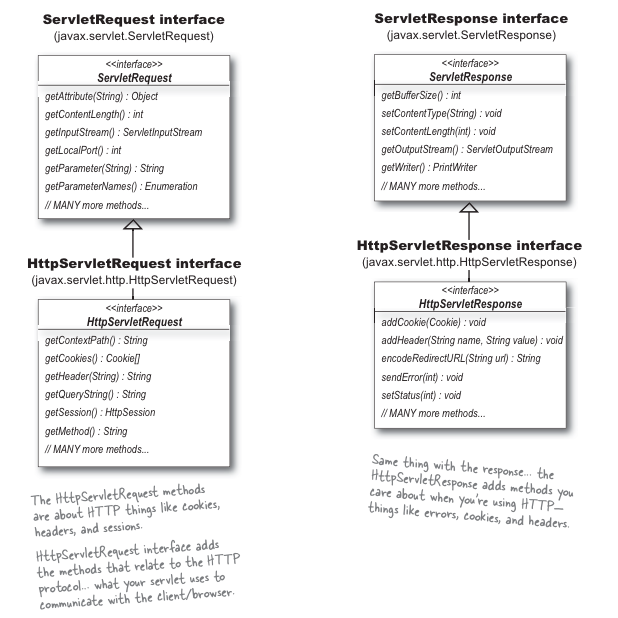
**Chapter 7**

A Servlet's real job is to handle requests and provide appropiate response. We already know that we are handed a request and response as arguments to the doGet()and doPost() method, but what powers do those request and response objects give you? What can you do with them and Why do you care? We will try to answer these questions in this chapter.

We can say that request and response are the key to everything and the arguments to service() method. Let's check the class hierarchy of request and response objects.



Need to ponder that HttpServletRequest and HttpServletResponse both are interfaces, then who implements these interfaces? Are those classes in the API?

These interfaces are implemented by The Container(Tomcat in our case) and NO, the classes are n't in the API because they are left to the vendor to implement. The good news is we don't have to worry about it. All we need to know are the methods we can call on the objects The Container gives us as part of request.

The client’s request, remember, always includes a specific HTTP Method. If the HTTP Method is a GET, the service() method calls doGet(). If the HTTP request Method is a POST, the service() method calls doPost().

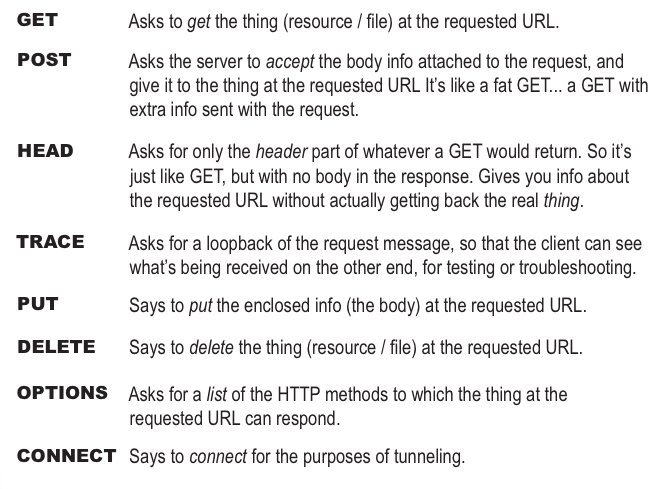
Yes, there *are* other HTTP Methods besides GET and POST. There’s also HEAD, TRACE, OPTIONS, PUT, DELETE, and CONNECT.

All but one of the eight has a matching doXXX() method in the HttpServlet class, so besides doGet() and doPost(), you’ve got doOptions(), doHead(), doTrace(), doPut(), and doDelete(). There’s no mechanism in the servlet API for handling doConnect(), so it’s not part of HttpServlet.

But while the other HTTP Methods might matter to, say, a web serv*er* developer, a servl*et* developer rarely uses anything but GET and POST.

For most (or probably *all*) servlet development, you’ll use either doGet() (for simple requests) or doPost() (to accept and process form data), and you won’t have to think about the others.

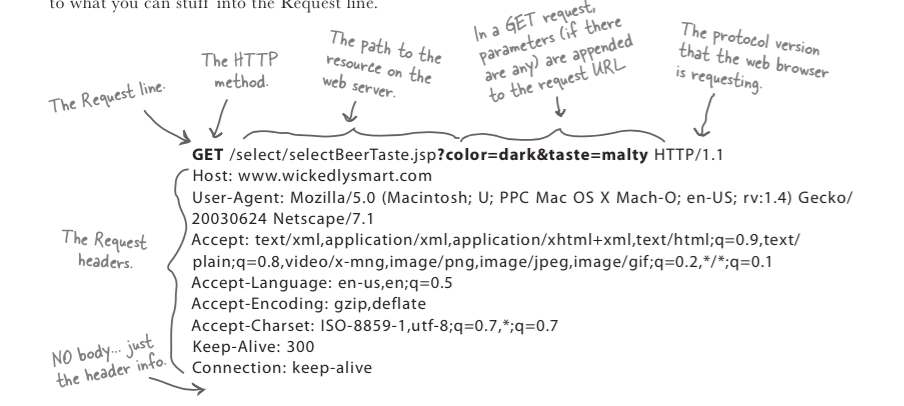
But Just for formality brief description is mentioned below:

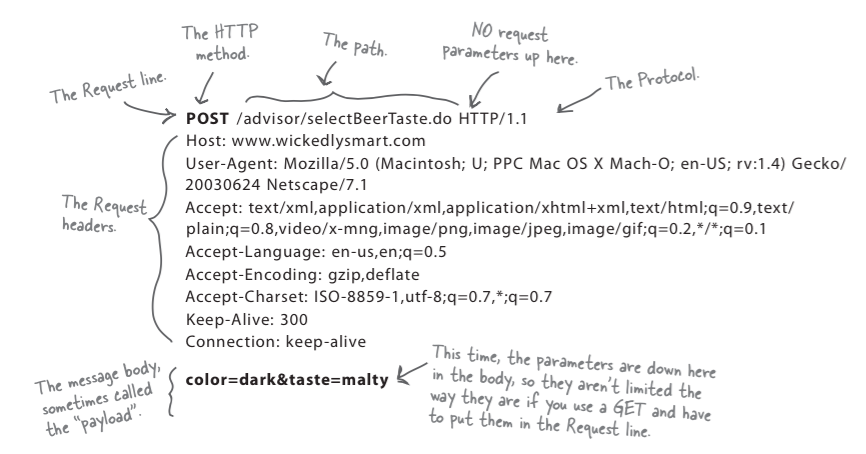


**Let's check out the difference between GET and POST**

*POST has a body.* That’s the key. Both GET and POST can send parameters, but with GET, the parameter data is limited to what you can stuff into the Request line.

Mentioned below are the structure of GET and POST request, which will help us to identify the differences between the two.

****

****

Seems like the difference between GET and POST is the size of the parameter data you can send!!!

NO, it's not just about the size, in GET the parameter data shows up in the browser’s input bar, right after actual URL (and separated with a “?”). Imagine a scenario in which you would not want the parameters to be visible.

Like : http://localhost:8080/MyApp/myServlet.do?name=nishant&password=nishant

So, security might be another issue.

Still another issue is whether you need or want end-users to be able to bookmark the request page. GET requests can be bookmarked; POST requests cannot. That might be really important if you have, say, a page that lets users specify search criteria. The users might want to come back a week later and try the same search again now that there’s new data on the server.

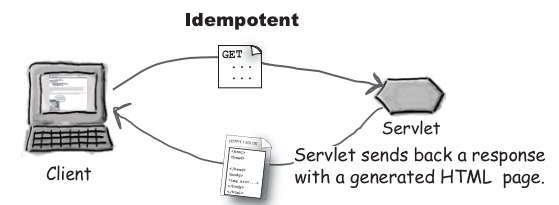
But *besides* size, security, and bookmarking, there’s another crucial difference between GET and POST—the way they’re *supposed* to be used. GET is meant to be used for *getting* things. Period. Simple retrieval. Sure, you might use the parameters to help figure out what to send back, but the point is—you’re not making any changes on the server! POST is meant to be used for *sending data to be processed*. This could be as simple as query parameters used to figure out what to send back, just as with a GET, but when you think of POST, think: *update*. Think: use the data from the POST body to *change something on the server*.

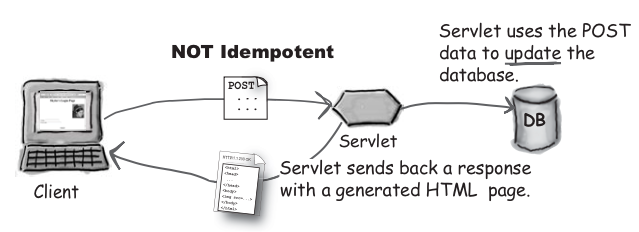
And that brings up another issue... whether the request is ***idempotent or not***.

**Idempotent or Non-Idempotent:**

An HTTP GET is just for getting things and it's not supposed to change anything on the server. So a GET is by definition or according to the HTTP spec Idempotent. It can be executed more than once without any bad side effects. Actually being idempotent is GOOD, it means we can do the same thing over and over again, with no unwanted side effects!.

POST is not Idempotent, the data submitted in the body of a POST might be destined for a transaction that can't be reversed. So we have to be careful with doPost() method functionality!.



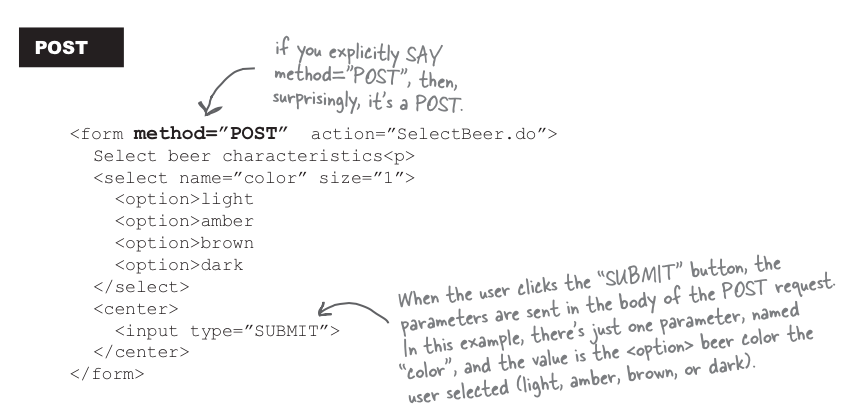


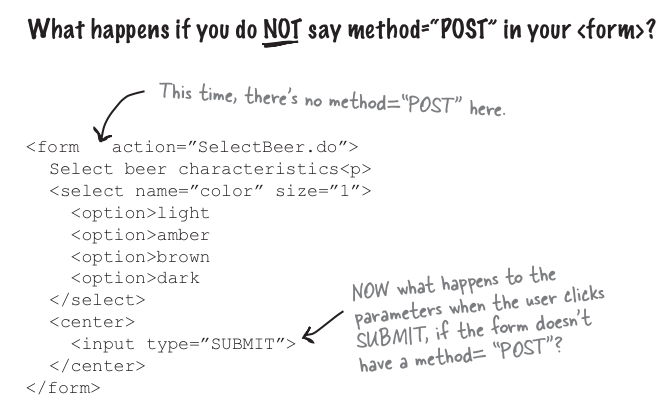
NOTE: There are several different uses of the word “Idempotent”, we are using it in the HTTP/servlet way to mean that the same request can be made twice with no negative consequences on the server. We do not use “Idempotent” to mean that the same request always returns the same response and we do not mean that a request has No side effects.

But in effect you can use, GET for making an update to the server, but it's not the suitable design principle to use it.

**What determines whether the browser sends a GET or POST request?**





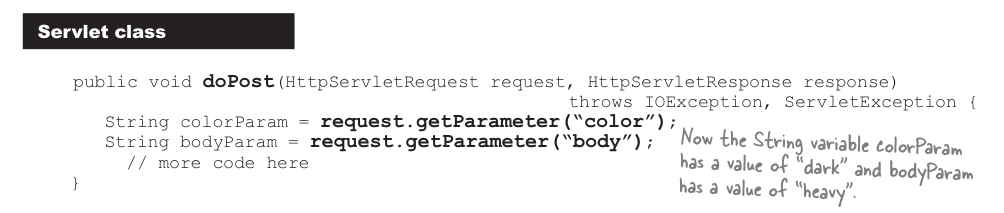


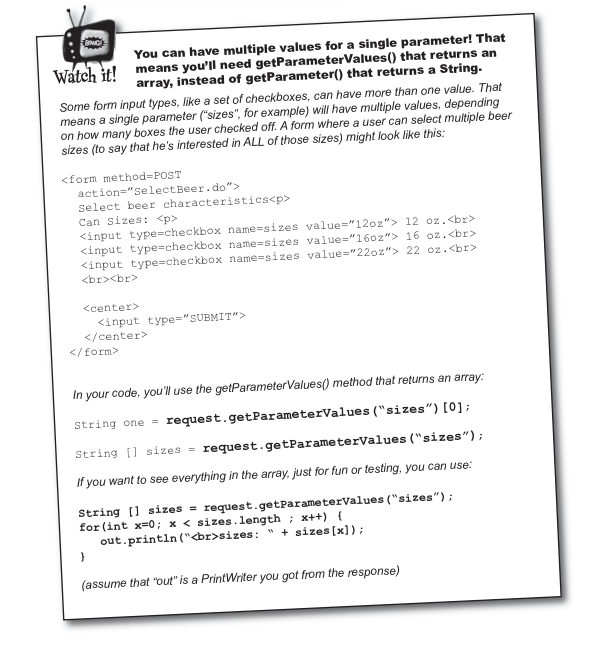
If we don't put method = “POST” into your form, the default is an HTTP GET request. That means the browser sends the parameters in the request url, but that's the least of your problems. Because if the request comes in as a GET, that means you'll run into big trouble at runtime if you have only doPost() method and not a doGet() method in your servlet!!.

Then we will get an error stating “HTTP method GET is not supported by this URL”.

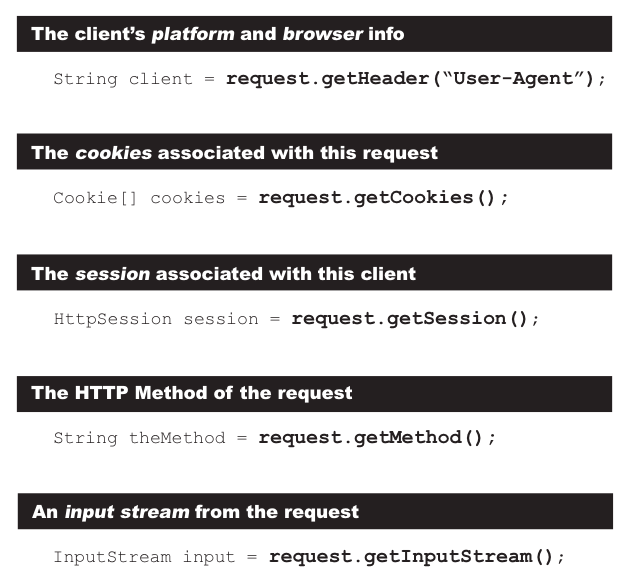
After submitting/posting data by form, we can get that parameters value by below mentioned code







The ServletRequest and HttpServletRequest interfaces have a ton of methods we can call, but you don't need to memorize them all. On your own, you really should look at the full API for javax.servlet.ServletRequest and javax.servlet.http.HttpServletRequest, but here we'll look at only the methods you're most likely to use in your work.



So that's the Request, **now let's see the Response**

The response is what goes back to the client. The thing the browser gets, parses, and renders for the user. Typically, you use the response object to get an output stream (usually a Writer) and you use that stream to write the HTML (or some other type of content) that goes back to the client. The response object has other methods besides just the I/O output, though, and we’ll look at some of them in a bit more detail.

Most of the time, you use the Response just to send data back to client. We can call two methods on the response: **setContentType() and getWriter()**. But we can also use the response to set other headres, send errors and add cookies.

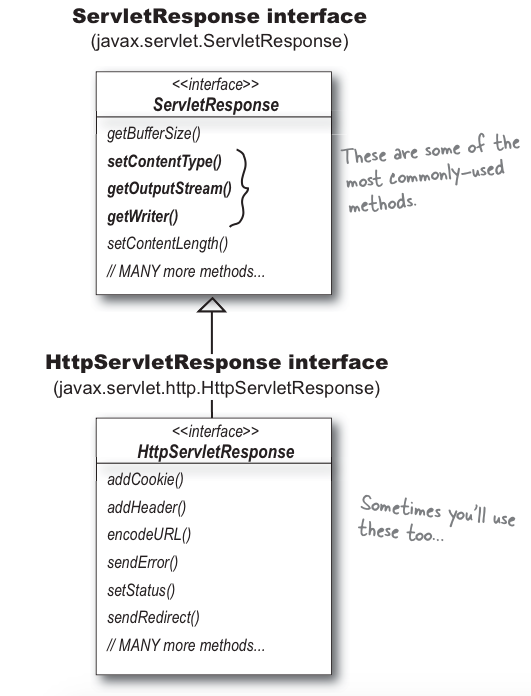
WAIT a minute, we thought we were n't going to send HTML from servlet, because it's so ugly to format it for the output stream...!!

Yes, we should use JSPs rather than sending HTML back in the response output stream from a servlet, but that does n't mean we will never have to work with an output stream from our servlet.

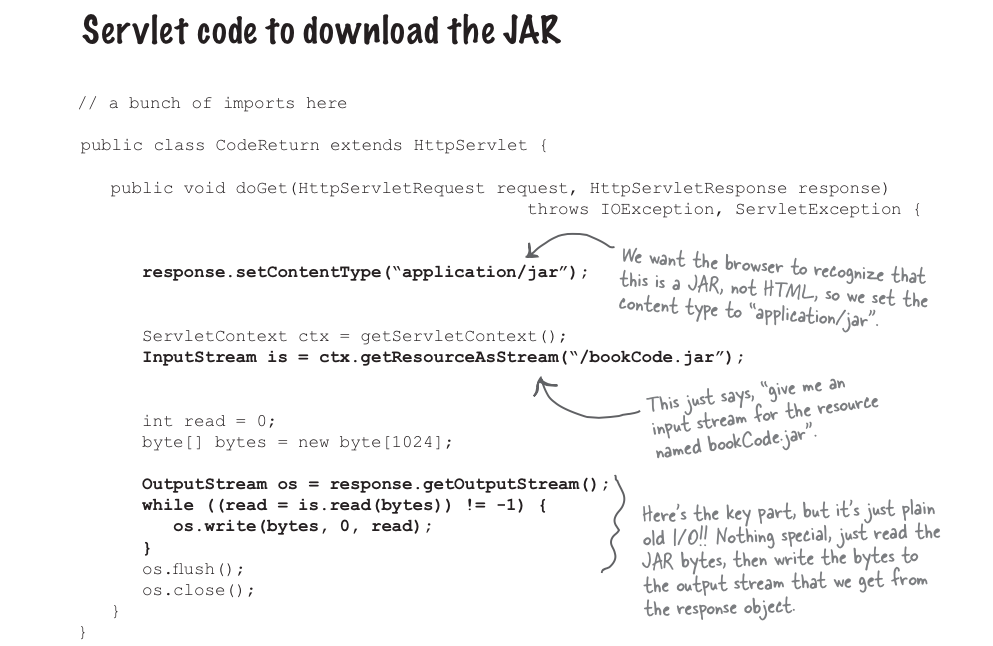
For some basic reasons we need to work with the servlet's output stream

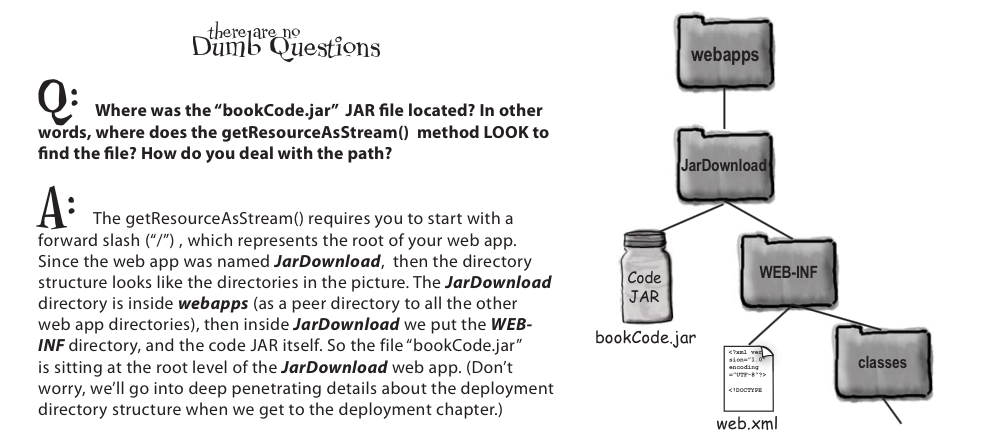
1. Your hosting provider might not support JSPs. There are plenty of older servers and containers out there that support servlets but not JSPs, so you’re stuck with it.

2. Who said that *HTML* was the only thing you could send back in a response? You might send something *other* than HTML back to the client. Something for which an output stream makes perfect sense.



Let’s say you’ve created a download page where the client can get code from JAR files. Instead of sending back an HTML page, the response contains the bytes representing the JAR. You *read* the bytes of the JAR file, then *write* them to the response’s output stream.





So, what's the deal with 'content type'??

response.setContentType(“application/jar”);

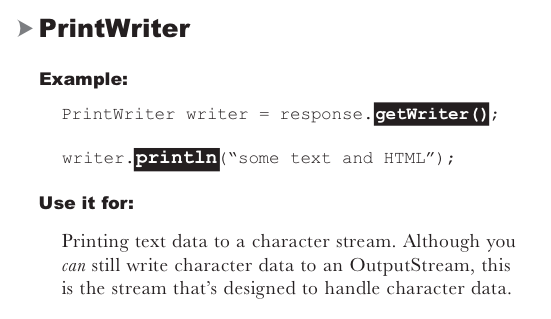
You have to tell the browser what you’re sending back, so the browser can ***do the right thing:*** launch a “helper” app like a PDF viewer or video player, render the HTML, save the bytes of the response as a downloaded file, etc. And since you’re wondering, yes when we say *content type* we mean the same thing as MIME type. Content type is an HTTP header that *must* be included in the HTTP response.

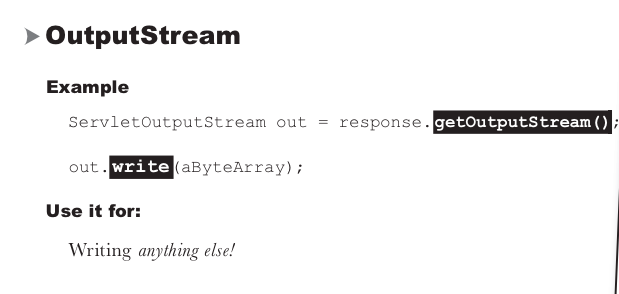
Question!!! ???

Why do we have to set the content type, can't server figure it out from the extension of the file??

Most servers *can*, for static content. In Apache, for example, you can set up MIME types by mapping a specific file extension (.txt, .jar, etc.) to a specific content type, and Apache will use that to set the content type in the HTTP header. But we’re talking about what happens inside a servlet where there IS no file! You’re the one who is sending back the response; the Container has no idea what you’re sending.

Okkkk!!! So I think we have got two choices for output!! characters or bytes





FYI: The PrintWriter actually “wraps” the ServletOutputStream. In other words, the PrintWriter has a reference to the ServletOutputStream and delegates calls to it. There’s just ONE output stream back to the client, but the PrintWriter “decorates” the stream by adding higher-level character-friendly methods.

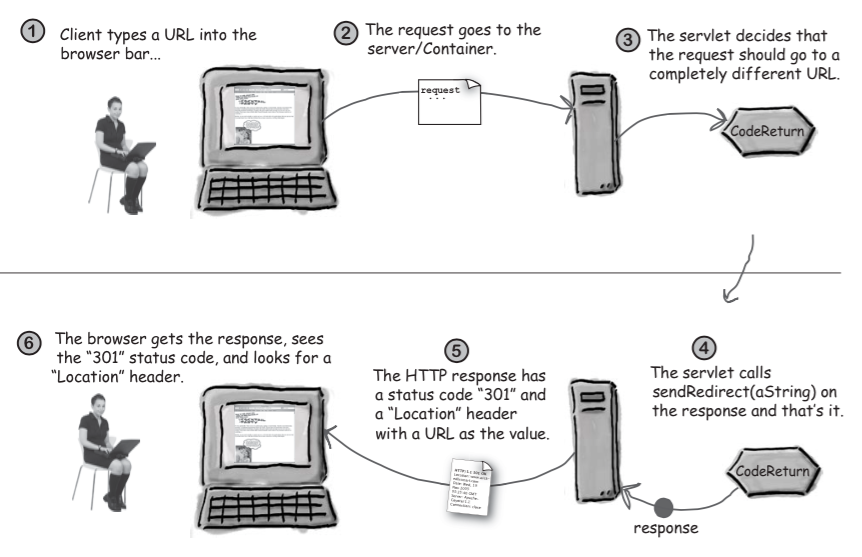
Now what about the headers in the response, actually “We can set response headers” of “We can add response headers” !!!!! don't get confuse with this, setHeader() overwrites the existing value and addHeader() adds an additional value.

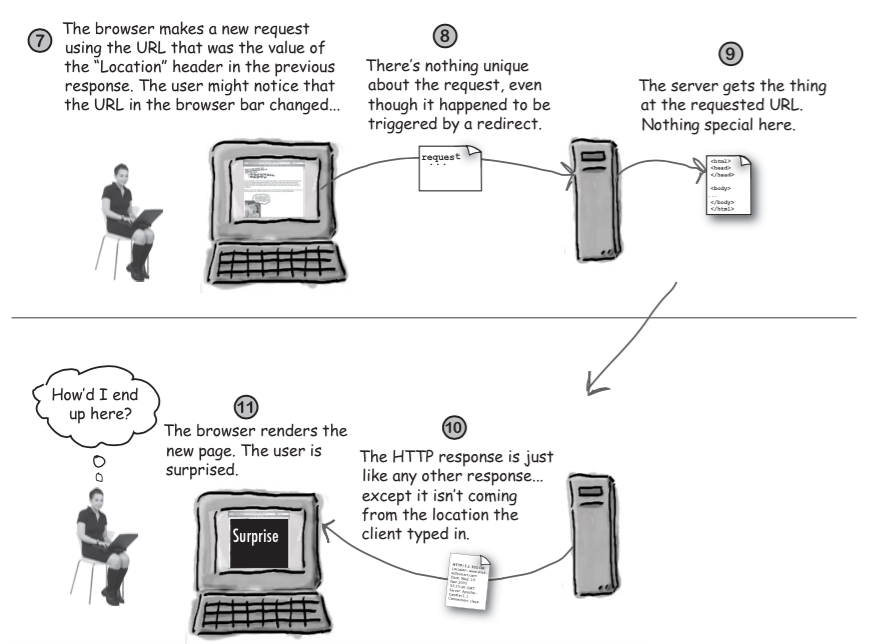
When you call setContentType(“text/html”), you’re setting a header just as if you said: setHeader(“content-type”, “text/html”);

So what’s the difference? No difference... *assuming* *you* *type* *the* “*content-type” header correctly.* The setHeader() method won’t complain if you misspell the header names—it just thinks you’re adding a new kind of header. But something else will fail later, because now you haven’t properly set the content type of the response!

**But some times we just don't want to deal with the response yourself**... You can choose to have something else handle the response for your request. You can either *redirect* the request to a completely different URL, or you can *dispatch the request* to some other component in your web app (typically a JSP).

**Redirect:**

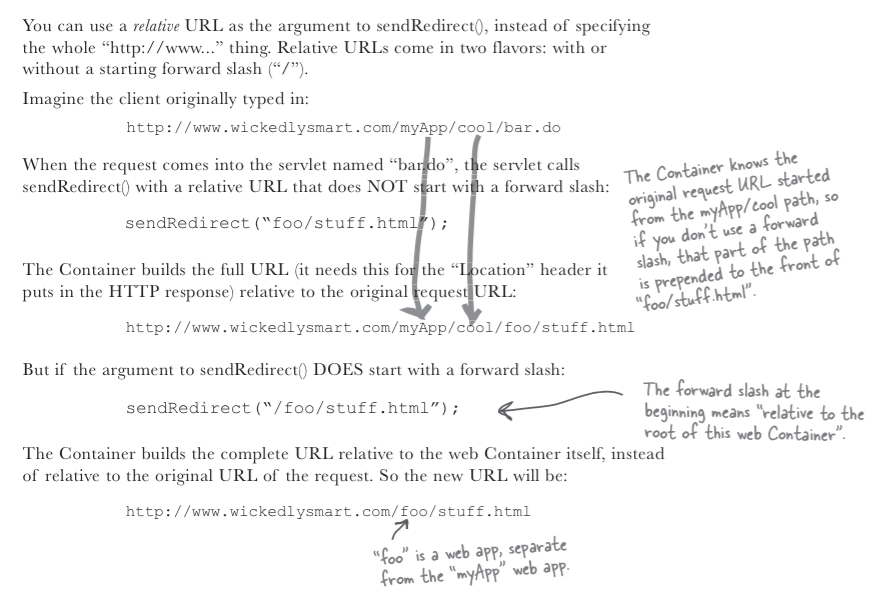




Servlet redirect makes the browser to do the work. A redirect lets the servlet off the hook completely. After deciding that it can’t do the work, the servlet simply calls the *sendRedirect()* method:

if (worksForMe) {  
// handle the request } else { response.sendRedirect(“http://www.oreilly.com”); }

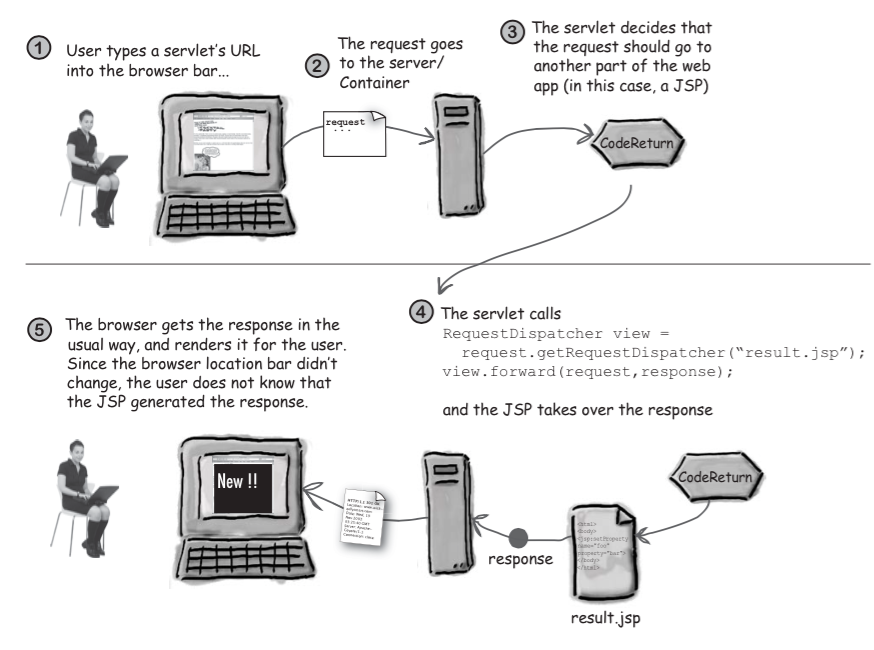
**Using relative URLs in sendRedirect() method:**

**Note :** we can't do a sendRedirect() after writing to the response or for practical purposes it means we can;t write to the response and then call sendRedirect();

**Request Dispatch**

RequestDispatcher rd = request.getRequestDispatcher(“index.jsp”);

rd.forward(request,response);

As of now, just look over “forward” technique we will cover “include” in our later chapters

And that’s the big difference between a redirect and a request dispatch—*redirect* makes the ***client***do the work while *request dispatch* makes something else on the ***server***do the work. So remember: **redirect = *client*, request dispatch = *server***. We’ll say more about request dispatch in a later chapter, but these two pages should give you a quick look at the highlights.

