Web Applications

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
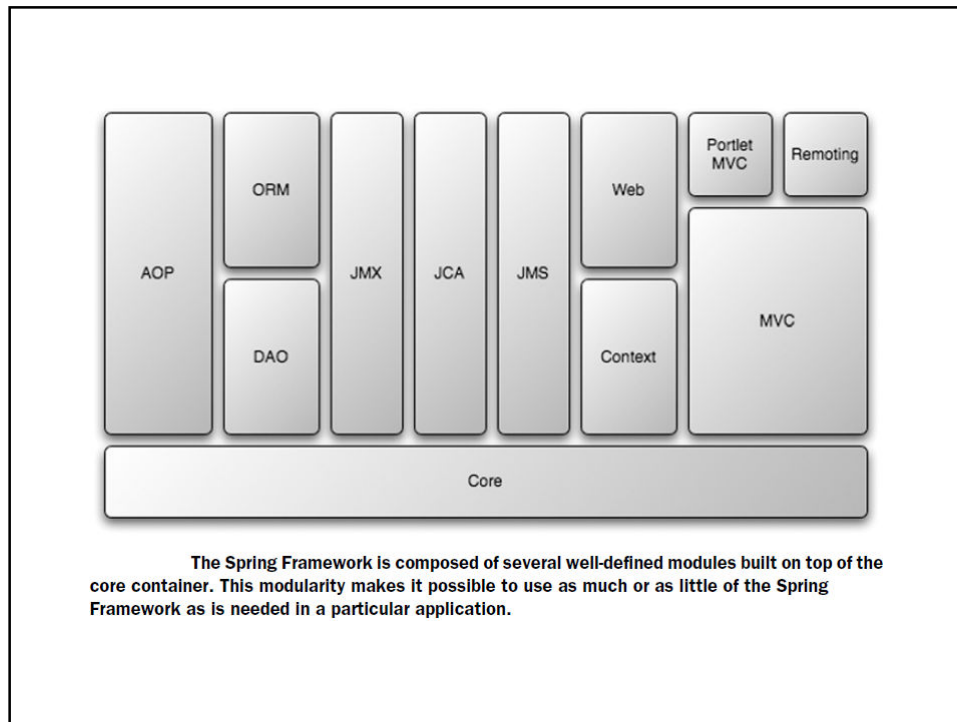
Open Source Frameworks

- 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.
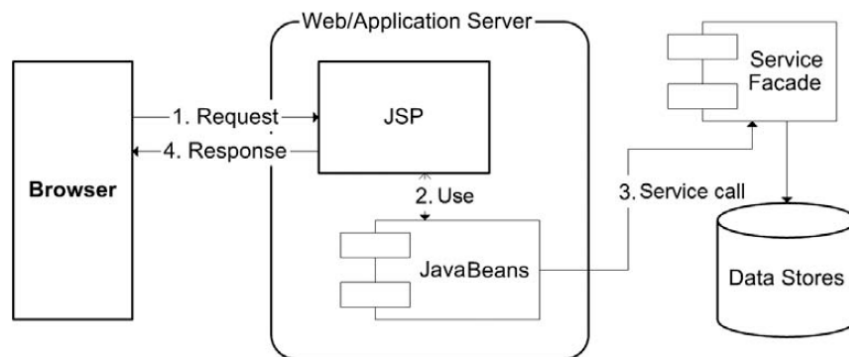
Spring MVC

- 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 execute.

Model 1 Architecture

Model 2 Architecture
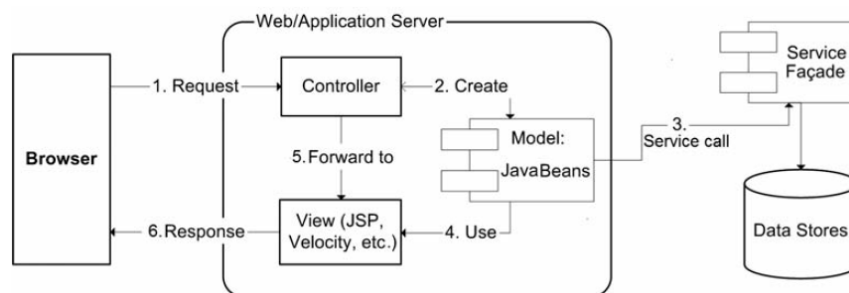
Spring MVC

- 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. Spring also provides mock classes to simulate Java HTTP objects (HttpServletRequest, for example) which makes unit testing of the web layer much simpler
- 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 need all the extra components required for form controllers.
- 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 (jakarta.apache.org/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.
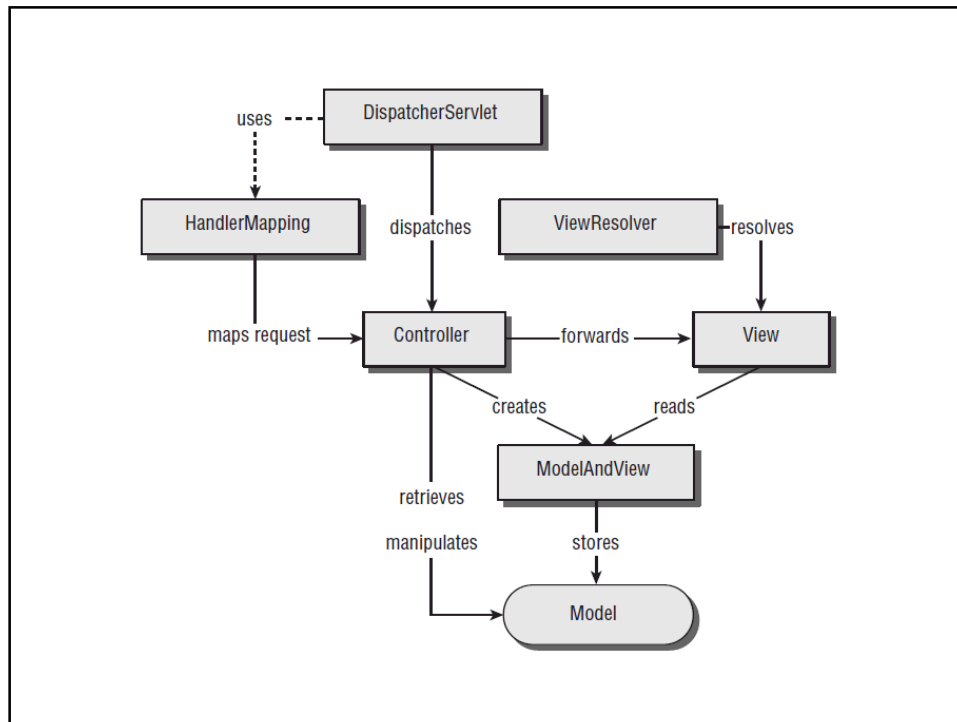
DispatcherServlet

- 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 application's web.xml file:

    ```
    <servlet>
            <servlet-name>roadrantz</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.
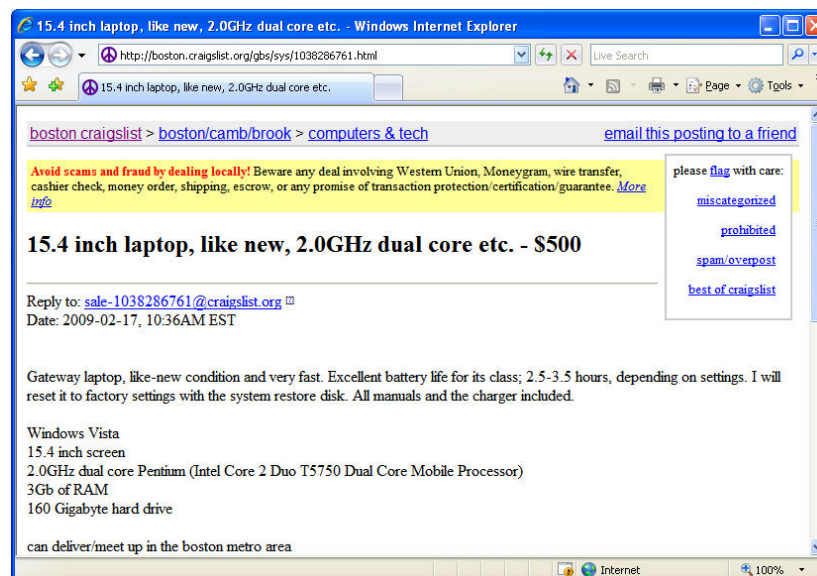
MVC Implementation

To create a web application with Spring, we need to start with the basic web.xml file, where we specify the DispatcherServlet and set the mapping for the specified url-pattern

```
Untitled - Notepad
File  Edit  Format  View  Help
    <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 we 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.
- Our 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.

Using Handler Mappings

- How does our web application know which servlet (controller implementation) 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

- We will start with the example of BeanNameUrlHandlerMapping. This is the simple HandlerMapping implementation that maps controller bean IDs to the servlet URLs.
- This HandlerMapping implementation is used by default if no HandlerMapping is defined in the Spring context files.
- Oftentimes you'll find yourself mapping your controllers to URL patterns that are quite similar to the class names of the controllers.
- 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.
- In fact, that's roughly what ControllerClassNameHandlerMapping does:
  `<bean id="urlMapping" 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.

BeanNameUrlHandlerMapping example

- Write the controller class that performs the logic behind the page.

- Configure the controller in the DispatcherServlet's context configuration file.

- Configure a view resolver to tie the controller to the JSP.
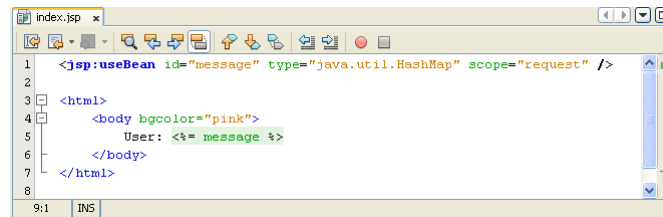
- Write the JSP that will render the page to the user.

YusufController.java

```java
package com.yusuf;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

import java.util.HashMap;
import java.util.Map;
import java.util.Date;

public class YusufController extends AbstractController {

    public YusufController() {
    }

    protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response) throws Exc
    {
        Map model = new HashMap();
        model.put("Greeting", "Hello World");
        model.put("Server time", new Date());
        return new ModelAndView("index", "message", model);
    }
}
```

dispatcher-servlet.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/sp
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <bean id="beanNameUrlHandlerMapping" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
    <bean name="/index.htm" class="com.yusuf.YusufController">
    </bean>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix">
            <value>/WEB-INF/jsp/</value>
        </property>
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
</beans>
```
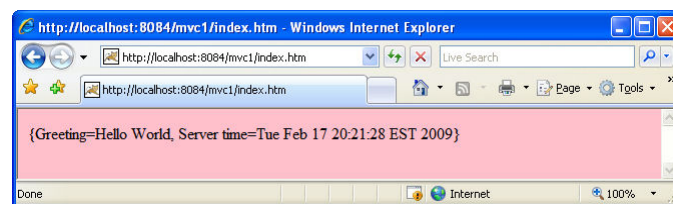
```
index.jsp  ×
1      <jsp:useBean id="message" type="java.util.HashMap" scope="request" />
2
3   <html>
4       <body bgcolor="pink">
5           User: <%= message %>
6       </body>
7   </html>
8
9:1    INS
```


http://localhost:8084/mvc1/index.htm - Windows Internet Explorer

http://localhost:8084/mvc1/index.htm

http://localhost:8084/mvc1/index.htm

{Greeting=Hello World, Server time=Tue Feb 17 20:21:28 EST 2009}

Done   Internet   100%

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. E 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.
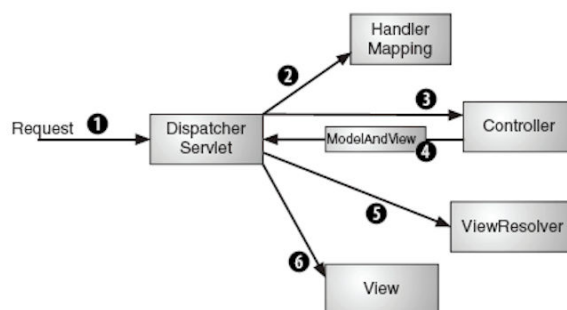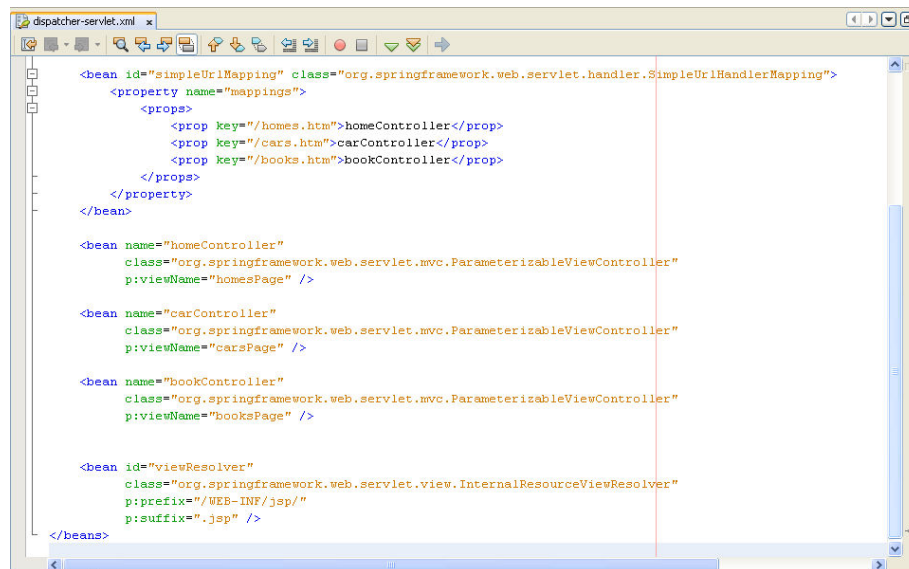

A request is dispatched by DispatcherServlet to a controller (which is chosen through a handler mapping). Once the controller is finished, the request is then sent to a view (which is chosen through a ViewResolver) to render output.

2. SimpleUrlHandlerMapping

- SimpleUrlHandlerMapping offers more flexibility in the request mappings.

- You can configure the mapping as key/value properties in the publicUrlMapping bean.

- A simple example of a Spring application context file containing the handler mapping configuration is on the next slide.

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

---

In fact, that's roughly what ControllerClassNameHandlerMapping does:

```
<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.roadrantz.mvc.RantsForVehicleController
                    |         |
                    |_____|
                         |
                         v
           /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 you will need to ask yourself to help you narrow down the choices.

Class diagram showing a partial list of Spring controllers
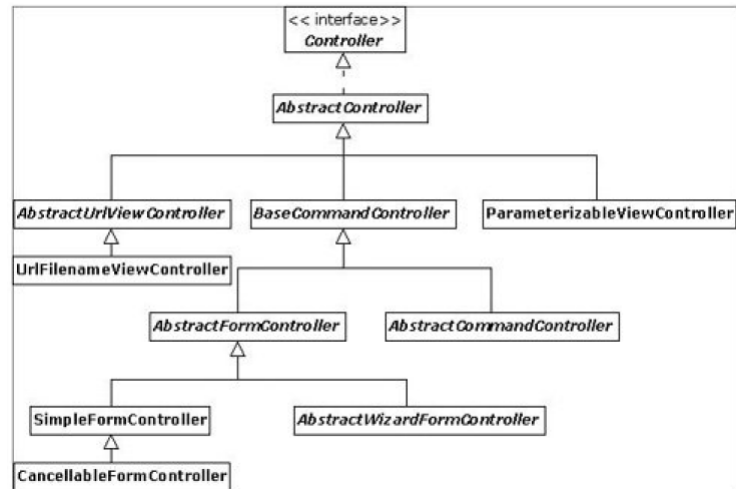
Spring MVC's selection of controller classes.

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 donothing implementations for all HandlerInterceptor methods.

As an example, we are going to implement a 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.

user — (search flights) → (select flights) → (confirm selection) → (enter personal details) → (enter payment details) → (confirm booking) → (BOOKED)

- 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