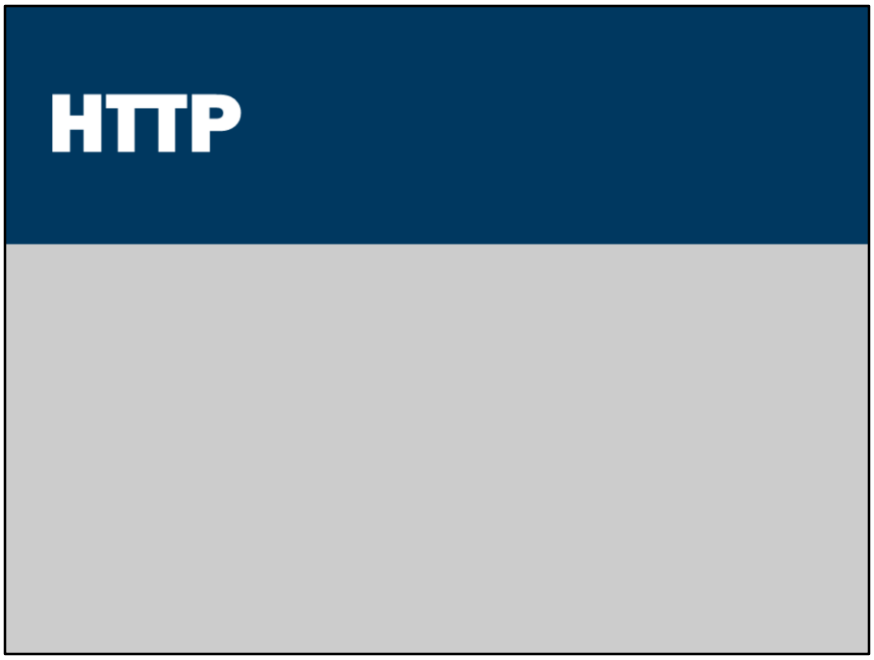
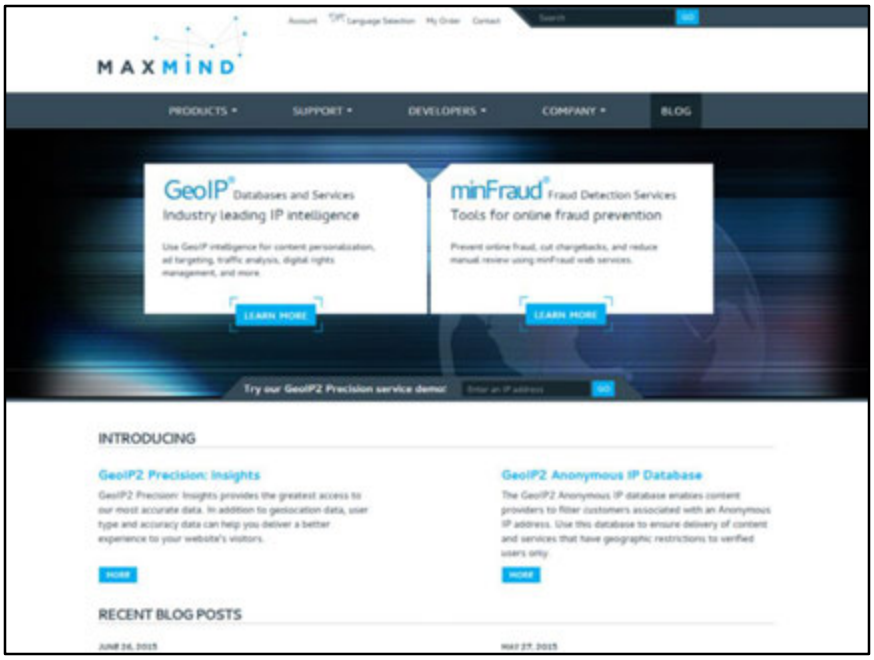


This is a quick refresher on the HTTP protocol.

An excellent explanation of how web browsers work:
http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/

See also:
http://grosskurth.ca/papers/browser-refarch.pdf
http://grosskurth.ca/papers/browser-archevol-20060619.pdf



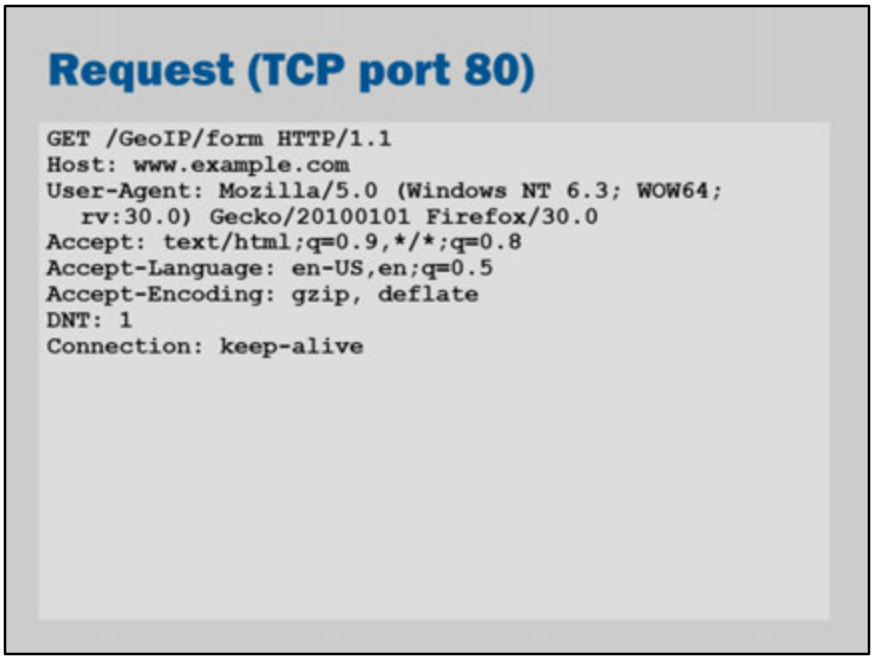
In the Internet Protocol (IP), there is no fundamental mapping between IP address and geographic places. IP addresses are allocated by regional internet registries (ARIN, APNIC, AFRINIC, LACNIC and RIPE) so there is some degree of high level locality. However, routing on the internet does not require sites to know the geographic location of other sites – only which network is the best 'next hop'. IP addresses belong to networks (which are free to assign IP addresses as they please) and the appropriate routing information is communicated using routing protocols such as BGP (Border Gateway Protocol).

Even though IP addresses are not required to map to physical locations, there tends to be a fairly static connection between IP addresses (and IP address blocks) and a physical location.

Internet businesses often need to know the location of an IP address (fraud detection, pricing, redirection to a regional office, language selection, copyright restrictions, laws). Companies such as MaxMind provide services that assist these businesses by mapping from IP addresses onto approximate locations.

Here's an example of using MaxMind's GeoIP database to lookup the location of an IP address.

I've used the Maxmind database to put together a very simple IP address lookup tool that I'll use in the next few slides.



The first two lines in this request are the most “important”.
The first one tells the server how to handle the request. The second one tells the server the name of the website.

More details:

Web pages are retrieved using Hyper Text Transfer Protocol (HTTP). HTTP 1.1 was introduced in RFC2068. The specification was revised in RFC2616:
http://tools.ietf.org/html/rfc2616
In 2014, a comprehensive update of the HTTP 1.1 to improve clarity and eliminate ambiguity was completed in RFCs 7230 to 7235:
http://tools.ietf.org/html/rfc7230
http://tools.ietf.org/html/rfc7231
http://tools.ietf.org/html/rfc7232
http://tools.ietf.org/html/rfc7233
http://tools.ietf.org/html/rfc7234
http://tools.ietf.org/html/rfc7235

HTTP 1.1 requests have the following syntax:

A request line:

Method SP Request-URI SP Version CRLF

Followed by any number of header fields:

Field-Name ":" OWS Field-value OWS CRLF

Followed by a blank line:

CRLF

Followed by the content of the request, if any.

Note that SP = space (0x20), CR = carriage return (0x0D), LF = line feed, (0x0A), OWS = optional white space (0x20 or 0x09).

HTTP 1.1 requires the "Host" header be present. It gives the name of the server (allowing multiple servers on a single IP address). HTTP 1.0 has the same syntax but does not require a "Host" header.

Response (same connection)

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open
   Source Edition  4.0  Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition  4.0
Content-Type: text/html;charset=UTF-8
Date: Sat, 12 Jul 2014 06:32:55 GMT
Content-Length: 231

<!DOCTYPE html>
<html>
  <head>
    <title>IP Address Lookup</title>
  </head>
  <body>
    <h1>IP Address Lookup</h1>
    <form action="lookup" method="POST">
      <input type="text" name="ip">
      <input type="submit">
    </form>
  </body>
</html>
```

The response follows the same syntax as a header, except the first line is the status line (i.e., whether the request was successful):

Version SP Status-code SP Human-readable-reason-phrase CRLF

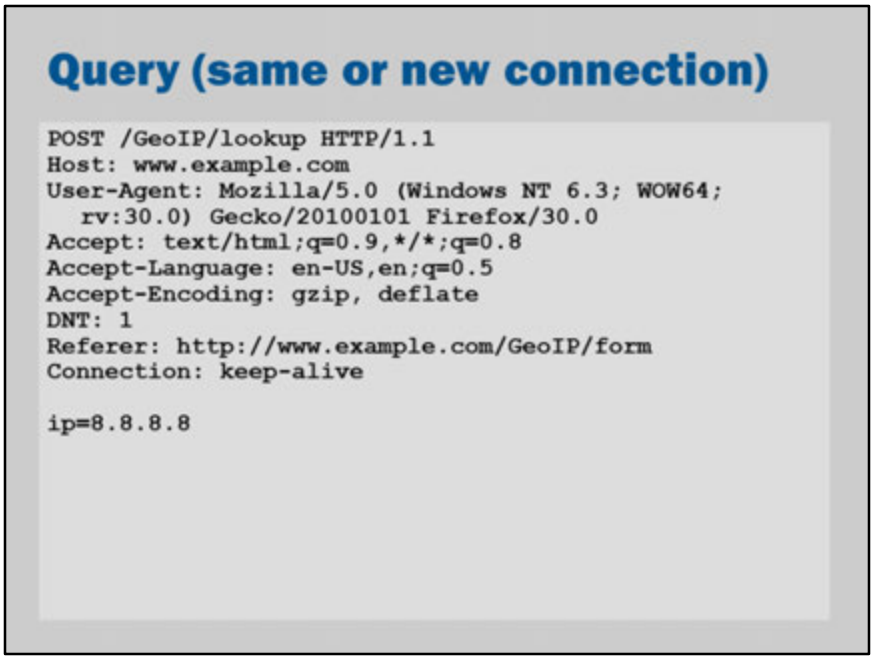
200 indicates a successful response.

Numbers starting with 4 are errors. For example, 404 is probably the most famous response code: the page is not found.

The payload (the body of the response) is the HTML document.


IP Address Lookup
www.example.com/GeoIP/form
Search
IP Address Lookup
Submit Query

Now, suppose that the user enters an IP address, then what does the browser post back to the server?…



In a GET request, all information is encoded in the URL.

In a POST request, field values are encoded into the content of the HTTP request message.

This is a POST request, so the field values are encoded in the body/payload of the HTTP request (ip=8.8.8.8).

How do we know it is a POST request? Two reasons:
1. The first line of the request says POST
2. The HTML on the previous page specified that the request is to be made using POST (<form action="lookup" method="**POST**">)

Challenge Question: What would this request have looked like if it was a GET request (i.e., on the previous page we had (<form action="lookup" method="**GET**">)?

Answer: The first line would have been:
GET /GeoIP/lookup?ip=8.8.8.8 HTTP/1.1

Response (same connection)

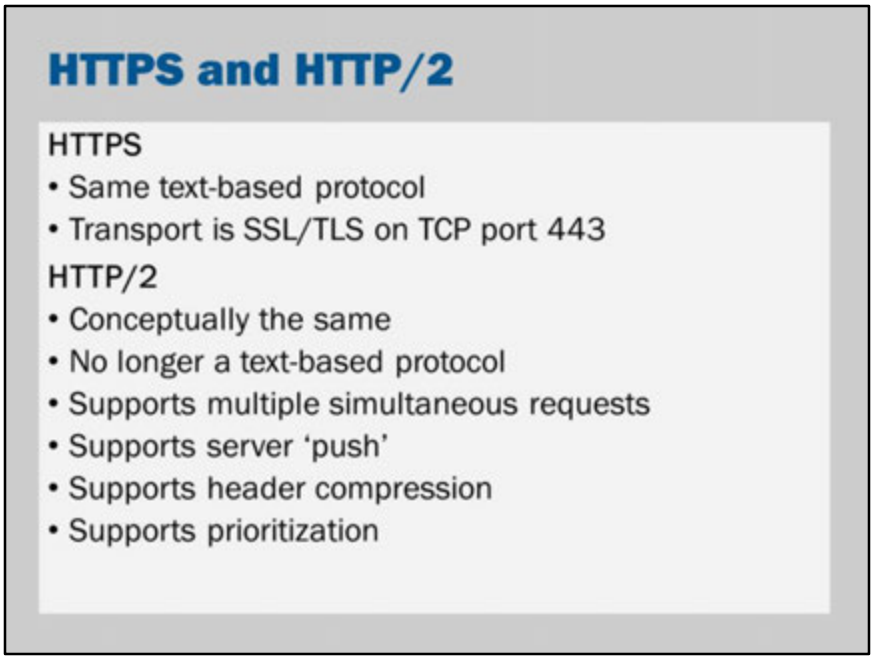
```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server
  Open Source Edition  4.0  Java/Oracle
  Corporation/1.7)
Server: GlassFish Server Open Source Edition  4.0
Content-Type: text/html;charset=UTF-8
Date: Sat, 12 Jul 2014 06:36:45 GMT
Content-Length: 179

<!DOCTYPE html>
<html>
  <head>
    <title>IP Address Lookup</title>
  </head>
  <body>
    <h1>IP Address Lookup</h1>
    <p>
      8.8.8.8 is located in: United States
    </p>
  </body>
</html>
```

This completes our quick recap of HTTP.

However, it brings us to the question: what are the ways that we could generate such dynamic content?

Do we need to write our own program that can listen for HTTP connections, that processes the request and analyzes the headers? Doing it from scratch can be a lot of work, especially if you want to build a system that is secure, fast, scalable, standards compliant, extensible and well designed.

This is what the response looks like when it is rendered in the browser.



HTTPS is how web-pages are accessed securely. HTTP/2 is the latest version of the HTTP protocol, that provides a whole range of new performance improvements (it is similar to Google's SPDY protcol)

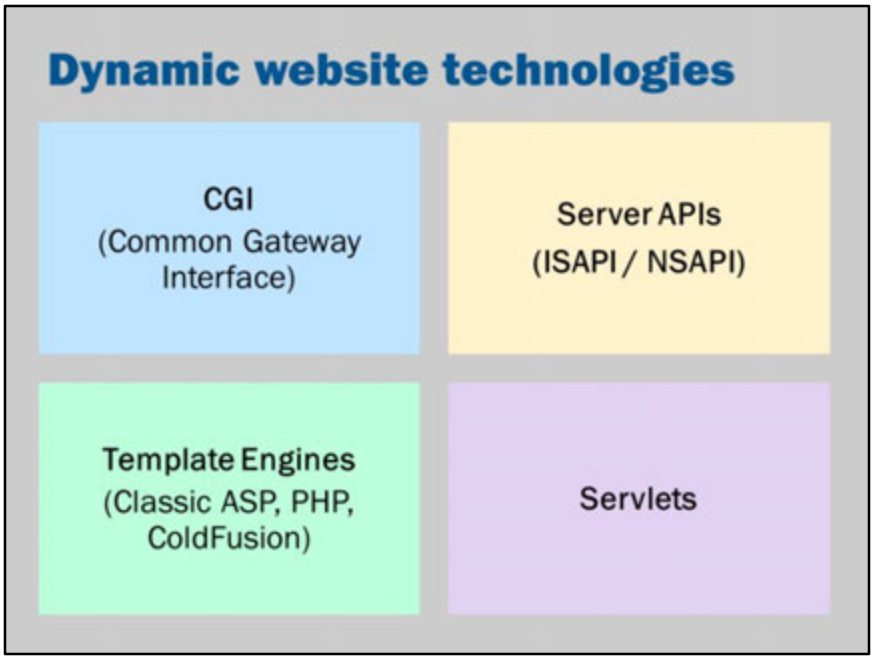
Conceptually, HTTPS and HTTP/2 work the same as HTTP. They have the same HTTP methods, the same use of headers and message body. The difference lies in how the data is transmitted between the browser and the server.

HTTPS uses an encrypted connection over TCP, rather than just plain TCP sockets. This encrypted connection uses a protocol called TLS.
The full specifications are here:
https://tools.ietf.org/html/rfc5246

The HTTP/2.0 specification is online here:
https://tools.ietf.org/html/rfc7540

You can see a demo of the performance benefits of HTTP/2 here:
https://http2.akamai.com/demo

Servlets



There are a number of approaches that have been designed to help simplify the task of building dynamic websites.

The basic idea is that a web server manages all the complexity of dealing with HTTP. Then there are a variety of technologies that "plug in" to provide the dynamic content.

CGI:
This is the oldest technology.
The web-server calls a separate program and uses its input/output to process requests.

A "CGI script" is an ordinary program in the operating system that produces text on standard output.

The CGI-compliant web server listens for incoming requests. When a request is received, the web server will execute the program. The details of the request (e.g., HTTP headers) are given to the program using the operating system's "environment

variables".

Any data in POST or PUT requests is delivered to the program using its standard input.

The CGI program responds on standard output with a partial HTTP response. The web server can then add additional headers (if appropriate) and returns the data to the client via the network connection.

Advantages:
- Very simple
- Can re-use existing command-line programs

Disadvantages:
- Resource intensive (processes tend to be costly in operating systems)
- Program-centric (as opposed to page-centric)

Server APIs:
- ISAPI (Internet Server Application Programming Interface)
- NSAPI (Netscape Server Application Programming Interface)
- Apache module API

Allows the server to be extended via a C/C++ API.

Advantages:
- Very efficient (native code running as part of the server)

Disadvantages:
- Complex to code
- Can crash the web-server
- Difficult to debug

Template Engines using Marked-up HTML:
The server interprets the file and executes operations.
This functionality be implemented in the server using Server APIs (e.g., mod_php on Apache).

Advantages:
- Page-centric
- Cross-platform portable (depending on the language)

Disadvantages:
- Poor performance (code is interpreted, rather than compiled)
- Difficult to debug

Servlets:
A Java technology for dynamic content. The Servlet is run “inside” the web server. If the web server is implemented in Java, it may be run inside the same JVM as the server.

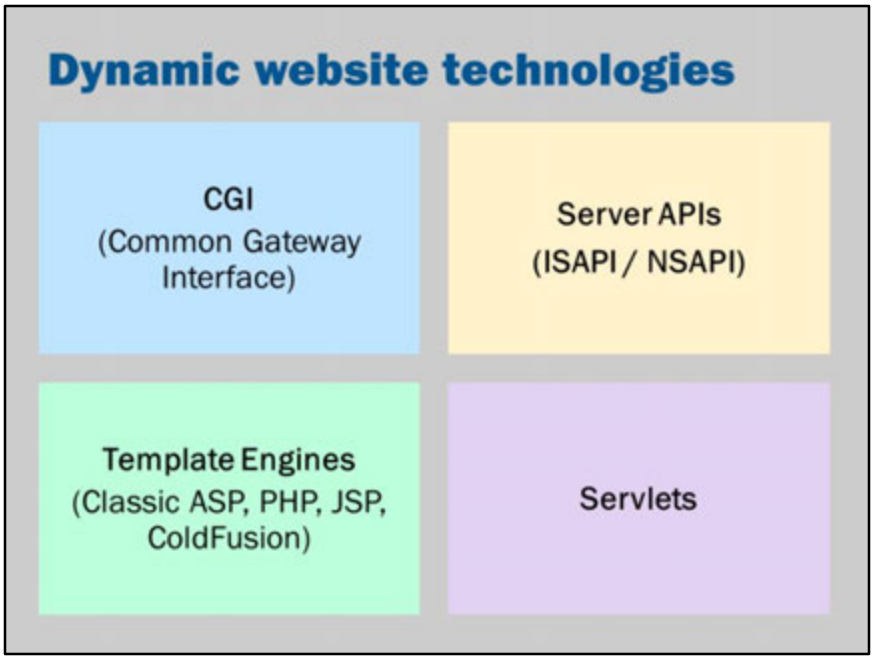
Advantages:
- Efficient (no heavy-weight processes, only threads; modern JVMs are very fast)
- Robust (Servlet crash causes no damage)
- Secure (Java and JVM provide security)
- Powerful (access to full Java SE and Java EE APIs)
- Cross-platform portability

Disadvantages:
- Difficult to create rich HTML (however, this is addressed by Java Server Pages and Java Server Faces)

Simple CGI script

```
public class CGI {

   public static void main(String[] args) {
     System.out.println("Content-type: text/html");
     System.out.println();
     System.out.println("<html><body>");
     System.out.println("Hello, World!");
     System.out.println("</body></html>");
   }

}
```

This is what a simple CGI script might look like. Essentially all a CGI script does is output headers and the HTML body.



This is just a copy of the earlier slide.

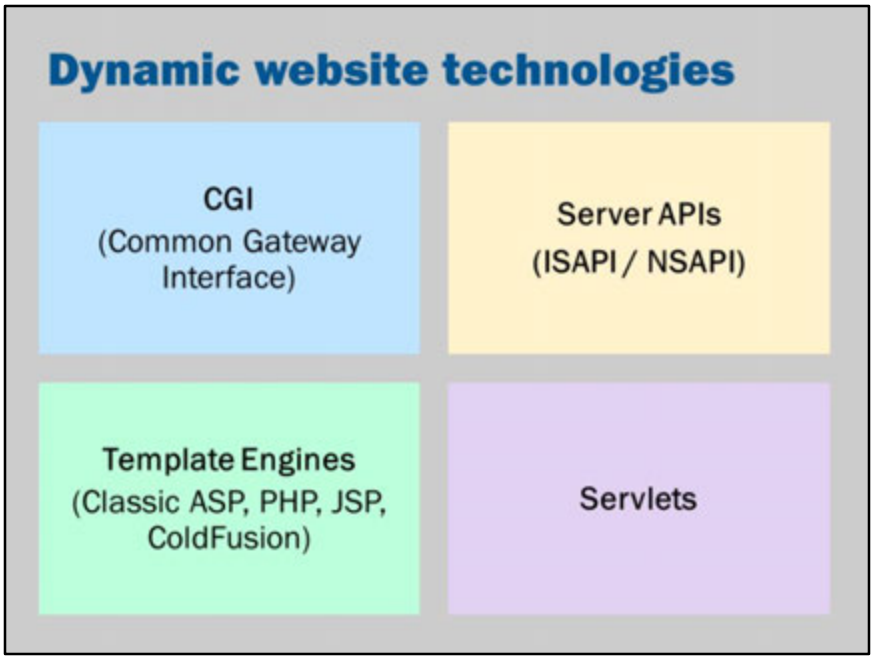
Simple JSP example

```
<%! private int counter = 0; %>

<%
    int visitorNum;
    synchronized (this) {
        counter++;
        visitorNum = counter;
    }
%>

<html>
<body>
<p>You are visitor: <%= visitorNum %></p>
</body>
</html>
```

This is an example of a template engine.
In particular it is an example of Java Server Pages.

You'll notice that HTML and Java code have been mixed together.

Don't worry about what this code actually means right now… I'll explain that next week.

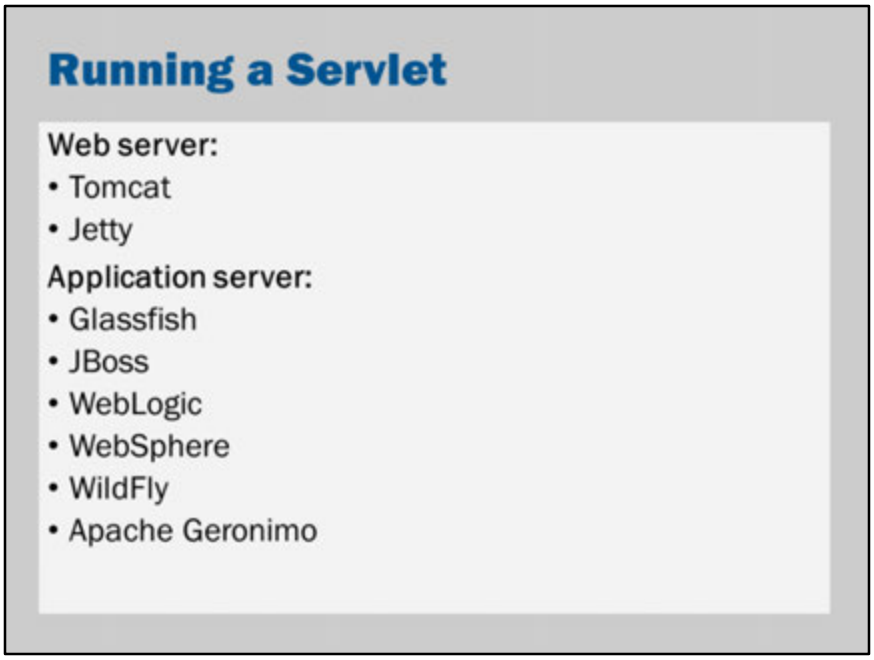


This is just a copy of the earlier slide.

Basic servlet

```
public class Greeting extends HttpServlet {

    @Override
    protected void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
                throws ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<body>");
        out.println("<p>Hello, World!</p>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

This is a simple Servlet. Next week, we'll look at what this means in greater detail.

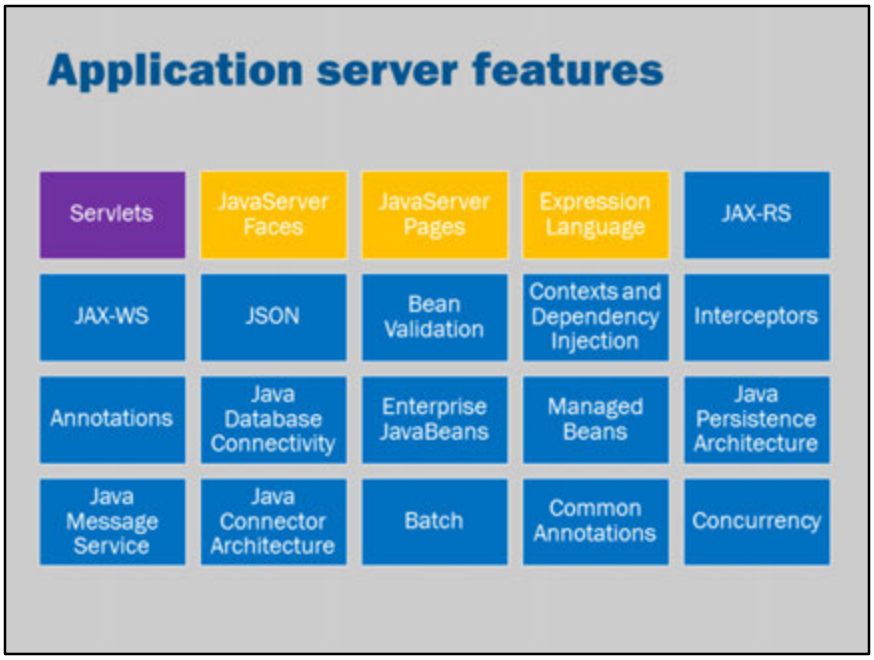


These are the names of well-known web servers that can run a Servlet.

Tomcat and Jetty are web servers. They can run Servlets (as well as Java Server Pages).

The application servers can also run Servlets. However, they also have a wide range of other features.

In theory, you can write your Serlvet code once, and it will run on all of these servers without needing to be changed at all.



Servlets help hide the complexity of building a scalable, robust, secure and standards-compliant web server. In a Servlet, you can focus on the business problem.

A Web Server or the Application Server deals with the protocols and the complexity of scalability and robustness. The server acts as a container for the servlet, and provides a range of other services that are useful for developing business applications.

This diagram illustrates some of the services provided by a Java EE 7 application server.

What is the difference between a web server and an application server?

An application server is a “superset” of a web server. A web server provides features and services relating to serving dynamic content on the web. Application servers provide many more features including remote method invocation (i.e., non-web clients), transactions, database connectivity, messaging services and so on.

Key points

- Servlets are the foundation for Java web technologies
- Servlets run on a web server or an application server

Creating servlets

I'm going to use this IP address lookup tool for the examples in upcoming slides.

GET Servlet

```
@WebServlet("/form")
public class FormServlet extends HttpServlet {

  @Override
  public void doGet(HttpServletRequest req,
                    HttpServletResponse resp)
              throws ServletException, IOException {


    resp.setContentType("text/html");

    PrintWriter out = resp.getWriter();
    out.println("<!DOCTYPE html>");
    out.println("<html>");
    out.println("  <head>");
    out.println("    <title>IP Address Lookup</title>");
    out.println("  </head>");
    out.println("  <body>");
    out.println("    <h1>IP Address Lookup</h1>");
    out.println("    <form action=\"lookup\" method=\"POST\">");
    out.println("      <input type=\"text\" name=\"ip\">");
    out.println("      <input type=\"submit\">");
    out.println("    </form>");
    out.println("  </body>");
    out.println("</html>");
  }

}
```

To create a servlet, just:

1. Extend HttpServlet
2. Extend/override a handler method such as doGet(request, response)

The default HttpServlet will dispatch GET and POST to doGet(…) and doPost(…) methods respectively.

More reading about GET, POST and the rest of the possible HTTP verbs:
http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
http://en.wikipedia.org/wiki/Representational_state_transfer
http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Generated HTTP response

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open
   Source Edition  4.0  Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition  4.0
Content-Type: text/html;charset=UTF-8
Date: Sat, 12 Jul 2014 06:32:55 GMT
Content-Length: 231

<!DOCTYPE html>
<html>
  <head>
    <title>IP Address Lookup</title>
  </head>
  <body>
    <h1>IP Address Lookup</h1>
    <form action="lookup" method="POST">
      <input type="text" name="ip">
      <input type="submit">
    </form>
  </body>
</html>
```

Here's what the servlet would generate….

…and the rendered webpage…

…suppose now that the user posts back some data…

Query (same or new connection)

```
POST /GeoIP/lookup HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64;
  rv:30.0) Gecko/20100101 Firefox/30.0
Accept: text/html;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://www.example.com/GeoIP/form
Connection: keep-alive

ip=8.8.8.8
```

…then this would be the posted back data…

POST Servlet with parameters

```
@WebServlet("/lookup")
public class LookupServlet extends HttpServlet {

  @Override
  public void doPost(HttpServletRequest req,
                     HttpServletResponse resp)
              throws ServletException, IOException {

    String ip = req.getParameter("ip");

    String location = GeoIP.getLocation(ip);

    resp.setContentType("text/html");

    PrintWriter out = resp.getWriter();
    out.println("<!DOCTYPE html>");
    out.println("<html>");
    out.println("  <head>");
    out.println("    <title>IP Address Lookup</title>");
    out.println("  </head>");
    out.println("  <body>");
    out.println("    <h1>IP Address Lookup</h1>");
    out.println("    <p>");
    out.println("      " + ip + " is located in: " + location);
    out.println("    </p>");
    out.println("  </body>");
    out.println("</html>");
  }

}
```

… and this is a servlet that could handle it.

It is a servlet, so it:

1. Extends HttpServlet
2. Extends/overrides a handler method such as doGet(request, response).
   In this case, it overrides doPost

The default HttpServlet will dispatch GET and POST to doGet(…) and doPost(…) methods respectively.

So, the doPost method will handle the posted data.

This line retrieves the value of the posted form data:

```
String ip = req.getParameter("ip");
```

This line is code that calls an external library to do Geo-location.
GeoIP is a class that I wrote (not shown) that does the IP lookup.
It isn't important for the demo.

You can just treat it as the calculation that the servlet does:

```
String location = GeoIP.getLocation(ip);
```

Then, finally the response is written out:

```
resp.setContentType("text/html");

PrintWriter out = resp.getWriter();
out.println("<!DOCTYPE html>");
out.println("<html>");
out.println("  <head>");
out.println("    <title>IP Address Lookup</title>");
out.println("  </head>");
out.println("  <body>");
out.println("    <h1>IP Address Lookup</h1>");
out.println("    <p>");
```

Note that here, we're inserting dynamic data into the generated HTML.

```
out.println("      " + ip + " is located in: " + location);
out.println("    </p>");
out.println("  </body>");
out.println("</html>");
```

Generated HTTP response

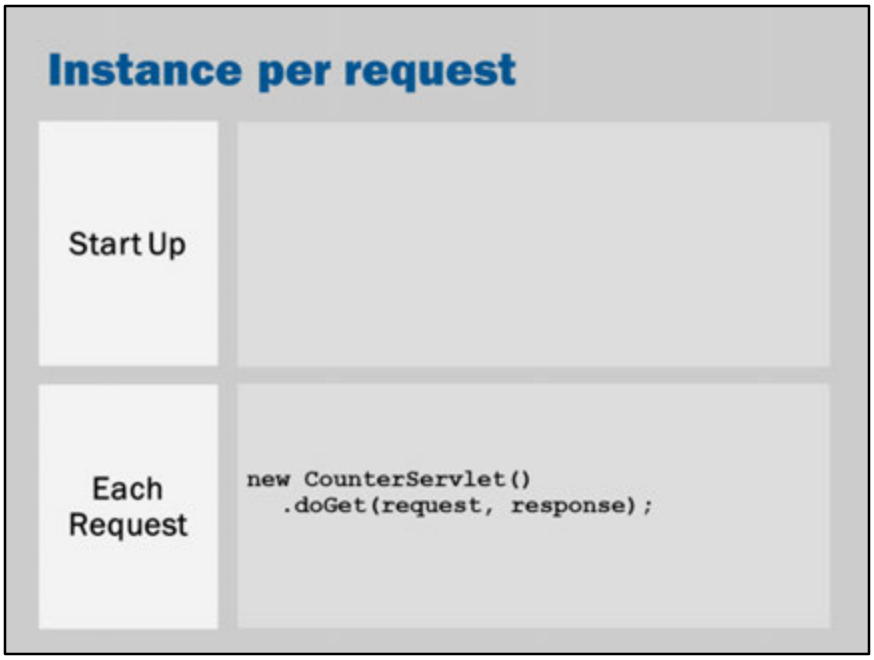
```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server
   Open Source Edition  4.0  Java/Oracle
   Corporation/1.7)
Server: GlassFish Server Open Source Edition  4.0
Content-Type: text/html;charset=UTF-8
Date: Sat, 12 Jul 2014 06:36:45 GMT
Content-Length: 179

<!DOCTYPE html>
<html>
  <head>
    <title>IP Address Lookup</title>
  </head>
  <body>
    <h1>IP Address Lookup</h1>
    <p>
      8.8.8.8 is located in: United States
    </p>
  </body>
</html>
```

… here's the result that gets generated…

… and this is what the user will see.

In application containers and other software platforms, “lifecycle” refers to the different stages of a component and the rules that dictate when and how the component may be used in those stages.

When using Servlets, it is important to understand when they are created and how. We will consider some possibilities, discuss the strengths and weaknesses of each and then see which model was used by Java EE.

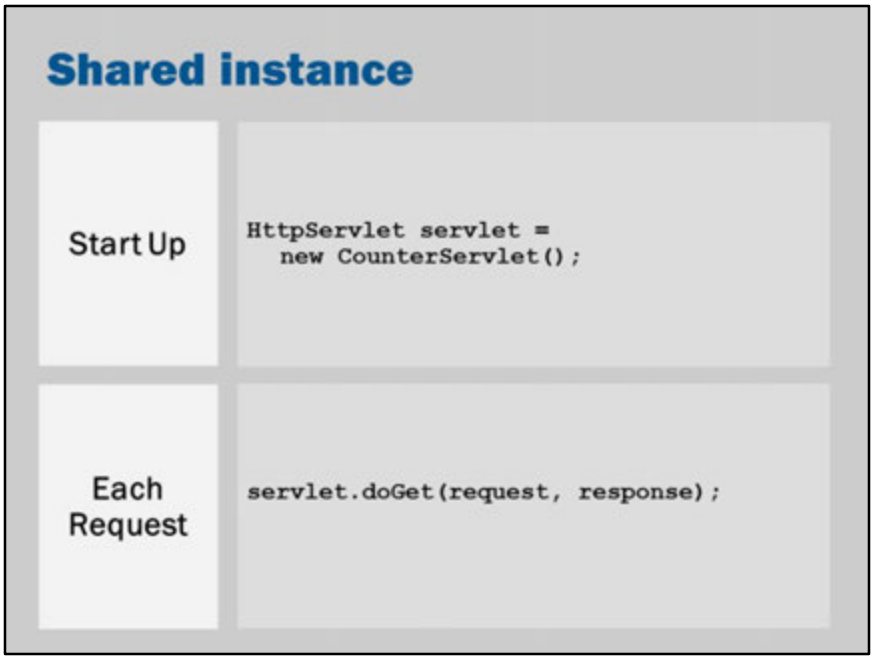


This slide and the next slides list four possibilities for how the Serlvet class might be instantiated by the web-server.

One possibility is to create a new instance of the Servlet class to handle every request.

Pro:
- Simple programming model – easy to understand
- Can use and modify instance state when handling a request

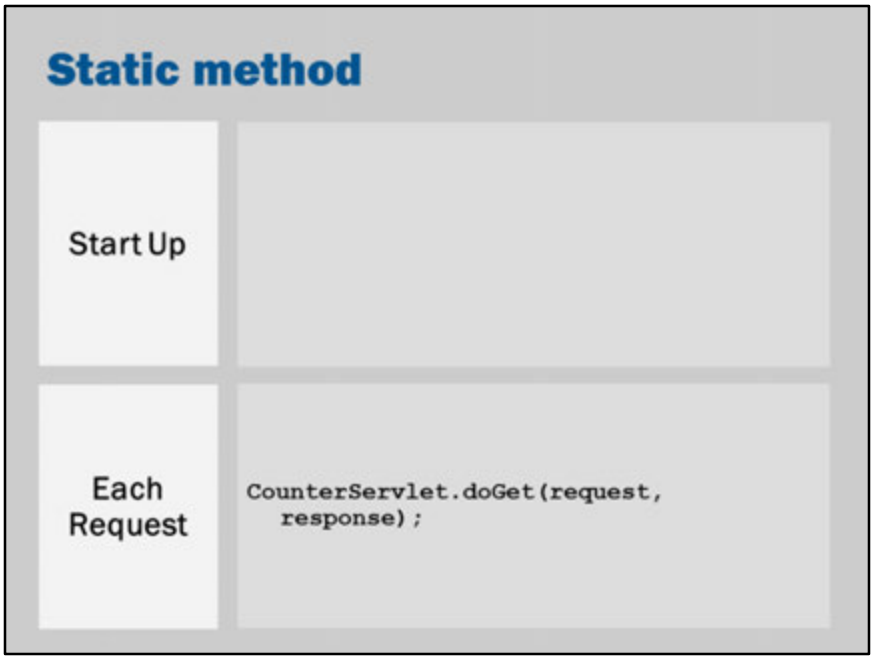
Con:
- Must create and garbage collect the object instance with each request (nowadays not so costly as it used to be)
- If there is expensive initialization (e.g., reading configuration files) this will be lost after each request



One possibility is to create just one instance of the Servlet class, and have that single instance handle every single request.

Pro:
- Fast
- Allows for expensive initialization to be done just once

Con:
- Need to be very careful about concurrent access
- Cannot use local variables when handling individual requests (as this will result in concurrent modification problems)
- More confusing



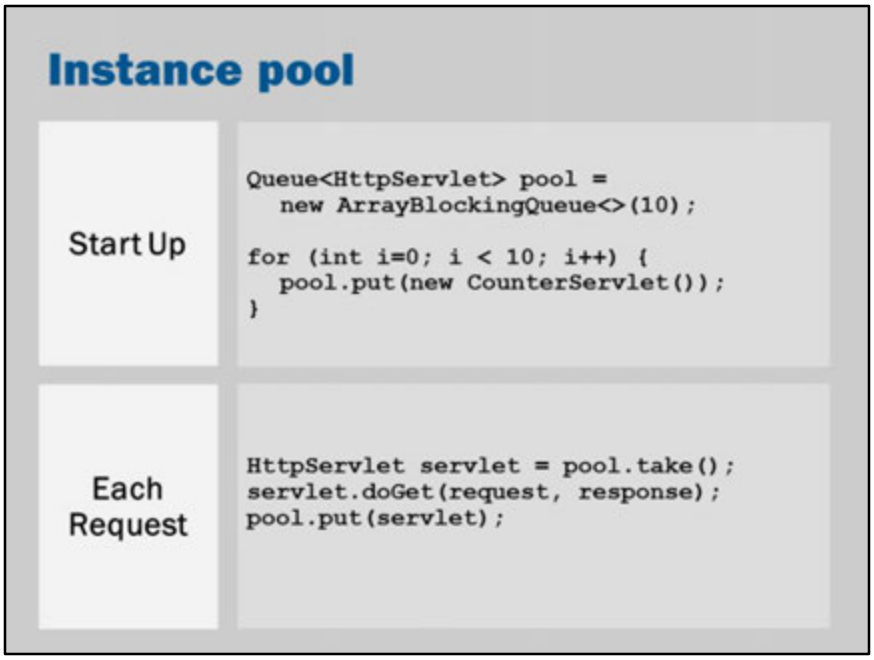
One possibility is to use only static methods.

Pro:

- No surprises – it isn't possible to have more than one instance because static methods aren't associated with instances

Con:

- Does not work well with inheritance or object-oriented design
- Not possible to enforce compile-time checks of method signatures



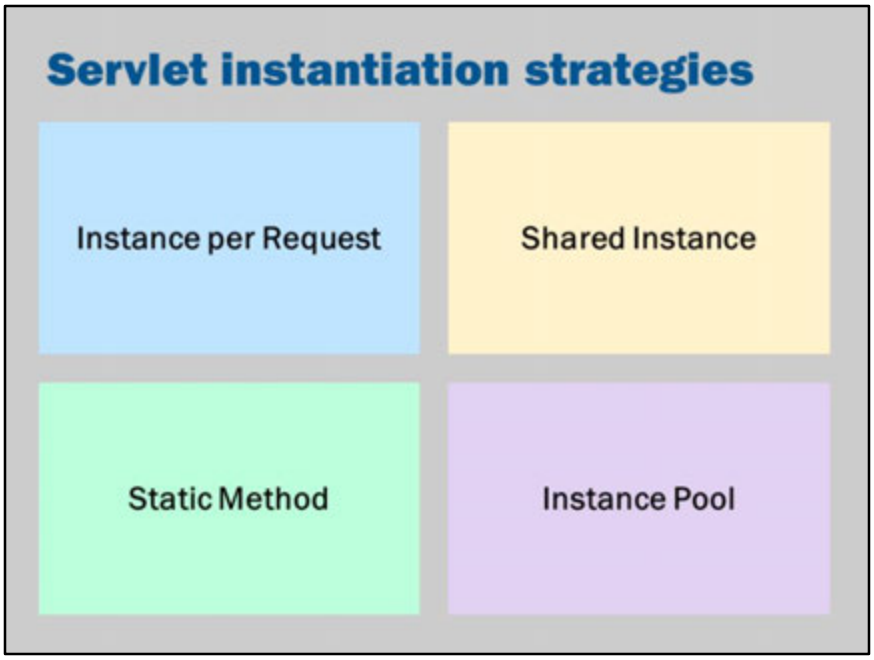
One possibility is to create several instances of the Servlet class, but ensure that each instance is processing no more than one request at a time (i.e., one instance per thread in the server).

Pro:

- Simple programming model – easy to understand
- Can safely manipulate instance state when handling a request

Con:

- Need to remember to clean up after each request – this can be confusing and may result in surprising bugs that only manifest in certain situations (e.g., after 10 requests when each item in the pool has been used once)



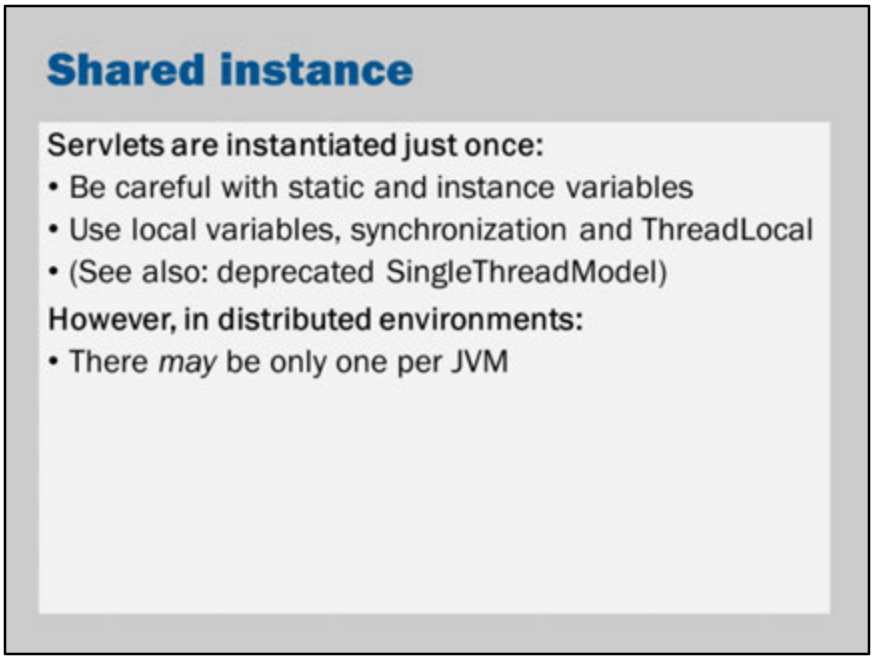
These aren't the only approaches: in the lecture. Another possibility could be to have a servlet per thread (and reusing threads) – this is similar to the Instance Pool approach.

Which would you use when designing an application container?

All of these approaches have there benefits and would be reasonable in one way or another.

For example, ASP.NET uses the instance per request approach.

Java EE uses shared instance. This is probably a historical decision made when the Java platform was slower: a shared instance is more efficient because there are less objects to create and less to clean up (i.e., garbage collect) for each request. However, the advantage probably does not make as much sense today: computers and Java are a lot faster.



In non-distributed Java EE application servers, there must be only one instance created by the application container. This means that, whenever possible you should use local variables.

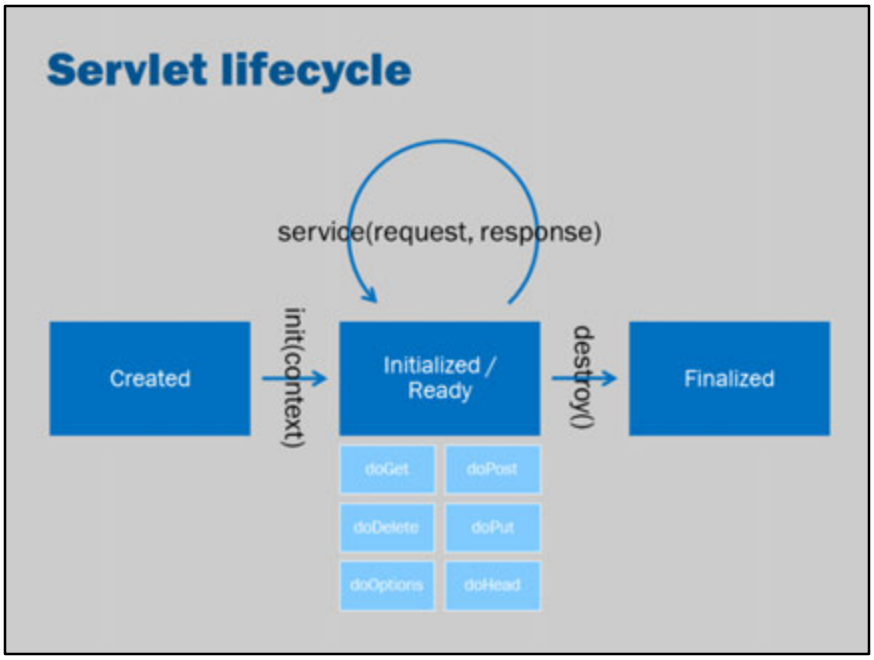
```
@WebServlet("/MyServlet"})
public class MyServlet extends HttpServlet {
   private static int staticVariable; // be very careful of other threads
   private int instanceVariable;  // be very careful of other threads

   @Override
   protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
      int localVariable; // this variable is local to the thread
                         // so you don't have to worry about concurrency

   }
}
```

In distributed Java EE application servers, all bets are off:

- There are multiple servers running on different computers, so there must be one

instance per computer.

- In addition, the spec allows for more than once instance in a JVM.

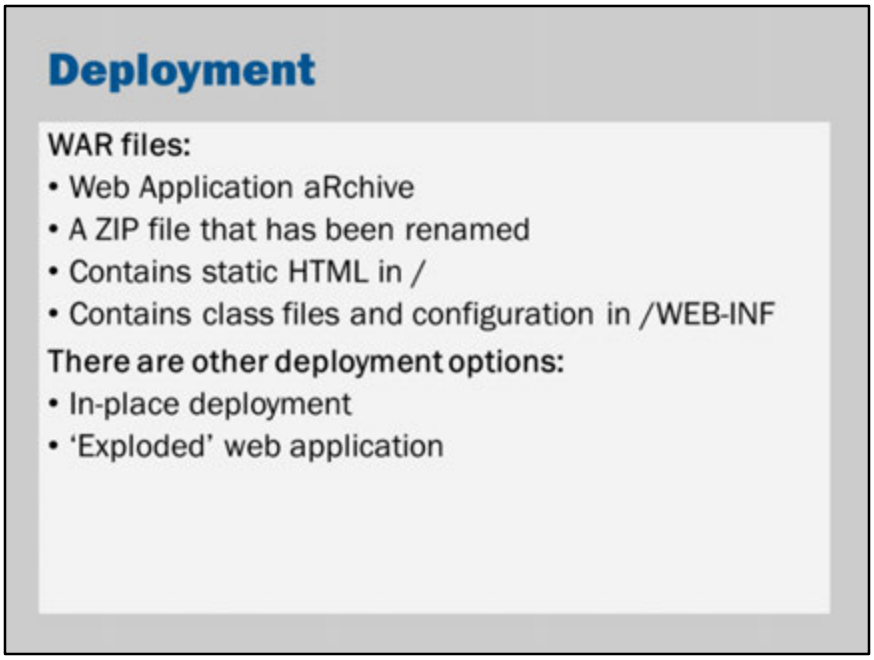


The lifecycle of a Servlet is controlled by the container. A Servlet is created/instantiated (i.e., new MyServlet()) but it will not receive any requests until after it has been initialized. It will also not receive any requests after it has been destroyed. These states are depicted in the diagram above.

Why do we need these states? Why not just use the constructor and finalizer that can be used on any class in Java?

The reason is that the application container needs to start up cleanly and safely. It may need to instantiate the object and inspect the object early. However, it can only initialize the Servlet when other services in the container are ready (e.g., it may need to wait for the database to be ready before allowing the Servlet to initialize).

You cannot assume that any of the application server's services are available in the constructor – any initialization that depends on databases or other application server services should be done in the init(…) method.



How can Servlets be deployed on a production server?

In NetBeans, you can test your project by clicking "play".

To deploy your project on a production server you will need a WAR file.

A WAR file is actually just a zip file. If you rename a WAR file into a zip file, you can decompress the file and see all of its contents.

A WAR file is a zip with a special directory structure:
/WEB-INF : configuration files, libraries and class files
/META-INF : basic naming information about the WAR
everything else : content to be directly hosted on the web server

In NetBeans, you can create a WAR file by right clicking on the project and selecting "Build". You will find the war file inside your project folder.

Deployment depends on the application server. Typically you can just copy (e.g., drag-and-drop) the file into a special directory. The application server will detect the

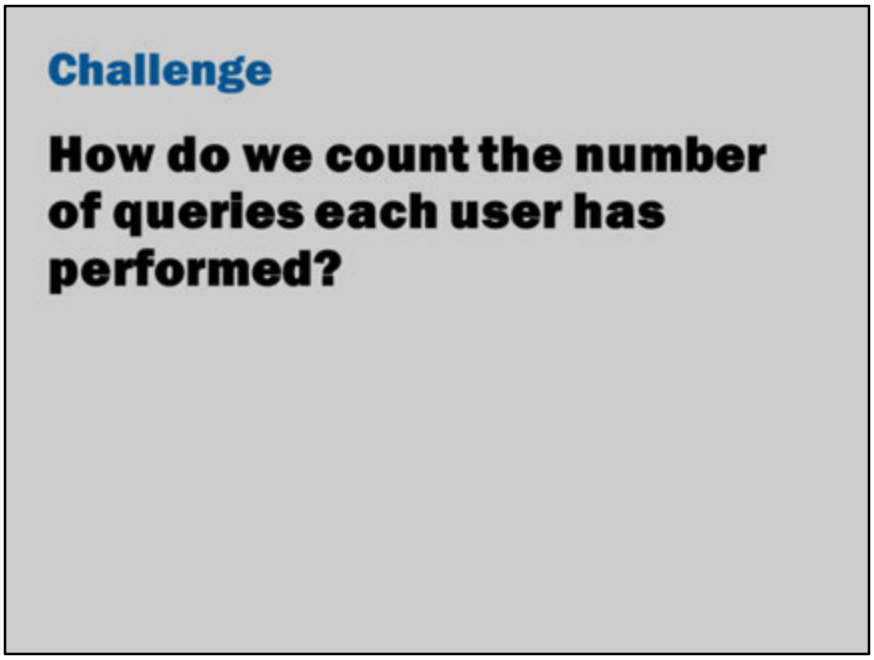
change, decompress the war file and run it.

Some application servers also have a special mode that allows you to deploy an uncompressed WAR file (i.e., “exploded”). This can create issues with concurrency but if you do not have an IDE it allows you to make small changes without needing to redeploy the entire application.

Key points

- Servlets follow a standard lifecycle:
  - *Only one instance is created*
  - *That instance is reused for all requests*
- Servlets and static content are deployed using a WAR file
  - *Just a ZIP file that has been renamed*

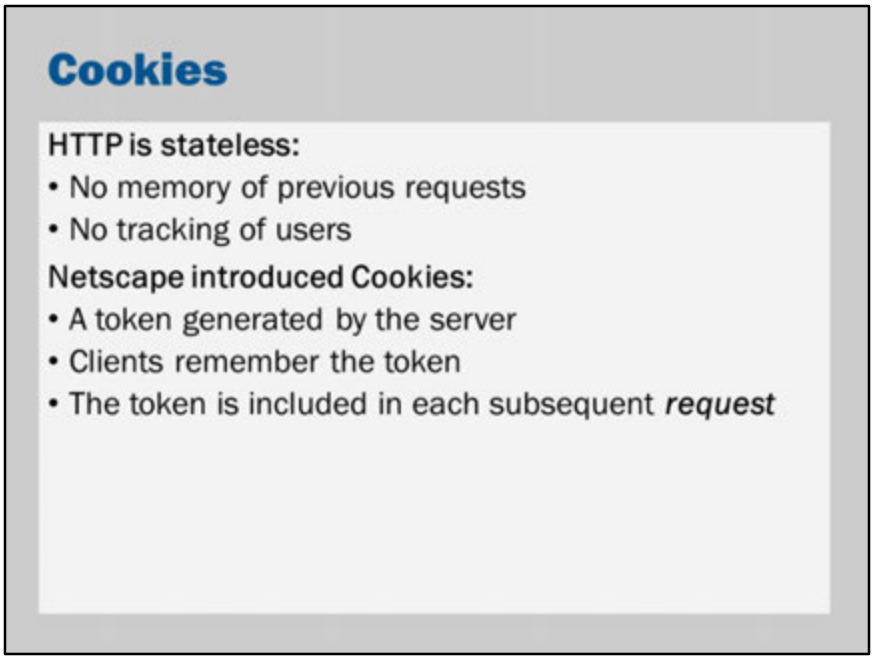
Sessions



In this section, consider the question of how we can count the number of page requests performed by each user.

HTTP is stateless, so we can't count the number of queries on a "connection".
The basic HTTP protocol provides no way of identifying users.
More than one user could be working behind a single IP address.

Also, a servlet has only one instance so there's no association with users on the server side either.



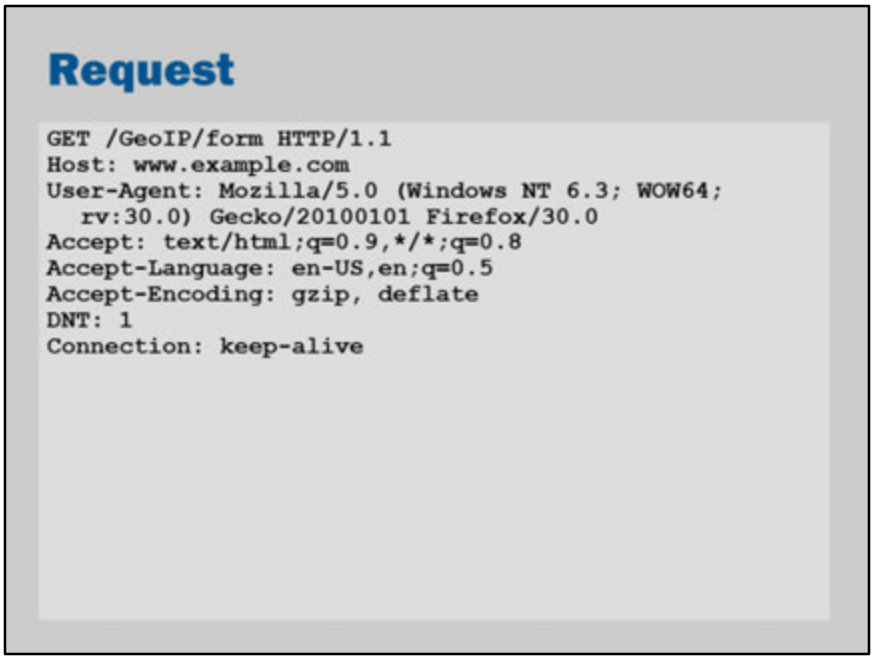
HTTP is a stateless protocol. Each request and response is independent. The protocol defines no way of linking or connecting separate requests.

This is where cookies and sessions come in. Cookies were introduced to the Netscape web browser in 1994 as a way to maintain state or tracking between requests.

Application servers use cookies (and URL re-writing) to provide sessions as a view of a user's interactions across multiple requests.

So, basically, cookies are a way that allows the server to give each web browser some identifying information.

More information:
The full Cookie and Set-Cookie specification is RFC 6265:
http://tools.ietf.org/html/rfc6265



When a user first visits our website, their web browser has no way of knowing if sessions / cookies are supported.

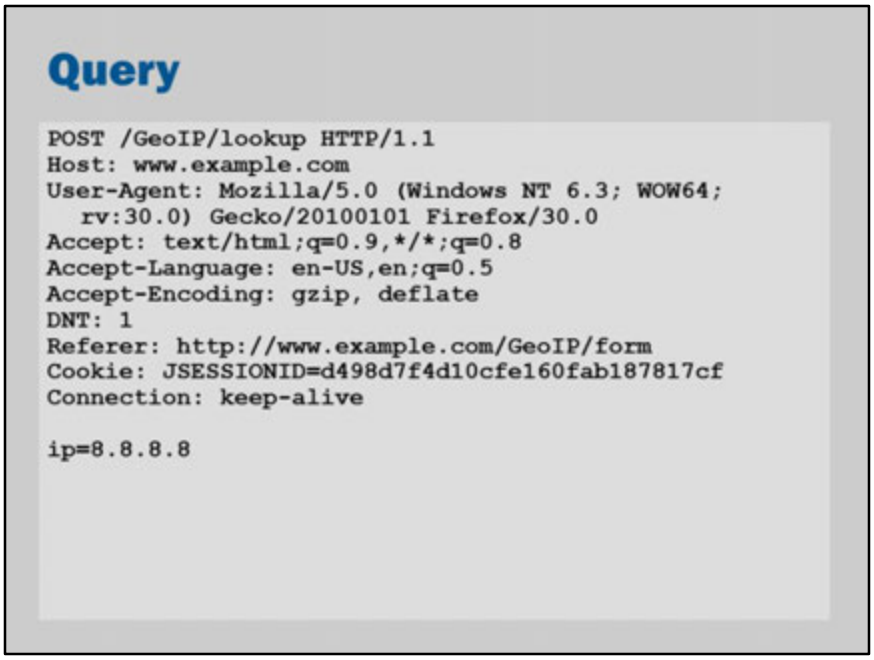
Thus, the request for a page is no different to what we saw earlier in the lecture.

Response

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open
   Source Edition  4.0  Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition  4.0
Set-Cookie: JSESSIONID=d498d7f4d10cfe160fab187817cf;
   Path=/GeoIP; HttpOnly
Content-Type: text/html;charset=UTF-8
Date: Sun, 13 Jul 2014 01:15:00 GMT
Content-Length: 231

<!DOCTYPE html>
<html>
  <head>
    <title>IP Address Lookup</title>
  </head>
  <body>
    <h1>IP Address Lookup</h1>
    <form action="lookup" method="POST">
      <input type="text" name="ip">
      <input type="submit">
    </form>
  </body>
</html>
```

This time, when the server responds it adds an extra header:
Set-Cookie: JSESSIONID=d498d7f4d10cfe160fab187817cf; Path=/GeoIP; HttpOnly

This tells the web-browser, “whenever you request a web page from me in future, please give me this cookie back”.



When the user has submitted their query, their browser makes a request to the server.

Notice that, this time, there is an extra header:
Cookie: JSESSIONID=d498d7f4d10cfe160fab187817cf

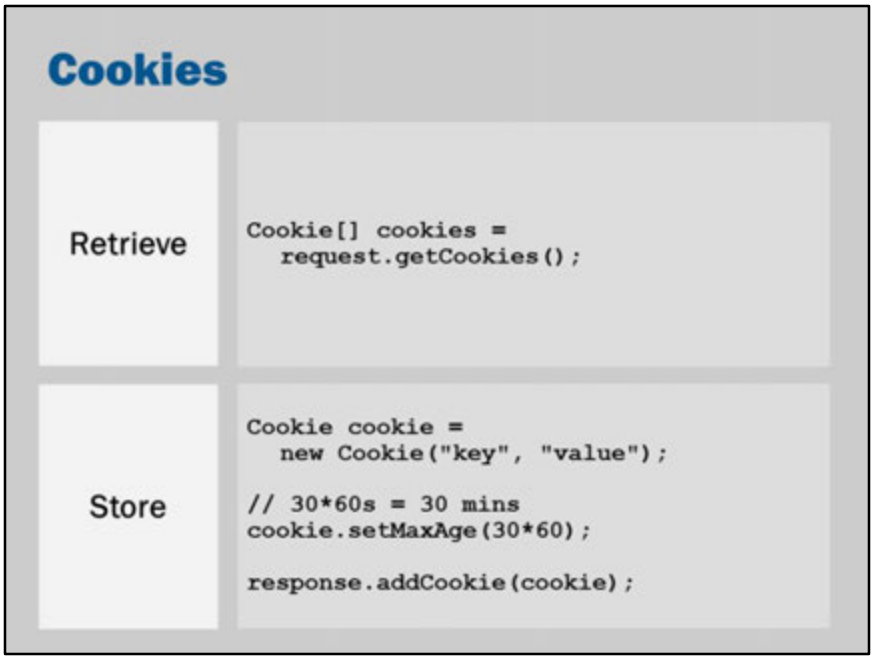
This header matches the Set-Cookie header that the browser sent the client in the last response.

The server can then link up the JSESSIONID values to see that they're the same, and recognize that it is the same user as before.

Response

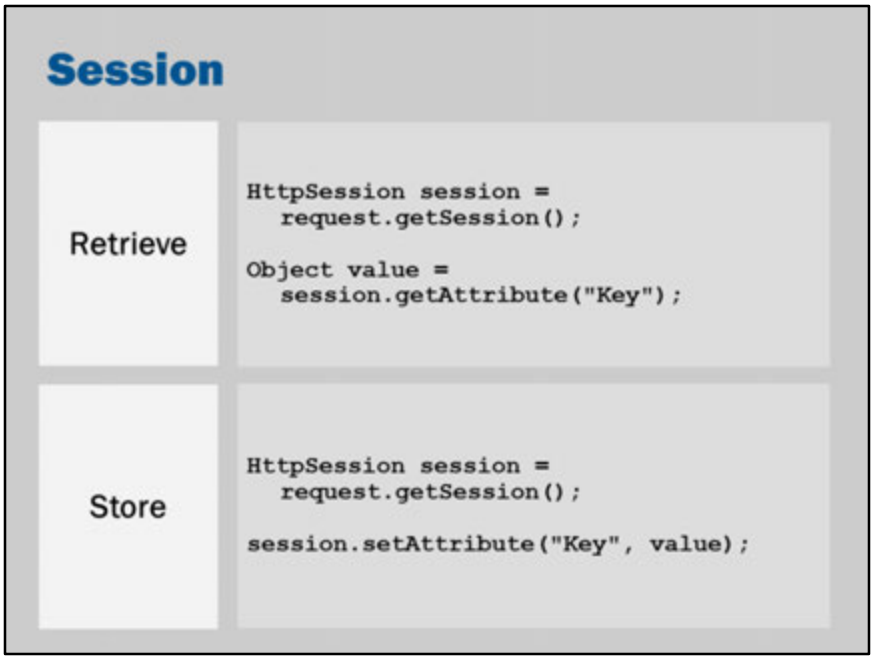
```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server
   Open Source Edition  4.0  Java/Oracle
   Corporation/1.7)
Server: GlassFish Server Open Source Edition  4.0
Content-Type: text/html;charset=UTF-8
Date: Sun, 13 Jul 2014 01:16:57 GMT
Content-Length: 179

<!DOCTYPE html>
<html>
  <head>
    <title>IP Address Lookup</title>
  </head>
  <body>
    <h1>IP Address Lookup (2)</h1>
    <p>
      8.8.8.8 is located in: United States
    </p>
  </body>
</html>
```

When the server now responds, it already knows that the user has a cookie, so it does not need to send another cookie.



As a developer on the Java EE platform, you **can** set cookies manually.
However, even though you **can** do it manually, it is usually **better** to use sessions (as we will see on the next slide).

This is a very low-level interface.
The store method simply causes “Set-Cookie:” headers to be sent to the client.
The Retrieve function returns any “Cookie:” headers provided by the client.



Sessions are an abstraction automatically provided and managed by the application server.

The application server will generate cookies for you (this is what the JSESSIONID comes from).
It will make sure that the cookies are generated in a secure way so that they can't easily be stolen by guessing.
It will store them securely on the server for you.
When a user returns to your site, it will look up the session information and retrieve it for use in your application.

You can store any kind of objects in a session.
The data is stored on the server (so you can store internal information in the session) and the client is just given a small code (JSESSIONID) for tracking.

Advanced Tip: If your session objects are Serializable (e.g., they implement java.io.Serializable so that they can be saved to disk), your application server can keep session data on the disk (so that it isn't lost when your application server restarts) or so that it can transfer the session data to another server.

Advanced Tip 2: You can configure the timeout of a session:

```
// 30*60s = 30 mins
session.setMaxInactiveInterval(30*60);
```

Counting with a Session

```
@WebServlet("/lookup")
public class LookupServlet extends HttpServlet {

  @Override
  public void doPost(HttpServletRequest req,
                     HttpServletResponse resp)
              throws ServletException, IOException {

    // Get the session
    HttpSession session = req.getSession();

    // How many times have they visited?
    Integer count = session.getAttribute("userCount");

    // Increase the counter
    if (count == null)              // first visit
      count = 1;
    else                            // not first visit
      count = (int)count + 1;

    // Then save the changes back in the session
    session.setAttribute("userCount", count);


    // Generate the response...
    String ip = req.getParameter("ip");
    ...
```

Because a Servlet only has one instance created to handle all requests, if we count visits for individual users, you need to store the requests in a Session.

This code illustrates how you might do that.

This code gets the session object from the web server:

```
HttpSession session = req.getSession();
```

The session object can be used to store and retrieve values by name. This code retrieves that "userCount" from the session:

```
Integer count = session.getAttribute("userCount");
```

The next few lines of code do some addition:

```
if (count == null)          // first visit
  count = 1;
else                        // not first visit
  count = (int)count + 1;
```

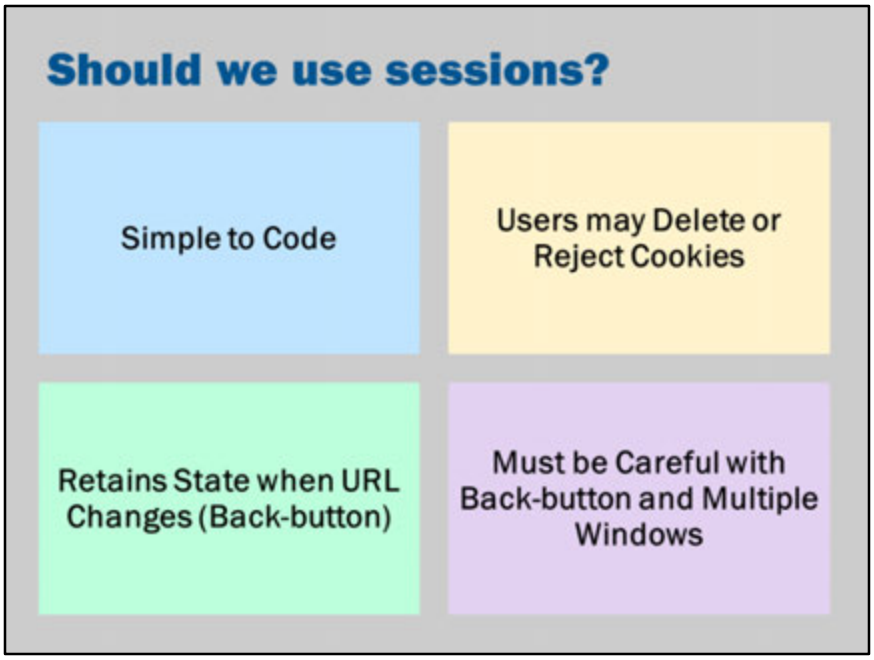
Then this line of code saves the result back into the session in the name "userCount".

```
session.setAttribute("userCount", count);
```

The counting has been done.

This code is then the start of the rest of the application, retrieving the user's submission as the first step in processing the results:

```
String ip = req.getParameter("ip");
```



Sessions are widely used for two main purposes:

- Keeping track of a shopping cart
- Keeping track of logged in users (especially “Remember me” on sites like Facebook/LinkedIn)

For these purposes, it is very helpful.

However, it is important to be careful.

One Cookie can be used in multiple tabs and multiple windows of the same web browser.
It usually makes sense to be able to browse in multiple windows and add items into one shopping cart.
Similarly, it makes sense that if you’ve logged in on one window, you should be logged in on all windows.

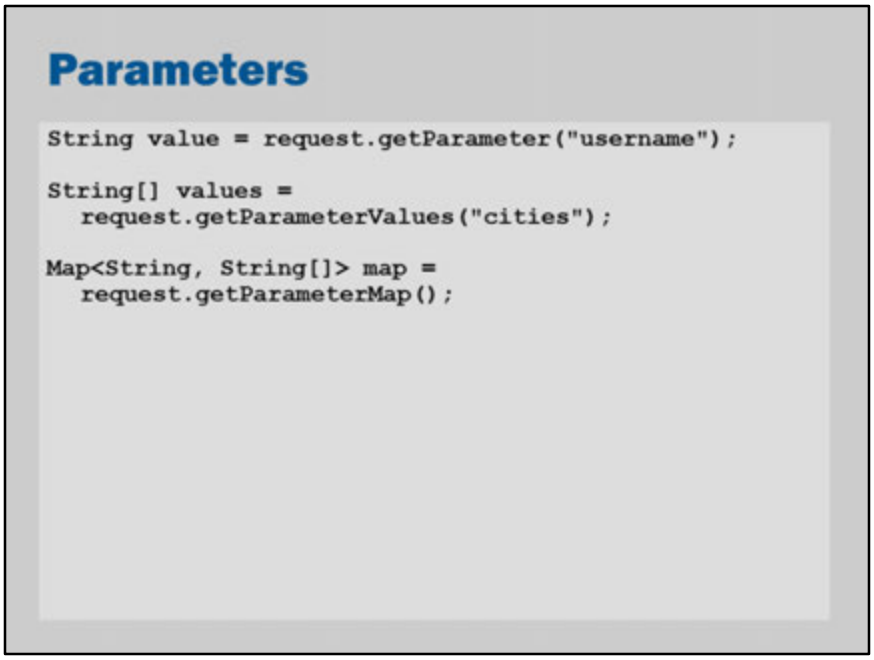
However, you can experience problems if you are using sessions to keep track of the user interface state.
For example:

- In a table of results, keeping track of the current page
- In a table of results, keeping track of which columns to sort by
- In a “wizard” style interface, keeping track of which page the user is currently on

Keeping track of user interface state in the session makes it difficult for the user to have two tabs open looking at or doing separate things.

In addition, you also need to be aware that cookies and sessions aren't reliable. Users can delete or reject cookies.
Also, malicious users could even generated fake cookies.

Bonus slides



Parameters are encoded into the URL of a GET request or into the message body of a POST request.

Form values are encoded by the web-browser into key/value pairs separated by an ampersand:
key1=value1&key2=value2&key3=value3

The Servlet container automatically decodes these values for the servlet.

If (for some reason) you wanted to decode the form values manually, you could use java.net.URLEncoder and java.net.URLDecoder.

These are three ways (but not the only three ways) to get the encoded parameters:

1. String value = request.getParameter("username");
   The value of a single parameter. This should only be used for singled-valued

parameters. If there is more than one value for a parameter, it will return the first. If there is no such named parameter, it will return null.

2. String[] values = request.getParameterValues("cities");
   The set of values of a parameter. HTTP/HTML allows for a parameter to have multiple values (this can happen with checkboxes). This retrieves every value associated with a parameter name (or null if there is no such parameter).
3. Map<String, String[]> map = request.getParameterMap();
   All values in a single map. This may be useful for iterating over every key/value submitted in a form. Ordinarily, you should know all the keys and values so this should not be necessary, but this may be useful for creating general purpose servlets that can deal with any form submission.

Study idea

- Look at the source for HttpServlet