

AJAX

□ **What You Should Already Know**

- Before you continue you should have a basic understanding of the following:
 - HTML / XHTML
 - JavaScript

□ **AJAX = Asynchronous JavaScript and XML**

- AJAX is not a new programming language, but a technique for creating better, faster, and more interactive web applications.

AJAX

- With AJAX, your JavaScript can communicate directly with the server, using the JavaScript XMLHttpRequest object.
- With this object, your JavaScript can trade data with a web server, without reloading the page
- AJAX uses asynchronous data transfer (HTTP requests) between the browser and the web server, allowing web pages to request small bits of information from the server instead of whole pages.
- The AJAX technique makes Internet applications smaller, faster and more user-friendly. AJAX is a browser technology independent of web server software.

AJAX is Based on Web Standards

- 
- AJAX is based on the following web standards, and these standards are well defined, and supported by all major browsers. AJAX applications are browser/platform independent.
 - **JavaScript**
 - **XML**
 - **HTML**
 - **CSS**

AJAX is About Better Internet Applications



- Web applications have many benefits over desktop applications;
 - they can reach a larger audience,
 - they are easier to install and support,
 - easier to develop.
- However, Internet applications are not always as "rich" and user-friendly as traditional desktop applications.
- With AJAX, Internet applications can be made richer and more user-friendly.
- There is nothing new to learn.
 - AJAX is based on existing standards.
 - These standards have been used by most developers for several years.

AJAX Uses HTTP Requests

- In traditional JavaScript coding, if you want to get any information from a database or a file on the server, or send user information to a server, you will have to make an HTML form and GET or POST data to the server.
- The user will have to click the "Submit" button to send/get the information, wait for the server to respond, then a new page will load with the results.
- Because the server returns a new page each time the user submits input, traditional web applications can run slowly and tend to be less user-friendly.
- With AJAX, your JavaScript communicates directly with the server, through the JavaScript **XMLHttpRequest** object. With an HTTP request, a web page can make a request to, and get a response from a web server - without reloading the page.
- The user will stay on the same page, and he or she will not notice that scripts request pages, or send data to a server in the background.

The XMLHttpRequest Object

- By using the XMLHttpRequest object, a web developer can update a page with data from the server after the page has loaded!
- AJAX was made popular in 2005 by Google (with Google Suggest).
- Google Suggest is using the XMLHttpRequest object to create a very dynamic web interface:
 - When you start typing in Google's search box, a JavaScript sends the letters off to a server and the server returns a list of suggestions.
- The XMLHttpRequest object is supported in Internet Explorer 5.0+, Safari 1.2, Mozilla 1.0 / Firefox, Opera 8+, and Netscape 7.
- **AJAX - Browser Support**
 - The keystone of AJAX is the XMLHttpRequest object.

```
1 <html>
2 <body>
3 <script type="text/javascript">
4     function ajaxFunction()
5     {
6         var xmlhttp; //Different browsers use different methods to create the XMLHttpRequest object
7
8         try // Firefox, Opera 8.0+, Safari
9         {
10             xmlhttp=new XMLHttpRequest();
11         }
12         catch (e)
13         {
14             try // Internet Explorer
15             {
16                 xmlhttp=new ActiveXObject("Msxml2.XMLHTTP");
17             }
18             catch (e)
19             {
20                 try
21                 {
22                     xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
23                 }
24                 catch (e)
25                 {
26                     alert("Your browser does not support AJAX!");
27                     return false;
28                 }
29             }
30         }
31
32         xmlhttp.onreadystatechange=function()
33         {
34             if(xmlhttp.readyState==4)
35             {
36                 document.myForm.time.value=xmlhttp.responseText;
37             }
38         }
39
40         xmlhttp.open("GET","time.php",true);
41         xmlhttp.send();
42     }
43 </script>
44
45 <form name="myForm">
46     Name: <input type="text" onkeyup="ajaxFunction(); name="username" />
47     Time: <input type="text" name="time" />
48 </form>
49 </body>
50 </html>
```

Did you notice something missing from the HTML form?

- There is no submit button.

AJAX - More About the XMLHttpRequest Object

- Before sending data to the server, we have to explain three important properties of the XMLHttpRequest object.

1. The onreadystatechange Property

- After a request to the server, we need a function that can receive the data that is returned by the server.
- The onreadystatechange property stores your function that will process the response from a server.
- This is not a method, the function is stored in the property to be called automatically.
- The following code sets the onreadystatechange property and stores an empty function inside it:

```
xmlHttp.onreadystatechange=function()  
{  
    // We are going to write some code here  
}
```

2. The readyState Property

- When a request to a server is sent, we want to perform some actions based on the response.
- The onreadystatechange event is triggered every time the readyState changes.
- The readyState property holds the status of the XMLHttpRequest.
- Three important properties of the XMLHttpRequest object:

Property	Description
onreadystatechange	Stores a function (or the name of a function) to be called automatically each time the readyState property changes
readyState	Holds the status of the XMLHttpRequest. Changes from 0 to 4: 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
status	200: "OK" 404: Page not found

- In the onreadystatechange event, we specify what will happen when the server response is ready to be processed.
- When readyState is 4 and status is 200, the response is ready:

```
xmlhttp.onreadystatechange=function()
{
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
  {
    document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
  }
}
```

Using a Callback Function

- A callback function is a function passed as a parameter to another function.
- If you have more than one AJAX task on your website, you should create ONE standard function for creating the XMLHttpRequest object, and call this for each AJAX task.
- The function call should contain the URL and what to do on onreadystatechange (which is probably different for each call):

```
function myFunction()
{
    loadXMLDoc("ajax_info.php", function() {
        if (xmlhttp.readyState==4 && xmlhttp.status==200)
        {
            document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
        }
    });
}
```

3. The responseText Property

The data sent back from the server can be retrieved with the `responseText` property.

In our code, we will set the value of our "time" input field equal to `responseText`:

```
xmlHttp.onreadystatechange=function()
{
  if (xmlHttp.readyState==4)
  {
    document.myForm.time.value=xmlHttp.responseText;
  }
}
```

AJAX - Sending a Request to the Server

The XMLHttpRequest object is used to exchange data with a server.

Send a Request To a Server

To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object:

```
xmlhttp.open("GET", "ajax_info.txt", true);
xmlhttp.send();
```

Method	Description
open(<i>method, url, async</i>)	Specifies the type of request, the URL, and if the request should be handled asynchronously or not. <i>method</i> : the type of request: GET or POST <i>url</i> : the location of the file on the server <i>async</i> : true (asynchronous) or false (synchronous)
send(<i>string</i>)	Sends the request off to the server. <i>string</i> : Only used for POST requests

GET or POST?

- GET is simpler and faster than POST, and can be used in most cases.
- However, always use POST requests when:
 - ▣ A cached file is not an option (update a file or database on the server)
 - ▣ Sending a large amount of data to the server (POST has no size limitations)
 - ▣ Sending user input (which can contain unknown characters), POST is more robust and secure than GET

GET Requests

A simple GET request:

Example

```
xmlhttp.open ("GET", "demo_get.php", true);  
xmlhttp.send();
```

In the example above, you may get a cached result.

To avoid this, add a unique ID to the URL:

Example

```
xmlhttp.open ("GET", "demo_get.php?t=" + Math.random(), true);  
xmlhttp.send();
```

If you want to send information with the GET method, add the information to the URL:

Example

```
xmlhttp.open ("GET", "demo_get2.php?fname=Henry&lname=Ford", true);  
xmlhttp.send();
```

POST Requests

A simple POST request:

Example

```
xmlhttp.open("POST","demo_post.php",true);
xmlhttp.send();
```

To POST data like an HTML form, add an HTTP header with `setRequestHeader()`. Specify the data you want to send in the `send()` method:

Example

```
xmlhttp.open("POST","ajax_test.php",true);
xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");
xmlhttp.send("fname=Henry&lname=Ford");
```

Method	Description
<code>setRequestHeader(header,value)</code>	Adds HTTP headers to the request. <i>header</i> : specifies the header name <i>value</i> : specifies the header value

The url - A File On a Server

- The url parameter of the open() method, is an address to a file on a server:

```
xmlhttp.open("GET","ajax_test.php",true);
```

- The file can be any kind of file, like .txt and .xml, or server scripting files like .asp and .php (which can perform actions on the server before sending the response back).

Asynchronous - True or False?

- AJAX stands for Asynchronous JavaScript and XML, and for the XMLHttpRequest object to behave as AJAX, the async parameter of the open() method has to be set to true:

```
xmlhttp.open("GET","ajax_test.php",true);
```
- Sending asynchronous requests is a huge improvement for web developers.
- Many of the tasks performed on the server are very time consuming.
- Before AJAX, this operation could cause the application to hang or stop.
- With AJAX, the JavaScript does not have to wait for the server response, but can instead:
 - execute other scripts while waiting for server response
 - deal with the response when the response ready

Async=true

- When using async=true, specify a function to execute when the response is ready in the onreadystatechange event:

- Example**

```
xmlhttp.onreadystatechange=function()
{
if (xmlhttp.readyState==4 && xmlhttp.status==200)
{
document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
}
}
xmlhttp.open("GET","ajax_info.php",true);
xmlhttp.send();
```

Async=false

- To use async=false, change the third parameter in the open() method to false:
`xmlhttp.open("GET","ajax_info.php",false);`
- Using async=false is not recommended, but for a few small requests this can be ok.
- Remember that the JavaScript will NOT continue to execute, until the server response is ready.
- If the server is busy or slow, the application will hang or stop.
- Note: When you use async=false, do NOT write an onreadystatechange function - just put the code after the send() statement:
- **Example**

```
xmlhttp.open("GET","ajax_info.txt",false);
xmlhttp.send();
document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
```

Server Response

To get the response from a server, use the `responseText` or `responseXML` property of the `XMLHttpRequest` object.

Property	Description
<code>responseText</code>	get the response data as a string
<code>responseXML</code>	get the response data as XML data

The `responseText` Property

If the response from the server is not XML, use the `responseText` property.

The `responseText` property returns the response as a string, and you can use it accordingly:

Example

```
document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
```

The responseXML Property

If the response from the server is XML, and you want to parse it as an XML object, use the `responseXML` property:

Example

Request the file [cd_catalog.xml](#) and parse the response:

```
xmlDoc=xmlhttp.responseXML;
txt="";
x=xmlDoc.getElementsByTagName ("ARTIST");
for (i=0;i<x.length;i++)
{
  txt=txt + x[i].childNodes[0].nodeValue + "<br>";
}
document.getElementById("myDiv").innerHTML=txt;
```

Dependency Injection

- One of the biggest issues in software systems today is managing the dependencies between objects. If my `ProcessOrdersService` class is using the `OrdersDAO` and `CustomersDAO` classes, it has dependencies on them and, through them, each of *their* dependencies.
- Unmanaged, those dependencies can get out of control without you even noticing.
- If you have ever changed a constructor signature and realized that you have just broken your code in 19 places, or ever tried to instantiate an object only to find that it needs the environment *just so* because of a dependency three levels down, you know the pain that I am describing.
- *Inversion of Control (IoC)* means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source (for example, an xml configuration file).
- *Dependency Injection (DI)* means that this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties.
- In object-oriented computer programming DI is a technique for supplying an external dependency (i.e. a reference) to a software component - that is, indicating to a part of a program which other parts it can use.

- Say your application has a text editor component and you want to provide spell checking.
- Here we create a dependency between the TextEditor and the SpellChecker.

```
public class TextEditor
{
    private SpellChecker checker;
    public TextEditor()
    {
        checker = new SpellChecker();
    }
}
```

The Problem – Tight Coupling

- The biggest issue with the code is tight coupling between classes.
- In other words the **TextEditor** class depends on the **checker** object. So for any reason SpellChecker class changes, it will lead to change and compiling of 'TextEditor' class also.
- So let's put down problems with this approach:
 - ▣ The biggest problem is that TextEditor class controls the creation of checker object.
 - ▣ SpellChecker class is directly referenced in the TextEditor class which leads to tight coupling between SpellChecker and TextEditor objects.
 - ▣ TextEditor class is aware of the SpellChecker class type.
 - ▣ So if we add new SpellChecker types, it will lead to changes in the TextEditor class also as TextEditor class is exposed to the actual SpellChecker implementation.
- So if for any reason the SpellChecker object is not able to create, the whole TextEditor class will fail in the constructor initialization itself.

Solution

- The solution definitely revolves around shifting the object creation control from the TextEditor class to some one else.
- The main problem roots from the TextEditor class creating the checker object.
- If we are able to shift this task/control of object creation from the TextEditor class to some other entity, we have solved our problem.
- In other words, if we are able to invert this control to a third party, we have found our solution.
- **So the solution name is IOC (Inversion of Control).**

IoC and DI patterns are all about removing dependencies from your code

- In an IoC scenario we would instead do something like this:

```
public class TextEditor
{
    private ISpellChecker checker;
    public TextEditor(ISpellChecker checker)
    {
        this.checker = checker;
    }
}
```

- Now, the client creating the `TextEditor` class has the control over which `SpellChecker` implementation to use.
- We're injecting the `TextEditor` with the dependency.

Dependency Injection (DI)

- Without the concept of dependency injection, a consumer who needs a particular service in order to accomplish a certain task would be responsible for handling the life-cycle (instantiating, opening and closing streams, disposing, etc.) of that service.
- Using the concept of dependency injection, however, the life-cycle of a service is handled by a dependency provider (typically a container) rather than the consumer.
- The consumer would thus only need a reference to an implementation of the service that it needed in order to accomplish the necessary task.

DI involves at least three elements:

□ a **dependent**,

□ its **dependencies** and

□ an **injector** (sometimes referred to as a **provider** or **container**).

- The dependent is a consumer that needs to accomplish a task in a computer program. In order to do so, it needs the help of various services (the dependencies) that execute certain sub-tasks.
- The provider is the component that is able to compose the dependent and its dependencies so that they are ready to be used, while also managing these objects' life-cycles.
- This injector may be implemented, for example, as a service locator, an abstract factory, a factory method or a more complex abstraction such as a framework.

Simple Example

- In this example we created a component that provides a list of movies directed by a particular director
- This stunningly useful function is implemented by a single method.

```
class MovieLister...
{
    public Movie[ ] moviesDirectedBy(String arg)
    {
        List allMovies = finder.findAll();

        for (Iterator it = allMovies.iterator(); it.hasNext();)
        {
            Movie movie = (Movie) it.next();
            if (!movie.getDirector().equals(arg))
                it.remove();
        }
        return (Movie[ ]) allMovies.toArray(new Movie[allMovies.size()]);
    }
}
```

- *The implementation of this function is naive in the extreme, it asks a finder object (which we'll get to in a moment) to return every film it knows about. Then it just hunts through this list to return those directed by a particular director.*
- The real point is this finder object, or particularly how we connect the lister object with a particular finder object.
- The reason why this is interesting is that I want this wonderful moviesDirectedBy method to be completely independent of how all the movies are being stored.
- So all the method does is refer to a finder, and all that finder does is know how to respond to the findAll method.
- We can bring this out by defining an interface for the finder.

```
public interface MovieFinder
{
    List findAll();
}
```

- Now all of this is very well decoupled, but at some point we have to come up with a concrete class to actually come up with the movies.
- In this case we put the code for this in the constructor of my lister class.

```
class MovieLister...  
{  
    private MovieFinder finder;  
    public MovieLister()  
    {  
        finder = new ColonDelimitedMovieFinder( "movies1.txt" );  
    }  
}
```

- *The name of the implementation class comes from the fact that we're getting the list from a colon delimited text file.*

- Now if we're using this class for just ourselves, this is all fine.
- But what happens when someone is overwhelmed by a desire for this wonderful functionality and would like a copy of this program?
- If they also store their movie listings in a colon delimited text file called "movies1.txt" then everything is wonderful.

If they have a different name for their movies file, then it's easy to put the name of the file in a properties file.

But what if they have a completely different form of storing their movie listing: a SQL database, an XML file, a web service, or just another format of text file?

In this case we need a different class to grab that data.

Now because we've defined a MovieFinder interface, this won't alter the moviesDirectedBy method.

But we still need to have some way to get an instance of the right finder implementation into place.

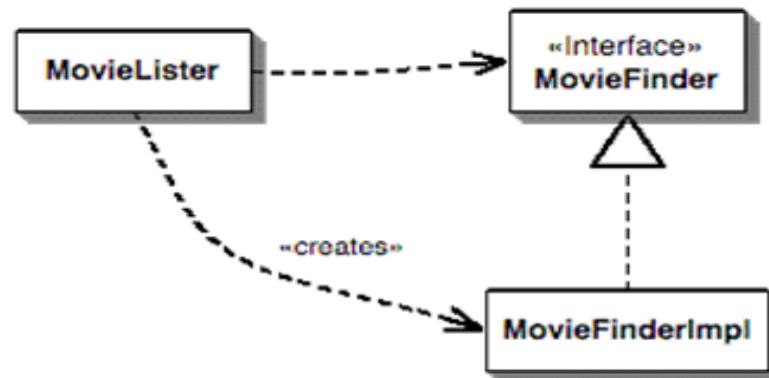


Figure 1: The dependencies using a simple creation in the lister class

Figure 1 shows the dependencies for this situation. The MovieLister class is dependent on both the MovieFinder interface and upon the implementation.



We would prefer it if it were only dependent on the interface, but then how do we make an instance to work with?

This situation could be described as a Plugin.

The implementation class for the finder isn't linked into the program at compile time, since we don't know what others are going to use.

Instead we want the lister to work with any implementation, and for that implementation to be plugged in at some later point, out of my hands.



The problem is how we can make that link so that the lister class is ignorant of the implementation class, but can still talk to an instance to do its work.

Expanding this into a real system, we might have dozens of such services and components.

In each case we can abstract our use of these components by talking to them through an interface (and using an adapter if the component isn't designed with an interface in mind).

But if we wish to deploy this system in different ways, we need to use plugins to handle the interaction with these services so we can use different implementations in different deployments.

- So the core problem is how do we assemble these plugins into an application?
- This is one of the main problems that this new breed of lightweight containers face, and universally they all do it using Inversion of Control.

The basic idea of the Dependency Injection is to have a separate object, an assembler, that populates a field in the lister class with an appropriate implementation for the finder interface, resulting in a dependency diagram along the lines of Figure 2

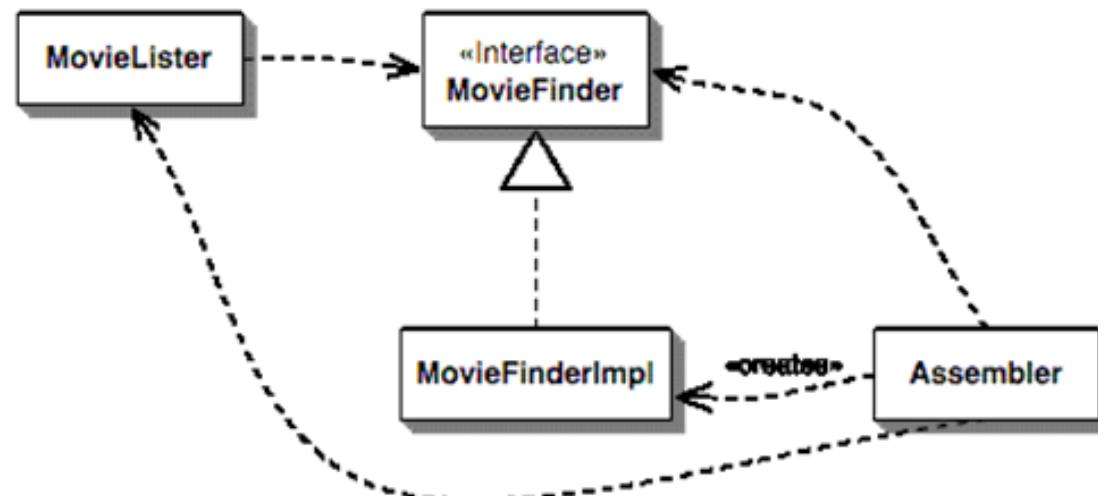


Figure 2: The dependencies for a Dependency Injector

Forms of Dependency Injection

There are three main styles of dependency injection. The names used for them are:

- Constructor Injection,
- Setter Injection, and
- Interface Injection.

If you read about this stuff in the current discussions about Inversion of Control you'll hear these referred to as

- type 1 IoC (interface injection),
- type 2 IoC (setter injection) and
- type 3 IoC (constructor injection).

You will find numeric names rather hard to remember, which is why the names are used here.

Constructor Injection

Let us start with showing how this injection is done using a lightweight container called PicoContainer. PicoContainer uses a constructor to decide how to inject a finder implementation into the lister class. For this to work, the movie lister class needs to declare a constructor that includes everything it needs injected.

```
class MovieLister...
    public MovieLister(MovieFinder finder) {
        this.finder = finder;
    }
```

The finder itself will also be managed by the pico container, and as such will have the filename of the text file injected into it by the container.

```
class ColonMovieFinder...
    public ColonMovieFinder(String filename) {
        this.filename = filename;
    }
```

The pico container then needs to be told which implementation class to associate with each interface, and which string to inject into the finder.

```
private MutablePicoContainer configureContainer() {  
    MutablePicoContainer pico = new DefaultPicoContainer();  
    Parameter[] finderParams = {new ConstantParameter("movies1.txt")};  
    pico.registerComponentImplementation(MovieFinder.class, ColonMovieFinder.class, finderParams);  
    pico.registerComponentImplementation(MovieLister.class);  
    return pico;  
}
```

This configuration code is typically set up in a different class. For our example, everyone who uses our lister might write the appropriate configuration code in some setup class of their own. Of course it's common to hold this kind of configuration information in separate config files. You can write a class to read a config file and set up the container appropriately.

Although PicoContainer doesn't contain this functionality itself, there is a closely related project called NanoContainer that provides the appropriate wrappers to allow you to have XML configuration files. Such a nano container will parse the XML and then configure an underlying pico container. The philosophy of the project is to separate the config file format from the underlying mechanism.

To use the container you write code something like this.

```
public void testWithPico() {  
    MutablePicoContainer pico = configureContainer();  
    MovieLister lister = (MovieLister) pico.getComponentInstance(MovieLister.class);  
    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");  
    assertEquals("Once Upon a Time in the West", movies[0].getTitle());  
}
```

Setter Injection with Spring

To get the movie lister to accept the injection we define a setting method for that service

```
class MovieLister...
private MovieFinder finder;

public void setFinder(MovieFinder finder) {
    this.finder = finder;
}
```

Similarly we define a setter for the filename.

```
class ColonMovieFinder...
public void setFilename(String filename) {
    this.filename = filename;
}
```

The third step is to set up the configuration for the files. Spring supports configuration through XML files and also through code, but XML is the expected way to do it.

```
<beans>
<bean id="MovieLister" class="spring.MovieLister">
<property name="finder">
<ref local="MovieFinder"/>
</property>
</bean>
<bean id="MovieFinder" class="spring.ColonMovieFinder">
<property name="filename">
<value>movies1.txt</value>
</property>
</bean>
</beans>
```

The test then looks like this.

```
MovieLister lister = (MovieLister) getBean("MovieLister");
Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
assertEquals("Once Upon a Time in the West", movies[0].getTitle());
```

Interface Injection

The third injection technique is to define and use interfaces for the injection. Avalon is an example of a framework that uses this technique in places.

With this technique we begin by defining an interface that we'll use to perform the injection through. Here's the interface for injecting a movie finder into an object.

```
public interface InjectFinder {  
    void injectFinder(MovieFinder finder);  
}
```

This interface would be defined by whoever provides the MovieFinder interface. It needs to be implemented by any class that wants to use a finder, such as the lister.

```
class MovieLister implements InjectFinder...  
public void injectFinder(MovieFinder finder) {  
    this.finder = finder;  
}
```

We may use a similar approach to inject the filename into the finder implementation.

```
public interface InjectFinderFilename {  
    void injectFilename (String filename);  
}  
class ColonMovieFinder implements MovieFinder, InjectFinderFilename.....  
public void injectFilename(String filename) {  
    this.filename = filename;  
}
```

Then, as usual, we need some configuration code to wire up the implementations.

```
class Tester...  
private Container container;  
  
private void configureContainer() {  
    container = new Container();  
    registerComponents();  
    registerInjectors();  
    container.start();  
}
```

The container way

A container is an abstraction responsible for object management, instantiation and configuration. So you can configure the objects using the container rather than writing client code like factory patterns to implement object management. There are many containers available which can help us manage dependency injection with ease. So rather than writing huge factory codes container identifies the object dependencies and creates and injects them in appropriate objects.

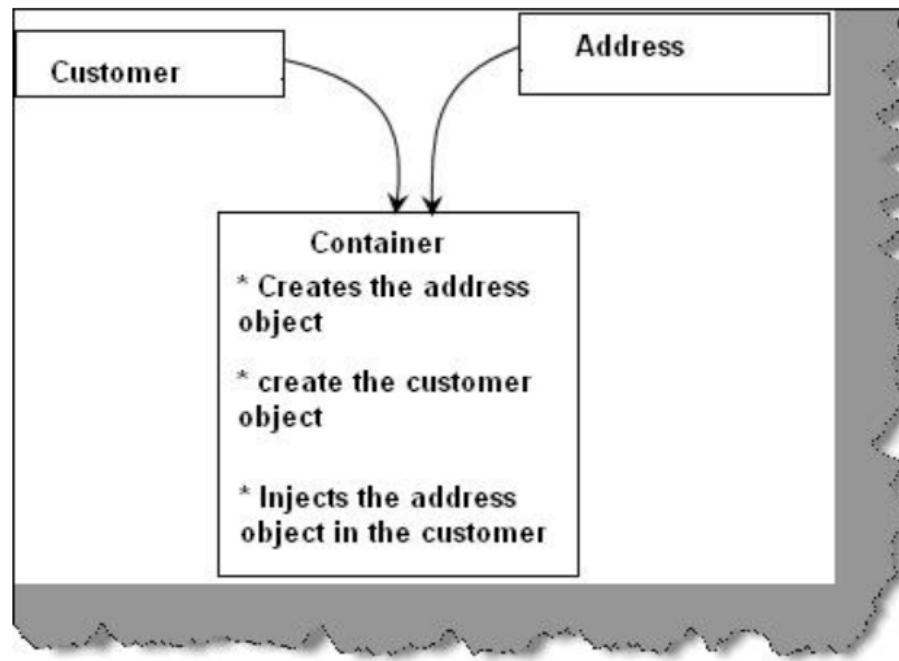


Figure: - Container in action

So you can think about container as a mid man who will register address and customer objects as separate entity and later the container creates the customer and address object and injects the address object in the customer. So you can visualize the high level of abstraction provided by containers.

What we will do is cover the customer and address example using one of the container Windsor container, you can get more details about the container

Spring IoC Container

- Central to the Spring Framework is its Inversion of Control container, which provides a consistent means of configuring and managing Java objects using callbacks.
- The container is responsible for managing object lifecycles: creating objects, calling initialization methods, and configuring objects by wiring them together.
- Objects created by the container are also called Managed Objects or Beans.
- Typically, the container is configured by loading XML files containing Bean definitions which provide the information required to create the beans.
- Objects can be obtained by means of Dependency injection.
- In many cases it's not necessary to use the container when using other parts of the Spring Framework, although using it will likely make an application easier to configure and customize.
- The Spring container provides a consistent mechanism to configure applications and integrates with almost all Java environments, from small-scale applications to large enterprise applications.
- The container can be turned into a partially-compliant EJB3 container by means of the Pitchfork project. The Spring Framework is criticized by some as not being standards compliant. However, SpringSource doesn't see EJB3 compliance as a major goal, and claims that the Spring Framework and the container allow for more powerful programming models.

Using JavaBeans Components

Topics Covered:

- Understanding the benefits of beans
- Creating beans
- Installing bean classes on your server
- Accessing bean properties
- Explicitly setting bean properties
- Automatically setting bean properties from request parameters
- Sharing beans among multiple servlets and JSP pages

We'll discuss the third general strategy for inserting dynamic content in JSP pages by means of JavaBeans components.

Simple application or
small development team.



Complex application or
large development team.

- **Call Java code directly.** Place all Java code in JSP page. Appropriate only for very small amounts of code.
- **Call Java code indirectly.** Develop separate utility classes. Insert into JSP page only the Java code needed to invoke the utility classes.
- **Use beans.** Develop separate utility classes structured as beans. Use `jsp:useBean`, `jsp:getProperty`, and `jsp:setProperty` to invoke the code.
- **Use the MVC architecture.** Have a servlet respond to original request, look up data, and store results in beans. Forward to a JSP page to present results. JSP page uses beans.
- **Use the JSP expression language.** Use shorthand syntax to access and output object properties. Usually used in conjunction with beans and MVC.
- **Use custom tags.** Develop tag handler classes. Invoke the tag handlers with XML-like custom tags.

Limiting the Amount of Java Code in JSP Pages

- As we have discussed the benefit of using separate Java classes instead of embedding large amounts of code directly in JSP pages, separate classes are easier to
 - Write
 - Compile
 - Test
 - Debug
 - Reuse

what do beans provide that other classes do not?

- After all, beans are merely regular Java classes that follow some simple conventions defined by the JavaBeans specification:
 - beans extend no particular class
 - are in no particular package, and
 - use no particular interface.
- Although it is true that beans are merely Java classes that are written in a standard format, there are several advantages to their use.
- With beans in general, visual manipulation tools and other programs can automatically discover information about classes that follow this format and can create and manipulate the classes without the user having to explicitly write any code.

advantages of using JavaBeans components over scriptlets

- **No Java syntax.**
 - By using beans, page authors can manipulate Java objects using only XML-compatible syntax: no parentheses, semicolons, or curly braces. This promotes a stronger separation between the content and the presentation and is especially useful in large development teams that have separate Web and Java developers.
- **Simpler object sharing.**
 - When you use the JSP bean constructs, you can much more easily share objects among multiple pages or between requests than if you use the equivalent explicit Java code.
- **Convenient correspondence between request parameters and object properties.**
 - The JSP bean constructs greatly simplify the process of reading request parameters, converting from strings, and putting the results inside objects.

What Are Beans?

- Beans are simply Java classes that are written in a standard format to expose data through properties (attributes).
- Full coverage of JavaBeans is beyond the scope of this class, but for the purposes of use in JSP, all you need to know about beans are the three simple points outlined in the following list.

1. A bean class must have a zero-argument (default) constructor.

- You can satisfy this requirement either
 - by explicitly defining such a constructor or
 - by omitting all constructors

Note:

- "default constructor" refers to a nullary constructor that is automatically generated by the compiler if no constructors have been defined for the class.
- The default constructor is also empty, meaning that it does nothing.
- A user defined constructor that takes no parameters is called a default constructor too.

2. A bean class should have no public instance variables (fields).

- To be a bean that is accessible from JSP, a class should use accessor methods instead of allowing direct access to the instance variables.
 - You should already be familiar with this practice since it is an important design strategy in object-oriented programming.

3. Persistent values should be accessed via ***getXxx*** and ***setXxx***.

- For example,
 - if your Car class stores the current number of passengers, you might have methods named `getNumPassengers` and `setNumPassengers`.
 - In such a case, the Car class is said to have a *property named numPassengers*.
- If the class has a ***getXxx*** method but no ***setXxx***, *the class is said to have a read-only property named xxx*.
- The one exception to this naming convention is with boolean properties: they are permitted to use a method called ***isXxx*** to look up their values.
- So, for example, your Car class might have methods called `isLeased` and `setLeased`, and would be said to have a boolean property named `leased`



Although you can use JSP scriptlets or expressions to access arbitrary methods of a class, standard JSP actions for accessing beans can only make use of methods that use the `getXxx`, `setXxx` or `isXxx`, `setXxx` naming convention.

Building a JavaBean



So what does a JavaBean look like?

- For this example, we'll define a JavaBean class called CarBean that we could use as a component in a car sales Web site.

- It'll be a component that will model a car and have one property—the make of the car.

Here's the code:

```
package packageName;

public class CarBean
{
    private String make = "Ford";

    public CarBean()  {}

    public String getMake()
    {
        return make;
    }

    public void setMake(String make)
    {
        this.make = make;
    }
}
```

Using Beans – Basic Tasks

- 
- **jsp:useBean**
 - **jsp:getProperty**
 - **jsp:setProperty.**

jsp : useBean

- In the simplest case, this element builds a new bean.
- It is normally used as follows:

```
<jsp:useBean id="beanName" class="package.ClassName" scope="scopeType" />
```

- If you supply a scope attribute, the jsp:useBean element can either build a new bean or access a preexisting one.

jsp : getProperty

- This element reads and outputs the value of a bean property.
- Reading a property is a shorthand notation for calling a method of the form getXxx
- *This element is used as follows:*

```
<jsp:getProperty name="beanName" property="propertyName" />
```

jsp : setProperty

- This element modifies a bean property (i.e., calls a method of the form `setXxx`).
- *It is normally used as follows:*

```
<jsp:setProperty name="beanName" property="propertyName" value="propertyValue" />
```

Installing Bean Classes



- The bean class definition should be placed in the same directories where servlets can be installed, *not in the directory that contains the JSP file.*
- *Just remember to use packages.*

In-class exercise: creating a simple bean

```
package com.me.cars;

public class CarBean
{
    private String make = "Ford";

    public CarBean() { }

    public String getMake()
    {
        return make;
    }

    public void setMake(String make)
    {
        this.make = make;
    }
}
```

Now, create a JSP page to use the Bean

```
<jsp:useBean id="myCar" class="com.yusuf.cars.CarBean" />

<html>
  <head>
    <title>Using a JavaBean</title>
  </head>
  <body>

    <h2>Using a JavaBean</h2>

    I have a <jsp:getProperty name="myCar" property="make" /><br>
    <jsp:setProperty name="myCar" property="make" value="Ferrari" />

    Now I have a <jsp:getProperty name="myCar" property="make" />

  </body>
</html>
```

Here is the output



Sharing Beans

- We have treated the objects that were created with *jsp:useBean*
- As though they were simply bound to local variables in the `_jspService` method (which is called by the `service` method of the servlet that is generated from the page).
- Although the beans are indeed bound to local variables, that is not the only behavior.
- They are also stored in one of four different locations, depending on the value of the optional `scope` attribute of *jsp:useBean*.

```
<jsp:useBean ... scope="page" />           (default)  
<jsp:useBean ... scope="request" />  
<jsp:useBean ... scope="session" />  
<jsp:useBean ... scope="application" />
```

Using scope

- When you use scope, the system first looks for an existing bean of the specified name in the designated location.
- Only when the system fails to find a preexisting bean, it creates a new one.
- This behavior lets a servlet handle complex user requests by
 - setting up beans,
 - storing them in one of the standard shared locations (the request, the session, or the servlet context),
 - then forwarding the request to one of several possible JSP pages to present results appropriate to the request data.
- We'll discuss this approach (Model View Controller Architecture) next.

JavaBeans or Enterprise JavaBeans?

- **EJBs are an advanced topic and beyond the scope of this class.**
- JSP is one part of the Java 2 Enterprise Edition (J2EE) architecture, namely the *presentation tier*.
- Enterprise JavaBeans (EJBs) are another part of this architecture.
- However, as you create bigger and better JSP Web applications, you'll inevitably come across the term.
- We want to warn you not to confuse the JavaBeans you've learned about in this lecture with EJBs.
- Although they share a similar name, they have very different capabilities, designs, and uses.
- JavaBeans are general-purpose components that can be used in a variety of different applications, from very simple to very complex.
- EJBs, on the other hand, are components designed for use in complex business applications.
They support features commonly used in these types of programs, such as automatically saving and retrieving information to a database, performing many tasks in a single transaction that can be safely aborted if parts of the transaction fail, or communicating other Java components across a network and so on
- Although you could accomplish any one of these EJB features with normal JavaBeans, EJBs make using these complex features easier.
- *However, EJBs are considerably more complicated to understand, use, and maintain than JavaBeans, and you'll have your hands full learning JSP and Servlets in this class, so we won't discuss EJBs.*

Summary



- JavaBeans are a simple but helpful addition to JSP.
- As used by JSP, a JavaBean is really nothing more than a fancy name for a way to code a Java class.
- By following certain design restrictions, it is easy to create a set of JSP actions that can manipulate and use any of those classes.
- JavaBeans are an example of a set of design restrictions, primarily get and set methods, and the JavaBean standard actions are available for use with JSP.

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the [JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999](#). JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

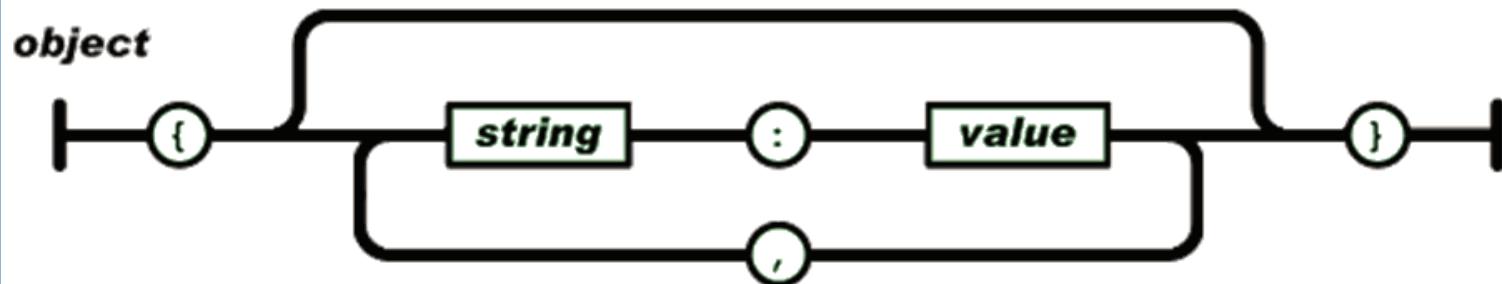
JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

In JSON, they take on these forms:

An *object* is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).



The following example shows the JSON representation of an object that describes a person.

```
{  
    "firstName": "John",  
    "lastName": "Smith",  
    "age": 25,  
    "address": {  
        "streetAddress": "21 2nd Street",  
        "city": "New York",  
        "state": "NY",  
        "postalCode": "10021"  
    },  
    "phoneNumber": [  
        { "type": "home", "number": "212 555-1234" },  
        { "type": "fax", "number": "646 555-4567" }  
    ]  
}
```

A possible equivalent for the above in XML could be:

```
<Person>  
    <firstName>John</firstName>  
    <lastName>Smith</lastName>  
    <age>25</age>  
    <address>  
        <streetAddress>21 2nd Street</streetAddress>  
        <city>New York</city>  
        <state>NY</state>  
        <postalCode>10021</postalCode>  
    </address>  
    <phoneNumber type="home">212 555-1234</phoneNumber>  
    <phoneNumber type="fax">646 555-4567</phoneNumber>  
</Person>
```

JSON in JavaScript

- JSON is a subset of the object literal notation of JavaScript. Since JSON is a subset of JavaScript, it can be used in the language with no muss or fuss.

```
var myJSONObject = {"bindings": [  
    {"ircEvent": "PRIVMSG", "method": "newURI", "regex": "^http://.*"},  
    {"ircEvent": "PRIVMSG", "method": "deleteURI", "regex": "^delete.*"},  
    {"ircEvent": "PRIVMSG", "method": "randomURI", "regex": "^random.*"}  
];};
```

- In this example, an object is created containing a single member "bindings", which contains an array containing three objects, each containing "ircEvent", "method", and "regex" members.
- Members can be retrieved using dot or subscript operators.

myJSONObject.bindings[0].method // "newURI"

```
<%@page contentType="text/html; charset=UTF-8"%>
<%@page import="org.json.simple.JSONObject"%>
<%
    JSONObject obj=new JSONObject();
    obj.put("name","foo");
    obj.put("num",new Integer(100));
    obj.put("balance",new Double(1000.21));
    obj.put("is_vip",new Boolean(true));
    obj.put("nickname",null);
    out.print(obj);
    out.flush();
%>
```

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
</head>

<script type="text/javascript">
function createXMLHttpRequest() {
    // See http://en.wikipedia.org/wiki/XMLHttpRequest
    // Provide the XMLHttpRequest class for IE 5.x-6.x:
    if( typeof XMLHttpRequest == "undefined" ) XMLHttpRequest = function() {
        try { return new ActiveXObject("Msxml2.XMLHTTP.6.0") } catch(e) {}
        try { return new ActiveXObject("Msxml2.XMLHTTP.3.0") } catch(e) {}
        try { return new ActiveXObject("Msxml2.XMLHTTP") } catch(e) {}
        try { return new ActiveXObject("Microsoft.XMLHTTP") } catch(e) {}
        throw new Error( "This browser does not support XMLHttpRequest." )
    };
    return new XMLHttpRequest();
}

var AJAX = createXMLHttpRequest();

function handler() {
    if(AJAX.readyState == 4 && AJAX.status == 200) {
        var json = eval('(' + AJAX.responseText + ')');
        alert('Success. Result: name => ' + json.name + ', ' + 'balance => ' + json.balance);
    }else if (AJAX.readyState == 4 && AJAX.status != 200) {
        alert('Something went wrong...');
    }
}

function show() {
    AJAX.onreadystatechange = handler;
    AJAX.open("GET", "service.jsp");
    AJAX.send("");
};
</script>

<body>
    <a href="#" onclick="javascript:show();"> Click here to get JSON data from the server side
</body>
</html>
```

JSP Introduction and Overview

- ▶ Understanding the need for JSP
- ▶ Evaluating the benefits of JSP
- ▶ Comparing JSP to other technologies
- ▶ JSP Elements



What are JavaServer Pages?

- ▶ JSP is a specification to create dynamic web pages based on the servlet specifications
- ▶ Server side processing
- ▶ Separates the graphical design from the dynamic content



Is JSP meant to replace Servlets?



JSP is not meant to replace Servlets

- ▶ JSP is an extension of the servlet technology, and it is a common practice to use both servlets and JSP pages in the same web application (MVC)
- ▶ Servlet technology may be efficient, scalable, platform independent, and buzzword compliant
 - ▶ but it is far from practical when building Web applications
- ▶ Servlets become too inflexible to survive in the dynamic environment of a Web application when used in generating the user interface
- ▶ JSP is a way to counter the shortcomings of servlets



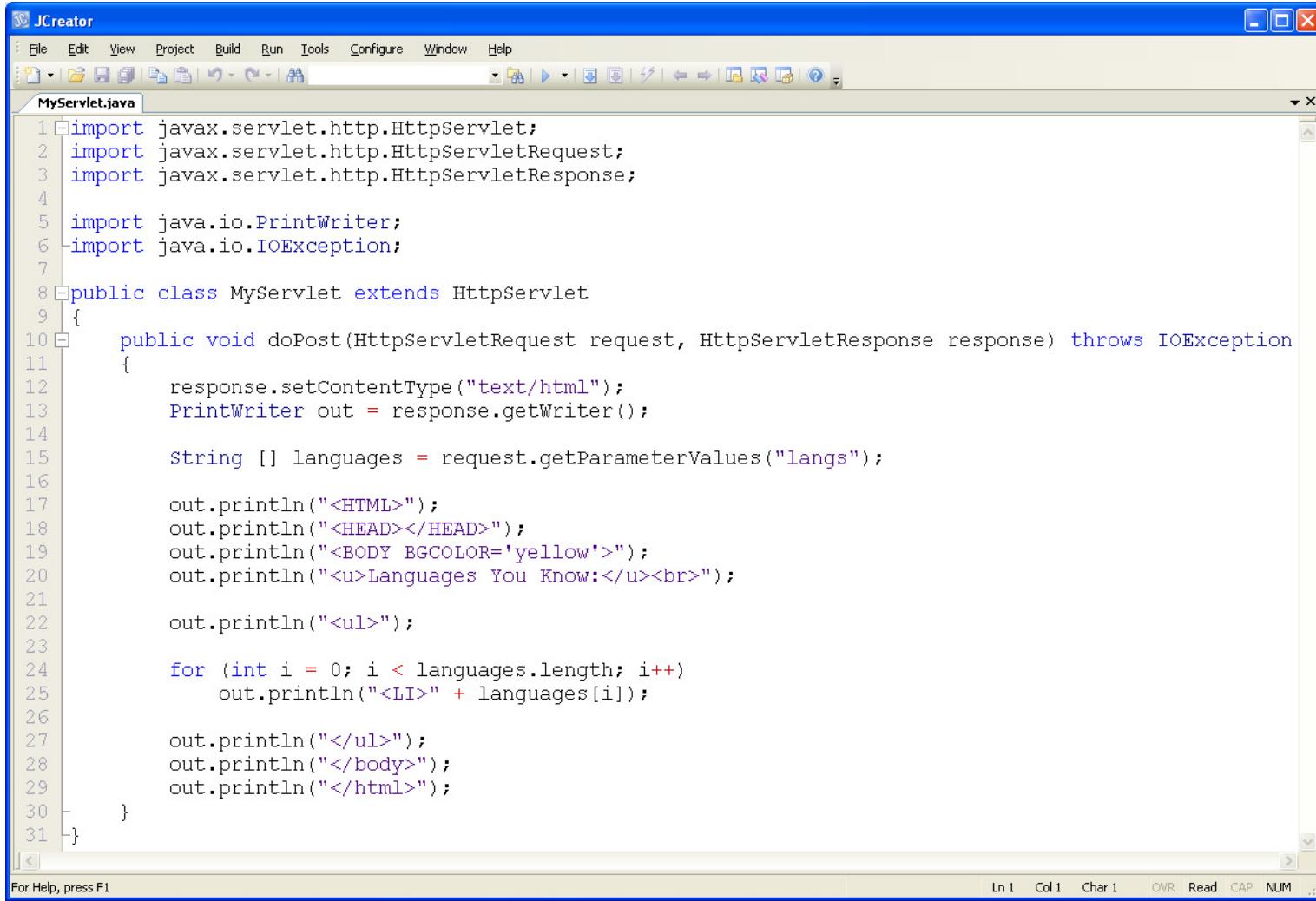
What is wrong with Servlets?



-
- ▶ Servlet programmers know how cumbersome it is to program with Servlets, especially when you have to send a long HTML page that includes little code.



Take a look at the following Servlet code



The screenshot shows the JCreator IDE interface with a Java file named "MyServlet.java" open. The code implements a servlet that prints a list of languages to the response. The code uses standard Java imports for servlet components and the PrintWriter class. It sets the response content type to text/html, gets the writer, and prints an HTML document with a yellow background color. The document contains a heading and a list of languages.

```
1 import javax.servlet.http.HttpServlet;
2 import javax.servlet.http.HttpServletRequest;
3 import javax.servlet.http.HttpServletResponse;
4
5 import java.io.PrintWriter;
6 import java.io.IOException;
7
8 public class MyServlet extends HttpServlet
9 {
10     public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException
11     {
12         response.setContentType("text/html");
13         PrintWriter out = response.getWriter();
14
15         String [] languages = request.getParameterValues("langs");
16
17         out.println("<HTML>");
18         out.println("<HEAD></HEAD>");
19         out.println("<BODY BGCOLOR='yellow'>");
20         out.println("<u>Languages You Know:</u><br>");
21
22         out.println("<ul>");
23
24         for (int i = 0; i < languages.length; i++)
25             out.println("<LI>" + languages[i]);
26
27         out.println("</ul>");
28         out.println("</body>");
29         out.println("</html>");
30     }
31 }
```

For Help, press F1 Ln 1 Col 1 Char 1 OVR Read CAP NUM .

-
- ▶ More than half of the content sent from doPost method is static HTML.
 - ▶ However, each HTML tag must be embedded in a String and sent using the println method of the PrintWriter object.
 - ▶ It is a tedious chore
 - ▶ Worse still, the HTML page may be much longer



What if we want to change the BGCOLOR to #FF0000?

- ▶ Another disadvantage of using Servlets is that every single change will require the intervention of the Servlet programmer.
- ▶ Even a slight modification, such as changing the value of bgcolor, will need to be done by the programmer.



What is the solution?

- ▶ Sun understood this problem and soon developed a solution.
- ▶ The result was JSP technology.
- ▶ According to the Sun's website, "JSP technology is an extension of the Servlet technology created to support authoring of HTML."
- ▶ JSP solves drawbacks in the Servlet technology by allowing the programmer to intersperse code with static content, for example.
- ▶ If the programmer has to work with an HTML page template written by a web designer, the programmer can simply add code into the HTML page, and save it as a .jsp file.
- ▶ If at a later stage the web designer needs to change the HTML body background color, he or she can do it without wasting the programmer's time. He or she can just open the .jsp file and edit it accordingly.



Demo

- ▶ Working with an HTML page template written by a web designer, the programmer can simply add code into the HTML page, and save it as a .jsp file.



With servlets, it is easy to

- ▶ Read form data
- ▶ Read HTTP request headers
- ▶ Set HTTP status codes and response headers
- ▶ Use cookies and session tracking
- ▶ Share data among servlets
- ▶ Remember data between requests
- ▶ Get fun, high-paying jobs



With servlets, it sure is a pain to

- ▶ Use those `println` statements to generate HTML
- ▶ Maintain that HTML

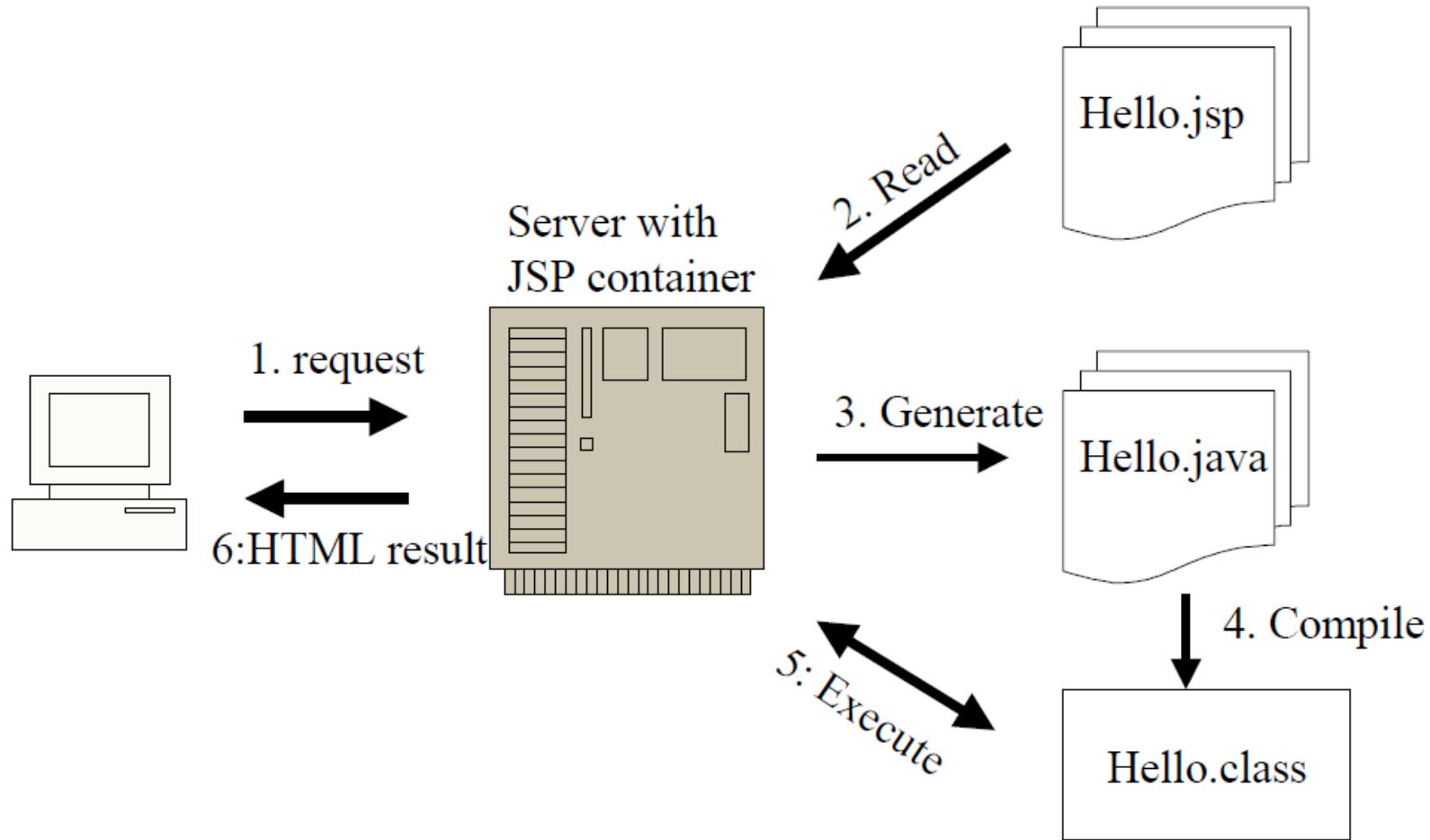


Architectural Overview

- ▶ JSP page is a simple text file consisting of HTML or XML content along with JSP elements (sort of shorthand for Java code)
- ▶ Inside the JSP container is a special servlet called the *page compiler*.
- ▶ The Servlet container is configured to forward to this page compiler all HTTP requests with URLs that match the .jsp extension.
- ▶ When a .jsp page is first called, the page compiler parses and compiles the .jsp page into a Servlet class.
- ▶ If compilation is successful, the jsp servlet class is loaded into the memory.



JSP Processing



- If JSP is already compiled, compilation is skipped

Why do you think that after the deployment,
the first user requests for a .jsp page will
experience slow response?



-
- ▶ It is because of the time spent for compiling .jsp page into a servlet class.



Confusing Translation Time with Request Time

- ▶ A JSP page is converted into a servlet. The servlet is compiled, loaded into the server's memory, initialized, and executed.
- ▶ But which step happens when?
 - ▶ To answer that question, remember two points:
 - ▶ The JSP page is translated into a servlet and compiled only the first time it is accessed after having been modified.
 - ▶ Loading into memory, initialization, and execution follow the normal rules for servlets.



JSP page translated into servlet	Servlet compiled	Servlet loaded into server's memory	<code>jspInit</code> called	<code>_jspService</code> called
----------------------------------	------------------	-------------------------------------	-----------------------------	---------------------------------

Page first written

Request 1	Yes	Yes	Yes	Yes	Yes
Request 2	No	No	No	No	Yes

Server restarted

Request 3	No	No	Yes	Yes	Yes
Request 4	No	No	No	No	Yes

Page modified

Request 5	Yes	Yes	Yes	Yes	Yes
Request 6	No	No	No	No	Yes



JSP vs Servlets

- ▶ JSP pages are translated into servlets. So, fundamentally, any task JSP pages can perform could also be accomplished by servlets.
- ▶ However, this underlying equivalence does not mean that servlets and JSP pages are equally appropriate in all scenarios.
- ▶ The issue is not the power of the technology, it is the convenience, productivity, and maintainability of one or the other.
- ▶ After all, anything you can do on a particular computer platform in the Java programming language you could also do in assembly language.
- ▶ But it still matters which you choose.



JSP provides the following benefits over servlets alone:

- ▶ **It is easier to write and maintain the HTML.**
 - ▶ Your static code is ordinary HTML: no extra backslashes, no double quotes, and no lurking Java syntax.
- ▶ **You can use standard Web-site development tools.**
 - ▶ For example, most people use Dreamweaver or FrontPage for creating JSP pages. Even HTML tools that know nothing about JSP can be used because they simply ignore the JSP tags.
- ▶ **You can divide up your development team.**
 - ▶ The Java programmers can work on the dynamic code. The Web developers can concentrate on the presentation layer. On large projects, this division is very important. Depending on the size of your team and the complexity of your project, you can enforce a weaker or stronger separation between the static HTML and the dynamic content.



-
- ▶ Now, this discussion is not to say that you should stop using servlets and use only JSP instead.
 - ▶ By no means. Almost all projects will use both.
 - ▶ For some requests in your project, you will use servlets.
 - ▶ For others, you will use JSP.
 - ▶ For still others, you will combine them with the MVC architecture (will discuss next week).



JSP developers need to know servlets for four reasons:

- ▶ 1. JSP pages get translated into servlets. You can't understand how JSP works without understanding servlets.
- ▶ 2. JSP consists of static HTML, special-purpose JSP tags, and Java code.
 - ▶ What kind of Java code?
 - ▶ Servlet code! You can't write that code if you don't understand servlet programming.
- ▶ 3. Some tasks are better accomplished by servlets than by JSP.
 - ▶ JSP is good at generating pages that consist of large sections of fairly well structured HTML or other character data.
 - ▶ Servlets are better for generating binary data, building pages with highly variable structure, and performing tasks (such as redirection) that involve little or no output.
- ▶ 4. Some tasks are better accomplished by a combination of servlets and JSP than by either servlets or JSP alone (MVC)



JSP vs. JavaScript

- ▶ Java Scripts provide client-side execution ability
 - ▶ Interpreted
 - ▶ Cumbersome and error prone
 - ▶ Non-portable
- ▶ JSP provide server-side execution
 - ▶ Compiled
 - ▶ Portable
 - ▶ Robust
 - ▶ Not integrated with HTML – Java creates HTML



JSP vs. Servlets

▶ Similarities

- ▶ Server-side execution
- ▶ Provide identical results to the end user
- ▶ JSP will convert to Servlet.

▶ Differences

- ▶ Servlet: “HTML in Java Code”
 - ▶ Mixes presentation with logic
- ▶ JSP: “Java Code Scriptlets in HTML”
 - ▶ Separates presentation from logic



JSP vs. ASP

- ▶ **Similarities**

- ▶ Server-side execution
- ▶ Separates presentation from logic
- ▶ JSP is Java's answer to ASP.

- ▶ **Differences**

- ▶ **ASP**

- ▶ Microsoft programming language, such as VB and etc.
- ▶ Microsoft platform
- ▶ Microsoft Product
- ▶ Although ASP are cached, they are always interpreted.

- ▶ **JSP**

- ▶ Java technology
- ▶ Platform independent
- ▶ Specification
- ▶ Compiled



JSP vs PHP or ColdFusion

- ▶ Better language for dynamic part
- ▶ Portable to multiple servers and operating systems



Different ways to generate dynamic content from JSP

Simple application or
small development team.



Complex application or
large development team.

- **Call Java code directly.** Place all Java code in JSP page. Appropriate only for very small amounts of code. This chapter.
- **Call Java code indirectly.** Develop separate utility classes. Insert into JSP page only the Java code needed to invoke the utility classes. This chapter.
- **Use beans.** Develop separate utility classes structured as beans. Use `jsp:useBean`, `jsp:getProperty`, and `jsp:setProperty` to invoke the code. Chapter 14.
- **Use the MVC architecture.** Have a servlet respond to original request, look up data, and store results in beans. Forward to a JSP page to present results. JSP page uses beans. Chapter 15.
- **Use the JSP expression language.** Use shorthand syntax to access and output object properties. Usually used in conjunction with beans and MVC. Chapter 16.
- **Use custom tags.** Develop tag handler classes. Invoke the tag handlers with XML-like custom tags. Volume 2.

-
- ▶ Each of these approaches has a legitimate place; the size and complexity of the project is the most important factor in deciding which approach is appropriate.
 - ▶ However, be aware that people err on the side of placing too much code directly in the page much more often than they err on the opposite end of the spectrum.
 - ▶ Although putting small amounts of Java code directly in JSP pages works fine for simple applications, using long and complicated blocks of Java code in JSP pages yields a result that is
 - ▶ hard to maintain,
 - ▶ hard to debug,
 - ▶ hard to reuse, and
 - ▶ hard to divide among different members of the development team.
-



JSP Scripting Elements

▶ **1. Expressions**

- ▶ `<%= Java Expression %>`
 - ▶ which are evaluated and inserted into the servlet's output.

▶ **2. Scriptlets**

- ▶ `<% Java Code %>`
 - ▶ which are inserted into the servlet's `_jspService` method

▶ **3. Declarations**

- ▶ `<%! Field/Method Declaration %>`
 - ▶ Which are inserted into the body of the servlet class, outside any existing methods.



Limiting the Amount of Java Code in JSP Pages

- ▶ You have 25 lines of Java code that you need to invoke.
You have two options:
 - ▶ (1) put all 25 lines directly in the JSP page,
 - ▶ (2) put the 25 lines of code in a separate Java class, put the Java class in WEB-INF/classes/directoryMatchingPackageName, and use one or two lines of JSP-based Java code to invoke it.
- ▶ Which is better?



Second Option Better

- ▶ The second. The second! And all the more so if you have 50, 100, 500, or 1000 lines of code.
- ▶ **Here's why:**
- ▶ **Development.**
 - ▶ You generally write regular classes in a Java-oriented environment (e.g., an IDE like JBuilder or Eclipse, etc). You generally write JSP in an HTML-oriented environment like Dreamweaver. The Java-oriented environment is typically better at balancing parentheses, providing tooltips, checking the syntax, colorizing the code, and so forth.
- ▶ **Compilation.**
 - ▶ To compile a regular Java class, you press the Build button in your IDE or invoke `javac`.
 - ▶ To compile a JSP page, start the server, open a browser, and enter the appropriate URL.
- ▶ **Debugging.**
 - ▶ We know this never happens to you, but when we *write* Java classes or JSP pages, we occasionally make syntax errors. If there is a syntax error in a regular class definition, the compiler tells you right away and it also tells you what line of code contains the error. If there is a syntax error in a JSP page, the server typically tells you what line of *the servlet* (*i.e.*, *the servlet into which the JSP page was translated*) contains the error. For tracing output at runtime, with regular classes you can use simple `System.out.println` statements if your IDE provides nothing better. In JSP, you can sometimes use print statements, but where those print statements are displayed varies from server to server.



- ▶ **Division of labor**

- ▶ Many large development teams are composed of some people who are experts in the Java language and others who are experts in HTML but know little or no Java. The more Java code that is directly in the page, the harder it is for the Web developers (the HTML experts) to manipulate it.

- ▶ **Testing**

- ▶ Suppose you want to make a JSP page that outputs random integers between designated 1 and some bound (inclusive). You use Math.random, multiply by the range, cast the result to an int, and add 1.
 - ▶ Hmm, that sounds right. But are you sure? If you do this directly in the JSP page, you have to invoke the page over and over to see if you get all the numbers in the designated range but no numbers outside the range. After hitting the Reload button a few dozen times, you will get tired of testing.
 - ▶ But, if you do this in a static method in a regular Java class, you can write a test routine that invokes the method inside a loop, and then you can run hundreds or thousands of test cases with no trouble. For more complicated methods, you can save the output, and, whenever you modify the method, compare the new output to the previously stored results.

- ▶ **Reuse**

- ▶ You put some code in a JSP page. Later, you discover that you need to do the same thing in a different JSP page. What do you do? Cut and paste?
 - ▶ Repeating code in this manner is a cardinal sin because if you change your approach, you have to change many different pieces of code. Solving the code reuse problem is what object-oriented programming is all about. Don't forget all your good OOP principles just because you are using JSP to simplify the generation of HTML.



Limit the amount of Java code that is in JSP pages

- ▶ “But wait!” you say, “I have an IDE that makes it easier to develop, debug, and compile JSP pages.” OK, good point. There is no hard and fast rule for exactly how much Java code is too much to go directly in the page.
 - ▶ But no IDE solves the testing and reuse problems, and your general design strategy should be centered around putting the complex code in regular Java classes and keeping the JSP pages relatively simple.
- ▶ Almost all experienced developers have seen gross excesses: JSP pages that consist of many lines of Java code followed by tiny snippets of HTML. That is obviously bad: it is harder to
 - ▶ develop, compile, debug, divvy up among team members, test, and reuse.
- ▶ A servlet would have been far better.
 - ▶ However, some of these developers have overreacted by flatly stating that it is *always wrong to have any Java code directly in the JSP page*.
 - ▶ Certainly, on some projects it is worth the effort to keep a strict separation between the content and the presentation and to enforce a style where there is no Java syntax in any of the JSP pages.
- ▶ But this is not always necessary (or even beneficial).
 - ▶ A few people go even further by saying that *all pages in all applications should use the Model-View-Controller (MVC) architecture*, preferably with the Apache Struts framework. This is also an overreaction.
 - ▶ Yes, MVC (will be discussed next week) is a great idea, and we use it all the time on real projects.
 - ▶ And, yes, Struts is a nice framework; we are using it on a large project as the book is going to press.
- ▶ The approaches are great when the situation gets moderately (MVC in general) or highly (Struts) complicated.
- ▶ But simple situations call for simple solutions.
 - ▶ all the approaches have a legitimate place; it depends mostly on the complexity of the application and the size of the development team.
 - ▶ Still, be warned: beginners are much more likely to err by making hard-to-manage JSP pages chock-full of Java code than they are to err by using unnecessarily large and elaborate frameworks.



Using JSP Expressions

- ▶ A JSP expression is used to insert values directly into the output. It has the following form:

<%= Java Expression %>

The expression is evaluated, converted to a string, and inserted in the page. This evaluation is performed at runtime (when the page is requested) and thus has full access to information about the request.

- ▶ For example, the following shows the date/time that the page was requested.

Current time: <%= new java.util.Date() %>



JSP/Servlet Correspondence

- ▶ Now, we just stated that a JSP expression is evaluated and inserted into the page output.
- ▶ Although this is true, it is sometimes helpful to understand what is going on behind the scenes.
- ▶ It is actually quite simple: JSP expressions basically become print (or write) statements in the servlet that results from the JSP page. Whereas regular HTML becomes print statements with double quotes around the text, JSP expressions become print statements with no double quotes. Instead of being placed in the doGet method, these print statements are placed in a new method called `_jspService` that is called by service for both GET and POST requests.



Seeing the exact code that your server generates



Example: JSP Expressions

```
<HTML>
<HEAD>
<TITLE>JSP Expressions</TITLE>
</HEAD>
<BODY>
<H2>JSP Expressions</H2>
<UL>
<LI>Current time: <%= new java.util.Date() %>
<LI>Server:    <%= application.getServerInfo() %>
<LI>Session ID: <%= session.getId() %>
<LI>The testParam form parameter: <%= request.getParameter("testParam") %>
</UL>
</BODY>
</HTML>
```



Writing Scriptlets

If you want to do something more complex than output the value of a simple expression, JSP scriptlets let you insert arbitrary code into the servlet's `_jspService` method (which is called by `service`).

Scriptlets have the following form:

```
<% Java Code %>
```

Scriptlets have access to the same automatically defined variables as do expressions (`request`, `response`, `session`, `out`, etc.). So, for example, if you want to explicitly send output to the resultant page, you could use the `out` variable, as in the following example.

```
<%  
String queryData = request.getQueryString();  
out.println("Attached GET data: " + queryData);  
%>
```



JSP/Servlet Correspondence

It is easy to understand how JSP scriptlets correspond to servlet code: the scriptlet code is just directly inserted into the `_jspService` method: no strings, no print statements, no changes whatsoever.



Scriptlet Example

```
<HTML>
<HEAD>
<TITLE>Color Testing</TITLE>
</HEAD>
<%
    String bgColor = request.getParameter("bgColor");
    if ((bgColor == null) || (bgColor.trim().equals("")))
    {
        bgColor = "WHITE";
    }
%>
<BODY BGCOLOR="<%= bgColor %>">
<H2 ALIGN="CENTER">Testing a Background of "<%= bgColor %>"</H2>
</BODY>
</HTML>
```



Using Declarations

- ▶ A JSP declaration lets you define methods or fields that get inserted into the main body of the servlet class (*outside the _jspService method that is called by service to process the request*). A declaration has the following form:

`<%! Field or Method Definition %>`

- ▶ Since declarations do not generate output, they are normally used in conjunction with JSP expressions or scriptlets.
- ▶ In principle, JSP declarations can contain field (instance variable) definitions, method definitions, inner class definitions, or even static initializer blocks: anything that is legal to put inside a class definition but outside any existing methods.
- ▶ In practice, however, declarations almost always contain field or method definitions.



JSP/Servlet Correspondence

- ▶ JSP declarations result in code that is placed inside the servlet class definition but outside the `_jspService` method. Since fields and methods can be declared in any order, it does not matter whether the code from declarations goes at the top or bottom of the servlet.



Declaration Example

```
<HTML>
<HEAD>
<TITLE>JSP Declarations</TITLE>
<LINK REL=STYLESCHEET
HREF="JSP-Styles.css"
TYPE="text/css">
</HEAD>
<BODY>
<H1>JSP Declarations</H1>
<%! private int accessCount = 0; %>
<H2>Accesses to page since server reboot:
<%= ++accessCount %></H2>
</BODY></HTML>
```



Architecture of Web Applications

- One approach to Web application development is to gather the requirements and quickly chalk out some JSP pages and JavaBeans.
- Though this approach would probably work to some degree,
 - ▣ the end result is unlikely to be maintainable,
 - ▣ it may well be hard to change a certain feature or to introduce new features without breaking other parts of the application.
 - ▣ It's also probably not *reusable*.
- If you need to introduce new functionality, you'd have to repeat a lot of work that you had already done but that's locked up in JSP pages and beans in such a way that you can't use it.
- By structuring your applications correctly, you can go much further toward achieving the goals of maintainability and reusability.

- When considering how to construct an application, you must decide how to divide up responsibilities and how the pieces will interact with one another.
- In a Web application, that means that you'll assign work to static HTML pages, JSPs, servlets, and other objects. The container and application server will pitch in and take care of some things for you.
- You want applications that are useful, easy to construct, and maintainable.
- Because requirements or uses seem to always change, you're also interested in software that is flexible or extensible.
- **The trick in building software that does all of this for you is to properly delegate the work to the right components.**
- For example, presenting static content to the client can be accomplished using a plain HTML page or a servlet or JSP. Writing a servlet to do this is more work than you need to do and makes the application complex.
- Also, you probably recognize that, in the workplace, employees are often assigned tasks making the best use of their skills.
- From a business perspective, it's useful to design software so that the work can be divided into parcels that are completed by employees whose training and talents are appropriate to the tasks.
- **One of the objectives of the JSP and servlet specifications is to make it possible to neatly divide the labor between graphics designers who work on the presentation and programmers who define the behavior of an application.**

Why Use a Design Pattern?

□ Reduce Development Time:

A good design pattern helps conceptually to break down a complex system into manageable tasks. This allows developers to individually code parts of the Web Application they are best suited for. It also allows individual components to be built and replaced without harming the existing code base.

□ Reduced Maintenance Time:

The majority of projects involve maintenance of an already existing system. Maintenance can be a nightmare should a system be hacked together when it was initially built. Good design plans ahead to simplify future maintenance concerns.

□ Collaboration:

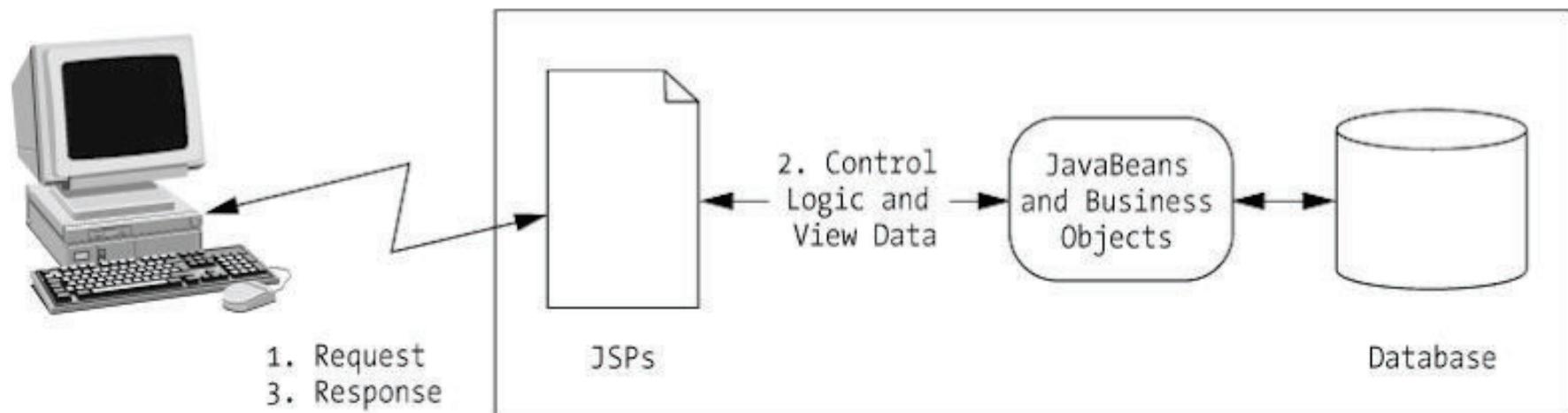
Not all developers share the same expertise. Often a mixed group of developers is assigned to a task especially in the case of a larger project. A good design can successfully enforce separation of a project's functionality into areas that collaborating developers are most familiar, and ensure separate parts of a project seamlessly fit together

Model 1 Architecture

- Model 1 is used to refer to what is usually the intuitive approach to using JSP and Servlets; a Model 1 architecture is what a new JSP and Servlet developer are likely to build.
- The concept behind a Model 1 architecture is simple: code functionality wherever the functionality is needed.
- The approach is very popular because it's both simple and provides instant gratification.
 - Should security be needed, code it in.
 - Should a JSP need information from a database, code in the query.
- In the Model 1 architecture, JSPs accept client requests, decide which actions to take next, and present the results.
- JSPs work with JavaBeans or other services to affect business objects and generate the content.
- **Model 1 is acceptable for applications containing up to a few thousand lines of code, and especially for programmers, but the JSP pages still have to handle the HTTP requests, and this can cause headaches for the page designers.**

Model 1 Architecture

In Model 1 architecture, the application is page-centric. The client browser navigates through a series of JSP pages in which any JSP page can employ a JavaBean that performs business operations. However, the highlight of this architecture is that each JSP page processes its own input. Applications implementing this architecture normally have a series of JSP pages where the user is expected to proceed from the first page to the next. If needed, a servlet or an HTML page can be substituted for the JSP page in the series.



Features of the JSP Model 1 Architecture

- You use HTML or JSP files to code the presentation.
- The JSP files can use JavaBeans or other Java objects to retrieve data if required.
- JSP files are also responsible for all the business and processing logic, such as receiving incoming requests, routing to the correct JSP page, instantiating the correct JSP pages, and so on.
- This means that the Model 1 architecture is a page-centric design: All the business and processing logic is either present in the JSP page itself or called directly from the JSP page.
- Data access is usually performed using custom tags or through JavaBean calls.
- Some quick-and-dirty projects use scriptlets in their JSP pages instead.
- Therefore, there's a tight coupling between the pages and the logic in Model 1 applications.
- The flow of the application goes from one JSP page to another using anchor links in JSP pages or the action attribute of HTML forms.
- This is the only kind of application you've seen so far.

Drawbacks of the JSP Model 1 Architecture

- The Model 1 architecture has one thing going for it:
 - simplicity.
- If your project is small, simple, and self-contained, it's the quickest way to get up and running. But the previous example, although by no means large or complicated, already illustrates a number of the disadvantages of Model 1 architecture:
- It becomes very hard to change the structure of such Web applications because the pages are *tightly coupled*.
- They have to be aware of each other. What if you decide that, after updating the quantities in a shopping cart, you want to redirect the user back to the catalog? This could require moving code from the shopping cart page to the catalog page.
- Large projects often involve teams of programmers working on different pages, and in the Model 1 scenario, each team would have to have a detailed understanding of the pages on which all of the other teams were working; otherwise, modifying pages could break the flow of the application.

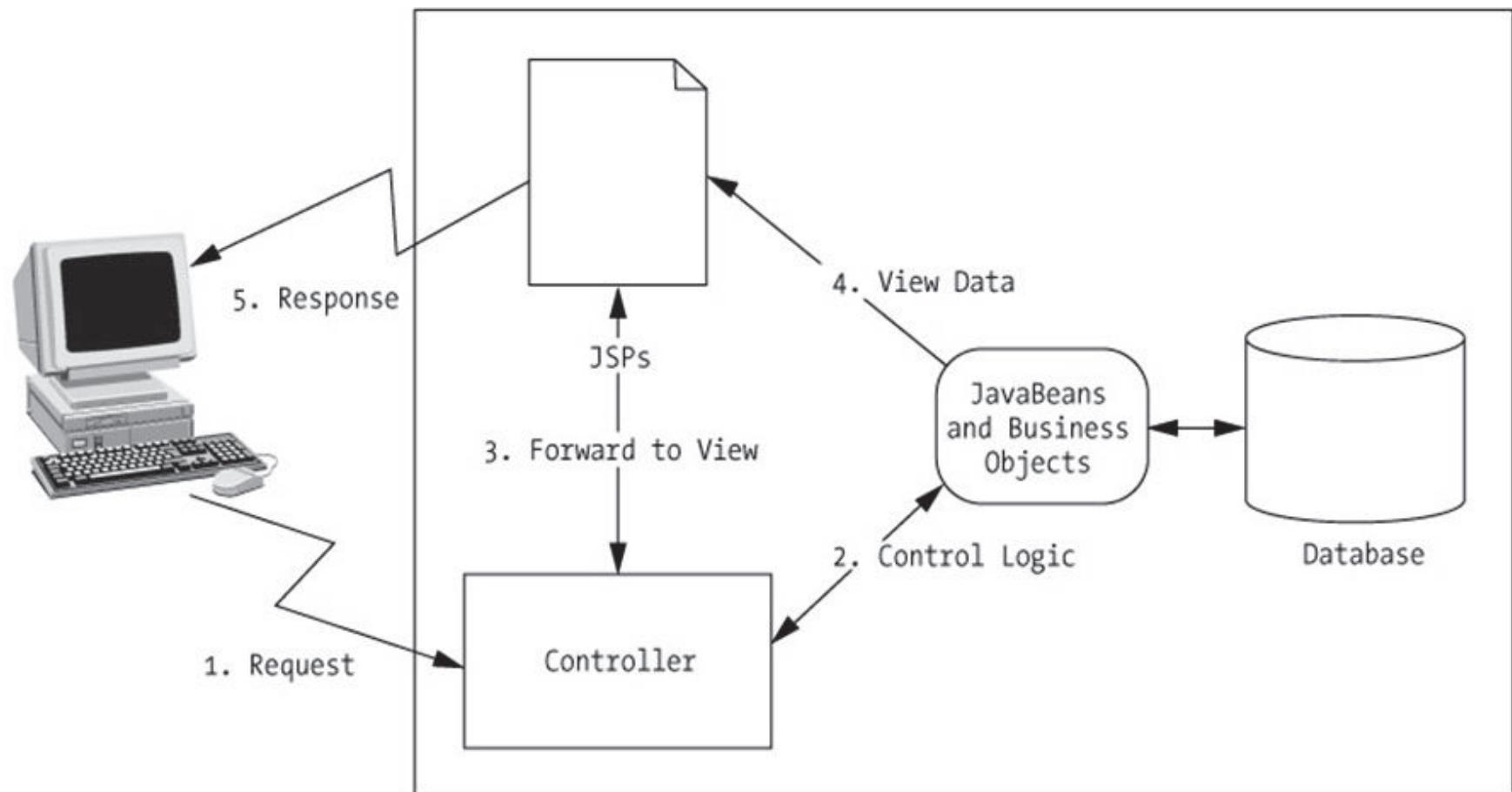
Drawbacks Cont'd

- Pages that are linked to from many other pages have to handle those other pages' logic, such as a cart update. In this way, they can accumulate a large amount of code that hasn't got an awful lot to do with the page itself. This reduces their *coherence*, making them harder to understand and to maintain.
- Presentation and application control logic are mixed up, making it hard for a Web designer to change the pages without messing up the Java code.
- And vice versa: it's very hard for a developer to change control logic that may be hidden in lots of HTML markup.
- The lack of separation between presentation and logic means that providing multiple presentations carries a very high cost.
- What if you decide to sell pizzas via Wireless Application Protocol (WAP) or personal digital assistant (PDA) devices?
 - These devices are radically different from Web browsers and require completely different presentation structure and formatting. Your only choice would be to produce a whole new set of pages and duplicate the application logic.
 - You'd have to implement every subsequent change to the logic in more than one place. Soon enough, changes will become very hard to manage, and the implementations will start to diverge.

JSP Model 2 Architecture

- A better solution, also suitable for larger applications, is to separate application logic and page presentation.
- This solution comes in the form of the JSP Model 2 architecture, also known as the model-view-controller (MVC)
- With this model,
 - ▣ a servlet
 - processes the request,
 - handles the application logic, and
 - instantiates the Java beans.
 - ▣ JSP
 - obtains data from the beans and
 - can format the response without having to know anything about what's going on behind the scenes.

Model 2



Understanding the Need for MVC

- Servlets are great when your application requires a lot of programming to accomplish its task.
- As discussed in the previous lectures, servlets can
 - manipulate HTTP status codes and headers,
 - use cookies,
 - track sessions,
 - save information between requests,
 - compress pages,
 - access databases,
 - generate JPEG images on-the-fly, and
 - perform many other tasks flexibly and efficiently.
- But, generating HTML with servlets can be tedious and can yield a result that is hard to modify.
- That's where JSP comes in, JSP lets you separate much of the presentation from the dynamic content. That way, you can write the HTML in the normal manner, even using HTML-specific tools and putting your Web content developers to work on your JSP documents.
- JSP expressions, scriptlets, and declarations let you insert simple Java code into the servlet that results from the JSP page, and directives let you control the overall layout of the page.
- For more complex requirements, you can wrap Java code inside beans or even define your own JSP tags.

JSP provides a single overall presentation.

- *What if you want to give totally different results depending on the data that you receive?*
 - Scripting expressions, beans, and custom tags, although extremely powerful and flexible, don't overcome the limitation that the JSP page defines a relatively fixed, top-level page appearance.
- Similarly, what if you need complex reasoning just to determine the type of data that applies to the current situation?
 - JSP is poor at this type of business logic.

The solution is to use both servlets and JavaServerPages.

- *In this approach, known as the Model View Controller (MVC) or Model 2 architecture, you let each technology concentrate on what it excels at.*
 - The original request is handled by a servlet.
 - The servlet invokes the business-logic and data-access code and creates beans to represent the results (that's the *model*).
 - *Then, the servlet decides which JSP page is appropriate to present those particular results and forwards the request there (the JSP page is the *view*).*
 - *The servlet decides what business logic code applies and which JSP page should present the results (the servlet is the *controller*).*

Motivation behind the MVC

- The key motivation behind the MVC approach is the desire to separate the code that creates and manipulates the data from the code that presents the data.
- The basic tools needed to implement this presentation-layer separation are standard in the servlet API and are the topic of this lecture.
- However, in very complex applications, a more elaborate MVC framework is sometimes beneficial. Some popular of these frameworks are
 - Apache Struts
 - Spring MVC

Quick summary of the required steps in MVC

1. Define beans to represent the data.

Your first step is define beans to represent the results that will be presented to the user.

2. Use a servlet to handle requests.

In most cases, the servlet reads request parameters as described in previous lectures.

3. Populate the beans.

The servlet invokes business logic (applicationspecific code) or data-access code to obtain the results. The results are placed in the beans that were defined in step 1.

4. Store the bean in the request, session, or servlet context.

The servlet calls setAttribute on the request, session, or servlet context objects to store a reference to the beans that represent the results of the request.

5. Forward the request to a JSP page.

The servlet determines which JSP page is appropriate to the situation and uses the forward method of RequestDispatcher to transfer control to that page.

6. Extract the data from the beans.

The JSP page accesses beans with jsp:useBean and a scope matching the location of step 4.

The page then uses jsp:getProperty to output the bean properties.

The JSP page does not create or modify the bean; it merely extracts and displays data that the servlet created.

Defining Beans to Represent the Data

- Beans are Java objects that follow a few simple conventions.
- In this case, since a servlet or other Java routine (never a JSP page) will be creating the beans, the requirement for an empty (zero-argument) constructor is waived.
- So, your objects merely need to follow the normal recommended practices of keeping the instance variables private and using accessor methods that follow the get/set naming convention.
- Since the JSP page will only access the beans, not create or modify them, a common practice is to define *value objects*: *objects that represent results but have little or no additional functionality*.

Writing Servlets to Handle Requests

- Once the bean classes are defined, the next task is to write a servlet to read the request information.
- Since, with MVC, a servlet responds to the initial request, the normal approaches of previous lectures are used to read request parameters and request headers, respectively.
- Although the servlets use the normal techniques to read the request information and generate the data, they do not use the normal techniques to output the results.
- In fact, with the MVC approach the servlets do not create *any output; the output is* completely handled by the JSP pages.
- So, the servlets do not call `response.setContentType`, `response.getWriter`, or `out.println`.

Populating the Beans

- After you read the form parameters, you use them to determine the results of the request.
- These results are determined in a completely application-specific manner.
- You might call some business logic code, invoke an Enterprise JavaBeans component, or query a database.
- No matter how you come up with the data, you need to use that data to fill in the value object beans that you defined in the first step.

Storing the Results

- You have read the form information.
- You have created data specific to the request.
- You have placed that data in beans.
- Now you need to store those beans in a location that the JSP pages will be able to access
- A servlet can store data for JSP pages in three main places:
 - in the HttpServletRequest,
 - in the HttpSession, and
 - in the ServletContext.
- These storage locations correspond to the three nondefault values of the scope attribute of `jsp:useBean`: that is,
 - `request`,
 - `session`,
 - `application`.

Forwarding Requests to JSP Pages using RequestDispatcher

- You forward requests with the forward method of RequestDispatcher.
- You obtain a RequestDispatcher by calling the getRequestDispatcher method of ServletRequest, supplying a relative address.
- You are permitted to specify addresses in the WEB-INF directory; clients are not allowed to directly access files in WEB-INF, but the server is allowed to transfer control there.
- Using locations in WEB-INF prevents clients from inadvertently accessing JSP pages directly, without first going through the servlets that create the JSP data.
- Once you have a RequestDispatcher, you use forward to transfer control to the associated address. You supply the HttpServletRequest and HttpServletResponse as arguments.

forward vs. redirect

- Note that the `forward` method of `RequestDispatcher` is quite different from the `sendRedirect` method of `HttpServletRequest`.
- With `forward`, there is no extra response/request pair as with `sendRedirect`.
 - Thus, the URL displayed to the client does not change when you use `forward`.
- *When you use the `forward` method of `RequestDispatcher`, the client sees the URL of the original servlet, not the URL of the final JSP page.*

Summary of the behavior of *forward*

- Control is transferred entirely on the server.
- No network traffic is involved.
- The user does not see the address of the destination JSP page and pages can be placed in WEB-INF to prevent the user from accessing them without going through the servlet that sets up the data.
- This is beneficial if the JSP page makes sense only in the context of servlet generated data.

Summary of the behavior of `sendRedirect`

- Control is transferred by sending the client a 302 status code and a Location response header.
- Transfer requires an additional network round trip.
- The user sees the address of the destination page and can bookmark it and access it independently.
 - This is beneficial if the JSP is designed to use default values when data is missing.
- For example, this approach would be used when redisplaying an incomplete HTML form or summarizing the contents of a shopping cart.

Summary:

- We have covered the two models of Java web application design:
 - Model 1 and Model 2
- Model 1 architecture provides rapid development for small projects, and is suitable for small projects that will remain small or for building prototypes.
- Model 2 is the recommended architecture for any medium-sized to large projects.
- Model 2 is harder to build, but it provides more maintainability and extensibility.
- In very complex applications, a more elaborate MVC framework is sometimes beneficial, such as *Apache Struts, and Spring MVC*
- Although Struts, and Spring MVC are useful and widely used, you should not feel that you must use Struts or Spring MVC in order to apply the MVC approach.
- For simple and moderately complex applications, implementing MVC from scratch with RequestDispatcher is straightforward and flexible.

Session Management



In this lecture, you will learn

- HTTP is a stateless protocol, and the implications of this statelessness
- Techniques you can use to manage user sessions

HTTP Protocol is Stateless

- This simply means that the Hyper Text Transfer Protocol that is the backbone of the Web is unable to retain a memory of the identity of each client that connects to a Web site and therefore treats each request for a Web page as a unique and independent connection, with no relationship whatsoever to the connections that preceded it.
- For viewing statically generated pages the stateless nature of the HTTP protocol is not usually a problem because the page you view will be the same no matter what previous operations you had performed.
- However for applications such as shopping carts which accumulate information as you shop it is extremely important to know what has happened previously, for example what you have in your basket. What is needed for these applications is a way to "maintain state" allowing connections to be tracked so that the application can respond to a request based on what has previously taken place.

To see the problem more clearly, consider the following LoginServlet.

- Assuming that the form uses the POST method, the user information is captured in the doPost method, which does the authentication by calling the login method.
 - If the login is successful, the information is displayed.
 - If not, the login form is displayed again

```
public class LoginServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request, HttpServletResponse response)
    {
        String userName = request.getParameter("userName");
        String password = request.getParameter("password");

        if (login(userName, password))
            //login successful, display the information
        else
            //login failed, resend the login form
    }
}
```

Discussion:

- What if you have another servlet that also only allows authorized users to view the information?

User Authorization



- The second servlet does not know whether the same user has successfully logged in to the first servlet.
- Consequently, the user will be required to login again.

Is this practical?



- This is, of course, is not practical. Every time a user goes to request a protected servlet, the user has to login again even though all the servlets are part of the same application
- Fortunately, there are ways to get around this, using techniques for remembering a user's session.
- Once users have logged in, they don't have to login again. The application will remember them.
- This is called ***Session Management***.

Session Management does not change the nature of HTTP statelessness



- Session management goes beyond simply remembering a user who successfully logged in.
- Anything that makes the application remember information that has been entered or requested by the user can be considered session management.
- Session management does not change the nature of HTTP statelessness – it simply provides a way around it.

How do you manage a user's session?

- By performing the following to servlets that need to remember a user's state:
 1. When the user requests a servlet, in addition to sending the response, the servlet also sends a token or an identifier.
 2. If the user does not come back with the next request for the same or a different servlet, that is fine, the token/identifier is sent back to the server
 3. Upon encountering the token, the next servlet should recognize the identifier and can do a certain action based on the token.
 4. When the servlet responds to the request, it also sends the same or a different token.
 5. This goes on and on with all the servlets that need to remember a user's session.

There are 4 techniques for session management.

- They operate based on the same principle, although what is passed and how it is passed is different from one to another.
- The techniques are as follows:
 1. URL rewriting
 2. Hidden fields
 3. Cookies
 4. Session objects
- Which technique you use depend on what you need to do in your application.

1. URL Rewriting

- With URL Rewriting, you append a token or identifier to the URL of the next servlet. You can send parameter name/value pairs using the following format:

url?name1=value1&name2=value2&...

- When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server.
- From a servlet, you can use the HttpServletRequest interface's getParameter method to obtain a parameter value
- For example, to obtain the value of the second parameter, you write the following:

request.getParameter(name2);

The use of URL rewriting is easy.

- When using this technique, however, you need to consider several things:
 - I. The number of characters that can be passed in a URL is limited. Typically, a browser can pass up to 2,000 characters
 - II. The value that you pass can be seen in the URL. Sometimes this is not desirable. For example, some people prefer their password not to appear on the URL
 - III. You need to encode certain characters – such as & and ? Characters and white spaces – that you append to a URL.

Example

Half.com: Shopping Cart - Windows Internet Explorer
http://cart.half.ebay.com/ws/eBaySAPI.dll?ShoppingCart&action=view

My Account | Wish List | Sell My Stuff | Help | eBay Home | Sign in

Home Books Textbooks Music Movies Games Game Systems **Shopping Cart**

All Categories Go

Save on shipping. Buy 2 or more items from the same seller.

Continue Shopping Speedy Checkout Proceed to Checkout

Core Servlets and Javaserver Pages : Larry Brown, Marty Hall
(Paperback, 2003)
Condition: Good
Seller: betterworldbooks (255345) Shipping: Media Mail from Mishawaka, IN Change Shipping
Move to Wish List | Remove | Buy more from seller to save on shipping

Essentials of Management Information Systems : Jane P. Laudon, Kenneth C. Laudon, Kenneth Laudon (Paperback, 2007)
Condition: Brand New
Seller: cheapestbooks123 (3) Shipping: Media Mail from New York, NY Change Shipping
Move to Wish List | Remove | Buy more from seller to save on shipping

http://cart.half.ebay.com/ws/eBaySAPI.dll?ShoppingCart&action=view

Half.com: Shopping Cart - Windows Internet Explorer

http://cart.half.ebay.com/ws/eBayISAPI.dll?ShoppingCart&action=view

Half.com: Shopping Cart

My Account | Wish List | Sell My Stuff | Help | eBay Home | Sign in

Home Books Textbooks Music Movies Games Game Systems Shopping Cart

All Categories Go

Save on shipping. Buy 2 or more items from the same seller.

Continue Shopping Speedy Checkout Proceed to Checkout

Core Servlets and Javaserver Pages : Larry Brown, Marty Hall
(Paperback, 2003)
Condition: Good
Seller: betterworldbooks (255345) Shipping: Media Mail from Mishawaka, IN Change Shipping
Move to Wish List | Remove | Buy more from seller to save on shipping

Essentials of Management Information Systems : Jane P. Laudon, Kenneth C. Laudon, Kenneth Laudon (Paperback, 2007)
Condition: Brand New
Seller: cheapestbooks123 (3) Shipping: Media Mail from New York, NY Change Shipping
Move to Wish List | Remove | Buy more from seller to save on shipping

PRICE: \$17.98
SHIPPING: \$3.49

PRICE: \$123.93
SHIPPING: \$3.49

http://product.half.ebay.com/ws/eBayISAPI.dll?HalfProductDetails&pr=2340011

Internet 100%

Half.com: Shopping Cart - Windows Internet Explorer

http://cart.half.ebay.com/ws/eBaySAPI.dll?ShoppingCart&action=view

Half.com: Shopping Cart

My Account | Wish List | Sell My Stuff | Help | eBay Home | Sign in

Home Books Textbooks Music Movies Games Game Systems Shopping Cart

All Categories Go

Save on shipping. Buy 2 or more items from the same seller.

Continue Shopping Speedy Checkout Proceed to Checkout

Core Servlets and Javaserver Pages : Larry Brown, Marty Hall
(Paperback, 2003)
Condition: Good
Seller: betterworldbooks (255345) Shipping: Media Mail from Mishawaka, IN Change Shipping
Move to Wish List | Remove | Buy more from seller to save on shipping

Essentials of Management Information Systems : Jane P. Laudon, Kenneth C. Laudon, Kenneth Laudon (Paperback, 2007)
Condition: Brand New
Seller: cheapestbooks123 (3) Shipping: Media Mail from New York, NY Change Shipping
Move to Wish List | Remove | Buy more from seller to save on shipping

PRICE: \$17.98
SHIPPING: \$3.49

PRICE: \$123.93
SHIPPING: \$3.49

http://cart.half.ebay.com/ws/eBaySAPI.dll?ShoppingCart&action=deleteitem&itemid=34

Internet 100%

In-class exercise

- Use URL Rewriting in the following web application:
 - Download from Blackboard under Course Materials

2. Hidden Fields

- Another technique for managing user sessions is by passing a token as the value for an HTML hidden field. Unlike the URL rewriting, the value does not show on the URL but can still be read by viewing the HTML source code.
- HTML forms have an entry that looks like the following:
`<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">`

This means that, when the form is submitted, the specified name and value are included in the GET or POST data. This can be used to store information about the session.

Disadvantages:

- it only works if every page is dynamically generated, since the whole point is that each session has a unique identifier.
- an HTML form is always required.

Example

The screenshot shows a Windows Internet Explorer browser window displaying the British Airways website (http://www.britishairways.com/travel/home/public/en_us). The page features the British Airways logo at the top, followed by a navigation bar with links to Home, Flights and more, Manage My Booking, Information, and Executive Club. A search bar is also present. The main content area includes a 'Welcome to ba.com' message and a promotional banner for '2 FREE NIGHTS IN LONDON WHEN YOU FLY FROM \$156'. To the right, there is a 'My Booking' sidebar with options for managing bookings and checking in online. Below the browser window, a separate Notepad window titled 'en_us[1] - Notepad' displays the HTML source code of the page, showing various form inputs and hidden fields.

```
<form action="bookFlight.do">
<input type="hidden" name="hostname" value="www.britishairways.com" />
<input type="hidden" name="protocol" value="http" />
<input type="hidden" name="audience" value="travel" />
<input type="hidden" name="pageid" value="HOME" />
<input type="hidden" name="logintype" value="public" />
<input type="hidden" name="scheme" value="" />
<input type="hidden" name="tier" value="" />
<input type="hidden" name="language" value="en" />
<input type="hidden" name="country" value="us" />
```

In-class exercise

- Use Hidden Fields in the following web application: Use
 - Download from Blackboard under Course Materials

Cookies



- The third technique that you can use to manage user sessions is by using cookies.
- A cookie is a small piece of information that is passed back and forth in the HTTP request and response.
- Even though a cookie can be created on the client side using some scripting language such as JavaScript, it is usually created by a server resource, such as a servlet.
- The cookie sent by a servlet to the client will be passed back to the server when the client requests another page from the same application.

Creating Cookies

- In servlet programming, a cookie is represented by the `Cookie` class in the `javax.servlet.http` package. You can create a cookie by calling the `Cookie` class constructor and passing two `String` objects: the name and value of the cookie.
- For instance, the following code creates a cookie object called `c1`. The cookie has the name "myCookie" and a value of "secret":
`Cookie c1 = new Cookie("myCookie", "secret");`
- You then can add the cookie to the HTTP response using the `addCookie` method of the `HttpServletResponse` interface:
`response.addCookie(c1);`
- Note that because cookies are carried in the request and response headers, you must not add a cookie after an output has been written to the `HttpServletResponse` object. Otherwise, an exception will be thrown.

Example

- The following example shows how you can create two cookies called userName and password.

```
Cookie c1 = new Cookie("userName", "yusuf");
Cookie c2 = new Cookie("password", "secret");
response.addCookie(c1);
response.addCookie(c2);
```

- To retrieve cookies, you use the getCookies method of the HttpServletRequest interface. This method returns a Cookie array containing all cookies in the request. It is your responsibility to loop through the array to get the cookie you want, as follows:

```
Cookie[] cookies = request.getCookies();
int length = cookies.length;

for (int i=0; i<length; i++)
{
    Cookie cookie = cookies[i];
    out.println("<B>Cookie Name:</B> " + cookie.getName() + "<BR>");
    out.println("<B>Cookie Value:</B> " + cookie.getValue() + "<BR>");
}
```

Disadvantage of using Cookies



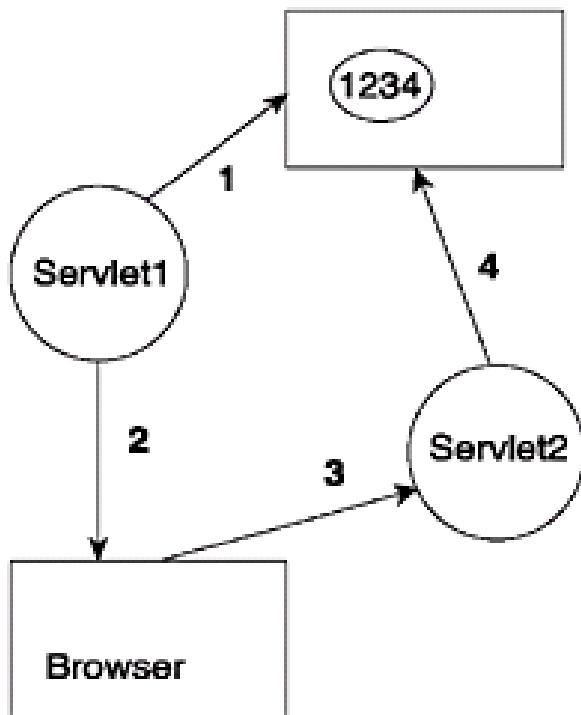
- The user can choose not to accept them.

- Even though browsers leave the factories with the cookie setting on, any user can (accidentally) change this setting.

Session Objects

- Session object, represented by the `javax.servlet.http.HttpSession` interface, is the easiest to use and the most powerful.
- For each user, the servlet can create an `HttpSession` object that is associated with that user only and can only be accessed by that particular user.
- The `HttpSession` object acts like a `Hashtable` into which you can store any number of key/object pairs.
- The `HttpSession` object is accessible from other servlets in the same application.
- To retrieve an object previously stored, you need only to pass the key.
- Unlike previous techniques, however, the server does not send any value.
- What it sends is simply a unique number called the *session identifier*.
- This session identifier is used to associate a user with a Session object in the server.
- Therefore, if there are 10 simultaneous users, 10 Session objects will be created in the server and each user can access only his or her own `HttpSession` object.

How session tracking works with the HttpSession object.



- An HttpSession object is created by a servlet called Servlet1. A session identifier is generated for this HttpSession object. In this example, the session identifier is 1234, but in reality, the servlet container will generate a longer random number that is guaranteed to be unique. The HttpSession object then is stored in the server and is associated with the generated session identifier. Also the programmer can store values immediately after creating an HttpSession.
- In the response, the servlet sends the session identifier to the client browser.
- When the client browser requests another resource in the same application, such as Servlet2, the session identifier is sent back to the server and passed to Servlet2 in the javax.servlet.http.HttpServletRequest object.
- For Servlet2 to have access to the HttpSession object for this particular client, it uses the getSession method of the javax.servlet.http.HttpServletRequest interface. This method automatically retrieves the session identifier from the request and obtains the HttpSession object associated with the session identifier.

Discussion

- What if the user never comes back after an HttpSession object is created?

- Then the servlet container waits for a certain period of time and removes that HttpSession object.

Problems?



- One worry about using Session objects is scalability.
- In some servlet containers, Session objects are stored in memory, and as the number of users exceeds a certain limit, the server eventually runs out of memory.

getSession method

- The getSession method of the javax.servlet.http.HttpServletRequest interface has two overloads. They are as follows:

```
HttpSession getSession()
```

```
HttpSession getSession(boolean create)
```

- The first overload returns the current session associated with this request, or if the request does not have a session identifier, it creates a new one.
- The second overload returns the HttpSession object associated with this request if there is a valid session identifier in the request. If no valid session identifier is found in the request, whether a new HttpSession object is created depends on the create value. If the value is true, a new HttpSession object is created if no valid session identifier is found in the request. Otherwise, the getSession method will return null.

The javax.servlet.http.HttpSession Interface

This interface has the following methods:

- getAttribute
- getAttributeNames
- getCreationTime
- getId
- getLastAccessedTime
- getMaxInactiveInterval
- getServletContext
- getSessionContext
- getValue
- getValueNames
- invalidate
- isNew
- putValue
- removeAttribute
- removeValue
- setAttribute
- setMaxInactiveInterval

In-class exercise

- Use Session Object in the following web application:
 - Download from Blackboard under Course Materials

Knowing Which Technique to Use



- Having learned the four techniques for managing user sessions, you may be wondering which one you should choose to implement.

Session Object

- Clearly, using Session objects is the easiest and you should use this if your servlet container supports swapping Session objects from memory to secondary storage.
- If you are using Tomcat 4 and later, this feature is available to you.

Cookies

- Using cookies is not as flexible as using Session objects.
- However, cookies are the way to go if you don't want your server to store any client-related information or if you want the client information to persist when the browser is closed.

Hidden Fields

- If you need to split a form into several smaller ones, however, using hidden fields is the cheapest and most efficient method.
- You don't need to consume server resources to temporarily store the values from the previous forms, and you don't need to rely on cookies.
- I would suggest hidden fields over Session objects, URL-rewriting, or cookies in this case.

URL Rewriting

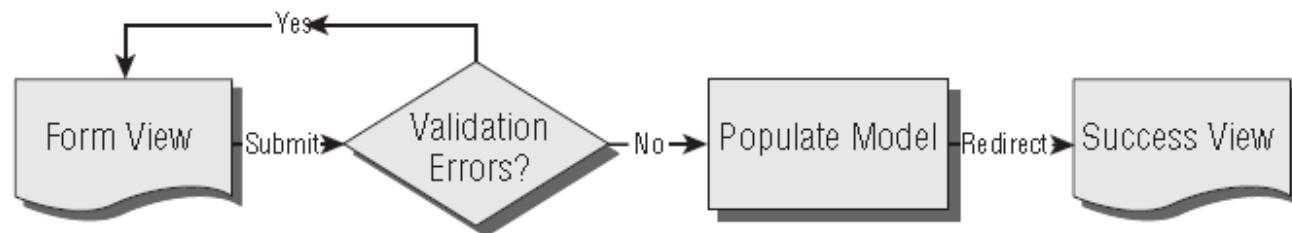
- I. The number of characters that can be passed in a URL is limited. Typically, a browser can pass up to 2,000 characters
- II. The value that you pass can be seen in the URL. Sometimes this is not desirable. For example, some people prefer their password not to appear on the URL
- III. You need to encode certain characters – such as & and ? Characters and white spaces – that you append to a URL.

Getting Data from the User with Forms

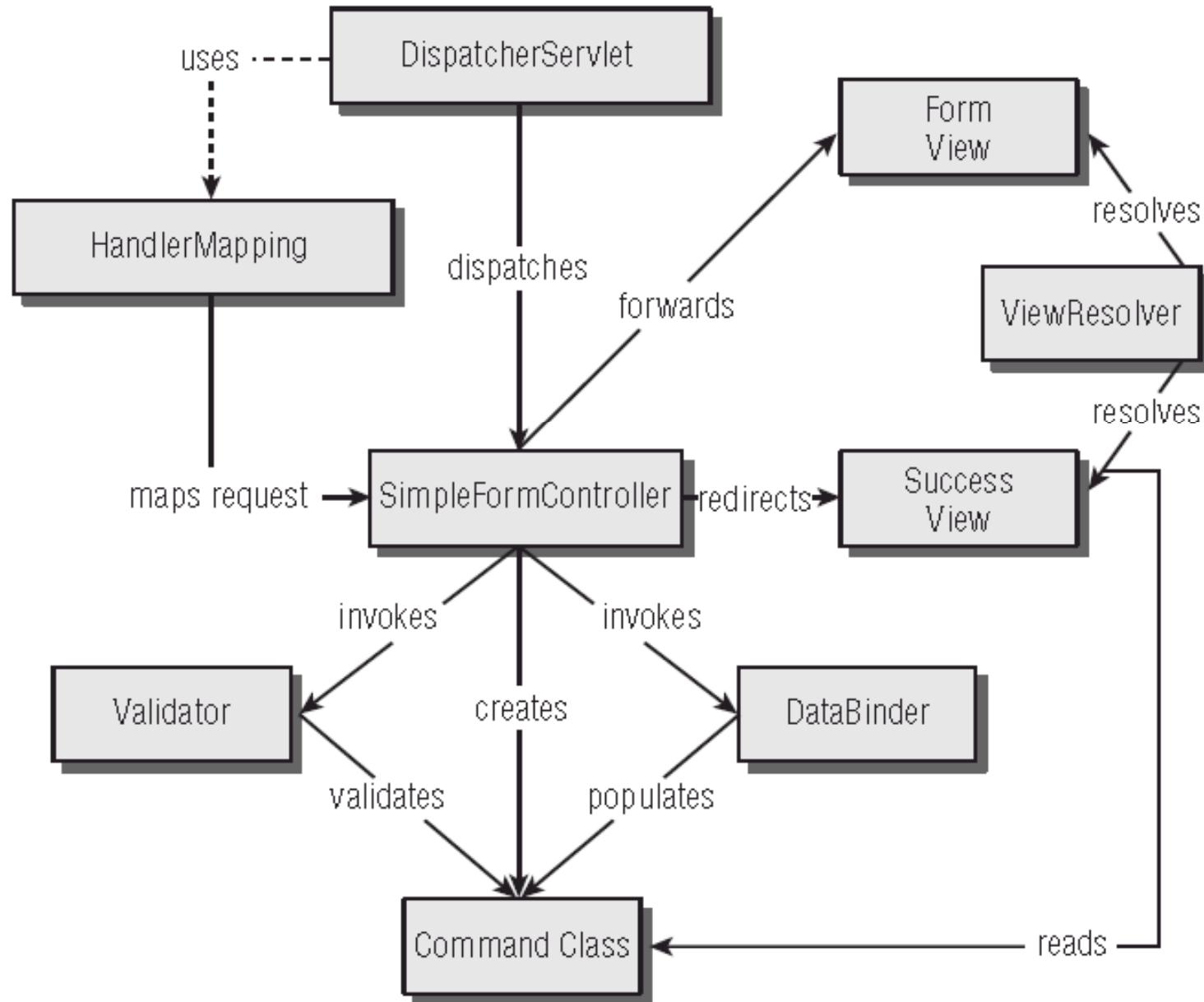
- Providing the ability to capture user data is critical for most dynamic web applications.
- Users often need to register for a user account in order to access certain sections of a site, or fill out forms so that an online purchase order can be processed.
- Whatever site you're building, chances are you will want to capture user data.

A Basic Form-Submission Workflow

In the previous lecture on Spring MVC development you saw how a request reaches a controller and how the controller can retrieve data from the request to interact with the model. The controller then decides which view to use to render the data. The workflow for processing web forms follows a similar path, with the addition of a couple of extra steps to make sure that the data is valid before it is submitted to the database. Form validation is key to processing web forms, as users can never be trusted to enter data the way you would like them to.



In theory you could start writing this workflow yourself by implementing the `Controller` interface. But obviously we wouldn't be talking about form submissions if there weren't an easier way to go about this. As you will see next, Spring MVC provides out-of-the-box implementations for handling form submissions, which will make your life a lot easier.



FormHandling Components

- The DispatcherServlet and the HandlerMapping are used to map URLs to the controller, whereas the ViewResolver takes care of mapping a logical view name to a view implementation.
- The controller at the center of Spring MVC's web form processing is the SimpleFormController classs
- The SimpleFormController provides the boilerplate code to support the basics of a form-submission workflow.
- As an application developer, you would typically subclass the SimpleFormController and configure a number of auxiliary components invoked by the SimpleFormController to validate form data and pass it to the model.
- Validator implementations can be plugged in to validate the data posted to a command class.

- The command class is simply a JavaBean class that has fields used to store the data submitted by the user. These fields are accessible through getters and setters.
- Data from the web form is mapped to the command class's getters and setters with the help of a DataBinder.
- The DataBinder is responsible for converting form data to the command class's field types.
- In case any data validation errors occur, the SimpleFormController returns the user to the original form view so that error messages can be presented to the user.
- When the form is successfully processed, the user is taken to the next page.

Sample Form



Pix
my photo albums

You have not yet created a photo album. Please use the form below to get started.

Album Name *

Description

Album Labels

Holidays

Business

Family

Creation Date

Understanding Form Submissions in Spring MVC

- Spring's SimpleFormController is used, as its name implies, to process simple web forms consisting of a single page.
- The form shown in the previous slide can be used to create and populate a new photo album.
- It has four fields on it, which are used to provide some basic information about a photo album.

Understanding Form Submissions in Spring MVC

- The album name is required.
- The description, album labels, and creation date are optional.
- The user can associate multiple labels with a photo album by checking the box next to a label.
- The creation date must be entered as *dd/mm/yyyy*; *12/29/2007 is valid but 29/12/2007 is not*.
- *The labels on the form are retrieved from the database.*
- When the form is successfully validated, the user is redirected to the albums page.
- If any validation error occurs, an error message is written at the top of the form.

Command Class

- Typically the first step in creating a web form is to define the command class that will be used to store the data.
- A command class is a fancy name for an object that has fields that can be accessed via getter and setter methods.
- The JSP's field names follow a particular naming convention so that a DataBinder can automatically figure out where data from the form goes in the command class.
- Before you get into the details of mapping form fields to the command class, you should take a look at command class used by the sample application to create a new photo album.

Following are the relevant fields of the Album domain object that are populated by the form:

```
public class Album implements Serializable
{
    private integer id;
    private String name;
    private String description;
    private Date creationDate;
    private String[] labels;
    // Additional fields.

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    // Additional accessors.
}
```

Using the Form View

As mentioned in the previous section, to populate the model with form data you need to provide a mapping between the form's field names and the model's properties. Spring uses the standard JavaBean naming convention for accessing JavaBean properties. The following table shows an example of the mapping between form field name and JavaBean properties.

Form Field Name	JavaBean Properties
Name	getName(), setName(..)
Date	getDate(), setDate(..)
person.age	getPerson().getAge(), getPerson().setAge(..)

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<head>
    <title><spring:message code="title.create" /></title>
</head>
<body>
<jsp:include flush="true" page="header.jsp"></jsp:include>
<div align="left"><spring:message code="${message}" text="" /></div>
<form:form>
<form:errors path="*" />
<table>
<tr>
    <th>Album Name*</th>
    <td><form:input path="name" /></td>
</tr>
<tr>
    <th>Description</th>
    <td><form:textarea path="description" /></td>
    ....
```

Why not use ordinary HTML page that follows the naming conventions described in the previous table?



- There is one caveat: it is not clear how you can redisplay the form's values after the data has been submitted to confirm the input values with the user.
- When using static HTML, you should use another part of the Spring MVC's tag library —the form tags.
- Spring MVC's form tags take care of rendering the appropriate HTML along with the field values.
- As you can see, the tags are similar to standard HTML form tags.
 - Each references the model with the path attribute.
 - The value of path is used to render the field's name and references the corresponding JavaBean property.

Populating the Model

- You may be wondering how Spring manages to convert a form's date input field to a Date object.
- Remember that dates are represented differently depending on where you live.
- **PropertyEditor** is a standard JDK interface, found in the `java.beans` package.
- This class provides the capability to take a String representation of an object and convert it to the appropriate object type.
- Spring provides implementations of the **PropertyEditor** interface for common object types such as URL, File, String and Locale, as well as others.
(See the `org.springframework.beans.propertyeditors` package for a complete list of property editors provided by Spring.)
- **Property editors** are not used only for type conversion with dependency injection but also for converting data submitted by a web form to an appropriate type in the model.

- The Album model shown in the earlier listing accepts a Date object to set the album's creation date.
- As you know, dates come in many shapes and forms and take different formats depending on the user's region. Because there is no common date format, it is necessary to tell the application how dates are entered by the user.
- The web form presented in the following listing accepts the *dd/mm/yyyy* format.
- Spring's *CustomDateEditor* is used to provide support for this date format.
- The following listing shows the implementation and registration in CreateAlbumController:

```
public class CreateAlbumController extends SimpleFormController
{
    /**
     * ...
     */
    protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder) throws Exception
    {
        CustomDateEditor dateEditor = new CustomDateEditor(new SimpleDateFormat("dd/MM/yyyy"), true);
        binder.registerCustomEditor(Date.class, dateEditor);
    }
    /**
     * ...
     */
}
```

Registering the Album class with the controller

- Next, you need to register the Album class with the controller by passing `Album.class` to the controller's `setCommandClass` method.
- By default, a new instance of Album is created each time the form is loaded. You can override the SimpleFormController's `formBackingObject` method if you want to customize this behavior so that you can, for example, reload an existing album in the form for editing.
- The following code shows an example of how this method is overloaded to retrieve an existing album from the database in case an ID parameter is passed to the form. This listing also shows how the Album class is registered as a command class in the constructor.

```
public class CreateAlbumController extends SimpleFormController {  
    private AlbumRepository albumRepo;  
  
    public CreateAlbumController() {  
        setCommandClass(Album.class);  
    }  
    //...  
    protected Object formBackingObject(HttpServletRequest request) throws Exception {  
        int id = ServletRequestUtils.getIntParameter(request, ALBUM_ID);  
        if (id != null && !"".equals(id)) {  
            return albumRepo.retrieveAlbumById(id);  
        } else {  
            return super.formBackingObject(request);  
        }  
    }  
    //...  
    public void setAlbumRepo(AlbumRepository albumRepo) {  
        this.albumRepo = albumRepo;  
    }  
}
```

- The request data is saved to the Album instance returned by the formBackingObject method and then passed to the doSubmitAction method.
- You need to override doSubmitAction to retrieve a populated Album object and store it in the database.

```
public class CreateAlbumController extends SimpleFormController
{
    private AlbumRepository albumRepo;
    //...
    protected void doSubmitAction(Object album) throws Exception
    {
        albumRepo.persistAlbum( (Album) album);
    }

    //...
    public void setAlbumRepo(AlbumRepository albumRepo)
    {
        this.albumRepo = albumRepo;
    }
}
```

- Your CreateAlbumController is almost ready for deployment.
- The last missing piece is telling the controller where to point the user after the form has successfully been saved to the database. In your application you want to redirect the user to the albums page.
- To do so, you need to add the following code to the constructor:

```
public CreateAlbumController()  
{  
    setCommandClass(Album.class);  
    setSuccessView("redirect:albums.htm");  
}
```

- At this point your form is ready and you should be able to create new albums - that is, if you enter data correctly. Since you cannot depend on your users to do this, you need to learn how to add validation rules to your form.

Implementing Form Validation

- To enforce the form's data-entry rules you cannot rely on user discipline alone.
- Users often forget to fill out required fields or do not respect valid e-mail or date formats.
- To filter out these human errors before the data is submitted to the rest of the application, you need to enforce validation rules.
- Spring provides a flexible validation mechanism that can be used to preserve data integrity.

Data validation is provided by implementations of the `Validator` interface. Following is a `Validator` implementation for the `Album` class that checks to determine whether the name field is filled out:

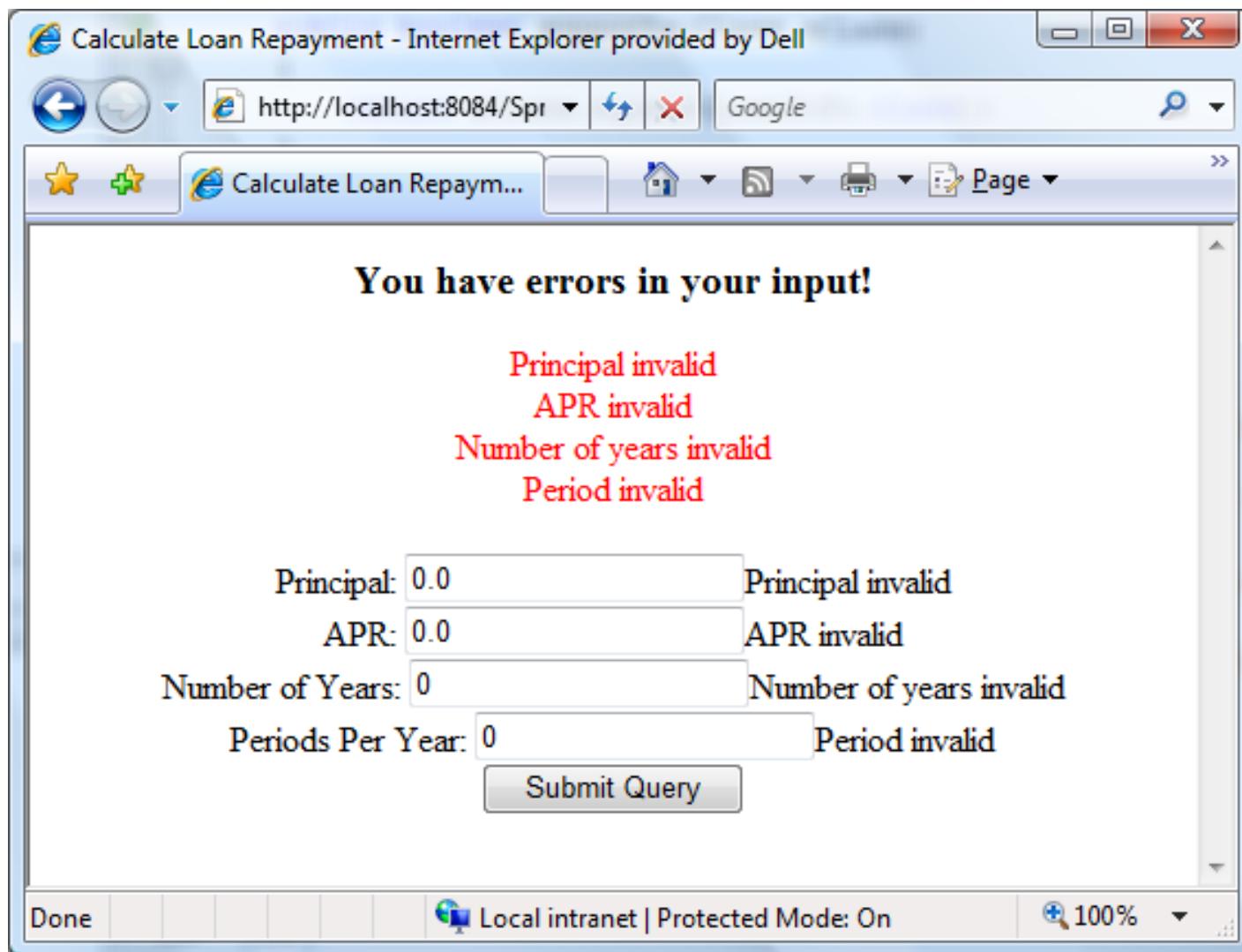
```
public class AlbumValidator implements Validator {  
  
    public boolean supports(Class clazz) {  
        return Album.class.isAssignableFrom(clazz);  
    }  
  
    public void validate(Object target, Errors errors) {  
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name",  
            "error.required.name");  
    }  
}
```

`Validator` implementations are fairly straightforward to write. A `validator` implements the `supports` method to provide information about the classes it knows to validate. The `validate` method receives an instance of this class along with an `Errors` collection, which can be used to pass validation errors to the application.

An `Errors` collection can be accessed in a JSP with the `form:errors` tag. `<spring:errors path="*"/>` prints out the entire collection of errors. It is also possible to print errors for a particular form field. For example, `<spring:errors path="name"/>` prints only validation errors that occur on the `Name` field. The path syntax is similar to the one used for form-field binding.

Validation can be used to enter user input errors that can be anticipated in advance. However, there are often user errors, coding errors, and system errors that cannot be easily anticipated.

In-class exercise



Web Applications



- In this lecture, you will learn how to develop web applications using Spring MVC.

Agile J2EE: Where do we want to go?



- Need to be able to produce high quality applications, faster and at lower cost
- Need to be able to cope with changing requirements
 - Waterfall is no longer an option
- Need to simplify the programming model
 - **Need to reduce complexity rather than rely on tools to hide it**
 - ...but must keep the power of the J2EE platform

Agile J2EE: Why is this important?



- Java™ technology/J2EE is facing challenges at the low end
 - .NET
 - PHP
 - Ruby
- Concerns from high end clients (banking in particular) that J2EE development is slow and expensive
- Complacency is dangerous

Agile J2EE: Aren't we there yet?



Problems with traditional J2EE architecture...

- Difficult to test traditionally architected J2EE apps
 - EJBs depend heavily on the runtime framework, making them relatively hard to author and test
- Simply too much code
 - Pet Store as an example
 - Much of that code is mundane “glue” code
- Heavyweight runtime environment
 - Components need to be explicitly deployed to be able to run, even for testing
 - Slow change-deploy-test cycle

Lightweight Containers

- **Frameworks** are central to modern J2EE development
- Many projects encounter the same problems
 - Service location
 - Consistent exception handling
 - Parameterizing application code...
- J2EE “out of the box” does not provide a complete (or ideal) programming model
- Result: many in-house frameworks
 - Expensive to maintain and develop
 - Better to share experience across many projects

Open Source Frameworks



- Responsible for much innovation in last 2–3 years
 - Flourishing open source is one of the great strengths of the Java platform
- Successful projects are driven by actual common problems to be solved
- Ideally placed to learn from collective developer experience
- Several products aim to simplify the development experience and remove excessive complexity from the developer's view

The Spring Framework

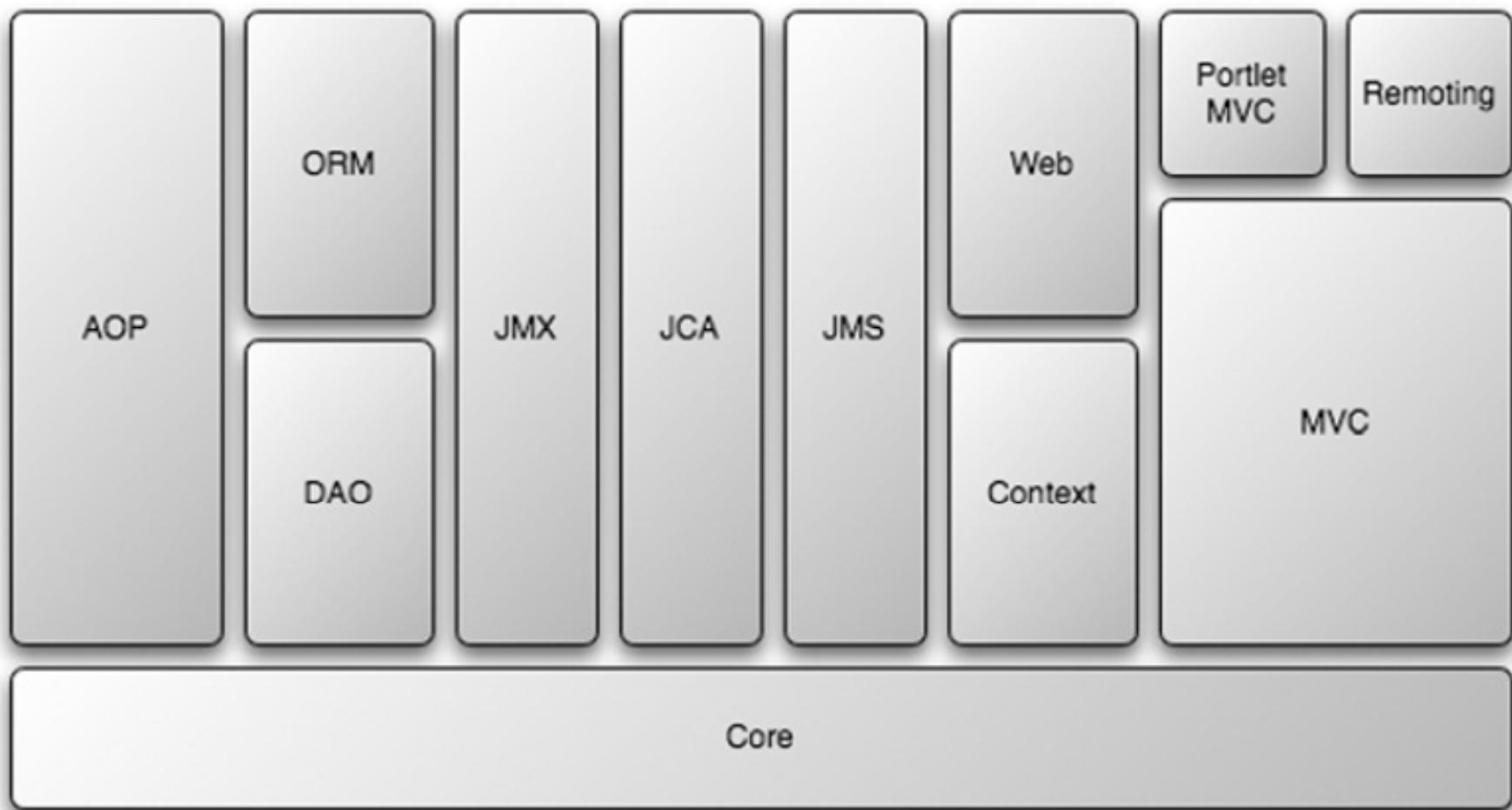


- Open source project
 - Apache 2.0 license
 - 21 developers
 - Interface21 lead development effort, with seven committers (and counting), including the two project leads
- Aims
 - Simplify J2EE development
 - Provide a comprehensive solution to developing applications built on POJOs
 - Provide services for applications ranging from simple web apps up to large financial/"enterprise" applications

Who is using Spring?



- Spring is widely used in many industries, including...
 - Banking
 - Transactional Web applications, message-driven middleware
 - Retail and investment banking
 - Scientific research
 - Defence
 - A growing number of Fortune 500 companies
 - High volume Web sites
 - **Significant enterprise usage, not merely adventurous early adopters**



The Spring Framework is composed of several well-defined modules built on top of the core container. This modularity makes it possible to use as much or as little of the Spring Framework as is needed in a particular application.

Spring MVC

- We will focus on Spring MVC

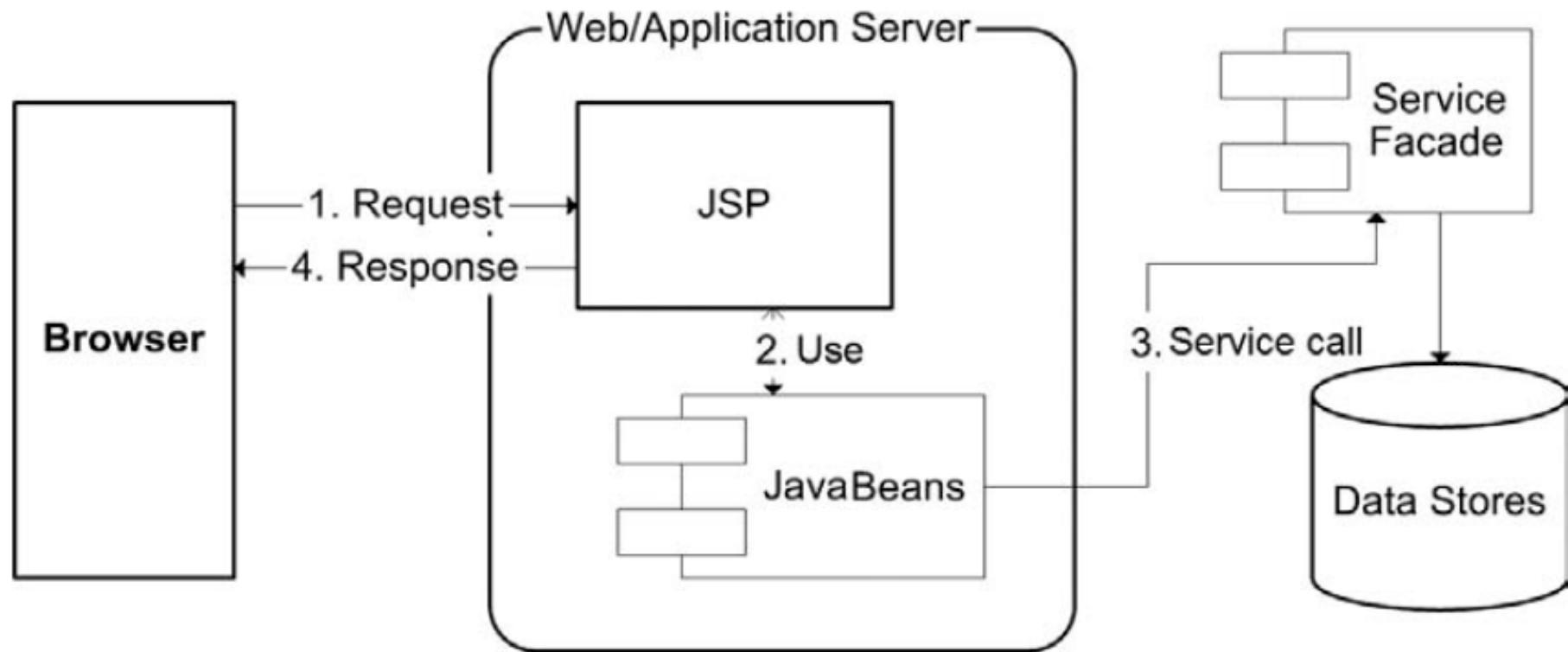
Web Framework

- Web applications have become a very important part of any enterprise system.
- The key requirement for a web framework is to simplify development of the web tier as much as possible.
- In this lecture, you will learn how to develop web applications using Spring.
- We will start with an explanation of Spring MVC architecture and the request cycle of Spring web applications, introducing handler mappings, interceptors, and controllers.
- Then we will discuss how we can use different technologies to render HTML in the browser.
- Before we dive into a discussion of Spring MVC, let us review MVC.

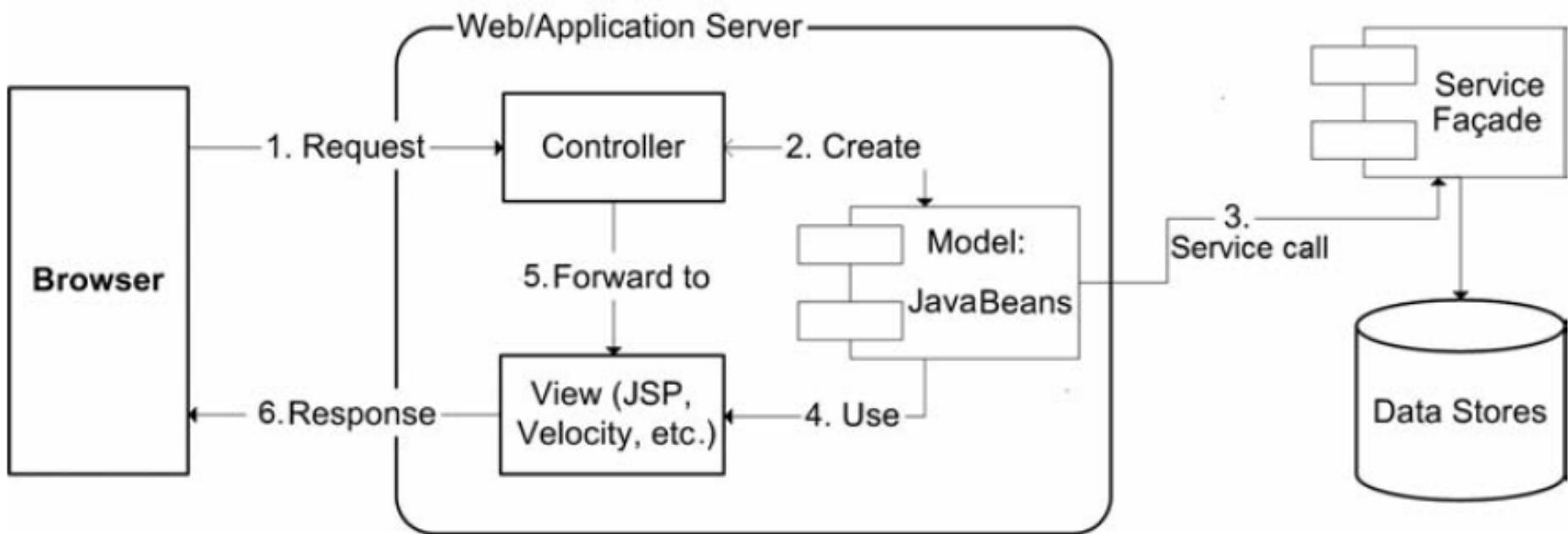
MVC Architecture

- MVC is the acronym for the model view controller architectural pattern.
- The purpose of this pattern is to simplify the implementation of applications that need to act on user requests and manipulate and display data.
- There are three distinct components of this pattern:
 - **The model** represents data that the user expects to see. In most cases, the model will consist of JavaBeans.
 - **The view** is responsible for rendering the model. A view component in a text editor will probably display the text in appropriate formatting; in a web application, it will, in most cases, generate HTML output that the client's browser can interpret.
 - **The controller** is a piece of logic that is responsible for processing and acting on user requests: it builds an appropriate model and passes it to the view for rendering
- In the case of Java web applications, the controller is usually a servlet.
- Of course the controller can be implemented in any language a web container can run

Model 1 Architecture



Model 2 Architecture



Spring MVC



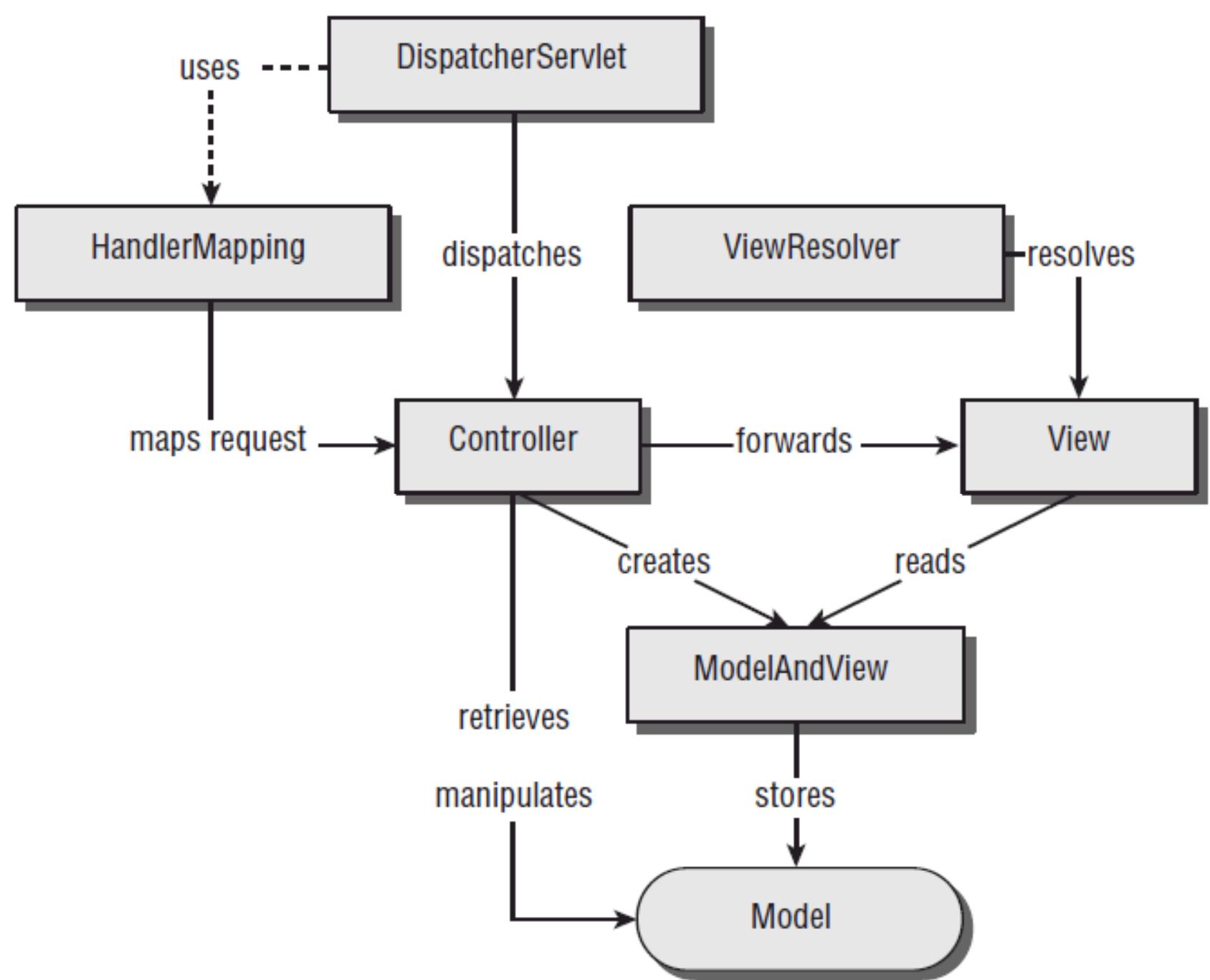
- Spring MVC support allows us to build flexible applications using MVC model two.
- The implementation is truly generic.
 - **the model** is a simple Map that holds the data;
 - **the view** is an interface whose implementations render the data;
 - **the controller** is an implementation of the Controller interface.

Benefits of the Spring Web MVC Framework

- **Easier testing.** The fact that most of Spring's classes are designed as JavaBeans enables you to inject test data using the setter methods of these classes.
- **Bind directly to business objects** Spring MVC does not require your business (model) classes to extend any special classes; this enables you to reuse your business objects by binding them directly to the HTML form fields
- **Clear separation of roles** Spring MVC nicely separates the roles played by the various components that make up this web framework.
- **Adaptable controllers** If your application does not require an HTML form, you can write a simpler version of a Spring controller that does not need all the extra components required for form controllers.

Benefits of the Spring Web MVC Framework Cont'd

- **Simple but powerful tag library** Spring's tag library is small, straightforward, but powerful.
- **Web Flow** This module is a subproject and is not bundled with the Spring core distribution. It is built on top of Spring MVC and adds the capability to easily write wizard like web applications that span across several HTTP requests.
- **View technologies and web frameworks** Although we are using JSP as our view technology, Spring supports other view technologies as well, such as Apache Velocity and FreeMarker (freemarker.org).
- **Lighter-weight environment.** Spring enables you to build enterprise-ready applications using POJOs; the environment setup can be simpler and less expensive because you could develop and deploy your application using a lighter-weight servlet container.



DispatcherServlet

- At the heart of Spring MVC is DispatcherServlet, a servlet that functions as Spring MVC's front controller.
- Like any servlet, DispatcherServlet must be configured in your web application's web.xml file. Place the following <servlet> declaration in your web.xml file:

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
```

- This servlet processes requests and invokes appropriate Controller elements to handle them.
- The DispatcherServlet intercepts incoming requests and determines which controller will handle the request.

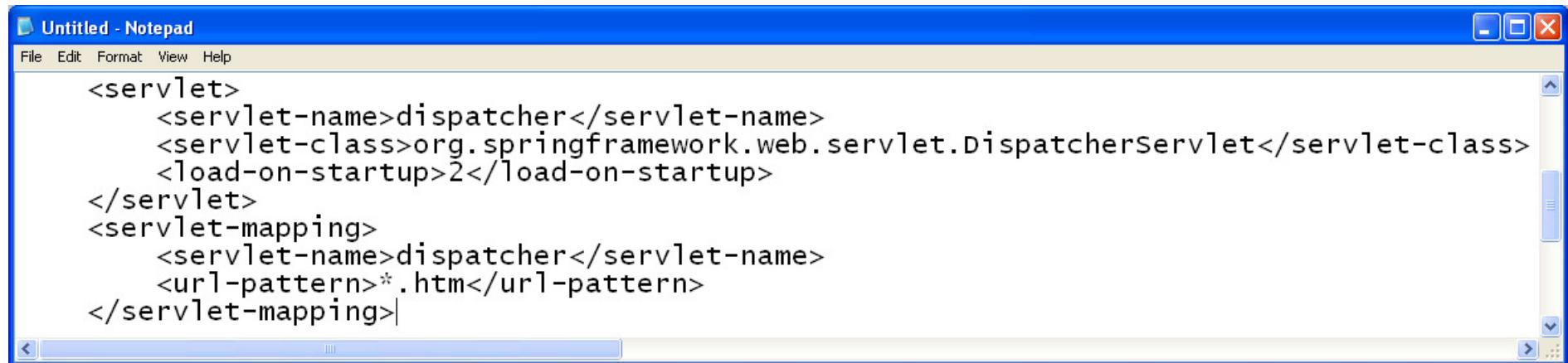
ModelAndView

- The Spring controllers return a ***ModelAndView*** class from their handling methods.
- The ***ModelAndView*** instance holds a reference to a view and a model.
- The ***Model*** is a simple Map instance that holds JavaBeans that the View interface is going to render.
- The ***View*** interface defines the render method.
- It follows that the View implementation can be virtually anything that can be interpreted by the client.

MVC Implementation

To create a web application with Spring,

1. we need to start with the basic web.xml file,
2. where we specify the DispatcherServlet and
3. set the mapping for the specified url-pattern



A screenshot of a Windows Notepad window titled "Untitled - Notepad". The window contains XML code for a web application's configuration. The code defines a DispatcherServlet with a servlet name of "dispatcher", a class of "org.springframework.web.servlet.DispatcherServlet", and a load-on-startup value of 2. It also defines a servlet mapping for the "dispatcher" servlet with a URL pattern of "*.htm".

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

Why .htm URL-Pattern?

- It could be because all of the content produced by our application is HTML.
- It could also be because we want to fool our friends into thinking that our entire application is composed of static HTML files.
- And it could be that Spring Developers think .do is a silly extension.

- But the truth of the matter is that the URL pattern is somewhat arbitrary and we could've chosen any URL pattern for DispatcherServlet.

- The main reason for choosing *.htm is that this pattern is the one used by convention in most Spring MVC applications that produce HTML content.

- The reasoning behind this convention is that the content being produced is HTML and so the URL should reflect that fact.

15.4 inch laptop, like new, 2.0GHz dual core etc. - Windows Internet Explorer

http://boston.craigslist.org/gbs/sys/1038286761.html

boston craigslist > boston/camb/brook > computers & tech

email this posting to a friend

Avoid scams and fraud by dealing locally! Beware any deal involving Western Union, Moneygram, wire transfer, cashier check, money order, shipping, escrow, or any promise of transaction protection/certification/guarantee. [More info](#)

please flag with care:

[miscategorized](#)

[prohibited](#)

[spam/overpost](#)

[best of craigslist](#)

15.4 inch laptop, like new, 2.0GHz dual core etc. - \$500

Reply to: sale-1038286761@craigslist.org

Date: 2009-02-17, 10:36AM EST

Gateway laptop, like-new condition and very fast. Excellent battery life for its class; 2.5-3.5 hours, depending on settings. I will reset it to factory settings with the system restore disk. All manuals and the charger included.

Windows Vista
15.4 inch screen
2.0GHz dual core Pentium (Intel Core 2 Duo T5750 Dual Core Mobile Processor)
3Gb of RAM
160 Gigabyte hard drive

can deliver/meet up in the boston metro area

Internet 100%

Using Handler Mappings

- How does our web application know which **controller** to invoke?
- This is where Spring handler mappings kick in.
- In a few easy steps, you can configure URL mappings to Spring controllers.
- All you need to do is edit the Spring application context file.

- Spring uses HandlerMapping implementations to identify the controller to invoke and provides three implementations of HandlerMapping, as shown below.
- All three HandlerMapping implementations extend the AbstractHandlerMapping base class.
- 99% of users will be using BeanNameUrlHandlerMapping or SimpleUrlHandlerMapping

HandlerMapping	Description
BeanNameUrlHandlerMapping	The bean name is identified by the URL. If the URL were /product/index.html, the controller bean ID that handles this mapping would have to be set to /product/index.html. This mapping is useful for small applications, as it does not support wildcards in the requests.
SimpleUrlHandlerMapping	This handler mapping allows you to specify in the requests (using full names and wildcards) which controller is going to handle the request.
ControllerClassNameHandlerMapping	This handler mapping is part of the convenience over configuration approach introduced with Spring 2.5. It automatically generates URL paths from the class names of the controllers.

1. BeanNameUrlHandlerMapping

- A very simple, but very powerful handler mapping is the **BeanNameUrlHandlerMapping**, which maps incoming HTTP requests to names of beans, defined in the web application context.
- Let's say we want to enable a user to insert an account and we've already provided an appropriate Controller and a JSP view.
- When using the BeanNameUrlHandlerMapping, we could map the HTTP request with URL

http://hostName/editaccount.htm
to the appropriate Controller as follows:

```
<bean name="handlerMapping"  
      class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>  
<bean name="/editaccount.htm" class="AccountController"></bean>
```

In-Class example



1. Write the controller class that performs the logic behind the page.
2. Configure the controller in the DispatcherServlet's context configuration file
3. Configure a view resolver to tie the controller to the JSP.
4. Write the JSP that will render the page to the user.

The screenshot shows a Java code editor window titled "YusufController.java". The code implements a Spring MVC controller. It imports necessary classes from javax.servlet.http, org.springframework.web.servlet, and org.springframework.web.servlet.mvc. The class extends AbstractController and overrides handleRequestInternal to return a ModelAndView containing a greeting message and the current server time.

```
1 package com.yusuf;
2
3 import javax.servlet.http.HttpServletRequest;
4 import javax.servlet.http.HttpServletResponse;
5 import org.springframework.web.servlet.ModelAndView;
6 import org.springframework.web.servlet.mvc.AbstractController;
7
8 import java.util.HashMap;
9 import java.util.Map;
10 import java.util.Date;
11
12 public class YusufController extends AbstractController {
13
14     public YusufController() {
15     }
16
17     @Override
18     protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response) throws Exception {
19         Map model = new HashMap();
20         model.put("Greeting", "Hello World");
21         model.put("Server time", new Date());
22         return new ModelAndView("index", "message", model);
23     }
24 }
```

```
dispatcher-servlet.xml x
[File] [Edit] [View] [Search] [Help]
[New] [Open] [Save] [Save As] [Print] [Find] [Replace] [Copy] [Cut] [Delete] [Format] [Properties] [Toolbars] [Status Bar]
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:p="http://www.springframework.org/schema/p"
5      xmlns:aop="http://www.springframework.org/schema/aop"
6      xmlns:tx="http://www.springframework.org/schema/tx"
7      xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
8          http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
9          http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
10
11     <bean id="beanNameUrlHandlerMapping" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
12     <bean name="/index.htm" class="com.yusuf.YusufController">
13         </bean>
14
15     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
16         <property name="prefix">
17             <value>/WEB-INF/jsp/</value>
18         </property>
19         <property name="suffix">
20             <value>.jsp</value>
21         </property>
22     </bean>
23 </beans>
```

index.jsp x

File Edit View Insert Tools Window Help

Back Forward Stop Refresh Home

1 <jsp:useBean id="message" type="java.util.HashMap" scope="request" />

2

3 <html>

4 <body bgcolor="pink">

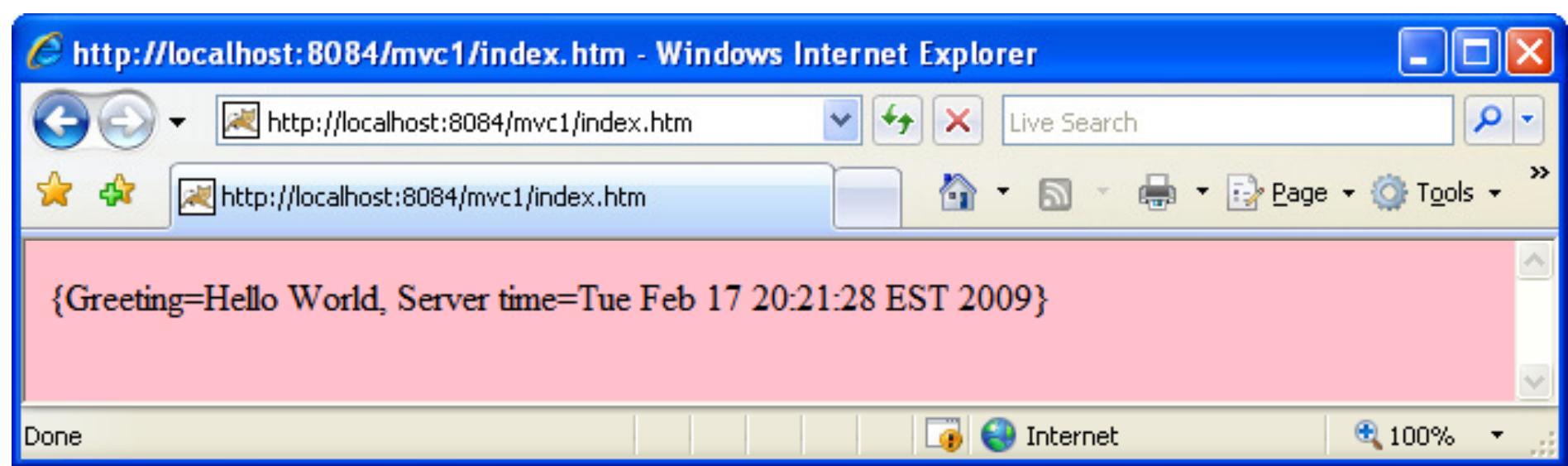
5 User: <%= message %>

6 </body>

7 </html>

8

9:1 INS



1. The first stop in the request's travels is Spring's DispatcherServlet.

Like most Java-based MVC frameworks, Spring MVC funnels requests through a single front controller servlet. A front controller is a common web-application pattern where a single servlet delegates responsibility for a request to other components of an application to perform the actual processing. In the case of Spring MVC, DispatcherServlet is the front controller.

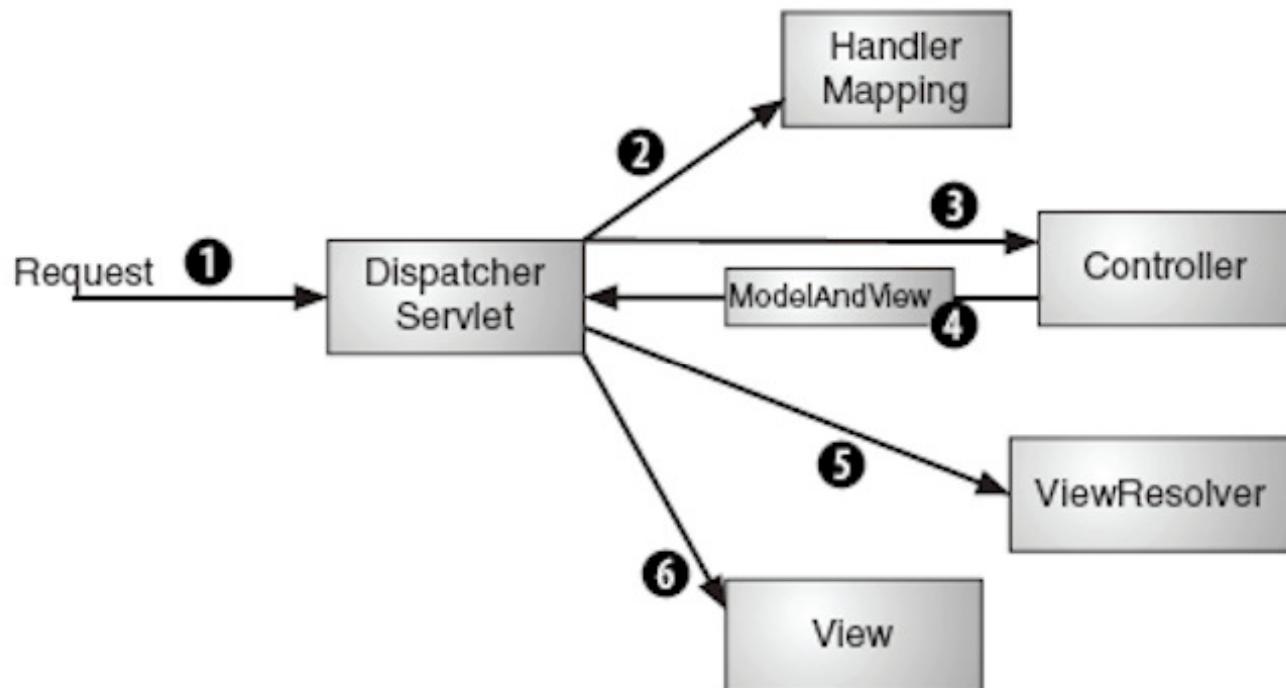
2. The DispatcherServlet's job is to send the request on to a Spring MVC controller.

A controller is a Spring component that processes the request. But a typical application may have several controllers and DispatcherServlet needs help deciding which controller to send the request to. So, the DispatcherServlet consults one or more handler mappings to figure out where the request's next stop will be. The handler mapping will pay particular attention to the URL carried by the request when making its decision.

3. Once an appropriate controller has been chosen, DispatcherServlet sends the request on its merry way to the chosen controller. At the controller, the request will drop off its payload (the information submitted by the user) and patiently wait for the controller to process that information. (Actually, a well-designed Controller performs little or no processing itself and instead delegates responsibility for the business logic to one or more service objects.)

4. So, the last thing that the controller will do is package up the model data and the name of a view into a *ModelAndView* object. It then sends the request, along with its new *ModelAndView* parcel, back to the *DispatcherServlet*. As its name implies, the *ModelAndView* object contains both the model data as well as a hint to what view should render the results.
5. So that the controller isn't coupled to a particular view, the *ModelAndView* doesn't carry a reference to the actual JSP. Instead it only carries a logical name that will be used to look up the actual view that will produce the resulting HTML. Once the *ModelAndView* is delivered to the *DispatcherServlet*, the *DispatcherServlet* asks a view resolver to help find the actual JSP.
6. Now that the *DispatcherServlet* knows which view will render the results, the request's job is almost over. Its final stop is at the view implementation (probably a JSP) where it delivers the model data. With the model data delivered to the view, the request's job is done. The view will use the model data to render a page that will be carried back to the browser by the (not-so-hard-working) response object.

The request will make several stops from the time that it leaves the browser until the time that it returns a response.



A request is dispatched by **DispatcherServlet** to a controller (which is chosen through a handler mapping). Once the controller is finished, the request is then sent to a view (which is chosen through a **ViewResolver**) to render output.

2. SimpleUrlHandlerMapping

- A further - and much more powerful handler mapping - is the SimpleUrlHandlerMapping.
- This mapping is configurable in the application context and has Ant-style path matching capabilities.
- A couple of example will probably makes thing clear enough.

```
<web-app>
  ...
  <servlet>
    <servlet-name>sample</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Maps the sample dispatcher to /*.form -->
  <servlet-mapping>
    <servlet-name>sample</servlet-name>
    <url-pattern>*.form</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>sample</servlet-name>
    <url-pattern>*.html</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

Allows all requests ending with .html and .form to be handled by the sample dispatcher servlet.

```
<beans>
  <bean id="handlerMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/account.form">editAccountFormController</prop>
        <prop key="/editaccount.form">editAccountFormController</prop>
        <prop key="/ex/view*.html">someViewController</prop>
        <prop key="/**/help.html">helpController</prop>
      </props>
    </property>
  </bean>

  <bean id="someViewController"
        class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

  <bean id="editAccountFormController"
        class="org.springframework.web.servlet.mvc.SimpleFormController">
    <property name="formView"><value>account</value></property>
    <property name="successView"><value>account-created</value></property>
    <property name="commandName"><value>Account</value></property>
    <property name="commandClass"><value>samples.Account</value></property>
  </bean>
</beans>
```

The screenshot shows a Java IDE interface with the title bar "dispatcher-servlet.xml". The main area displays the XML configuration for a Spring MVC application. The code defines four controller beans (homeController, carController, bookController) and a view resolver (viewResolver). The homeController maps to "/homes.htm", carController to "/cars.htm", and bookController to "/books.htm". The view resolver is configured to look for JSP files under the prefix "/WEB-INF/jsp/" with a suffix ".jsp".

```
<!-- Dispatcher Servlet Configuration -->
<beans>

    <bean id="simpleUrlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/homes.htm">homeController</prop>
                <prop key="/cars.htm">carController</prop>
                <prop key="/books.htm">bookController</prop>
            </props>
        </property>
    </bean>

    <bean name="homeController"
          class="org.springframework.web.servlet.mvc.ParameterizableViewController"
          p:viewName="homesPage" />

    <bean name="carController"
          class="org.springframework.web.servlet.mvc.ParameterizableViewController"
          p:viewName="carsPage" />

    <bean name="bookController"
          class="org.springframework.web.servlet.mvc.ParameterizableViewController"
          p:viewName="booksPage" />

    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver"
          p:prefix="/WEB-INF/jsp/"
          p:suffix=".jsp" />

</beans>
```

3. ControllerClassNameHandlerMapping

- Oftentimes you'll find yourself mapping your controllers to URL patterns that are quite similar to the class names of the controllers.
- For example, mapping
 - rantForVehicle.htm to RantsForVehicleController
 - rantsForDay.htm to RantsForDayController.
- Notice a pattern?
 - In those cases, the URL pattern is the same as the name of the controller class, dropping the Controller portion and adding .htm.
 - It seems that with a pattern like that it would be possible to assume a certain default for the mappings and not require explicit mappings

```
class ControllerClassNameHandlerMapping extends AbstractControllerUrlHandlerMapping
```

- Implementation of HandlerMapping that follows a simple convention for generating URL path mappings from the class names of registered Controller beans.
- For simple Controller implementations, the convention is to take the short name of the Class, remove the 'Controller' suffix if it exists and return the remaining text, lower-cased, as the mapping, with a leading .
- For example:
 - WelcomeController → /welcome*
 - HomeController → /home*

The `ControllerClassNameHandlerMapping` class is a `HandlerMapping` implementation that uses a convention to determine the mapping between request URLs and the `Controller` instances that are to handle those requests.

Consider the following simple `Controller` implementation. Take special notice of the *name* of the class.

```
public class ViewShoppingCartController implements Controller {  
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {  
        // the implementation is not hugely important for this example...  
    }  
}
```

Here is a snippet from the corresponding Spring Web MVC configuration file:

```
<bean class="org.springframework.web.servlet.support.ControllerClassNameHandlerMapping"/>  
  
<bean id="viewShoppingCart" class="x.y.z.ViewShoppingCartController">  
    <!-- inject dependencies as required... -->  
</bean>
```

The `ControllerClassNameHandlerMapping` finds all of the various handler (or `Controller`) beans defined in its application context and strips `Controller` off the name to define its handler mappings. Thus, `ViewShoppingCartController` maps to the `/viewshoppingcart*` request URL.

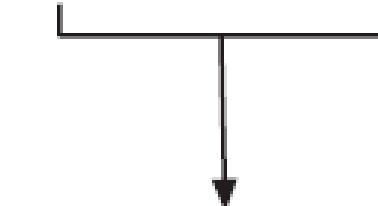
Let's look at some more examples so that the central idea becomes immediately familiar. (Notice all lowercase in the URLs, in contrast to camel-cased `Controller` class names.)

- `WelcomeController` maps to the `/welcome*` request URL
- `HomeController` maps to the `/home*` request URL
- `IndexController` maps to the `/index*` request URL
- `RegisterController` maps to the `/register*` request URL

```
<bean id="classNameHandlerMapping"  
      class="org.springframework.web.servlet.mvc.ControllerClassNameHandlerMapping" />
```

- By configuring ControllerClassNameHandlerMapping, you are telling Spring's DispatcherServlet to map URL patterns to controllers following a simple convention. Instead of explicitly mapping each controller to a URL pattern, Spring will automatically map controllers to URL patterns that are based on the controller's class name.
- Put simply, to produce the URL pattern, the Controller portion of the controller's class name is removed (if it exists), the remaining text is lowercased, a slash (/) is added to the beginning, and ".htm" is added to the end to produce the URL pattern.
- Consequently, a controller bean whose class is RantsForVehicle-Controller will be mapped to /rantsforvehicle.htm. Notice that the entire URL pattern is lowercased, which is slightly different from the convention we were following with SimpleUrlHandlerMapping.

```
com.roaddrantz.mvc.RantsForVehicleController
```



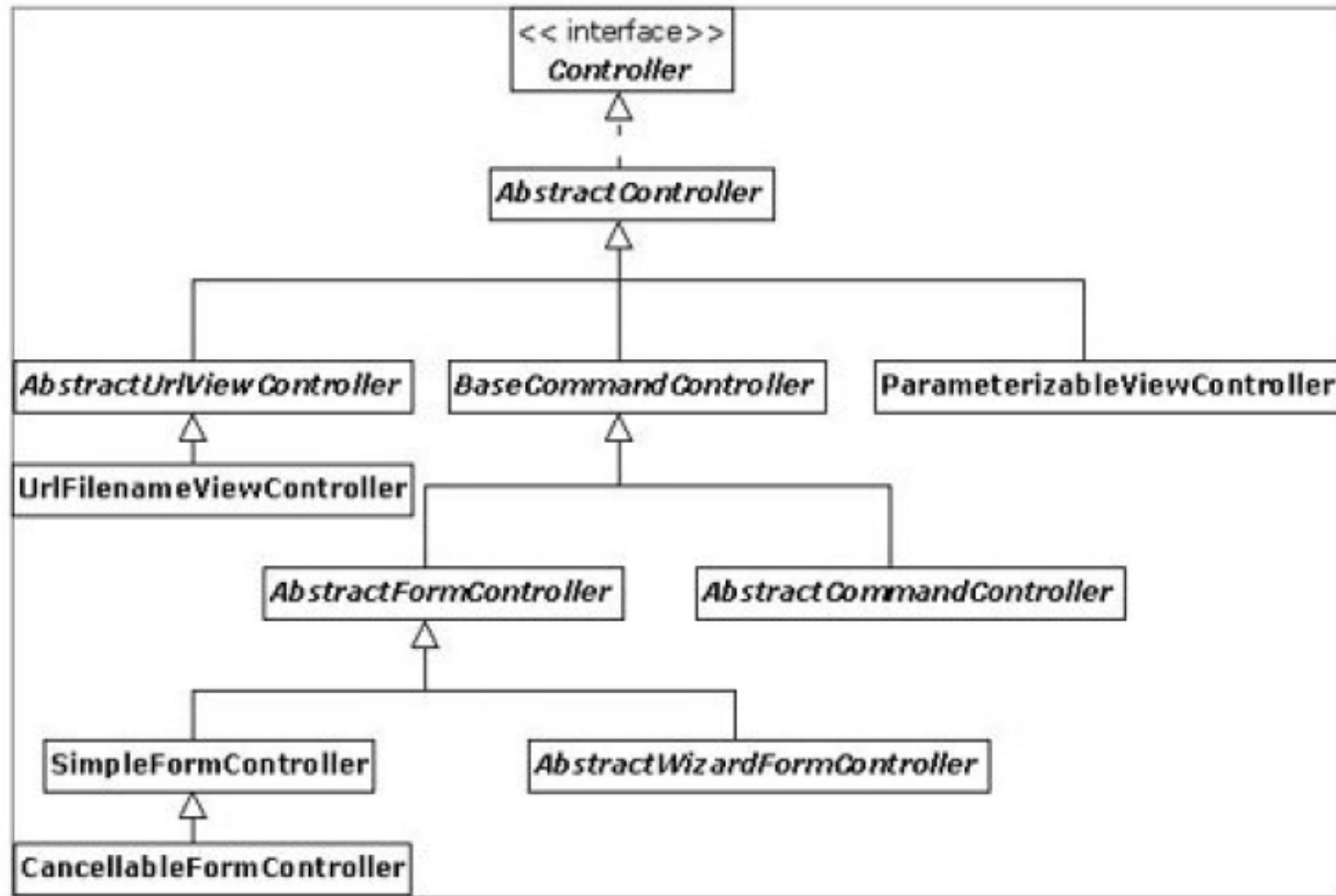
```
/rantsforvehicle.htm
```

ControllerClassNameHandler -
Mapping maps a request to a controller by
stripping **Controller** from the end of the
class name and normalizing it to lowercase.

Spring Controllers

- Controllers do all the work to process the request, build the model based on the request, and pass the model to the view for rendering.
- Spring's DispatcherServlet intercepts the requests from the client and uses a HandlerAdapter implementation that is responsible for delegating the request for further processing.
- You can implement the HandlerAdapter yourself, allowing you to modify the chain of command the request must pass through.
- Spring provides many types of controllers. This can be both good and bad. The good thing is that you have a variety of controllers to choose from, but that also happens to be the bad part because it can be a bit confusing at first about which one to use.
- The best way to decide which controller type to use probably is by knowing what type of functionality you need. For example,
 - Do your screens contain a form?
 - Do you need wizardlike functionality?
 - Do you just want to redirect to a JSP page and have no controller at all?
- These are the types of questions to ask yourself to help you narrow down the choices.

Class diagram showing a partial list of Spring controllers



Spring MVC's selection of controller classes.

Controller type	Classes	Useful when...
View	ParameterizableViewController UrlFilenameViewController	Your controller only needs to display a static view—no processing or data retrieval is needed.
Simple	Controller (interface) AbstractController	Your controller is extremely simple, requiring little more functionality than is afforded by basic Java servlets.
Throwaway	ThrowawayController	You want a simple way to handle requests as commands (in a manner similar to WebWork Actions).
Multiaction	MultiActionController	Your application has several actions that perform similar or related logic.
Command	BaseCommandController AbstractCommandController	Your controller will accept one or more parameters from the request and bind them to an object. Also capable of performing parameter validation.
Form	AbstractFormController SimpleFormController	You need to display an entry form to the user and also process the data entered into the form.
Wizard	AbstractWizardFormController	You want to walk your user through a complex, multipage entry form that ultimately gets processed as a single form.

Interceptors



- Interceptors are closely related to mappings, as you can specify a list of interceptors that will be called for each mapping.
- HandlerInterceptor implementations can process each request before or after it has been processed by the appropriate controller.
- You can choose to implement the HandlerInterceptor interface or extend HandlerInterceptorAdapter, which provides default do nothing implementations for all HandlerInterceptor methods.

Example: BigBrotherHandlerInterceptor that will process each request.

```
public class BigBrotherHandlerInterceptor extends HandlerInterceptorAdapter  
{  
    public void postHandle(HttpServletRequest request, HttpServletResponse response,  
        Object handler, ModelAndView modelAndView) throws Exception {  
        // process the request  
    }  
}
```

- The actual implementation of such an interceptor would probably process the request parameters and store them in an audit log.

To use the interceptor, we will create a URL mapping and interceptor bean definitions in the Spring application context file as shown below

```
<bean id="bigBrotherHandlerInterceptor"
      class="com.apress.prospring2.ch17.web.BigBrotherHandlerInterceptor"/>
<bean id="publicUrlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="interceptors">
<list>
<ref local="bigBrotherHandlerInterceptor"/>
</list>
</property>
<property name="mappings">
<value>
    /index.html=indexController
    /product/index.html=productController
    /product/view.html=productController
    /product/edit.html=productFormController
</value>
</property>
</bean>
```

View Resolvers

- A ViewResolver is a strategy interface that Spring MVC uses to look up and instantiate an appropriate view based on its name and locale.
- There are various view resolvers that all implement the ViewResolver interface's single method: `View resolveViewName(String viewName, Locale locale)` throws `Exception`.
- This allows your applications to be much easier to maintain. The locale parameter suggests that the ViewResolver can return views for different client locales, which is indeed the case

ViewResolver Implementations

Implementation	Description
BeanNameViewResolver	This simple ViewResolver implementation will try to get the View as a bean configured in the application context. This resolver may be useful for small applications, where you do not want to create another file that holds the view definitions. However, this resolver has several limitations; the most annoying one is that you have to configure the views as Spring beans in the application context. Also, it does not support internationalization.
ResourceBundleViewResolver	This resolver is far more complex. The view definitions are kept in a separate configuration file, so you do not have to configure the View beans in the application context file. This resolver supports internationalization.
UrlBasedViewResolver	This resolver instantiates the appropriate view based on the URL, which can configure the URL with prefixes and suffixes. This resolver gives you more control over views than BeanNameViewResolver but can become difficult to manage in a large application and does not support internationalization.
XmlViewResolver	This view resolver is similar to ResourceBundleViewResolver, as the view definitions are kept in a separate file. Unfortunately, this resolver does not support internationalization.

Choosing a View Resolver

- Many projects rely on JSP (or some other template language) to render the view results.
- Assuming that your application isn't internationalized or that you won't need to display a completely different view based on a user's locale, we recommend InternalResourceViewResolver because it is simply and tersely defined (as opposed to the other view resolvers that require you to explicitly define each view).
- If, however, your views will be rendered using a custom View implementation (e.g., RSS, PDF, Excel, images, etc.), you'll need to consider one of the other view resolvers.
- We favor BeanNameViewResolver and XmlFileViewResolver over ResourceBundleViewResolver because they let you define your View beans in a Spring context configuration XML file.
- By defining the View in a Spring application context, you're able to configure it using the same syntax as you use for the other components in your application.
- Given the choice between BeanNameViewResolver and XmlFileViewResolver, I'd settle on BeanNameViewResolver only when you have a handful of View beans that would not significantly increase the size of DispatcherServlet's context file.
- If the view resolver is managing a large number of View objects, I'd choose XmlFileViewResolver to separate the View bean definitions into a separate file. In the rare case that you must render a completely different view depending on a user's locale, you have no choice but to use ResourceBundleViewResolver.

Spring and Other Web Technologies

- In the previous sections, we used JSP pages to generate output that is sent to the client's browser for rendering.
- We could naturally build the entire application using just JSP pages, but the application and JSP pages would probably become too complex to manage.
- Even though JSP pages are very powerful, they can present a considerable processing overhead.
- Because Spring MVC fully decouples the view from the logic, the JSP pages should not contain any Java scriptlets.
- Even if this is the case, the JSP pages still need to be compiled, which is a lengthy operation, and their runtime performance is sometimes not as good as we would like.
- The Velocity templating engine from Apache (we might discuss in the next lectures) is a viable alternative, offering much faster rendering times while not restricting the developer too much.
- In addition to Velocity, we might explore the Tiles framework, which allows you to organize the output generated by the controllers into separate components that can be assembled together using a master template. This greatly simplifies the visual design of the application, and any changes to the overall layout of the output can be made very quickly with fewer coding mistakes and easier code management.

Spring Web Flow

- Almost every web application developer nowadays must have been confronted with the requirement to limit the user's navigational freedom and guide the user through a series of consecutive pages in a specific way for a business process to be completed.
- If you haven't had to implement such a process yourself yet, you have certainly participated in one the last time you placed an order with your favorite online retailer or booked a flight online.

Basic flowchart of a simplified version of such an airline ticket booking process

At the beginning, the user can search for flights until she has picked a suitable one. So far, the process is pretty straightforward. However, by confirming her flight selection, she enters a more complex booking process involving a set of steps that all need to be completed successfully before the selected flight can be booked. In our simple example, the user will have to enter her personal details correctly before she is asked to provide the airline with payment details. Once those details have been accepted, a final confirmation is requested before the tickets are booked, and the user can finally start looking forward to visiting her travel destination.



- This is a very simple example, but we're sure you get the idea.
- Page sequences like this and more complex conversations usually require some sort of state management. HTTP is a stateless protocol, meaning that each request is completely independent of previous or later requests. Information is passed on through request parameters or session attributes. Achieving stateful navigational control spanning a sequence of pages in a stateless environment can be quite cumbersome.
- There are other situations that can also cause problems in a web application.
- What if a user in the example flow entered the postal code of his old address in the personal details form, but only realized it after submitting his entries? Even if a link to the previous page is provided, many users will just click the Back button to go back. Theoretically, this should prompt the browser to display the last page purely retrieved from its own cache, but in practice, all browsers implement their own strategies. Some browsers even reload data from the server. Surely a web application should behave the same with all browsers; and especially in a way that the web developer can predict.
- Another situation of concern is related to a user moving through the pages in the other direction. By knowing the correct URLs and the parameters that these URLs expect, a user can theoretically hop from one stage of a process to another while leaving out other stages in between.
- In our ticket booking example, we would want to make sure the user can't take an illegal shortcut and just skip, say, the page for entering payment details.

Another Common Problem

- To mention a last common problem in web applications, think of a situation where a page seems to hang after you click a link or submit a form. Instead of just waiting in front of the screen, most of us would probably press the refresh button. The undesirable state this can lead to, especially if your last request was a POST request, is known as the *double-submit problem*.
- *When you were just posting a comment for a blog, the worst thing that could happen was to post the same comment twice.*
- People might think you're an impatient user; but now imagine if your last post had nothing to do with a blog but was a confirmation to take money out of your account. How painful could that be?

why do we list all these problems?



Web Flow

- In the next lecture, we will start a Spring module that offers solutions to all of them.
- Spring Web Flow is a controller framework for implementing page flows in web applications that are based on MVC frameworks like Spring MVC, Struts, or JSF.

Summary

- J2EE development can and should be simpler
 - Priorities include testability and simplified API
 - Should move to a POJO model
 - Lightweight containers make this reality today!
- Spring is the leading lightweight container
 - Robust and mature
 - Makes J2EE development much simpler
 - Does not mean sacrificing the power of the J2EE platform

Servlet Basics

- Evaluating servlets vs. other technologies
- Understanding the role of servlets
- Building Web pages dynamically
- Looking at servlet code
- A servlet that generates HTML
- The basic structure of servlets
- The servlet life cycle
- How to deal with multithreading problems
- In-Class Exercise

The Advantages of Servlets Over “Traditional” CGI

- Java servlets are more efficient, easier to use, more powerful, more portable, safer, and cheaper than traditional CGI and many alternative CGI-like technologies.

1. Efficient

- With traditional CGI, a new process is started for each HTTP request. If the CGI program itself is relatively short, the overhead of starting the process can dominate the execution time.
- With servlets, the Java virtual machine stays running and handles each request with a lightweight Java thread, not a heavyweight operating system process.
- Similarly, in traditional CGI, if there are N requests to the same CGI program, the code for the CGI program is loaded into memory N times.
- With servlets, however, there would be N threads, but only a single copy of the servlet class would be loaded. This approach reduces server memory requirements and saves time by instantiating fewer objects.
- Finally, when a CGI program finishes handling a request, the program terminates. This approach makes it difficult to cache computations, keep database connections open, and perform other optimizations that rely on persistent data.
- Servlets, however, remain in memory even after they complete a response, so it is straightforward to store arbitrarily complex data between client requests.

2. Convenient

- Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such high-level utilities.
- In CGI, you have to do much of this yourself.
- Besides, if you already know the Java programming language, why learn Perl too? You're already convinced that Java technology makes for more reliable and reusable code than does Visual Basic, VBScript, or C++. Why go back to those languages for server-side programming?

3. Powerful

- Servlets support several capabilities that are difficult or impossible to accomplish with regular CGI.
- Servlets can talk directly to the Web server, whereas regular CGI programs cannot, at least not without using a server-specific API.
- Communicating with the Web server makes it easier to translate relative URLs into concrete path names, for instance.
- Multiple servlets can also share data, making it easy to implement database connection pooling and similar resource-sharing optimizations.
- Servlets can also maintain information from request to request, simplifying techniques like session tracking and caching of previous computations.

4. Portable

- Servlets are written in the Java programming language and follow a standard API.
- *Servlets are supported directly or by a plugin on virtually every major Web server.*
- *Consequently,* servlets written for, say, Macromedia JRun can run virtually unchanged on
 - Apache Tomcat,
 - Microsoft Internet Information Server (with a separate plugin),
 - IBM WebSphere
 - iPlanet Enterprise Server,
 - Oracle9i AS,
 - StarNine WebStar.
- They are part of the Java 2 Platform, Enterprise Edition (J2EE), so industry support for servlets is becoming even more pervasive.

5. Inexpensive

- A number of free or very inexpensive Web servers are good for development use or deployment of low- or medium-volume Web sites.
- Thus, with servlets and JSP you can start with a free or inexpensive server and migrate to more expensive servers with high-performance capabilities or advanced administration utilities only after your project meets initial success.
- This is in contrast to many of the other CGI alternatives, which require a significant initial investment for the purchase of a proprietary package

6. Secure

- One of the main sources of vulnerabilities in traditional CGI stems from the fact that the programs are often executed by general-purpose operating system shells.
- So, the CGI programmer must be careful to filter out characters such as backquotes and semicolons that are treated specially by the shell. Implementing this precaution is harder than one might think, and weaknesses stemming from this problem are constantly being uncovered in widely used CGI libraries.
- A second source of problems is the fact that some CGI programs are processed by languages that do not automatically check array or string bounds. For example, in C and C++ it is perfectly legal to allocate a 100-element array and then write into the 999th “element,” which is really some random part of program memory.
- *So, programmers who forget to perform this check open up their system to deliberate or accidental buffer overflow attacks.*
- *Servlets suffer from neither of these problems. Even if a servlet executes a system call (e.g., with Runtime.exec or JNI) to invoke a program on the local operating system, it does not use a shell to do so. And, of course, array bounds checking and other memory protection features are a central part of the Java programming language.*

7. Mainstream

- There are a lot of good technologies out there.
- But if vendors don't support them and developers don't know how to use them, what good are they?
- Servlet and JSP technology is supported by servers from Apache, Oracle, IBM, Sybase, BEA, Macromedia, Caucho, Sun/iPlanet, New Atlanta, ATG, Fujitsu, Lutris, Silverstream, the World Wide Web Consortium (W3C), and many others. Several low-cost plugins add support to Microsoft IIS and Zeus as well.
- They run on Windows, Unix/Linux, MacOS, VMS, and IBM mainframe operating systems.
- They are arguably the most popular choice for developing medium to large Web applications.
- They are used by the airline industry (most United Airlines and Delta Airlines Web sites), e-commerce (ofoto.com), online banking (First USA Bank, CitiBank), Web search engines/portals (excite.com), large financial sites (American Century Investments), and hundreds of other sites that you visit every day.

A Servlet's Job

Servlets are Java programs that run on Web or application servers, acting as a middle layer between requests coming from Web browsers or other HTTP clients and databases or applications on the HTTP server. Their job is to perform the following tasks, as illustrated in Figure 1–1.

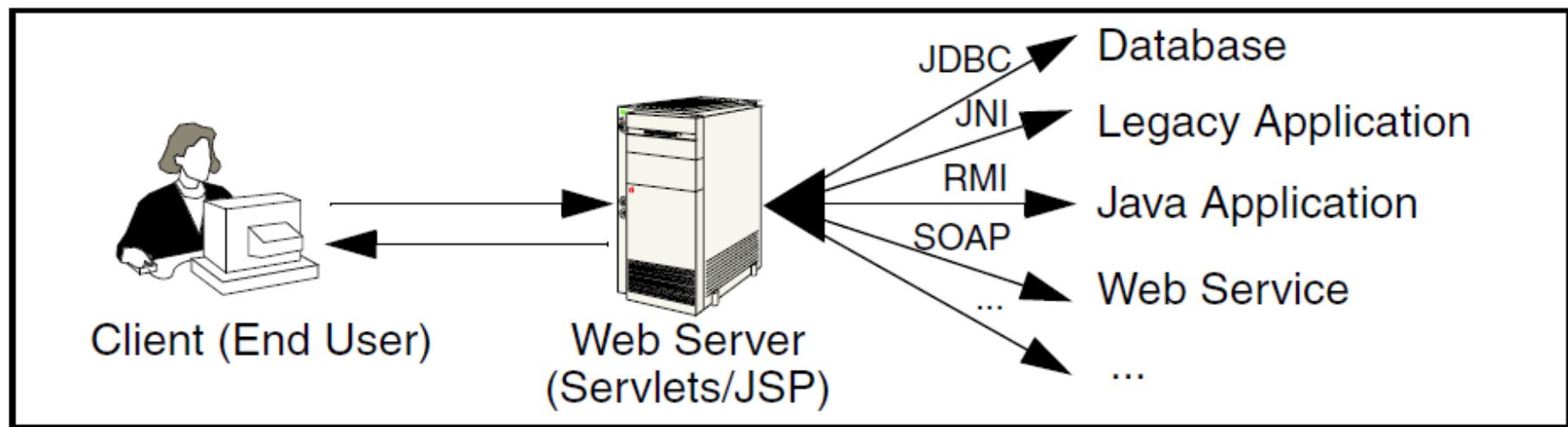


Figure 1–1 The role of Web middleware.

1. Read the explicit data sent by the client.

- The end user normally enters this data in an HTML form on a Web page.

2. Read the implicit HTTP request data sent by the browser.

- Figure 1–1 shows a single arrow going from the client to the Web server (the layer where servlets and JSP execute), but there are really *two varieties of data*:
 - *the explicit data that the end user enters in a form* and
 - the behind-the-scenes HTTP information.
- Both varieties are critical.
- The HTTP information includes cookies, information about media types and compression schemes the browser understands, and so forth; it is discussed later.

3. Generate the results.

- This process may require talking to a database, executing an RMI or EJB call, invoking a Web service, or computing the response directly.
- Your real data may be in a relational database.
- Fine. But your database probably doesn't speak HTTP or return results in HTML, so the Web browser can't talk directly to the database.
- Even if it could, for security reasons, you probably would not want it to.
- The same argument applies to most other applications. You need the Web middle layer to extract the incoming data from the HTTP stream, talk to the application, and embed the results inside a document.

4. Send the explicit data (i.e., the document) to the client.

- This document can be sent in a variety of formats, including text (HTML or XML), PDF, binary (GIF images), or even a compressed format like gzip that is layered on top of some other underlying format.
- But, HTML is by far the most common format, so an important servlet/JSP task is to wrap the results inside of HTML.

5. Send the implicit HTTP response data.

- Figure 1–1 shows a single arrow going from the Web middle layer (the servlet or JSP page) to the client.
- But, there are really *two varieties* of data sent:
 - the document itself and
 - the behind-the-scenes HTTP information.
- Again, both varieties are critical to effective development.
- Sending HTTP response data involves telling the browser or other client what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.
- *These tasks are discussed in week 4*

Why Build Web Pages Dynamically?

- **The Web page is based on data sent by the client.**
For instance, the results page from search engines and order-confirmation pages at online stores are specific to particular user requests. You don't know what to display until you read the data that the user submits. Just remember that the user submits two kinds of data: explicit (i.e., HTML form data) and implicit (i.e., HTTP request headers). Either kind of input can be used to build the output page. In particular, it is quite common to build a user-specific page based on a cookie value.
- **The Web page is derived from data that changes frequently.**
If the page changes for every request, then you certainly need to build the response at request time. If it changes only periodically, however, you could do it two ways: you could periodically build a new Web page on the server (independently of client requests), or you could wait and only build the page when the user requests it. The right approach depends on the situation, but sometimes it is more convenient to do the latter: wait for the user request. For example, a weather report or news headlines site might build the pages dynamically, perhaps returning a previously built page if that page is still up to date.
- **The Web page uses information from corporate databases or other server-side sources.**
If the information is in a database, you need server-side processing even if the client is using dynamic Web content such as an applet. Imagine using an applet by itself for a search engine site:

HelloServlet.java

A Quick Peek at Servlet Code

Now, this is hardly the time to delve into the depths of servlet syntax. Don't worry, you'll get plenty of that throughout the book. But it is worthwhile to take a quick look at a simple servlet, just to get a feel for the basic level of complexity.

Listing 1.1 shows a simple servlet that outputs a small HTML page to the client. Figure 1–2 shows the result.

The code is explained in detail in Chapter 3 (Servlet Basics), but for now, just notice four points:

- **It is regular Java code.** There are new APIs, but no new syntax.
- **It has unfamiliar import statements.** The servlet and JSP APIs are not part of the Java 2 Platform, Standard Edition (J2SE); they are a separate specification (and are also part of the Java 2 Platform, Enterprise Edition—J2EE).
- **It extends a standard class (`HttpServlet`).** Servlets provide a rich infrastructure for dealing with HTTP.
- **It overrides the `doGet` method.** Servlets have different methods to respond to different types of HTTP commands.

The Servlet Life Cycle

- Only a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate.
- When the servlet is first created, its init method is invoked, so init is where you put one-time setup code. After this, each user request results in a thread that calls the service method of the previously created instance.
- Multiple concurrent requests normally result in multiple threads calling service simultaneously, although your servlet can implement a special interface (SingleThreadModel) that stipulates that only a single thread is permitted to run at any one time.
- The service method then calls doGet, doPost, or another doXxx method, depending on the type of HTTP request it received.
- Finally, if the server decides to unload a servlet, it first calls the servlet's destroy method.

The service Method

- Each time the server receives a request for a servlet, the server spawns a new thread and calls service.
- The service method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc., as appropriate.
- A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified.
- A POST request results from an HTML form that specifically lists POST as the METHOD.
- Other HTTP requests are generated only by custom clients.
- 99% of the time, you only care about GET or POST requests, so you override doGet and/or doPost.

Handling both POST and GET requests identically

Now, if you have a servlet that needs to handle both POST and GET requests identically, you may be tempted to override `service` directly rather than implementing both `doGet` and `doPost`. This is not a good idea. Instead, just have `doPost` call `doGet` (or vice versa), as below.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet code
}

public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
```

Although this approach takes a couple of extra lines of code, it has several advantages over directly overriding `service`. First, you can later add support for other HTTP request methods by adding `doPut`, `doTrace`, etc., perhaps in a subclass. Overriding `service` directly precludes this possibility. Second, you can add support for modification dates by adding a `getLastModified` method

The init Method

Most of the time, your servlets deal only with per-request data, and `doGet` or `doPost` are the only life-cycle methods you need. Occasionally, however, you want to perform complex setup tasks when the servlet is first loaded, but not repeat those tasks for each request. The `init` method is designed for this case; it is called when the servlet is first created, and *not* called again for each user request. So, it is used for one-time initializations, just as with the `init` method of applets. The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

The `init` method definition looks like this:

```
public void init() throws ServletException {  
    // Initialization code...  
}
```

The `init` method performs two varieties of initializations: general initializations and initializations controlled by initialization parameters.

The destroy Method

- The server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator or perhaps because the servlet is idle for a long time.
- Before it does, however, it calls the servlet's `destroy` method.
- This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.
- Be aware, however, that it is possible for the Web server to crash (remember those Massachusetts power outages?). So, don't count on `destroy` as the *only mechanism for saving state to disk*.
- *If your servlet performs* activities like counting hits or accumulating lists of cookie values that indicate special access, you should also proactively write the data to disk periodically.

The SingleThreadModel Interface

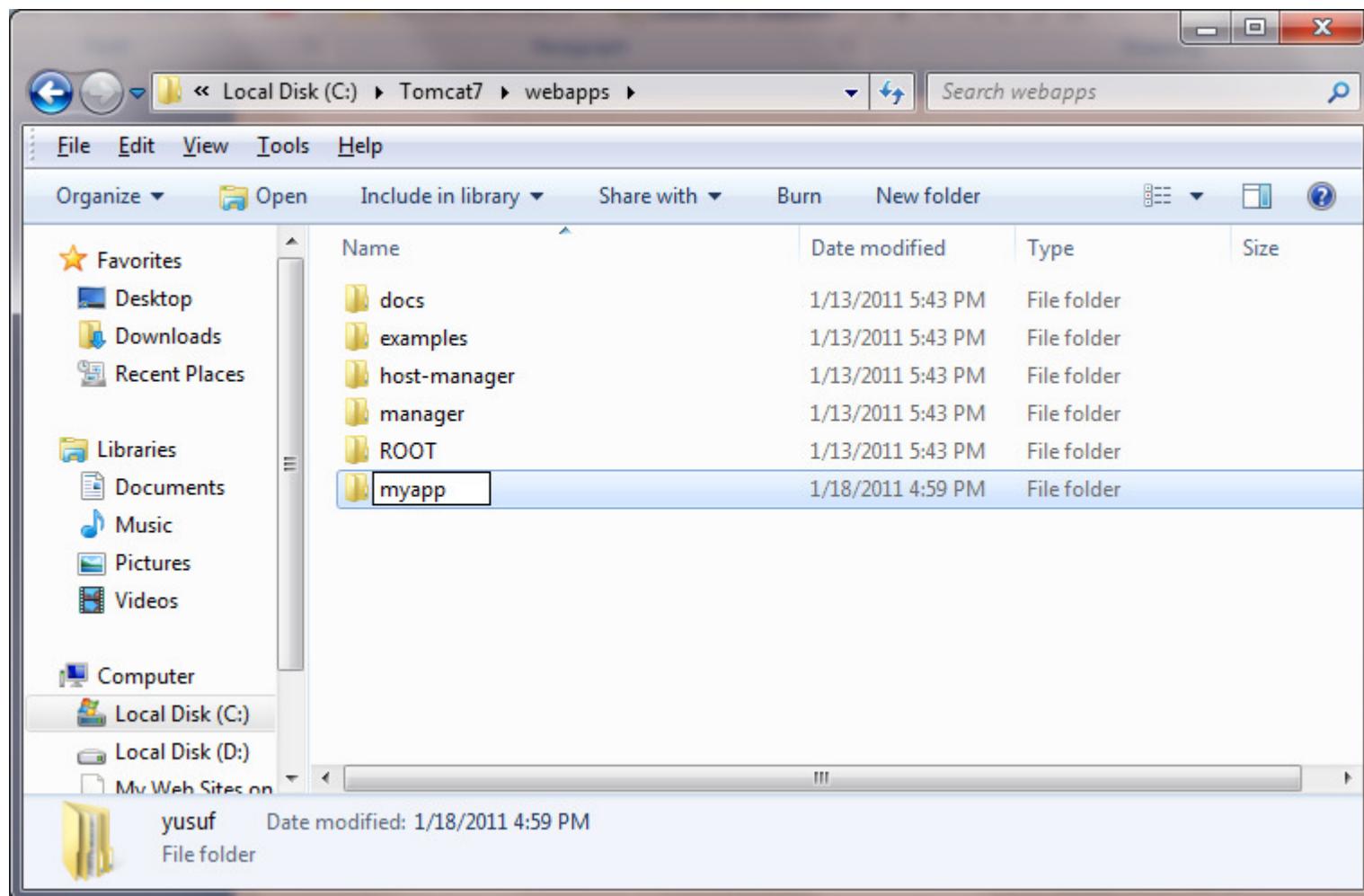
- Normally, the system makes a single instance of your servlet and then creates a new thread for each user request. This means that if a new request comes in while a previous request is still executing, multiple threads can concurrently be accessing the same servlet object.
- Consequently, your doGet and doPost methods must be careful to synchronize access to fields and other shared data (if any) since multiple threads may access the data simultaneously.
- Note that local variables are not shared by multiple threads, and thus need no special protection. In principle, you can prevent multithreaded access by having your servlet implement the SingleThreadModel interface, as below.

```
public class YourServlet extends HttpServlet implements SingleThreadModel  
{ ... }
```

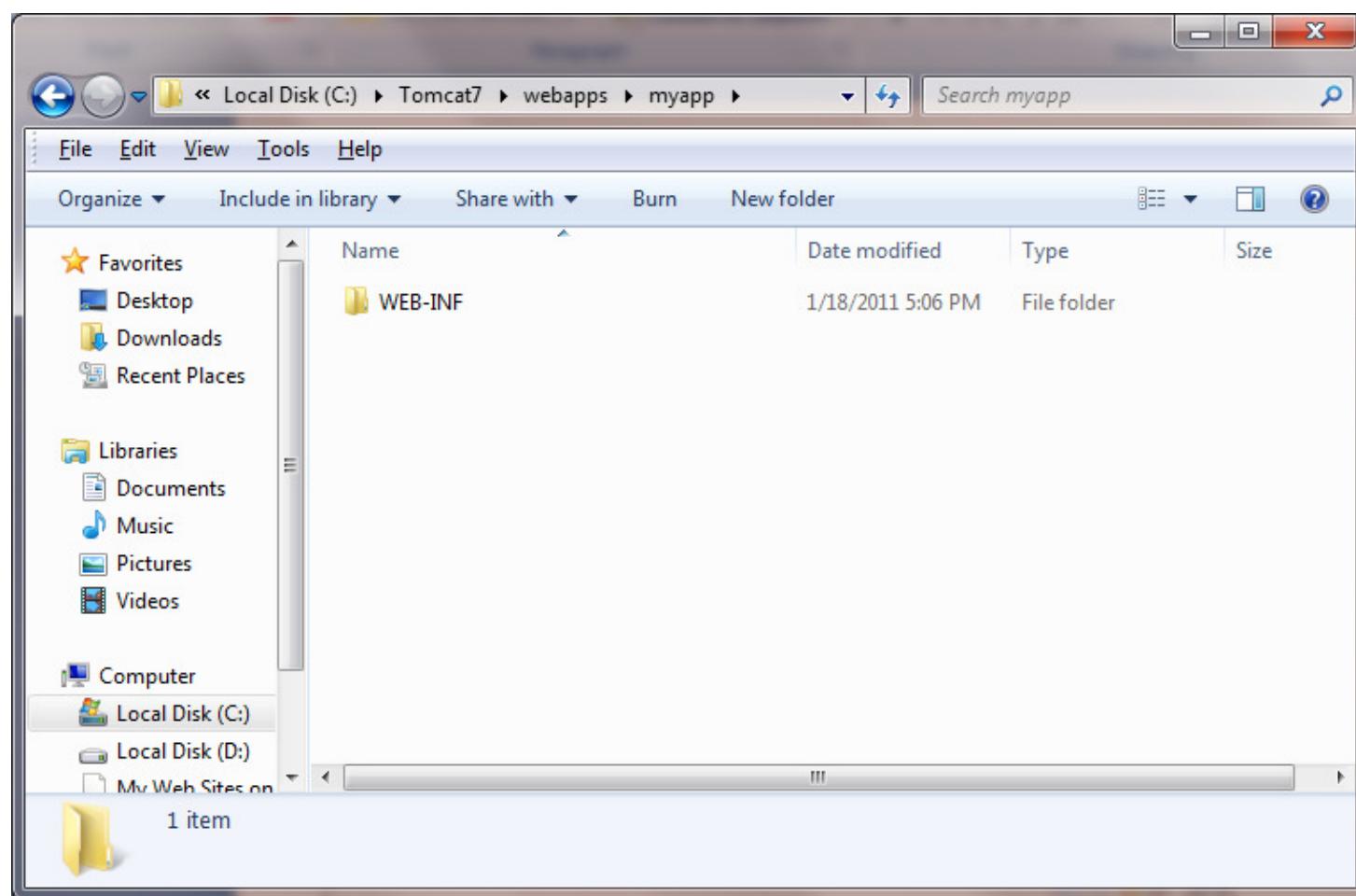
- If you implement this interface, the system guarantees that there is never more than one request thread accessing a single instance of your servlet.
- Although SingleThreadModel prevents concurrent access in principle, in practice there are two reasons why it is usually a poor choice.
 1. *First, synchronous access to your servlets can significantly hurt performance (latency) if your servlet is accessed frequently.*
 2. *The second problem with SingleThreadModel stems from the fact that the specification permits servers to use pools of instances instead of queueing up the requests to a single instance. As long as each instance handles only one request at a time, the pool-of-instances approach satisfies the requirements of the specification. But, it is a bad idea.*

In-Class Exercise

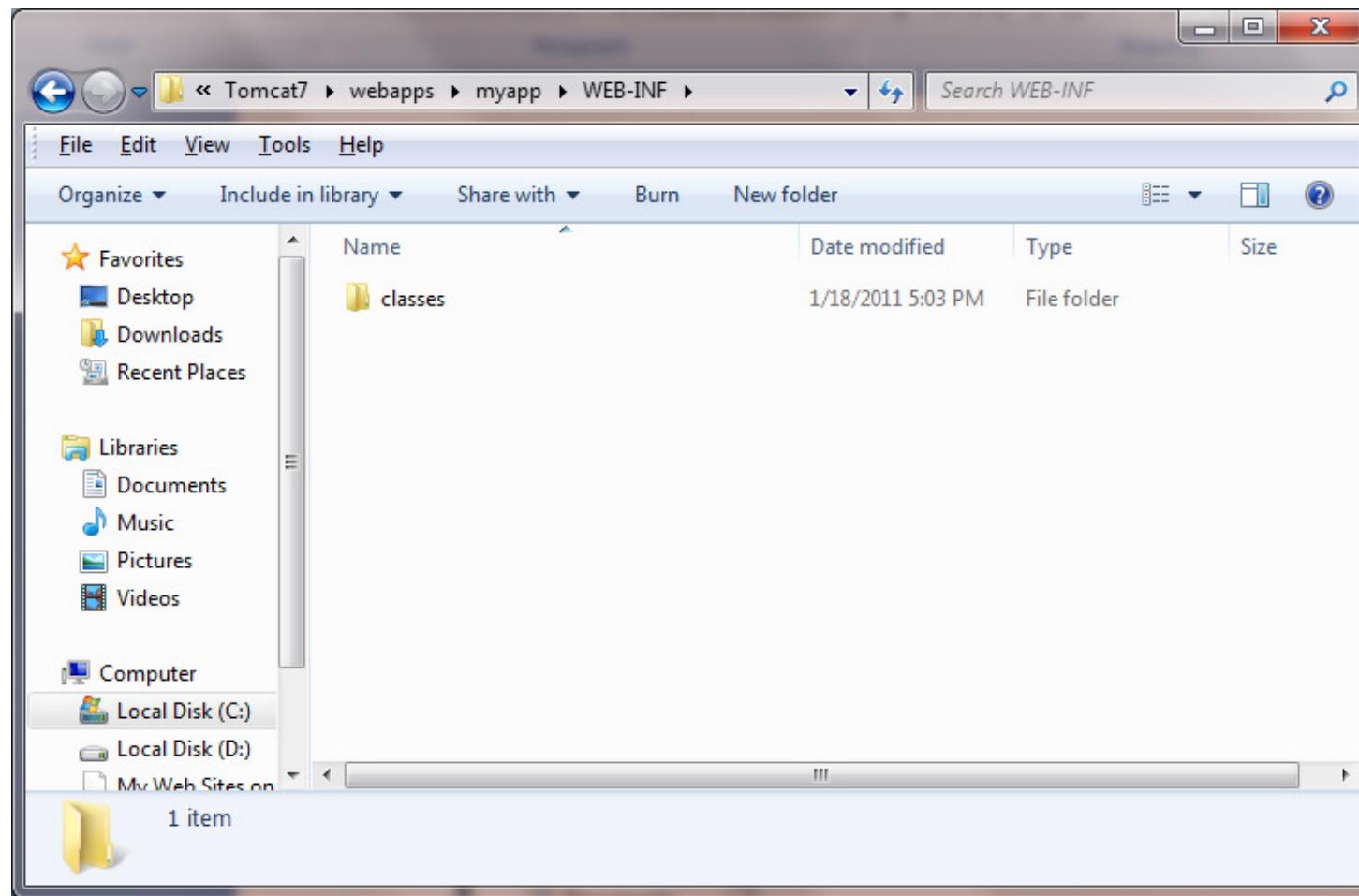
1. Create a new app under C:\yourTomcatInstallation\webapps



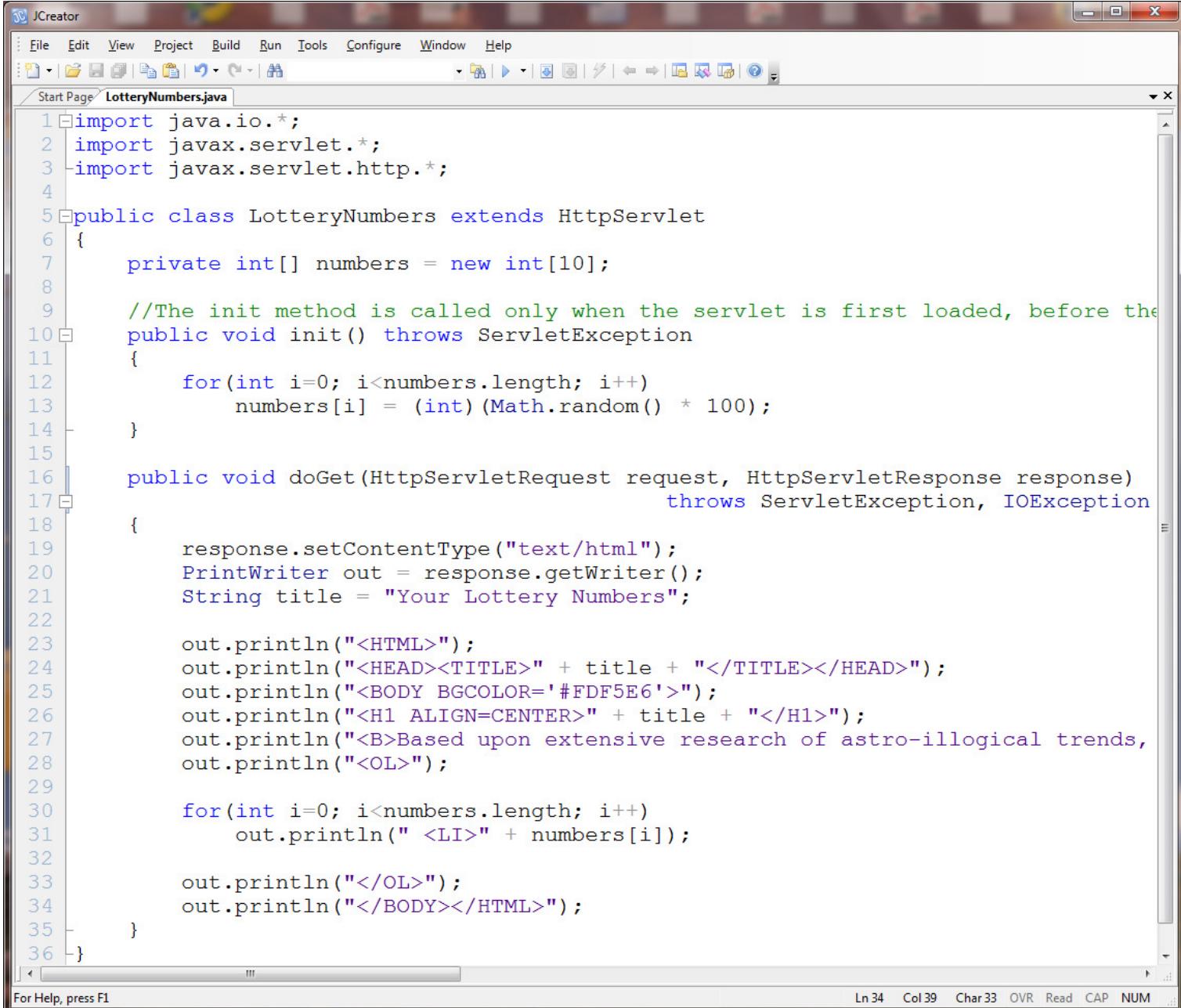
2. Create the WEB-INF directory



3. Create classes directory under WEB-INF



4. Create and Compile Your Servlet

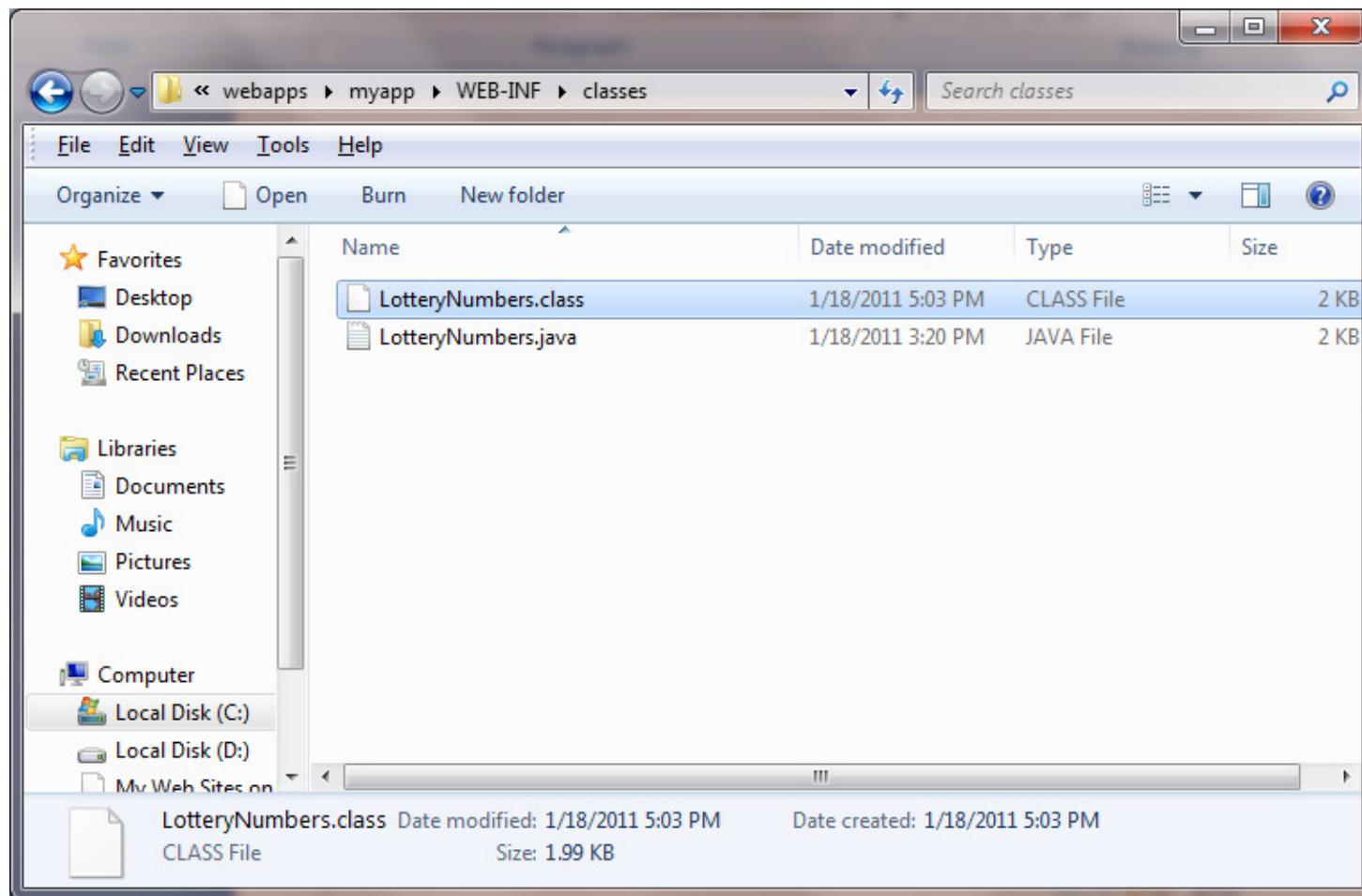


The screenshot shows the JCreator IDE interface with the file `LotteryNumbers.java` open. The code implements a servlet to generate lottery numbers.

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
4
5 public class LotteryNumbers extends HttpServlet
6 {
7     private int[] numbers = new int[10];
8
9     //The init method is called only when the servlet is first loaded, before the
10    public void init() throws ServletException
11    {
12        for(int i=0; i<numbers.length; i++)
13            numbers[i] = (int)(Math.random() * 100);
14    }
15
16    public void doGet(HttpServletRequest request, HttpServletResponse response)
17                     throws ServletException, IOException
18    {
19        response.setContentType("text/html");
20        PrintWriter out = response.getWriter();
21        String title = "Your Lottery Numbers";
22
23        out.println("<HTML>");
24        out.println("<HEAD><TITLE>" + title + "</TITLE></HEAD>");
25        out.println("<BODY BGCOLOR='#FDF5E6'>");
26        out.println("<H1 ALIGN=CENTER>" + title + "</H1>");
27        out.println("<B>Based upon extensive research of astro-illogical trends,</B>");
28        out.println("<OL>");
29
30        for(int i=0; i<numbers.length; i++)
31            out.println(" <LI>" + numbers[i]);
32
33        out.println("</OL>");
34        out.println("</BODY></HTML>");
35    }
36}
```

The code imports necessary packages, defines a `LotteryNumbers` class extending `HttpServlet`, initializes an array of integers, and handles the `doGet` method to print an HTML page with lottery numbers.

5. Compiled servlet needs to be under the classes directory

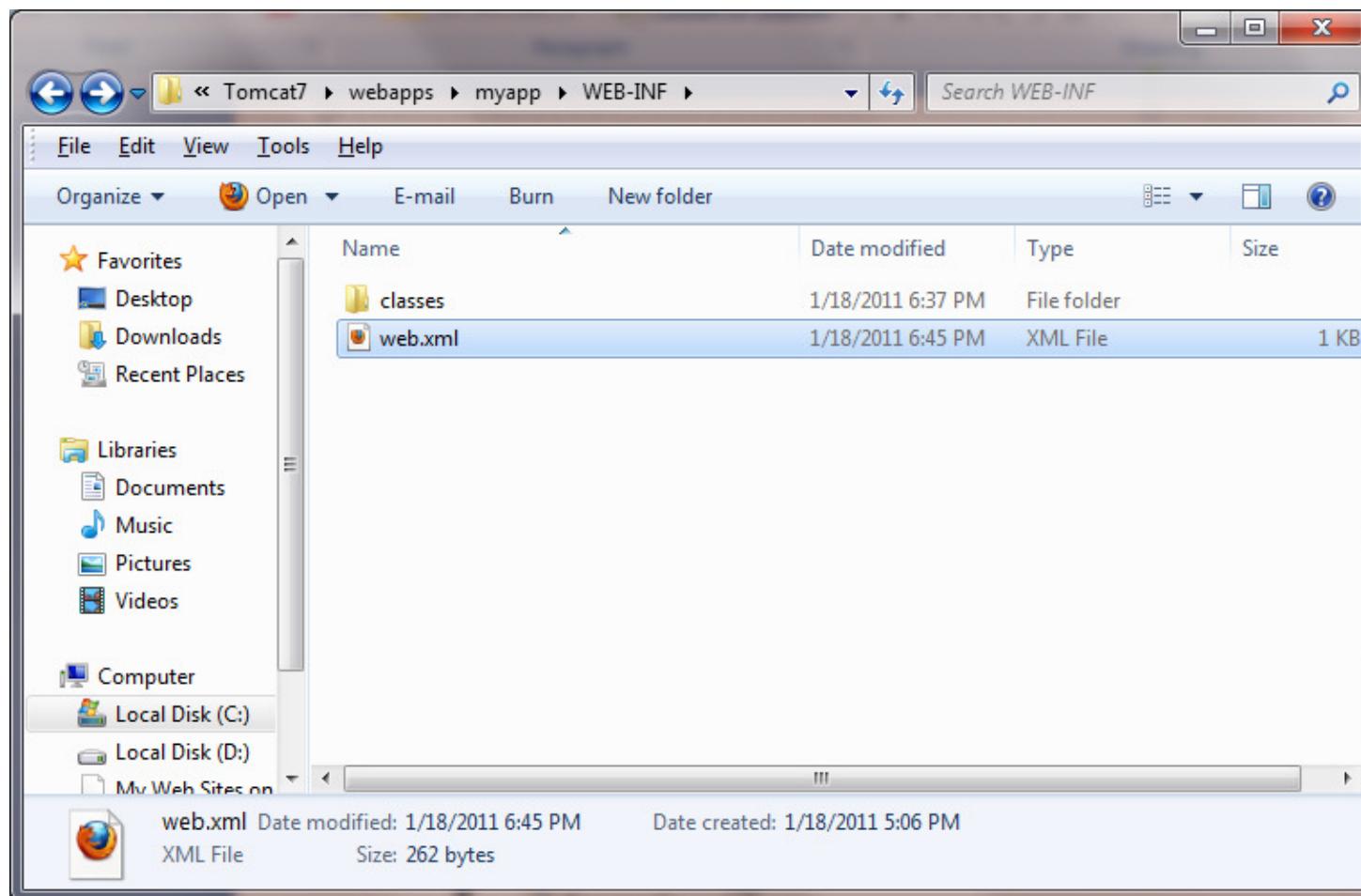


6. Create deployment descriptor web.xml

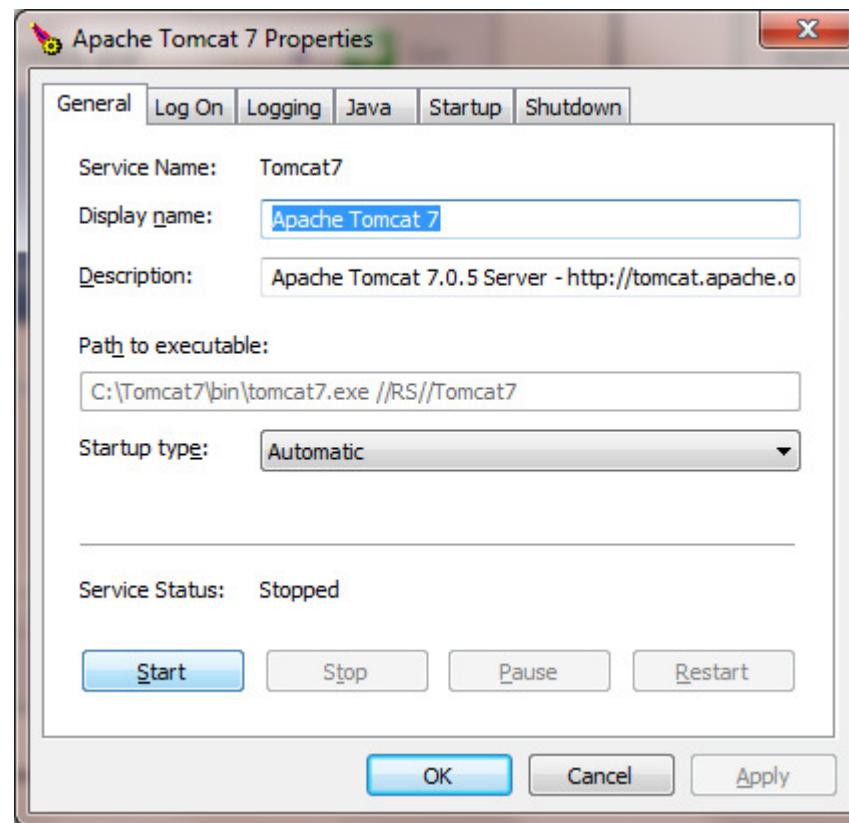
```
<web-app>
  <servlet>
    <servlet-name>lottery</servlet-name>
    <servlet-class>LotteryNumbers</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>lottery</servlet-name>
    <url-pattern>/lottery.do</url-pattern>
  </servlet-mapping>
</web-app>
```

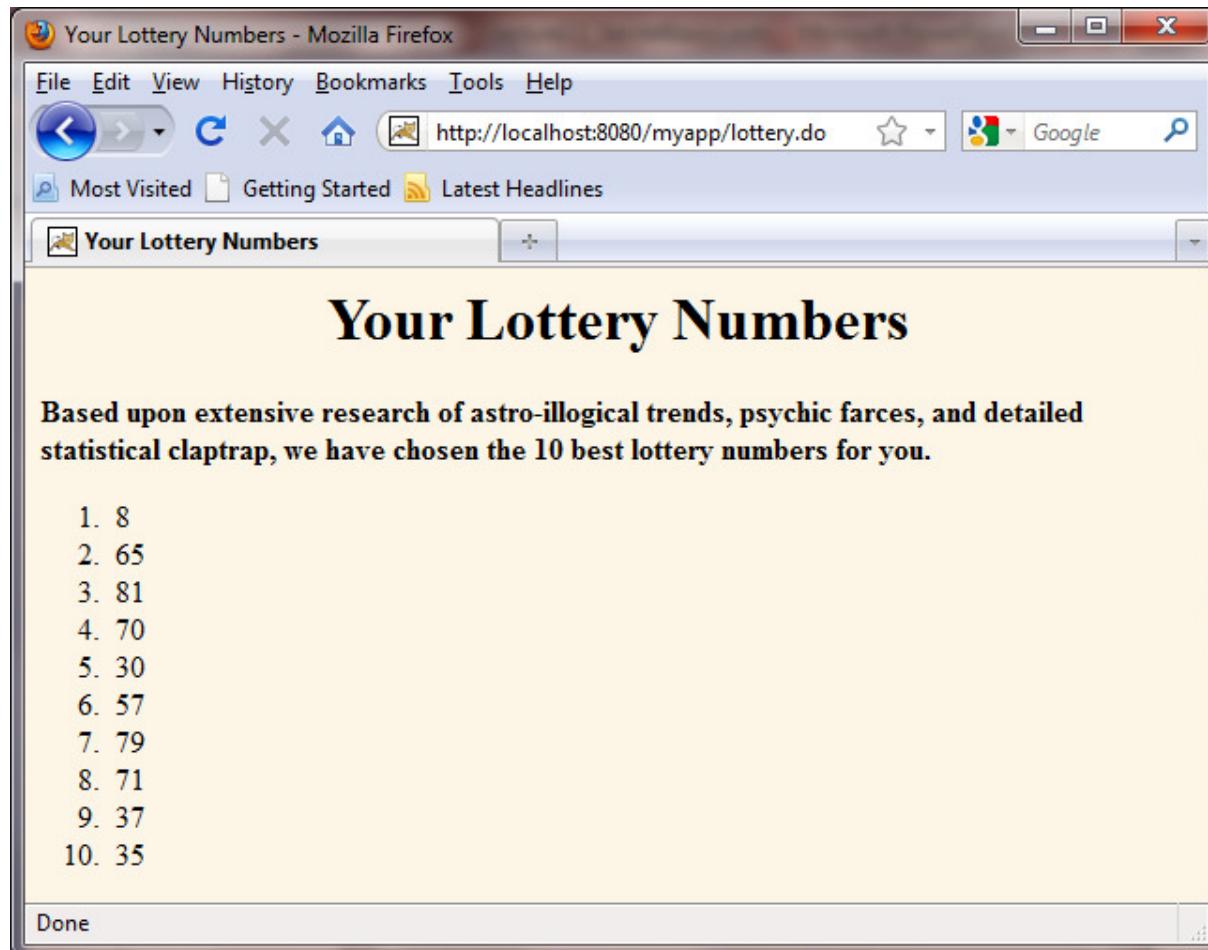
7. Save the web.xml under WEB-INF folder



8. Restart Tomcat



9. Run your web application



Handling The Client Request: Form Data



- One of the main motivations for building Web pages dynamically is so that the result can be based upon user input.
- This lecture will teach you how to access that input.

The Role of Form Data

- If you've ever used a search engine, visited an online bookstore, tracked stocks on the Web, or asked a Web-based site for quotes on plane tickets, you've probably seen funny-looking URLs like
<http://www.amazon.com/gp/search?rh=k%3Aj5ee%2Ci%3Astripbooks&keywords=j5ee&ie=UTF8&qid=1295541226>
- The part after the question mark
[rh=k%3Aj5ee%2Ci%3Astripbooks&keywords=j5ee&ie=UTF8&qid=1295541226](http://www.amazon.com/gp/search?rh=k%3Aj5ee%2Ci%3Astripbooks&keywords=j5ee&ie=UTF8&qid=1295541226) is known as **QUERY STRING** (*form data or query data*) and is the most common way to get information from a Web page to a server-side program.
- Form data can be attached to the end of the URL after a question mark (as above) for GET requests; form data can also be sent to the server on a separate line for POST requests.

Form Basics

1. **Use the FORM element to create an HTML form.** Use the ACTION attribute to designate the address of the servlet or JSP page that will process the results; you can use an absolute or relative URL. For example:

```
<FORM ACTION=" . . ."> . . . </FORM>
```

If ACTION is omitted, the data is submitted to the URL of the current page.

2. **Use input elements to collect user data.** Place the elements between the start and end tags of the FORM element and give each input element a NAME. Textfields are the most common input element; they are created with the following.

```
<INPUT TYPE="TEXT" NAME=" . . .">
```

3. **Place a submit button near the bottom of the form.** For example:

```
<INPUT TYPE="SUBMIT">
```

When the button is pressed, the URL designated by the form's ACTION is invoked. With GET requests, a question mark and name/value pairs are attached to the end of the URL, where the names come from the NAME attributes in the HTML input elements and the values come from the end user. With POST requests, the same data is sent, but on a separate request line instead of attached to the URL.

Do I Use GET or POST ?

- In HTML, one can specify two different submission methods for a form.
- The method is specified inside a FORM element, using the METHOD attribute.
- The difference between METHOD="GET" (the default) and METHOD="POST" is primarily defined in terms of form data encoding.
- There's a mixture of opinion on this one; some people say you should almost never use the GET method, due to its insecurity and limit on size; others maintain that you can use GET to retrieve information, while POST should be used whenever you modify data on the web server.

- One disadvantage of POST is that
 - pages loaded with POST cannot be properly book-marked,
 - whereas pages loaded with GET contain all the information needed to reproduce the request right in the URL.

- Extracting the needed information from this form data is traditionally one of the most tedious parts of server-side programming.
- First of all, before servlets you generally had to read the data one way for GET requests (in traditional CGI, this is usually through the QUERY_STRING environment variable) and a different way for POST requests (by reading the standard input in CGI).
- Second, you have to chop the pairs at the ampersands, then separate the parameter names (left of the equal signs) from the parameter values (right of the equal signs).
- Third, you have to *URL-decode the values*: *reverse the encoding that the browser uses* on certain characters. Alphanumeric characters are sent unchanged by the browser, but spaces are converted to plus signs and other characters are converted to %XX, where XX is the ASCII (or ISO Latin-1) value of the character, in hex. For example, if someone enters a value of “~hall, ~gates, and ~mcnealy” into a textfield with the name users in an HTML form, the data is sent as
 - “users=%7Ehall%2C+%7Egates%2C+and+%7Emcnealy”,
and the server-side program has to reconstitute the original string.
- Finally, the fourth reason that it is tedious to parse form data with traditional server-side technologies is that values can be omitted
 - (e.g., “param1=val1¶m2=¶m3=val3”)
 - or a parameter can appear more than once
 - (e.g., “param1=val1¶m2=val2¶m1=val3”),
so your parsing code needs special cases for these situations.

- Fortunately, servlets help us with much of this tedious parsing.

Reading Form Data from Servlets

- One of the nice features of servlets is that all of this form parsing is handled automatically.
- You call `request.getParameter` to get the value of a form parameter.
- You can also call `request.getParameterValues` if the parameter appears more than once, or
- you can call `request.getParameterNames` if you want a complete list of all parameters in the current request.
- In the rare cases in which you need to read the raw request data and parse it yourself, call `getReader` or `getInputStream`.

Java EE Documentation

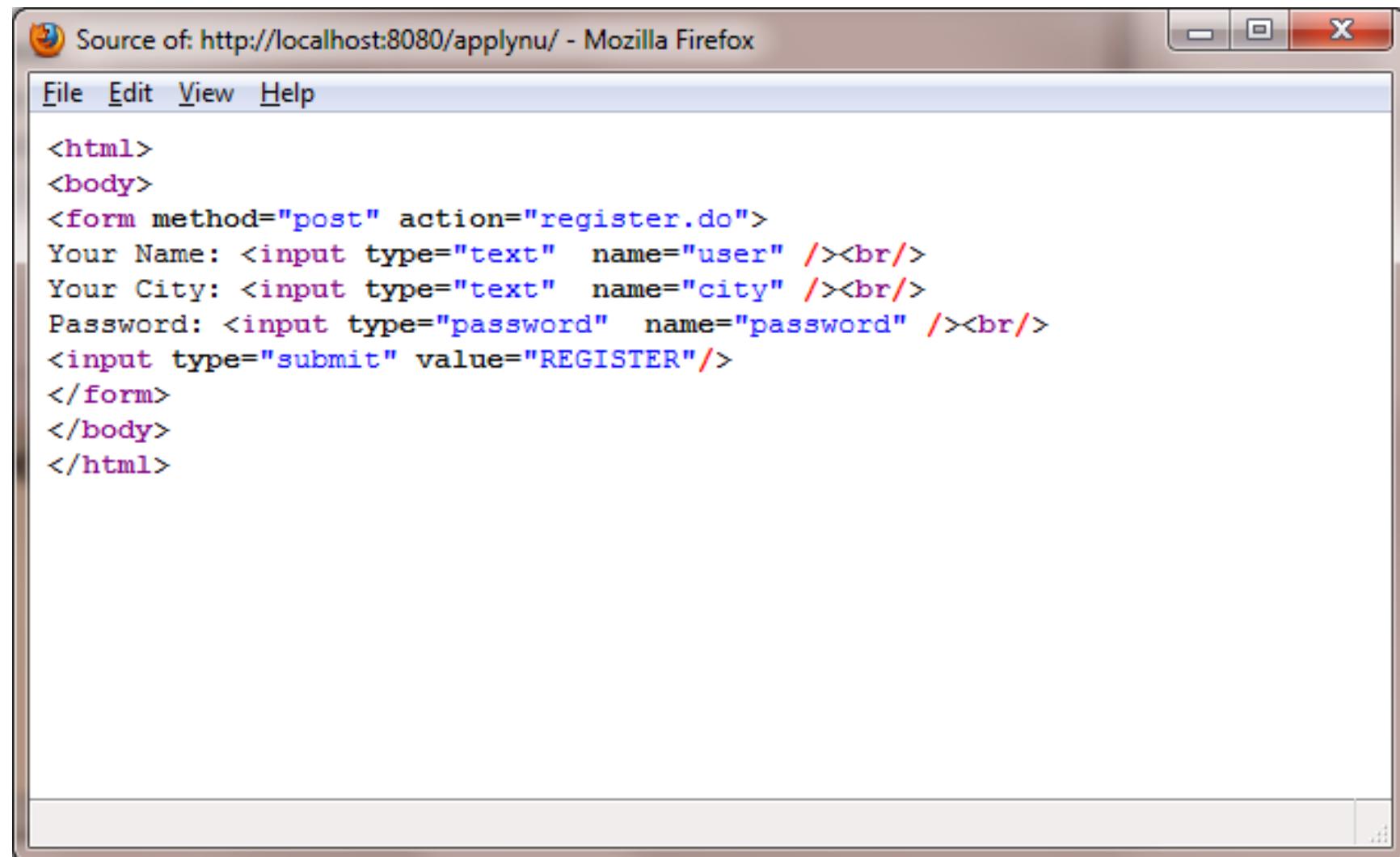


java.lang.String	getParameter(java.lang.String name) Returns the value of a request parameter as a String, or null if the parameter does not exist.
java.util.Map<java.lang.String,java.lang.String[]>	getParameterMap() Returns a java.util.Map of the parameters of this request.
java.util.Enumeration<java.lang.String>	getParameterNames() Returns an Enumeration of String objects containing the names of the parameters contained in this request.
java.lang.String[]	getParameterValues(java.lang.String name) Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist.

Reading Single Values: getParameter

- To read a request (form) parameter, you simply call the `getParameter` method of `HttpServletRequest`, supplying the case-sensitive parameter name as an argument.
- You supply the parameter name exactly as it appeared in the HTML source code, and you get the result exactly as the end user entered it; any necessary URL-decoding is done automatically.
- Unlike the case with many alternatives to servlet technology, you use `getParameter` exactly the same way when the data is sent by GET as you do when it is sent by POST; the servlet knows which request method the client used and automatically uses the appropriate method to read the data.
- An empty String is returned if the parameter exists but has no value (i.e., the user left the corresponding textfield empty when submitting the form), and null is returned if there was no such parameter.
- Parameter names are case sensitive, so the followings are *not* interchangeable.
`request.getParameter("Param1")` and `request.getParameter("param1")`

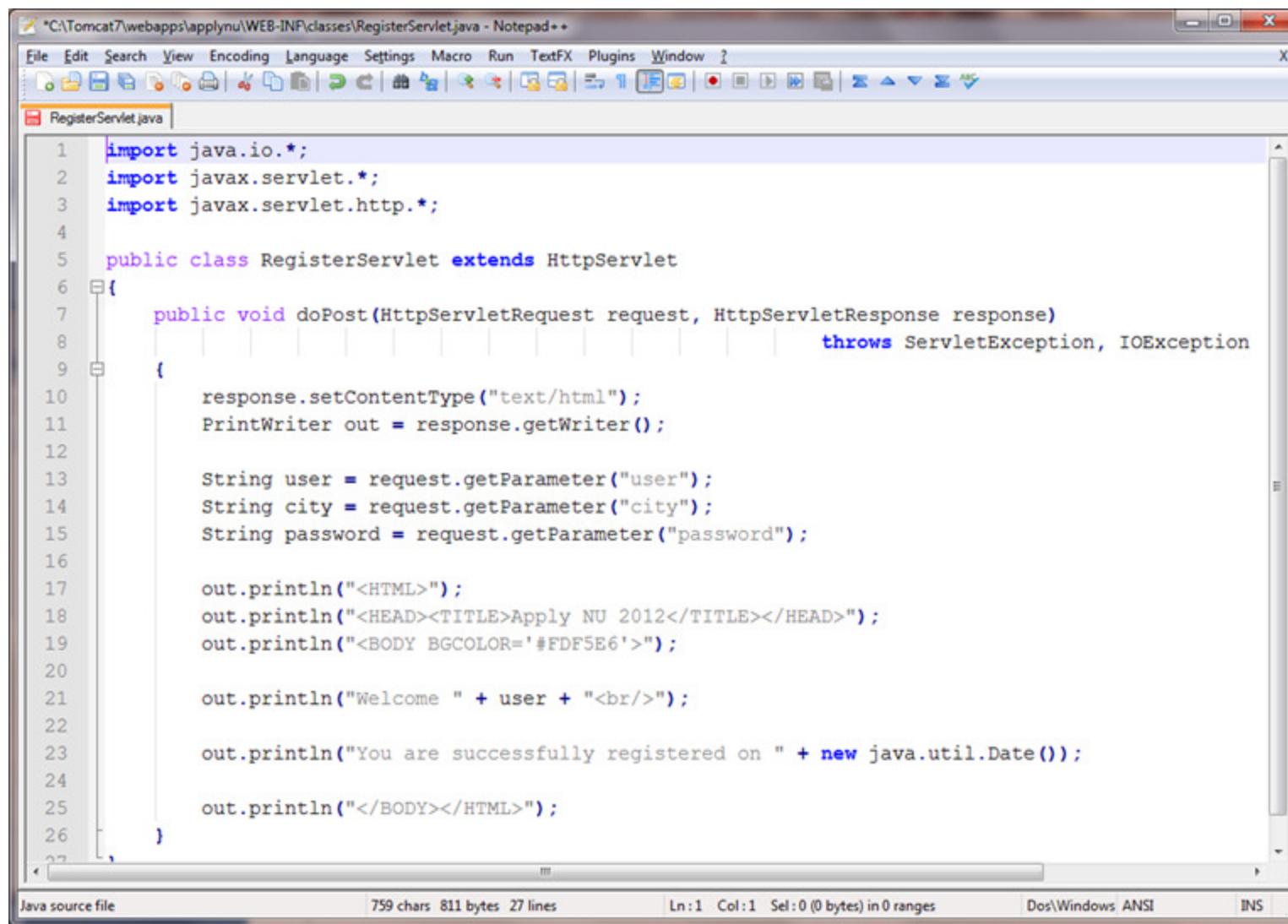
In-Class Exercise



The image shows a Mozilla Firefox browser window with the title "Source of: http://localhost:8080/applynu/ - Mozilla Firefox". The window displays the HTML source code of a registration form. The code includes fields for User Name, City, and Password, along with a "REGISTER" submit button.

```
<html>
<body>
<form method="post" action="register.do">
Your Name: <input type="text" name="user" /><br/>
Your City: <input type="text" name="city" /><br/>
Password: <input type="password" name="password" /><br/>
<input type="submit" value="REGISTER"/>
</form>
</body>
</html>
```

Servlet to handle the request

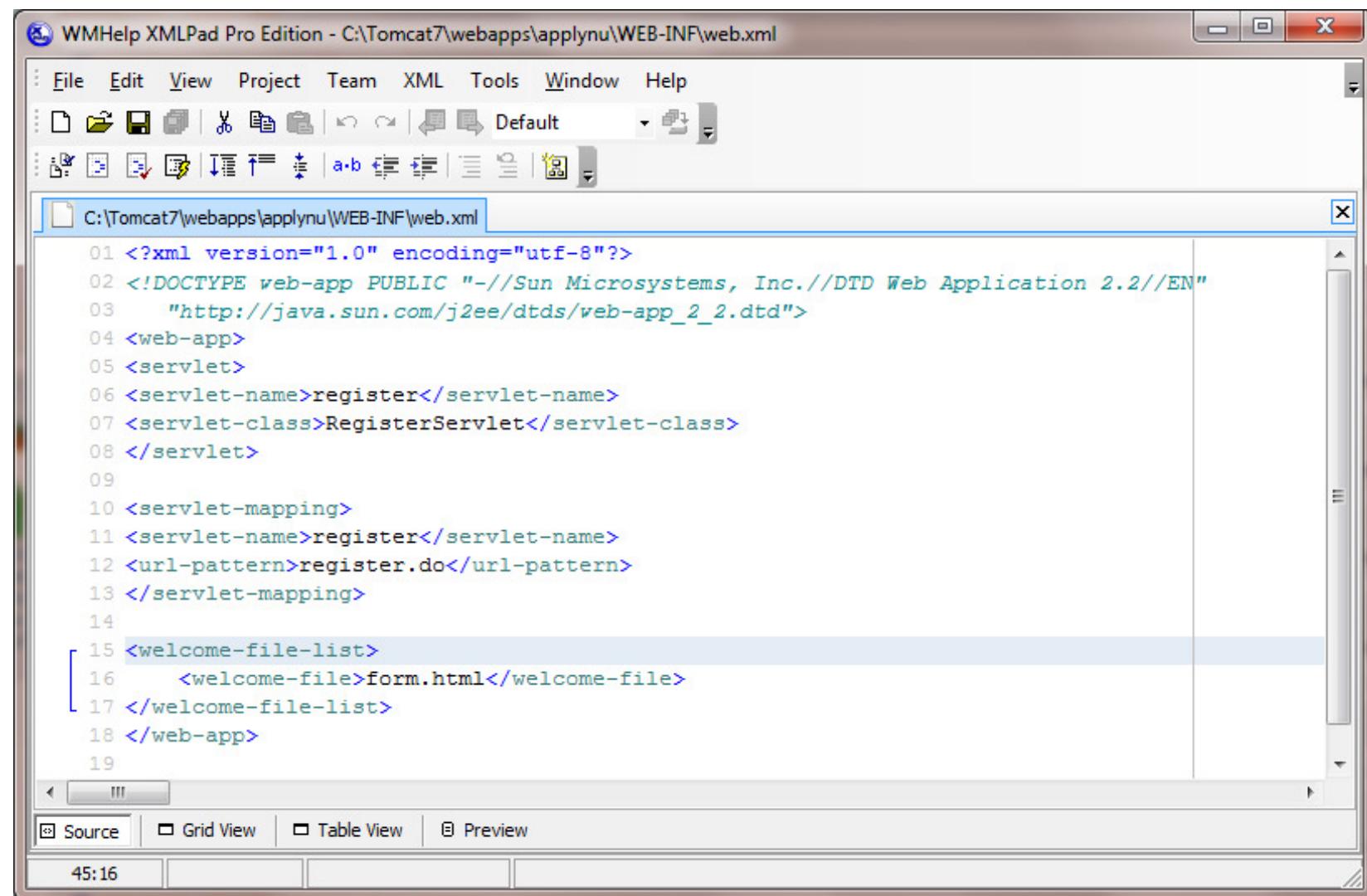


The screenshot shows a Notepad++ window displaying a Java source file named `RegisterServlet.java`. The code implements a servlet to handle registration requests. It imports necessary packages, defines a `RegisterServlet` class extending `HttpServlet`, and overrides the `doPost` method to process user input and generate an HTML response.

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
4
5 public class RegisterServlet extends HttpServlet
6 {
7     public void doPost(HttpServletRequest request, HttpServletResponse response)
8             throws ServletException, IOException
9     {
10        response.setContentType("text/html");
11        PrintWriter out = response.getWriter();
12
13        String user = request.getParameter("user");
14        String city = request.getParameter("city");
15        String password = request.getParameter("password");
16
17        out.println("<HTML>");
18        out.println("<HEAD><TITLE>Apply NU 2012</TITLE></HEAD>");
19        out.println("<BODY BGCOLOR='#FDF5E6'>");
20
21        out.println("Welcome " + user + "<br/>");
22
23        out.println("You are successfully registered on " + new java.util.Date());
24
25        out.println("</BODY></HTML>");
26    }
27}
```

Notepad++ status bar details:
Java source file 759 chars 811 bytes 27 lines Ln:1 Col:1 Sel:0 (0 bytes) in 0 ranges Dos\Windows ANSI INS

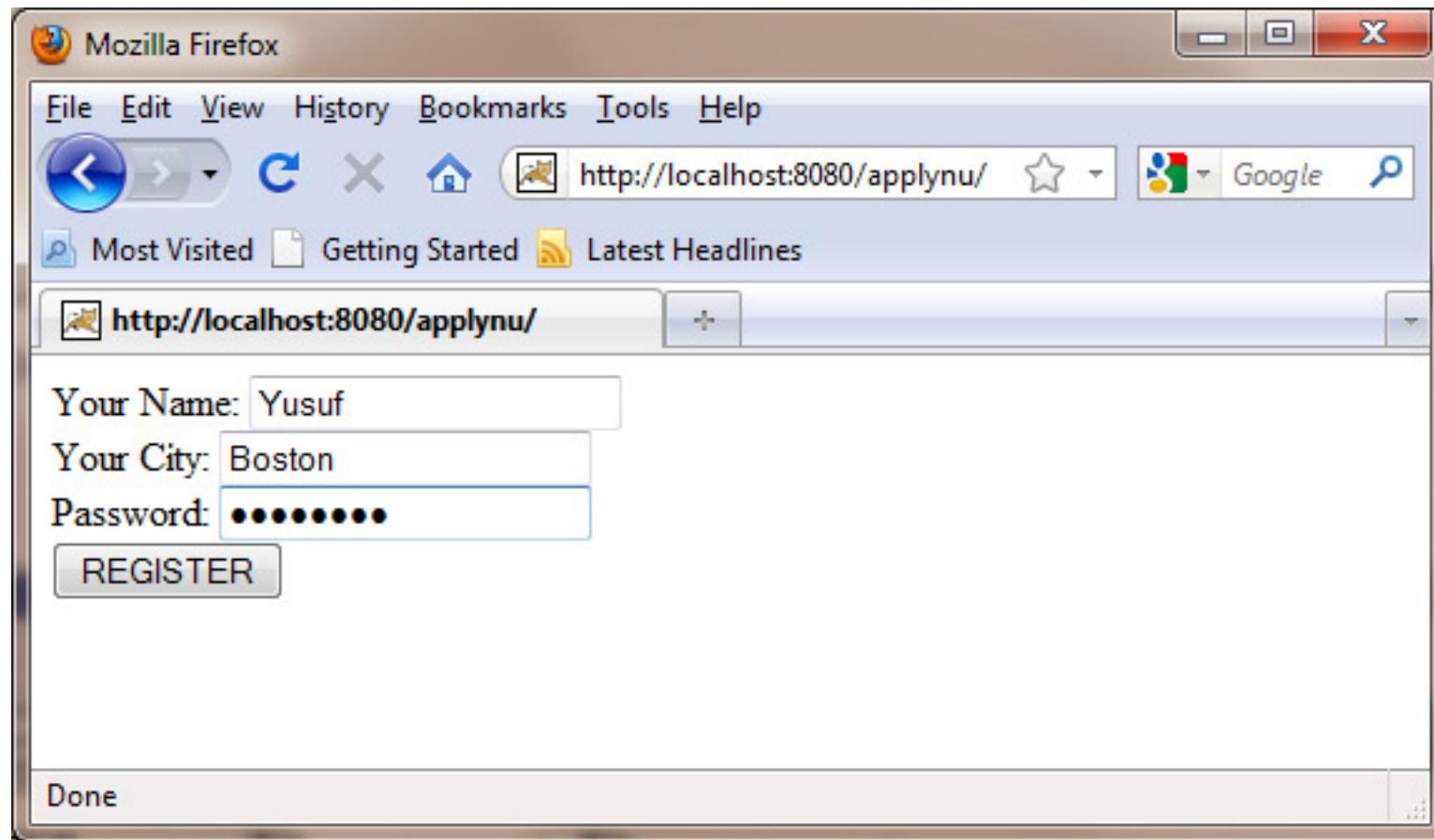
Deployment descriptor – web.xml



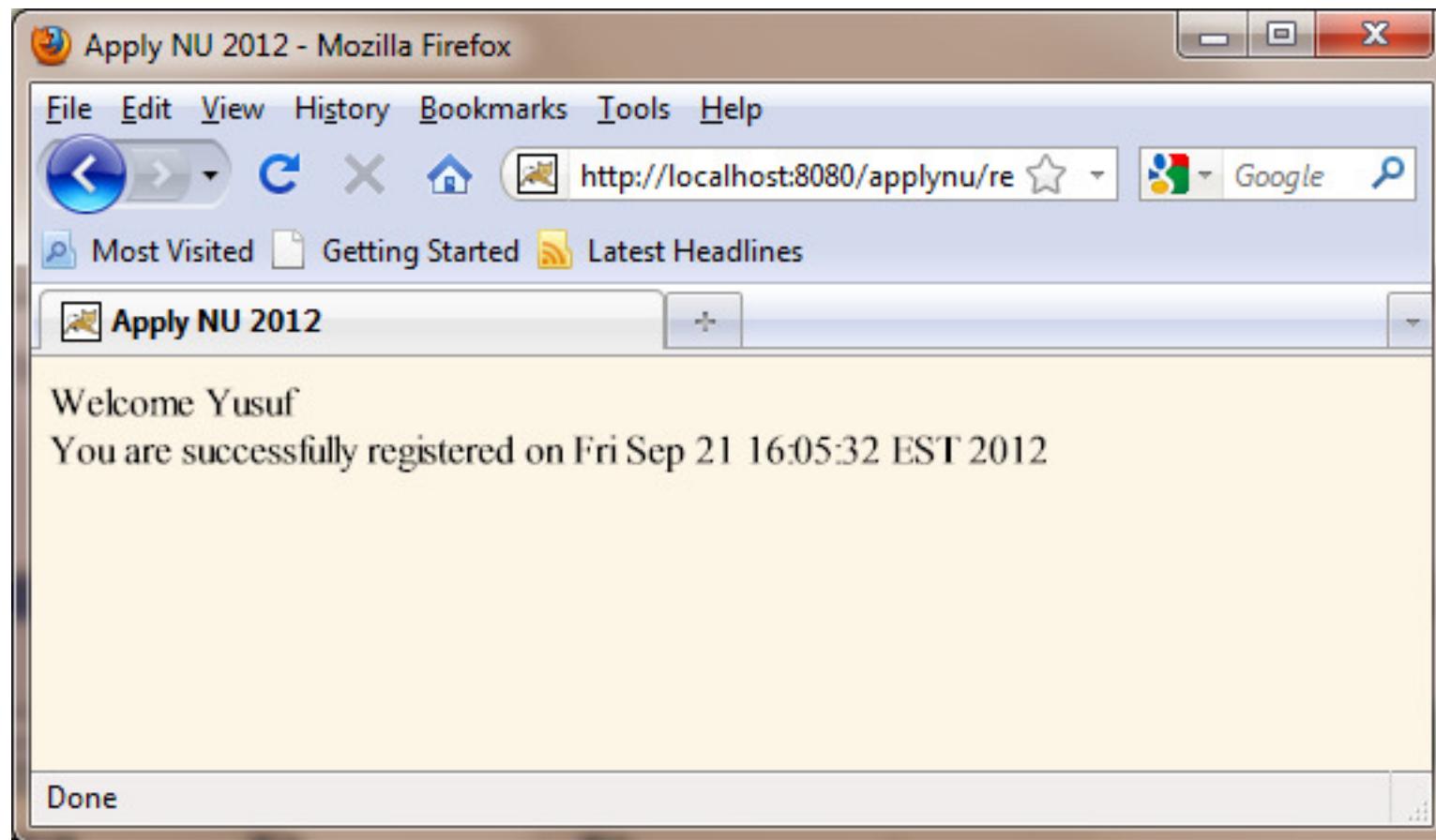
The screenshot shows the WMHelp XMLPad Pro Edition application window. The title bar reads "WMHelp XMLPad Pro Edition - C:\Tomcat7\webapps\applynu\WEB-INF\web.xml". The menu bar includes File, Edit, View, Project, Team, XML, Tools, Window, and Help. The toolbar contains various icons for file operations like Open, Save, Copy, Paste, and Find. The main pane displays the XML code for the web.xml file, which defines a servlet named "register" with a corresponding servlet class "RegisterServlet" and a URL pattern "register.do". It also specifies a welcome file "form.html". The bottom of the window has tabs for Source (selected), Grid View, Table View, and Preview, along with status information like "45:16".

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc./DTD Web Application 2.2//EN"
03   "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
04 <web-app>
05 <servlet>
06 <servlet-name>register</servlet-name>
07 <servlet-class>RegisterServlet</servlet-class>
08 </servlet>
09
10 <servlet-mapping>
11 <servlet-name>register</servlet-name>
12 <url-pattern>register.do</url-pattern>
13 </servlet-mapping>
14
15 <welcome-file-list>
16   <welcome-file>form.html</welcome-file>
17 </welcome-file-list>
18 </web-app>
19
```

Running the servlet



Generating HTML dynamically



Reading Multiple Values: getParameterValues

- If the same parameter name might appear in the form data more than once, you should call `getParameterValues` (which returns an array of strings) instead of `getParameter` (which returns a single string).
- The return value of `getParameterValues` is null for nonexistent parameter names and is a one-element array when the parameter has only a single value.
- Checkboxes, and multiselectable list boxes (i.e., HTML SELECT elements with the MULTIPLE attribute set repeat the parameter name for each selected element in the list return multiple values.

In-Class Exercise

- 
- Reading Multiple Value

Looking Up Parameter Names: `getParameterNames` and `getParameterMap`

- Most servlets look for a specific set of parameter names; in most cases, if the servlet does not know the name of the parameter, it does not know what to do with it either.
- So, your primary tool should be `getParameter`.
- However, it is sometimes useful to get a full list of parameter names. The primary utility of the full list is debugging, but you occasionally use the list for applications where the parameter names are very dynamic.
- For example, the names themselves might tell the system what to do with the parameters (e.g., `row-1-col-3-value`), the system might build a database update assuming that the parameter names are database column names, or the servlet might look for a few specific names and then pass the rest of the names to another application.
- Use `getParameterNames` to get this list in the form of an `Enumeration`, each entry of which can be cast to a `String` and used in a `getParameter` or `getParameterValues` call. If there are no parameters in the current request, `getParameterNames` returns an empty `Enumeration` (not `null`).
- An alternative to `getParameterNames` is `getParameterMap`. This method returns a `Map`: the parameter names (strings) are the table keys and the parameter values (string arrays as returned by `getParameterNames`) are the table values.

In-Class Exercise

- Reading Parameter Names