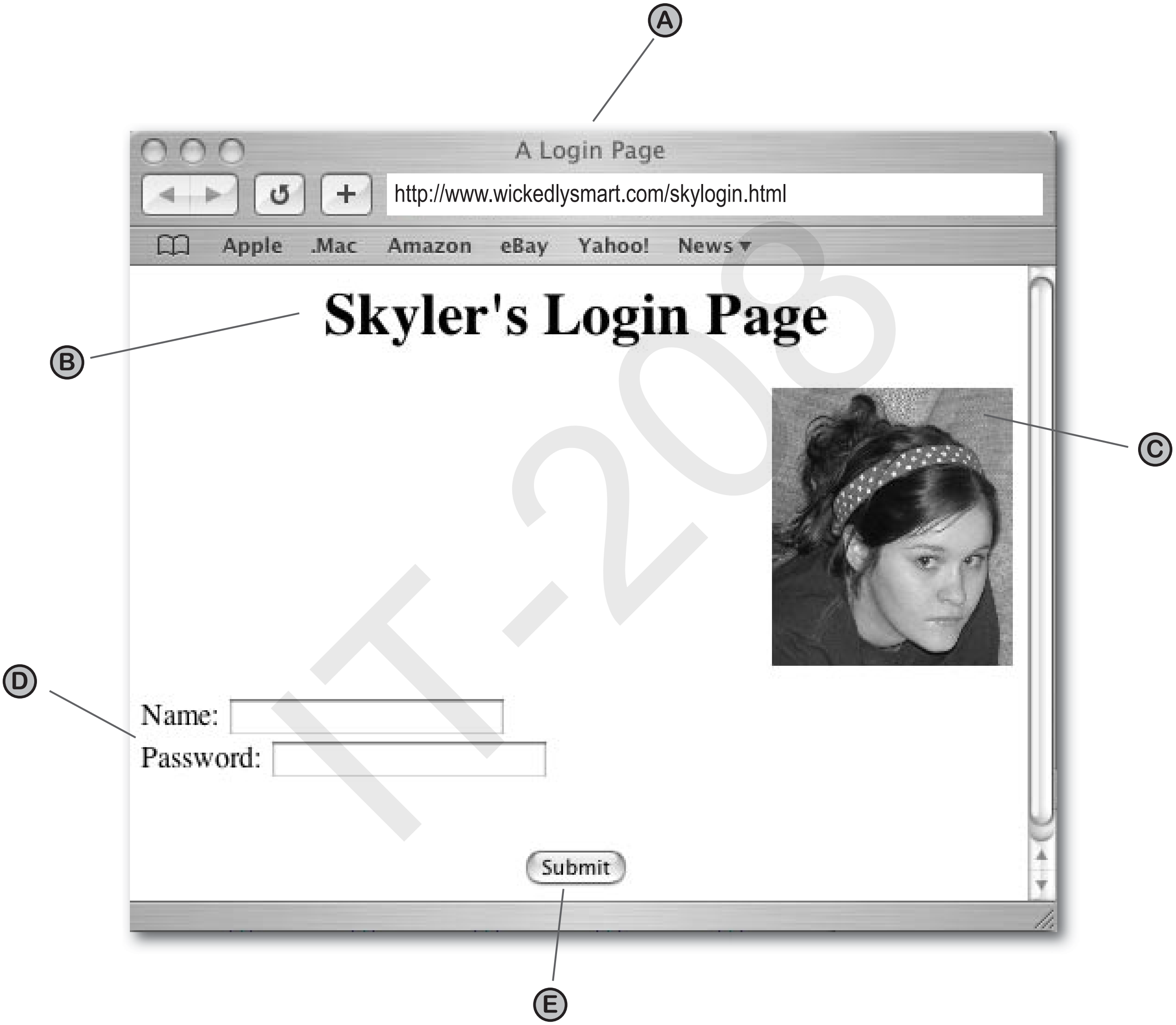
The “submit” button in the form.

Relax ***You need only the most basic HTML knowledge.***

HTML pops up all over the exam. But you’re not being *tested* on your HTML knowledge. You’ll see HTML in the context of a large chunk of questions, though, so you need at least some idea of what’s happening when you see simple HTML.

What the browser creates...

The browser reads through the HTML code, creates the web page, and renders it to the user's display.

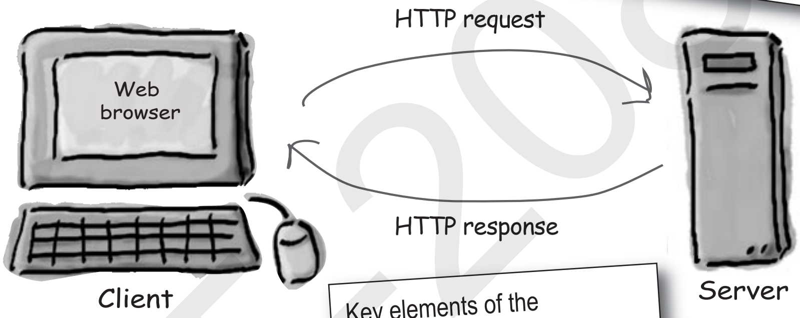


What is the HTTP protocol?

HTTP runs on top of TCP/IP. If you're not familiar with those networking protocols, here's the crash course: TCP is responsible for making sure that a file sent from one network node to another ends up as a complete file at the destination, even though the file is split into chunks when it's sent. IP is the underlying protocol that moves/routes the chunks (packets) from one host to another on their way to the destination. HTTP, then, is another network protocol that has Web-specific features, but it depends on TCP/IP to get the complete request and response from one place to another. The structure of an HTTP conversation is a simple **Request/Response** sequence; a browser *requests*, and a server *responds*.

Key elements of the **request** stream:

- HTTP method (the action to be performed)
- The page to access (a URL)
- Form parameters (like arguments to a method)



Key elements of the **response** stream:

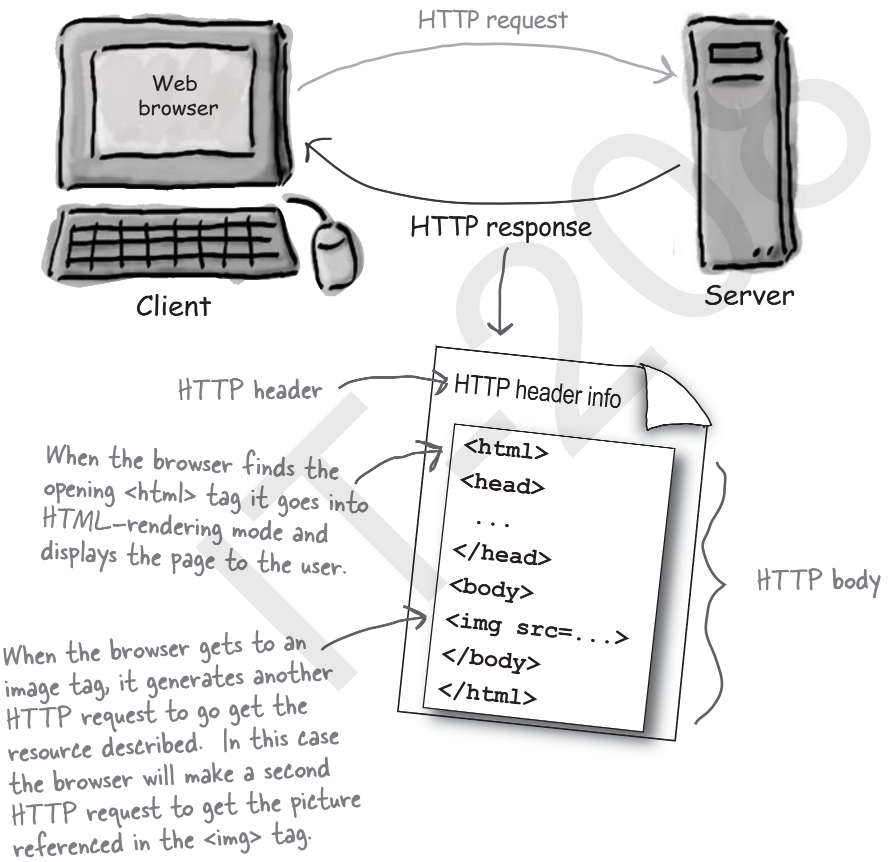
- A status code (for whether the request was successful)
- Content-type (text, picture, HTML, etc.)
- The content (the actual HTML, image, etc.)

***Relax* You don't have to memorize the HTTP spec.**

The HTTP protocol is an IETF standard, RFC 2616. If you care. (Fortunately, the exam doesn't expect you to.) Apache is an example of a Web server that processes HTTP requests. Mozilla is an example of a Web browser that provides the user with the means to make HTTP requests and to view the documents returned by the server.

HTML is part of the HTTP response

An HTTP response can *contain* HTML. HTTP adds header information to the top of whatever content is in the response (in other words, the *thing* coming back from the server). An HTML browser uses that header info to help process the HTML page. Think of the HTML content as data pasted inside an HTTP response.



```
<html>
<head>
 ...
</head>
<body>
<img src=...>
</body>
</html>
```

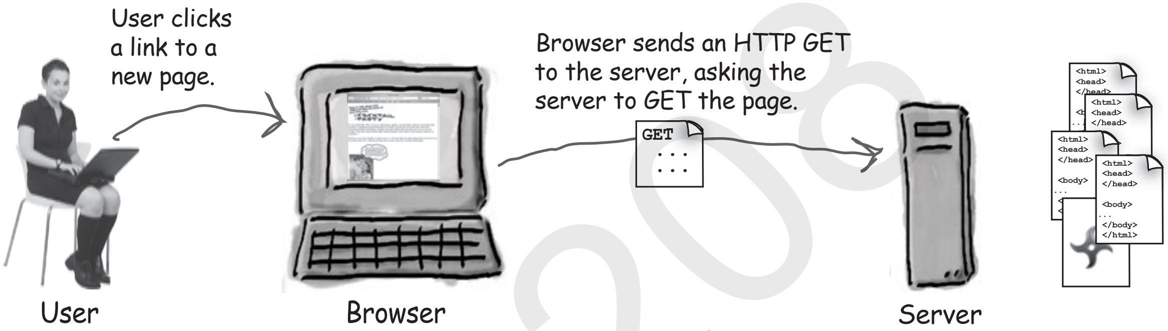
When the browser finds the opening <html> tag it goes into HTML—rendering mode and displays the page to the user.

When the browser gets to an image tag, it generates another HTTP request to go get the resource described. In this case the browser will make a second HTTP request to get the picture referenced in the <img> tag.

If that's the response, what's in the request?

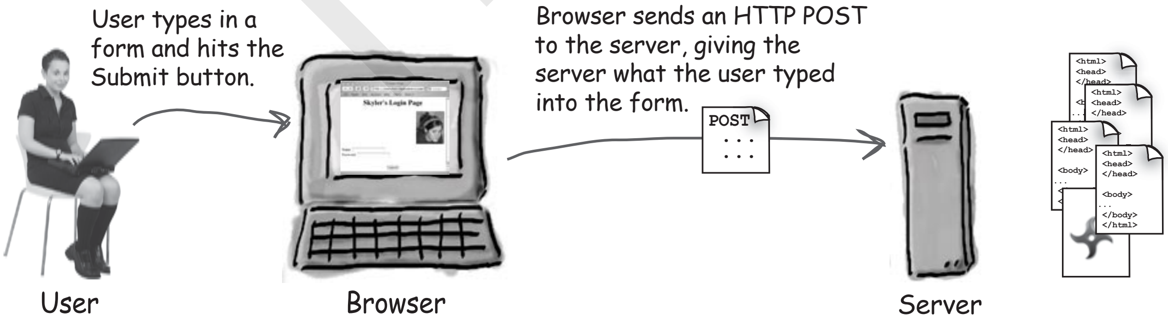
The first thing you'll find is an HTTP *method* name. These aren't *Java* methods, but the idea is similar. The method name tells the server the kind of request that's being made, and how the rest of the message will be formatted. The HTTP protocol has several methods, but the ones you'll use most often are *GET* and *POST*.

GET


User clicks a link to a new page.

Browser sends an HTTP GET to the server, asking the server to GET the page.

POST


User types in a form and hits the Submit button.

Browser sends an HTTP POST to the server, giving the server what the user typed into the form.

GET is a simple request, POST can send user data

GET is the simplest HTTP method, and its main job in life is to ask the server to *get* a resource and send it back. That resource might be an HTML page, a JPEG, a PDF, etc. Doesn't matter. The point of GET is to *get* something back from the server.

POST is a more powerful request. It's like a GET plus plus. With POST, you can *request* something and at the same time *send* form data to the server (later in this chapter we'll see what the server might do with that data).

Wait a minute... I could swear I've seen GET requests that *did* send some parameter data to the server.

there are no Dumb Questions

Q: So what about the other HTTP methods besides GET and POST?

A: Those are the two big ones that everybody uses. But there are a few rarely used methods (and Servlets can handle them) including HEAD, TRACE, PUT, DELETE, OPTIONS, and CONNECT.

You really don't need to know much about these others for the exam, although you might see them appear in a question. The Life and Death of a Servlet chapter covers the rest of the HTTP method details you'll need.

It's true... you can send a little data with HTTP GET

But you might not want to. Reasons you might use POST instead of GET include:

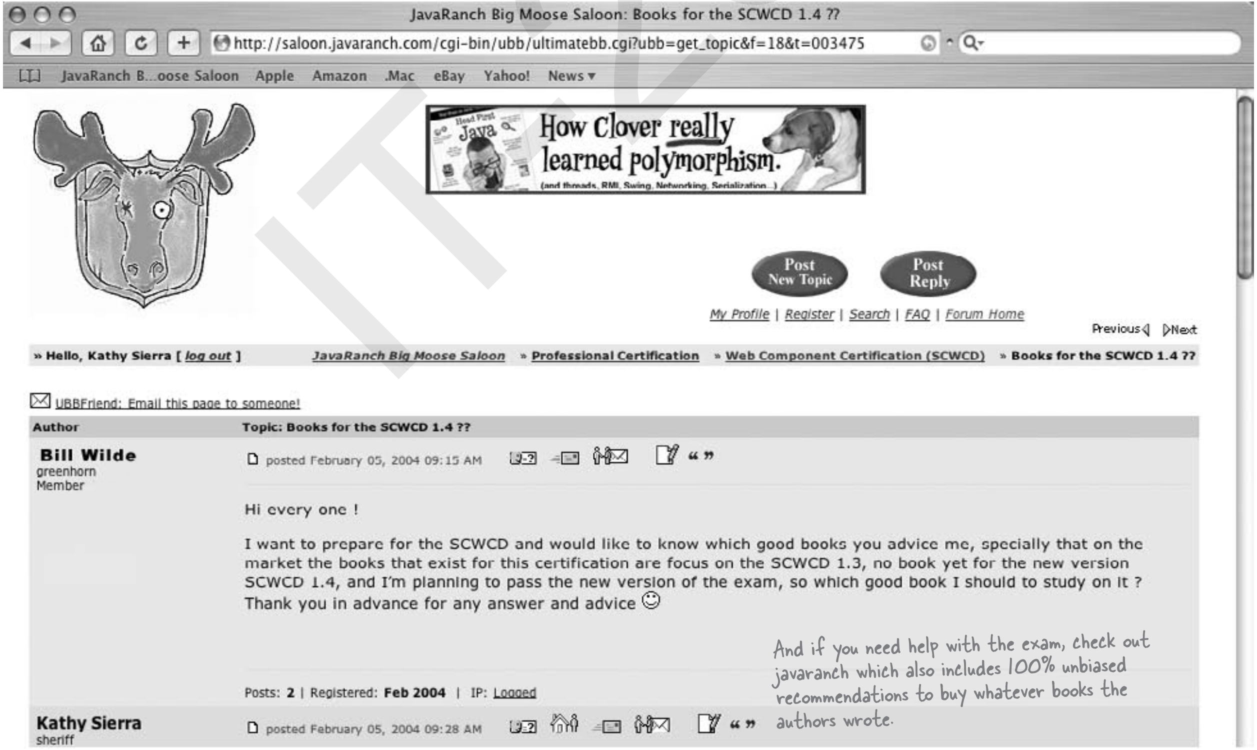
1. The total amount of characters in a GET is really limited (depending on the server). If the user types, say, a long passage into a "search" input box, the GET might not work.

2. The data you send with the GET is appended to the URL up in the browser bar, so whatever you send is exposed. Better not put a password or some other sensitive data as part of a GET!

3. Because of number two above, the user can't bookmark a form submission if you use POST instead of GET. Depending on your app, you may or may not want users to be able to bookmark the resulting request from a form submission.

The "?" separates the path and the parameters (the extra data). The amount of data you can send along with the GET is limited, and it's exposed up here in the browser bar for everyone to see. Together, the entire String is the URL that is sent with the request.

The original URL before the extra parameters.



JavaRanch Big Moose Saloon: Books for the SCWCD 1.4 ??

http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get_topic&f=18&t=003475

JavaRanch B...oose Saloon
Apple
Amazon
.Mac
eBay
Yahoo!
News ▾
Head First Java
How Clover really learned polymorphism.
(and threads, RMI, Swing, Networking, Serialization...)
Post New Topic
Post Reply
My Profile | Register | Search | FAQ | Forum Home
Previous
Next

» Hello, Kathy Sierra [log out]   JavaRanch Big Moose Saloon » Professional Certification » Web Component Certification (SCWCD) » Books for the SCWCD 1.4 ??

UBBFriend: Email this page to someone!

Author	Topic: Books for the SCWCD 1.4 ??
Bill Wilde greenhorn Member	posted February 05, 2004 09:15 AM Hi every one ! I want to prepare for the SCWCD and would like to know which good books you advice me, specially that on the market the books that exist for this certification are focus on the SCWCD 1.3, no book yet for the new version SCWCD 1.4, and I'm planning to pass the new version of the exam, so which good book I should to study on it ? Thank you in advance for any answer and advice ☺ Posts: 2 | Registered: Feb 2004 | IP: Logged
Kathy Sierra sheriff	posted February 05, 2004 09:28 AM

And if you need help with the exam, check out javaranch which also includes 100% unbiased recommendations to buy whatever books the authors wrote.

Anatomy of an HTTP GET request

The path to the resource, and any parameters added to the URL are all included on the "request line".

The Request line.

The HTTP Method.

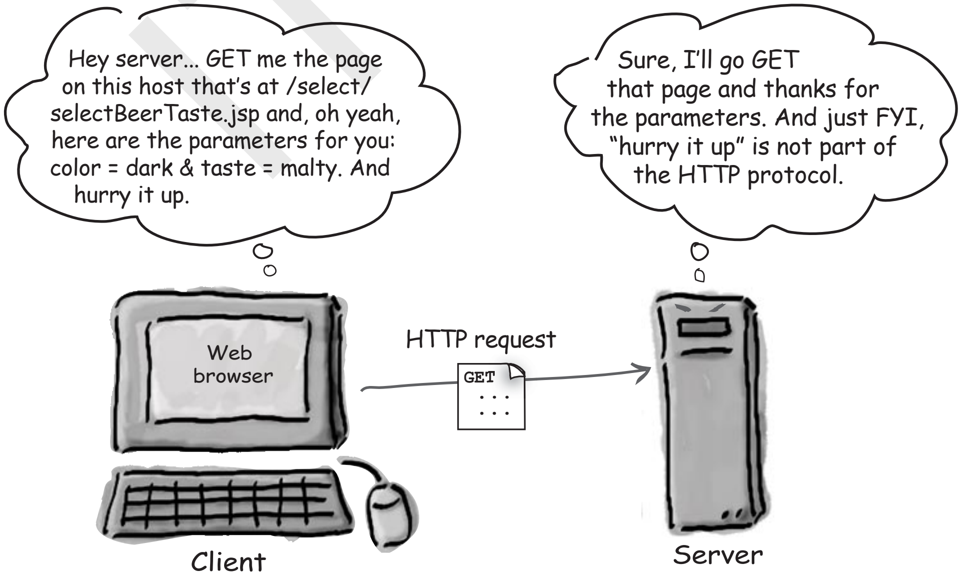
The path to the resource on the web server.

In a GET request, parameters (if there are any) are appended to the first part of the request URL, starting with a "?". Parameters are separated with an ampersand "&".

The protocol version that the web browser is requesting.

```
GET /select/selectBeerTaste.jsp?color=dark&taste=malty HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/
20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
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
```

The Request headers.


Hey server... GET me the page on this host that's at /select/selectBeerTaste.jsp and, oh yeah, here are the parameters for you: color = dark & taste = malty. And hurry it up.
Sure, I'll go GET that page and thanks for the parameters. And just FYI, "hurry it up" is not part of the HTTP protocol.
Web browser
HTTP request
GET
Client
Server

Anatomy of an HTTP POST request

HTTP POST requests are designed to be used by the browser to make complex requests on the server. For instance, if a user has just completed a long form, the application might want all of the form's data to be added to a database. The data to be sent back to the server is known as the "message body" or "payload" and can be quite large.

The HTTP Method.

The path to the resource on the web server.

The protocol version that the web browser is requesting.

The Request line.

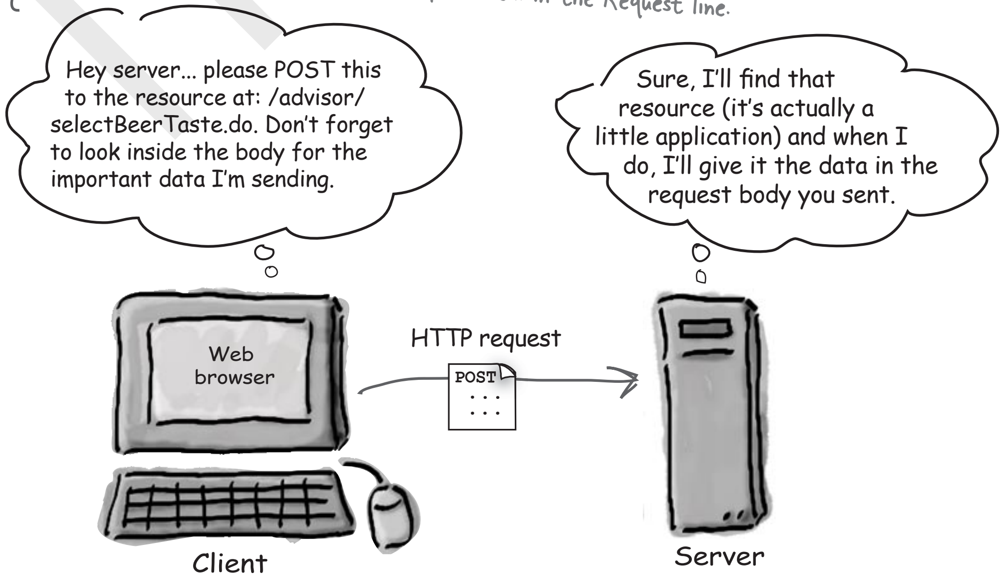
```
POST /advisor/selectBeerTaste.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/
20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
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

color=dark&taste=malty
```

The Request headers.

The message body, sometimes called the "payload".

This time, the parameters are down here in the body, so they aren't limited the way they are if you use a GET and have to put them in the Request line.


Hey server... please POST this to the resource at: /advisor/selectBeerTaste.do. Don't forget to look inside the body for the important data I'm sending.
Sure, I'll find that resource (it's actually a little application) and when I do, I'll give it the data in the request body you sent.
Web browser
HTTP request
POST
Client
Server

Anatomy of an HTTP response, and what the heck is a "MIME type"?

Now that we've seen the requests from the browser to the server, let's look at what the server sends back in response. An HTTP response has both a header and a body. The header info tells the browser about the protocol being used, whether the request was successful, and what kind of content is included in the body. The body contains the contents (for example, HTML) for the browser to display.

The protocol version that the web server is using.

The HTTP status code for the Response.

A text version of the status code.

HTTP Response headers.

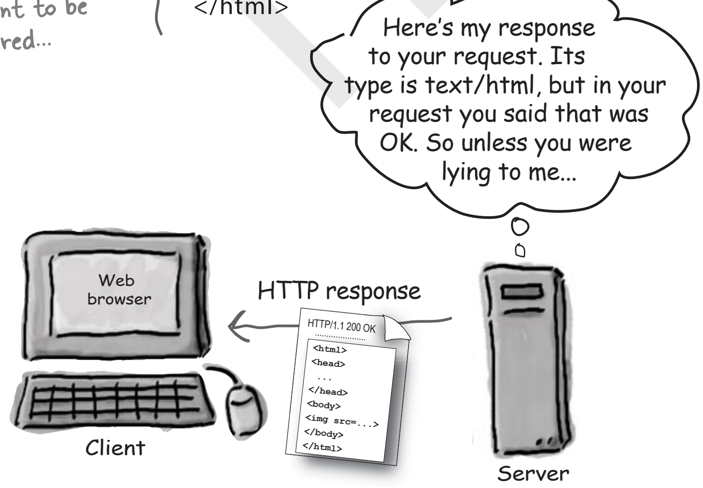
```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=0AAB6C8DE415E2E5F307CF334BFCA0C1; Path=/testEL
Content-Type: text/html
Content-Length: 397
Date: Wed, 19 Nov 2003 03:25:40 GMT
Server: Apache-Coyote/1.1
Connection: close

<html>
...
</html>
```

The content-type response header's value is known as a MIME type. The MIME type tells the browser what kind of data the browser is about to receive so that the browser will know how to render it.

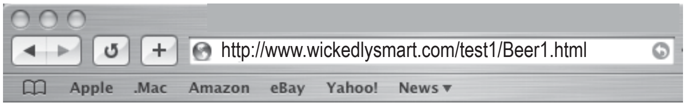
The body holds the HTML, or other content to be rendered...

Notice that the MIME type value relates to the values listed in the HTTP request's "Accept" header. (Go look at the Accept header from the previous page's POST request.)


Here's my response to your request. Its type is text/html, but in your request you said that was OK. So unless you were lying to me...
Web browser
HTTP response
HTTP/1.1 200 OK
<html>
<head>
...
</head>
<body>
<img src=...>
</body>
</html>
Client
Server

All the pieces. On one page.

The user types a URL.



The browser creates an HTTP GET request.

```
GET /test1/Beer1.html HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh...
. . .
```

The HTTP GET is sent to the server.

The server finds the page...

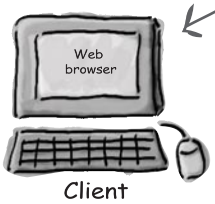
Beer1.html

```
<html><body>
<h1 align=center>Beer Login Page</h1>
<form>
  Select a beer type or buy beer
  making supplies?<p>
  <input type=radio name=select
    value=Select> Select a beer<br>
  <input type=radio name=select
    value=Buy> Buy supplies<br><br>
  <center>
    <input type=SUBMIT>
  </center>
</form>
</body></html>
```

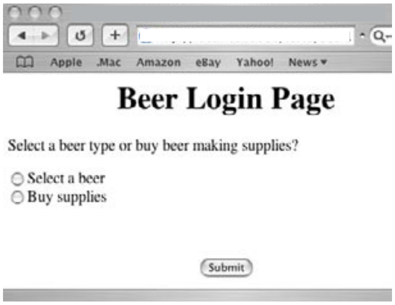
And generates an HTTP response.

```
HTTP/1.1 200 OK
Set-Cookie: . . .
. . .
<html><body>
<h1 align=center>Beer Login Page</h1>
<form>
 Select a beer type or buy beer . . .
```

The HTTP response is sent to the browser.



The browser renders the HTML.



Beer Login Page

Select a beer type or buy beer making supplies?

○ Select a beer
○ Buy supplies

Submit

Client looks forward to a successful beer transaction.

Sharpen your pencil

GET or POST?

For each description, circle either POST or GET depending on which HTTP method you'd choose for implementing that functionality. If you think it could be either, circle both. But be prepared to defend your answers...

POST GET *A user is returning a login name and password.*

POST GET *A user is requesting a new page via a hyperlink.*

POST GET *A chat room user is sending a written response.*

POST GET *A user hits the 'next' button to see the next page.*

POST GET *A user hits the 'log out' button on a secure banking site.*

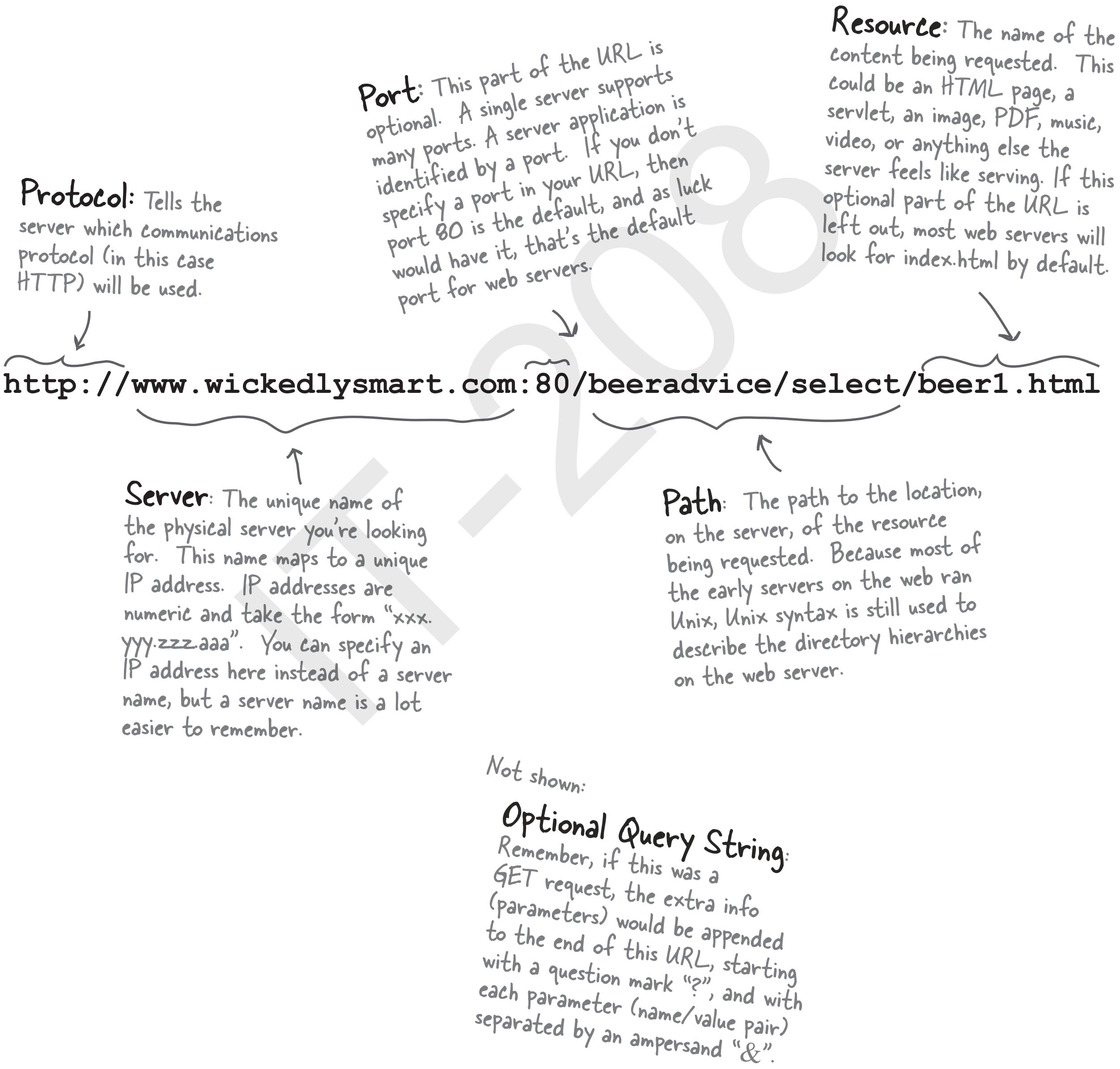
POST GET *A user hits the 'back' button on the browser.*

POST GET *A user sends a name and address form to the server.*

POST GET *A user makes a radio button selection.*

URL. Whatever you do, don't pronounce it "Earl".

When you get to the U's in the acronym dictionary there's a traffic jam... URI, URL, URN, where does it end? For now, we're going to focus on the URLs, or **U**niform **R**esource **L**ocators, that you know and love. Every resource on the web has its own unique address, in the URL format.



Protocol: Tells the server which communications protocol (in this case HTTP) will be used.

Port: This part of the URL is optional. A single server supports many ports. A server application is identified by a port. If you don't specify a port in your URL, then port 80 is the default, and as luck would have it, that's the default port for web servers.

Resource: The name of the content being requested. This could be an HTML page, a servlet, an image, PDF, music, video, or anything else the server feels like serving. If this optional part of the URL is left out, most web servers will look for index.html by default.

```
http://www.wickedlysmart.com:80/beeradvice/select/beer1.html
```

Server: The unique name of the physical server you're looking for. This name maps to a unique IP address. IP addresses are numeric and take the form "xxx.yyy.zzz.aaa". You can specify an IP address here instead of a server name, but a server name is a lot easier to remember.

Path: The path to the location, on the server, of the resource being requested. Because most of the early servers on the web ran Unix, Unix syntax is still used to describe the directory hierarchies on the web server.

Not shown:

Optional Query String: Remember, if this was a GET request, the extra info (parameters) would be appended to the end of this URL, starting with a question mark "?", and with each parameter (name/value pair) separated by an ampersand "&".

Off the path

A TCP port is just a number

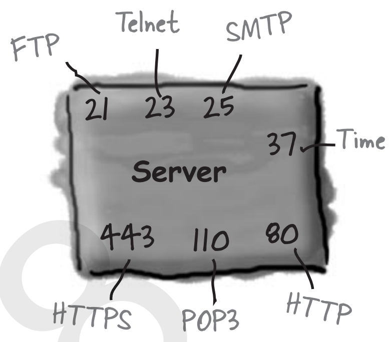
A 16-bit number that identifies a specific software program on the server hardware.

Your internet web (HTTP) server software runs on port 80. That’s a standard. If you’ve got a Telnet server, it’s running on port 23. FTP? 21. POP3 mail server? 110. SMTP? 25. The Time server sits at 37. Think of ports as unique identifiers. A port represents a logical connection to a particular piece of software running on the server *hardware*. That’s it. You can’t spin your hardware box around and find a TCP port. For one thing, you have 65536 of them on a server (0 to 65535). For another, they do *not* represent a place to plug in physical devices. They’re just numbers representing a server application.

Without port numbers, the server would have no way of knowing which application a client wanted to connect to. And since each application might have its own unique protocol, think of the trouble you’d have without these identifiers. What if your web browser, for example, landed at the POP3 mail server instead of the HTTP server? The mail server won’t know how to parse an HTTP request! And even if it did, the POP3 server doesn’t know anything about serving back an HTML page.

If you’re writing services (server programs) to run on a company network, you should check with the sys-admins to find out which ports are already taken. Your sys-admins might tell you, for example, that you can’t use any port number below, say, 3000.

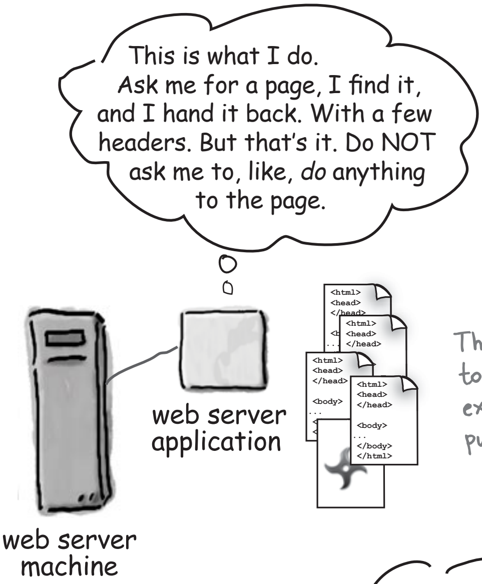
Well-known TCP port numbers for common server applications



Using one server app per port, a server can have up to 65536 different server apps running.

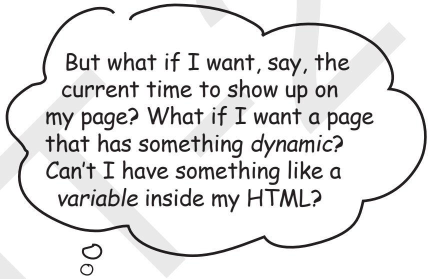
The TCP port numbers from 0 to 1023 are reserved for well-known services (including the Big One we care about—port 80). Don’t use these ports for your own custom server programs!

Web servers love serving static web pages


This is what I do. Ask me for a page, I find it, and I hand it back. With a few headers. But that's it. Do NOT ask me to, like, *do* anything to the page.
web server application
web server machine

A static page just sits there in a directory. The server finds it and hands it back to the client as is. Every client sees the same thing.

These pages go straight to the client just exactly as they were put on the server.


But what if I want, say, the current time to show up on my page? What if I want a page that has something *dynamic*? Can't I have something like a *variable* inside my HTML?

What if we want to stick something variable inside the HTML page?

```
<html>
<body>
The current time is [insertTimeOnServer].
</body>
</html>
```

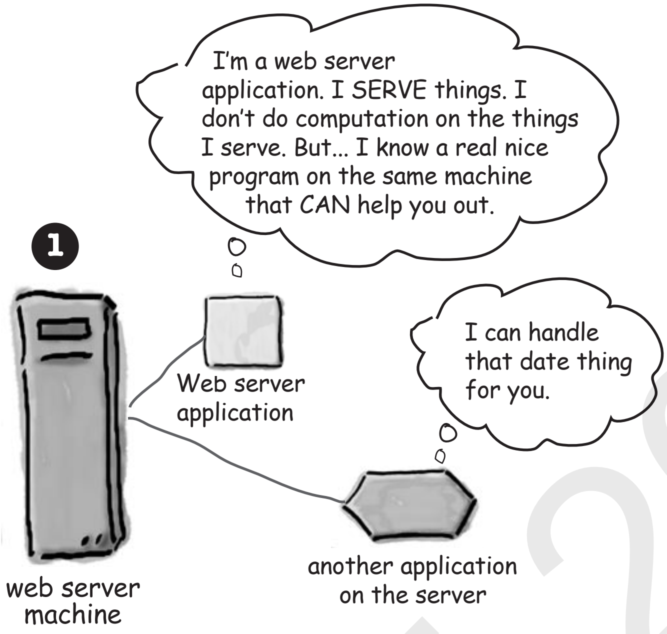
Answers from previous page:

B- www.wickedlysmart.com/skiingAdvice/

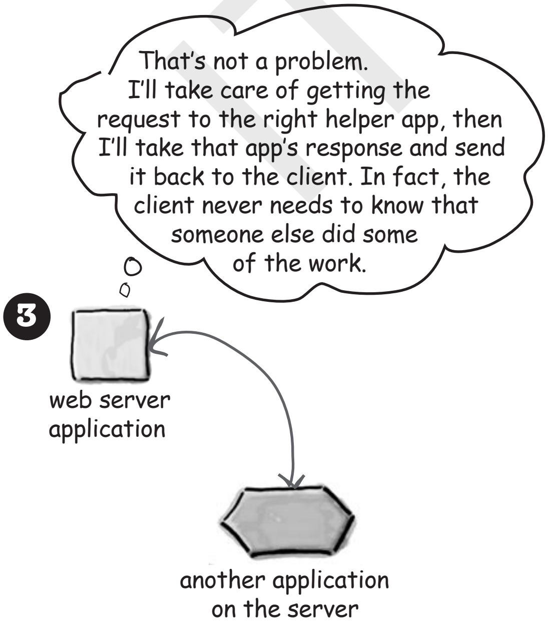
C- www.wickedlysmart.com/beerAdvice/

D- www.wickedlysmart.com/beerAdvice/select/selectBeer.html

But sometimes you need more than just the web server


1
I'm a web server application. I SERVE things. I don't do computation on the things I serve. But... I know a real nice program on the same machine that CAN help you out.
I can handle that date thing for you.
Web server application
another application on the server
web server machine

2
But how does that help? My clients are all *web* clients. The browser knows only about the web server... so it won't be able to call that other application.


That's not a problem. I'll take care of getting the request to the right helper app, then I'll take that app's response and send it back to the client. In fact, the client never needs to know that someone else did some of the work.
3
web server application
another application on the server

Two things the web server alone won't do

If you need just-in-time pages (dynamically-created pages that don't exist before the request) and the ability to write/save data on the server (which means writing to a file or database), you can't rely on the web server alone.

1 Dynamic content

The web server application serves only static pages, but a separate "helper" application that the web server can communicate with can build non-static, just-in-time pages. A dynamic page could be anything from a catalog to a weblog or even just a page that randomly chooses pictures to display.

Just-in-time pages don't exist before the request comes in. It's like making an HTML page out of thin air.

The request comes in, the helper app "writes" the HTML, and the web server gets it back to the client.

When instead of this:

```
<html>
<body>
The current time is
always 4:20 PM
on the server
</body>
</html>
```

You want this:

```
<html>
<body>
The current time is
[insertTimeOnServer]
on the server
</body>
</html>
```

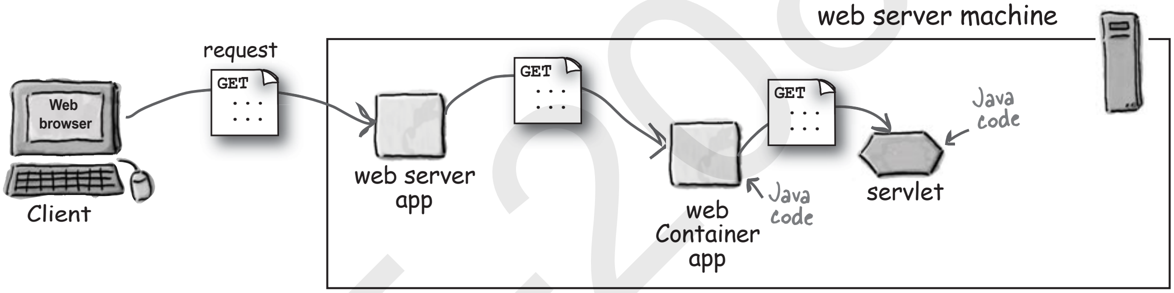
2 Saving data on the server

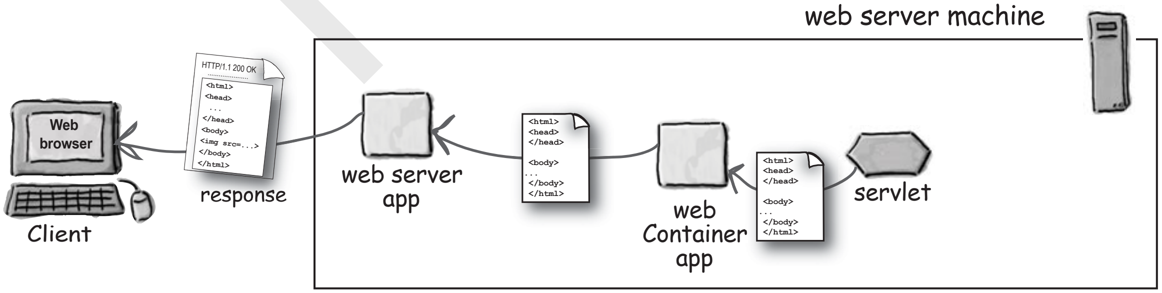
When the user submits data in a form, the web server sees the form data and thinks, "So? Like I care?". To process that form data, either to save it to a file or database or even just to use it to generate the response page, you need another app. When the web server sees a request for a helper app, the web server assumes that parameters are meant for that app. So the web server hands over the parameters, and gives the app a way to generate a response to the client.

What *is* a Container?

Servlets don't have a main() method. They're under the control of another Java application called a *Container*.

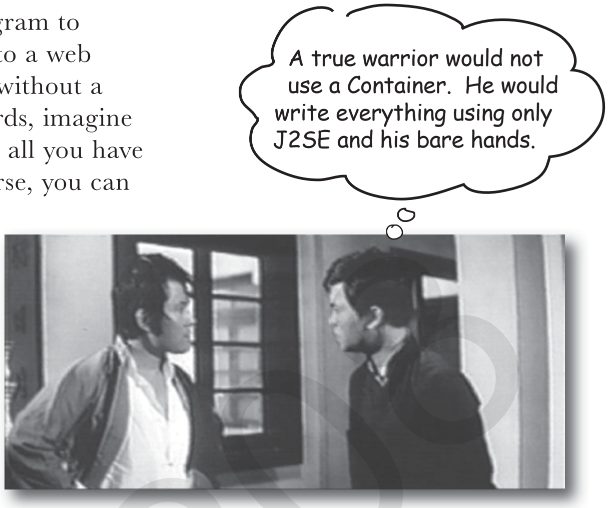
Tomcat is an example of a Container. When your web server application (like Apache) gets a request for a *servlet* (as opposed to, say, a plain old static HTML page), the server hands the request not to the servlet itself, but to the Container in which the servlet is *deployed*. It's the Container that gives the servlet the HTTP request and response, and it's the Container that calls the servlet's methods (like doPost() or doGet()).





What if you had Java, but no servlets or Containers?

What if you had to write a Java program to handle dynamic requests that come to a web server application (like Apache) but without a Container like Tomcat? In other words, imagine there's no such thing as servlets, and all you have are the core J2SE libraries? (Of course, you can assume you have the capability of configuring the web server application so that it can invoke your Java application.) It's OK if you don't yet know much about what the Container does. Just imagine you need server-side support for a web application, and all you have is plain old Java.


A true warrior would not use a Container. He would write everything using only J2SE and his bare hands.

List some of the functions you would have to implement in a J2SE application if no Container existed:

* Create a socket connection with the server, and create a listener for the socket.

Possible answers: create a thread manager, implement security, how about filtering for things like logging, JSP support - yikes, memory management...

What does the Container give you?

We know that it's the Container that manages and runs the servlet, but *why*? Is it worth the extra overhead?

Communications support The container provides an easy way for your servlets to talk to your web server. You don't have to build a ServerSocket, listen on a port, create streams, etc. The Container knows the protocol between the web server and itself, so that your servlet doesn't have to worry about an API between, say, the Apache web server and your own web application code. All you have to worry about is your own business logic that goes in your Servlet (like accepting an order from your online store).

Lifecycle Management The Container controls the life and death of your servlets. It takes care of loading the classes, instantiating and initializing the servlets, invoking the servlet methods, and making servlet instances eligible for garbage collection. With the Container in control, *you* don't have to worry as much about resource management.

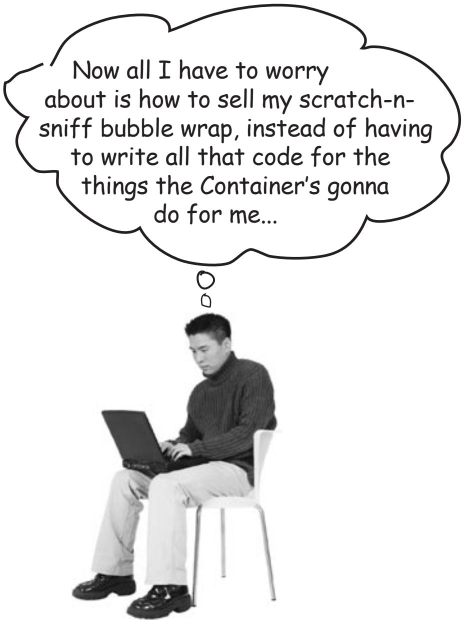
Multithreading Support The Container automatically creates a new Java thread for every servlet request it receives. When the servlet's done running the HTTP service method for that client's request, the thread completes (i.e. dies). This doesn't mean you're off the hook for thread safety—you can still run into synchronization issues. But having the server create and manage threads for multiple requests still saves you a lot of work.

Declarative Security With a Container, you get to use an XML deployment descriptor to configure (and modify) security without having to hard-code it into your servlet (or any other) class code. Think about that! You can manage and change your security without touching and recompiling your Java source files.

JSP Support You already know how cool JSPs are. Well, who do you think takes care of translating that JSP code into real Java? Of course. The *Container*.

Thanks to the Container, YOU get to concentrate more on your own business logic instead of worrying about writing code for threading, security, and networking.

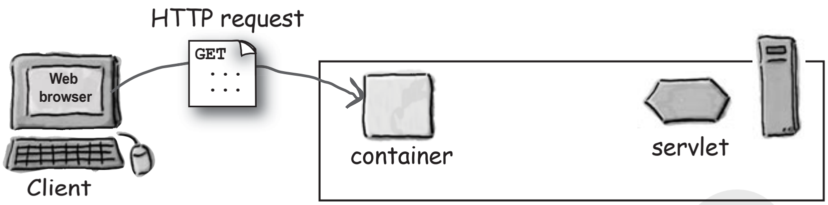
You get to focus all your energy on making a fabulous online bubble wrap store, and leave the underlying services like security and JSP processing up to the container.


Now all I have to worry about is how to sell my scratch-n-sniff bubble wrap, instead of having to write all that code for the things the Container's gonna do for me...

How the Container handles a request

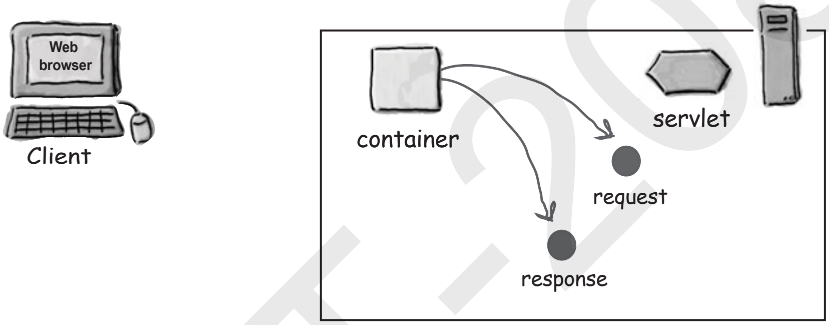
We'll save some of the juicier bits for later in the book, but here's a quick look:

1



User clicks a link that has a URL to a servlet instead of a static page.

2

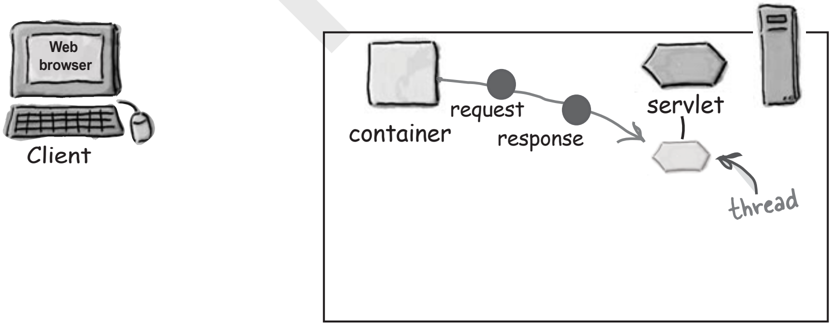


The container "sees" that the request is for a servlet, so the container creates two objects:

1) HttpServletResponse

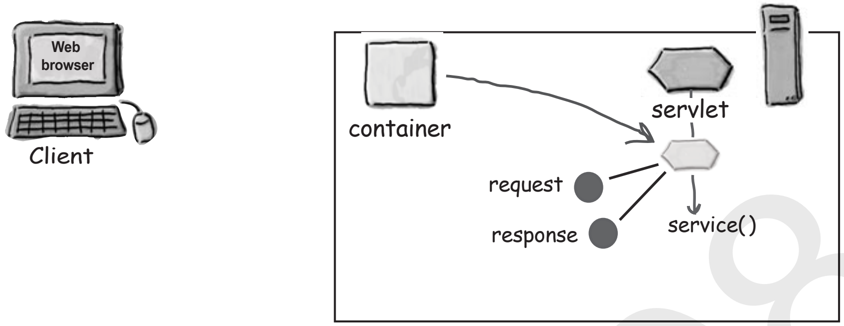
2) HttpServletRequest

3



The container finds the correct servlet based on the URL in the request, creates or allocates a thread for that request, and passes the request and response objects to the servlet thread.

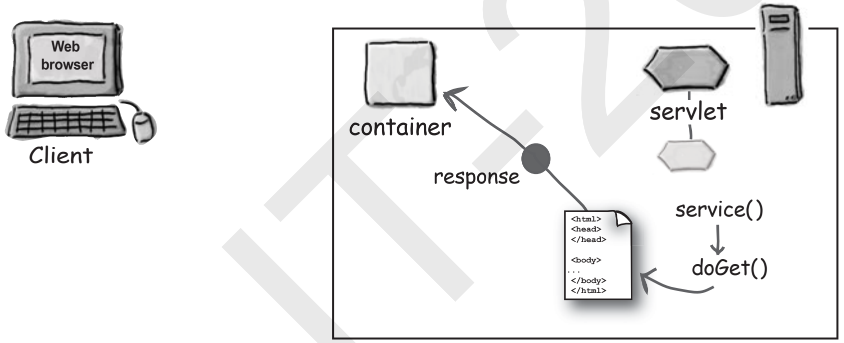
4



The container calls the servlet's service() method. Depending on the type of request, the service() method calls either the doGet() or doPost() method.

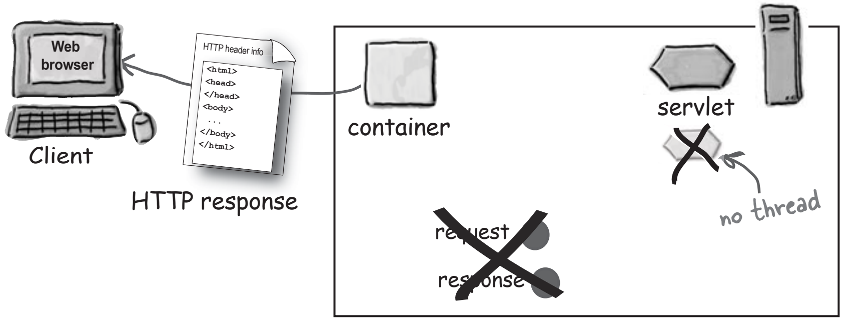
For this example, we'll assume the request was an HTTP GET.

5



The doGet() method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!

6



The thread completes, the container converts the response object into an HTTP response, sends it back to the client, then deletes the request and response objects.

How it looks in code (what makes a servlet a servlet)

In the real world, 99.9% of all servlets override either the doGet() or doPost() method.

Notice... no main() method. The servlet's lifecycle methods (like doGet()) are called by the Container.

99.9999% of all servlets are HttpServlets.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Ch2Servlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                    throws IOException {

    PrintWriter out = response.getWriter();
    java.util.Date today = new java.util.Date();
    out.println("<html> " +
               "<body>" +
               "<h1 style="text-align:center>" +
                "HF\'s Chapter2 Servlet</h1>" +
               "<br>" + today +
               "</body>" +
               "</html>");
  }
}
```

This is where your servlet gets references to the request and response objects which the container creates.

You can get a PrintWriter from the response object your servlet gets from the Container. Use the PrintWriter to write HTML text to the response object. (You can get other output options, besides PrintWriter, for writing, say, a picture instead of HTML text.)

there are no Dumb Questions

Q: I remember seeing *doGet()* and *doPost()*, but on the previous page, you show a *service()* method? Where did the *service()* method come from?

A: Your servlet inherited it from HttpServlet, which inherited it from GenericServlet which inherited it from... ahhh, we'll do class hierarchies to death in the Being a Servlet chapter, so you just need a little more patience.

Q: You wimped out on explaining how the container *found* the correct servlet... like, how does a URL relate to a servlet? Does the user have to type in the exact path and class file name of the servlet?

A: No. Good question, though. But it points to a Really Big Topic (servlet mapping and URL patterns), so we'll take only a quick look on the next few pages, but go into much more detail later in the book (in the Deployment chapter).

You're wondering how the Container found the Servlet...

Somehow, the URL that comes in as part of the request from the client is *mapped* to a specific servlet on the server. This mapping of URLs to servlets might be handled in a number of different ways, and it's one of the most fundamental issues you'll face as a web app developer. The user request must map to a particular servlet, and it's up to you to understand and (usually) *configure* that mapping. What do you think?

FLEX YOUR MIND

How should the Container map servlets to URLs?

The user does *something* in the browser (clicks a link, hits the "Submit" button, enters a URL, etc.) and that *something* is supposed to send the request to a *specific* servlet (or other web app resource like a JSP) you built. How might that happen?

For each of the following approaches, think about the pros and cons.

1. *Hardcode the mapping into your HTML page. In other words, the client is using the exact path and file (class) name of the servlet.*

   PROS:

   CONS:

2. *Use your Container vendor's tool to do the mapping:*

   PROS:

   CONS:

3. *Use some sort of properties table to store the mappings:*

   PROS:

   CONS:

A servlet can have THREE names

A servlet has a *file path name*, obviously, like classes/registration/SignUpServlet.class (a path to an actual class file). The original developer of the servlet class chose the *class* name (and the package name that defines part of the directory structure), and the location on the server defines the full path name. But anyone who deploys the servlet can also give it a special *deployment name*. A deployment name is simply a *secret internal* name that doesn't have to be the same as the class or file name. It *can* be the same as the servlet *class* name (registration.SignUpServlet) or the relative path to the class *file* (classes/registration/SignUpServlet.class), but it can also be something completely different (like *EnrollServlet*).

Finally, the servlet has a *public URL name*—the name the *client* knows about. In other words, the name coded into the HTML so that when the user clicks a link that's supposed to go to that servlet, this public URL name is sent to the server in the HTTP request.

1
I'll click the link to the **"register/registerMe"** servlet.

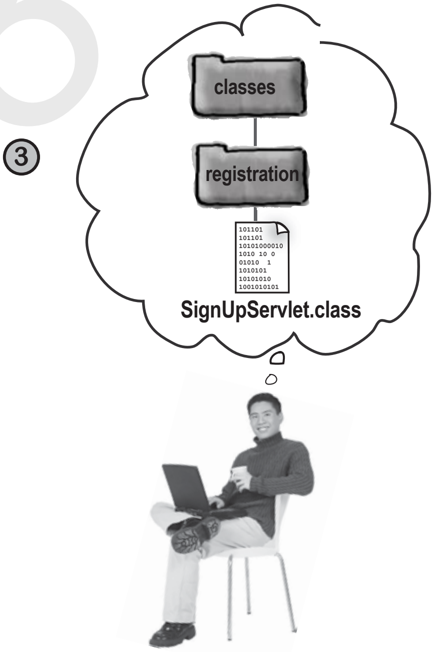
Client-known *URL* name

The client sees a URL for the servlet (in the HTML), but doesn't really know how that servlet name maps to real directories and files back on the server. The public URL name is a fake name, made up for clients.

2
I'm gonna call this servlet the **"EnrollServlet"**.

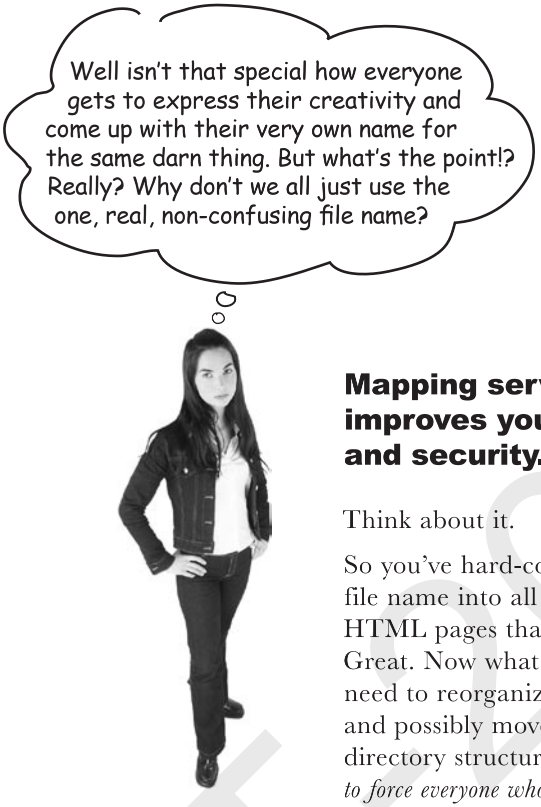
Deployer-known *secret internal* name

The deployer can create a name that's known only to the deployer and others in the real operational environment. This name, too, is a fake name, made up just for the deployment of the servlet. It doesn't have to match the public URL used by the client, OR the real file and path name of the servlet class.



Actual *file* name

The developer's servlet *class* has a fully-qualified name that includes both the class name and the package name. The servlet class *file* has a real path and file name, depending on where the package directory structure lives on the server.


Well isn't that special how everyone gets to express their creativity and come up with their very own name for the same darn thing. But what's the point!? Really? Why don't we all just use the one, real, non-confusing file name?

Mapping servlet names improves your app's flexibility and security.

Think about it.

So you've hard-coded the real path and file name into all the JSPs and other HTML pages that use that servlet. Great. Now what happens when you need to reorganize your application, and possibly move things into different directory structures? *Do you really want to force everyone who uses that servlet to know (and forever follow) that same directory structure?*

By mapping the name instead of coding in the real file and path name, you have the flexibility to move things around without having the maintenance nightmare of tracking down and changing client code that refers to the old location of the servlet files.

And what about security? Do you really want the client to know exactly how things are structured on your server? Do you want them to, say, attempt to navigate directly to the servlet without going through the right pages or forms? Because if the end-user can see the *real* path, she can type it into her browser and try to access it directly.

Using the Deployment Descriptor to map URLs to servlets

When you deploy your servlet into your web Container, you'll create a fairly simple XML document called the Deployment Descriptor (DD) to tell the Container how to run your servlets and JSPs. Although you'll use the DD for more than just mapping names, you'll use two XML elements to map URLs to servlets—one to map the client-known *public URL* name to your own *internal* name, and the other to map your own *internal* name to a fully-qualified *class name.*

The two DD elements for URL mapping:

1. **<servlet>**
   maps internal name to fully-qualified class name

2. **<servlet-mapping>**
   maps internal name to public URL name

This web app has two servlets.

There is a LOT more that goes into this opening <web-app> tag, but we don't want to show it right now (there's an example at the end of this chapter).

The <servlet-name> element is used to tie a <servlet> element to a specific <servlet-mapping> element. The end-user NEVER sees this name; it's used only in other parts of the DD.

The <servlet> element tells the Container which class files belong to a particular web application.

You put in the fully-qualified name of the class (but you don't add the ".class" extension).

```
<web-app ...>

  <servlet>
    <servlet-name>Internal name 1</servlet-name>
    <servlet-class>foo.Servlet1</servlet-class>
  </servlet>

  <servlet>
    <servlet-name>Internal name 2</servlet-name>
    <servlet-class>foo.Servlet2</servlet-class>
  </servlet>
```

```
  <servlet-mapping>
    <servlet-name>Internal name 1</servlet-name>
    <url-pattern>/Public1</url-pattern>
  </servlet-mapping>


 <servlet-mapping>
    <servlet-name>Internal name 2</servlet-name>
    <url-pattern>/Public2</url-pattern>
 </servlet-mapping>

</web-app>
```

Think of the <servlet-mapping> element as what the Container uses at runtime when a request comes in, to ask, "which servlet should I invoke for this requested URL?".

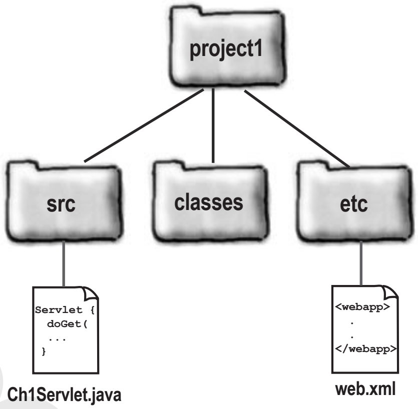
This is what the client sees (and uses) to get to the servlet... but it's a made-up name that is NOT the name of the actual servlet class.

It's possible to use wildcards in the <url-pattern> element... more on that and paths later.

Servlets Demystified (write, deploy, run)

Just so those new to servlets can stop holding their breath, here’s a quick guide to writing, deploying, and running a servlet. This might create more questions than it answers—***don’t panic***, you don’t have to *do* this right now. It’s just a quick demonstration for those who can’t wait. The next chapter includes a more thorough tutorial.

1. Build this directory tree (somewhere *not* under tomcat).



2. Write a servlet named Ch1Servlet.java and put it in the *src* directory (to keep this example simple, we aren’t putting the servlet in a package, but after this, all other servlet examples in the book will be in packages).

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Ch1Servlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws IOException {

    PrintWriter out = response.getWriter();
    java.util.Date today = new java.util.Date();
    out.println(“<html> “ +
              “<body>” +
              “<h1 align=center>HF\’s Chapter1 Servlet</h1>”
              + “<br>” + today +  “</body>” + “</html>”);
  }
}
```

Standard servlet declarations (there will be about 400 pages describing this stuff).

HTML embedded in a Java program. Looks lovely, doesn’t it?

3. Create a deployment descriptor (DD) named web.xml, put it in the *etc* directory

```
<?xml version=”1.0” encoding=”ISO-8851-1” ?>
<web-app xmlns=”http://java.sun.com/xml/ns/j2ee”
     xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
     xsi:schemaLocation=”http://java.sun.com/xml/ns/j2ee
     http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd”
    version=”2.4”>
   <servlet>
     <servlet-name>Chapter1 Servlet</servlet-name>
     <servlet-class>Ch1Servlet</servlet-class>
   </servlet>

   <servlet-mapping>
     <servlet-name>Chapter1 Servlet</servlet-name>
     <url-pattern>/Serv1</url-pattern>
   </servlet-mapping>
</web-app>
```

Highlights:

-One DD per web application.

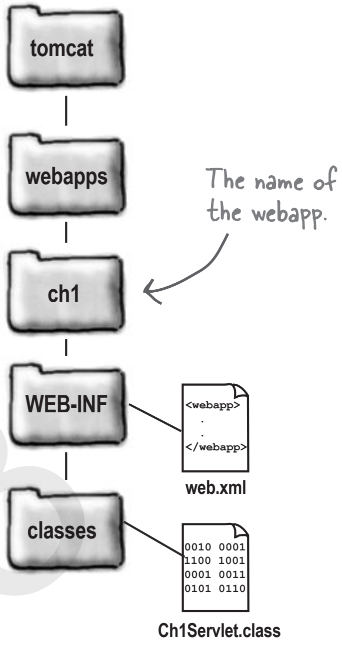
-A DD can declare many servlets.

- A <servlet-name> ties the <servlet> element to the <servlet-mapping> element.

- A <servlet-class> is the Java class.

- A <url-pattern> is the name the client uses for the request.

4 Build this directory tree under the existing *tomcat* directory...



The name of the webapp.

5 From the *project1* directory, compile the servlet...

```
%javac -classpath /your path/tomcat/common/lib/
servlet-api.jar -d classes src/Ch1Servlet.java
```

(This is all one command.)

(the Ch1Servlet.class file will end up in *project1/classes*)

6 Copy the Ch1Servlet.class file to *WEB-INF/classes*, and copy the web.xml file to *WEB-INF.*

7 From the *tomcat* directory, start Tomcat...

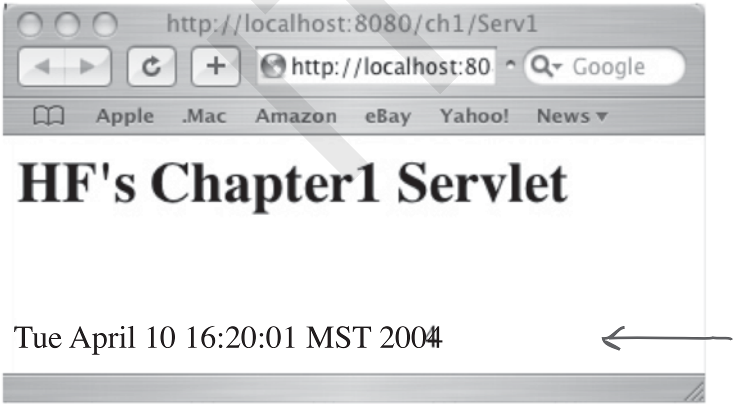
```
%bin/startup.sh
```

8 Launch your browser and type in:

```
http://localhost:8080/ch1/Serv1
```

The webapp is named 'ch1' and the servlet is named 'Serv1'.

it should display:



HF's Chapter1 Servlet

Tue April 10 16:20:01 MST 2004

Your date may vary...

9 For now, every time you update either a servlet class or the deployment descriptor, shutdown Tomcat:

```
%bin/shutdown.sh
```