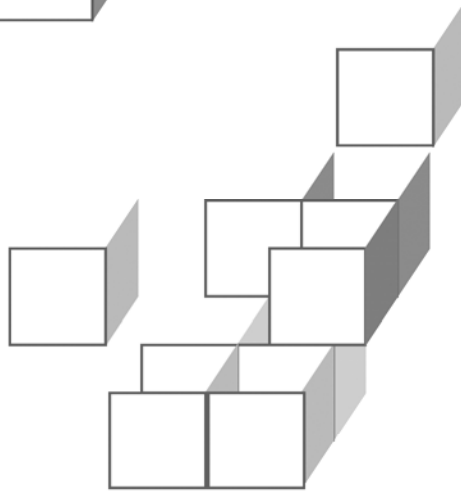
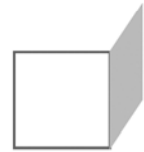
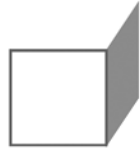


CHAPTER 4

THE WEB TIER



OBJECTIVES

After completing “The Web Tier,” you will be able to:

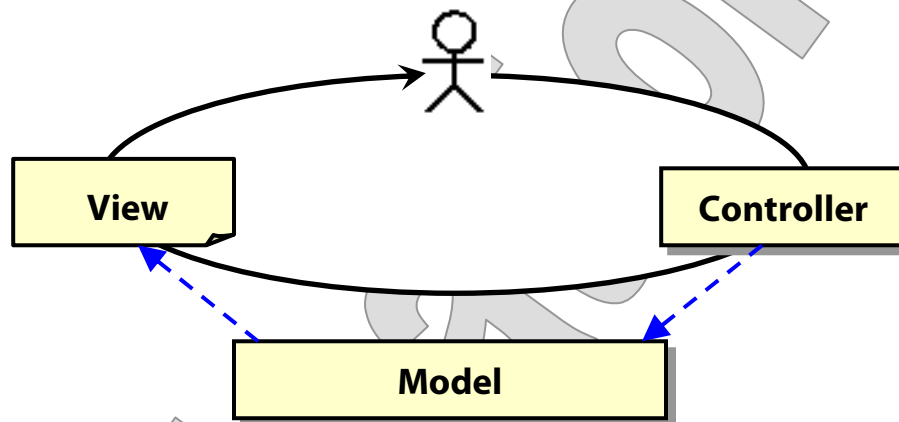
- Identify best practices and design patterns that have emerged in the years since Java EE was born, and explain how Spring facilitates these practices and patterns.
- Describe the lifecycle of an HTTP request/response roundtrip through a Spring web application: what components are involved in handling the request, carrying out work, presenting the next page, and handling errors.
- Refactor a traditional Java EE web application to use Spring.
- Make informed choices from among the many strategies Spring offers for implementing the major roles in request handling, including:
 - Handler mappings
 - Controllers
 - View resolvers

Servlets and JSPs: What's Missing?

- **Java servlets and JavaServer Pages (JSPs)** provide the basic means of responding to HTTP requests using Java code.
- There is a good deal of overlap in their capabilities, but each is best suited to a different sort of problem:
 - **Servlets** are Java classes and as such are strong on **processing**; producing HTML is possible but a bit awkward.
 - **JSPs** are more **presentation-oriented**, and best practice calls for all but true presentation logic to be deployed off-page and invoked using scriptlets, standard actions or custom tags.
- **Most Web applications are best developed to mix static HTML, JSPs, and servlets.**
 - The so-called **“Model 2”** architecture calls for servlets to implement business logic and then forward to JSPs to present the new information or system state as requested.
 - Thus servlets and JSPs each do what they're best at doing.
- **But the problem of how to coordinate these various components smoothly remains, and neither servlets nor JSP addresses this issue directly.**

The Model/View/Controller Pattern

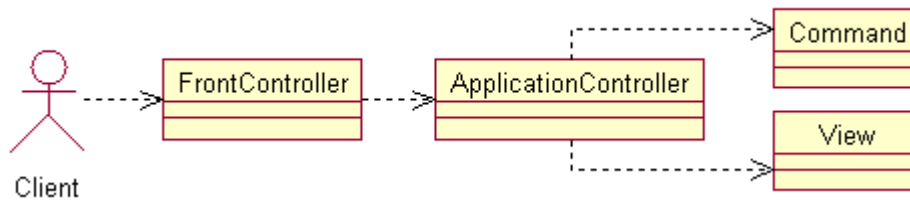
- As introduced in Chapter 1, MVC is a way of organizing any system – we’ll apply it to web applications specifically – into major roles **model**, **view**, and **controller**.



- As a prescription for decoupling a complex system, MVC succeeds based on a clear definition of dependencies:
 - Both the controller and view depend on the model’s semantics.
 - The model never depends on controllers and views. Think of this in terms of multi-tier architecture, too: the model may span the presentation and business tiers, or live entirely in the business tier, while the controller and view are purely presentation components.
 - Neither should there be interdependencies between controllers and views.
- Observing these rules keeps a system neatly organized, allows iterative development, and the best adaptability to change.
 - Especially, it facilitates **many-to-many relationships**: primarily from controller-to-model and view-to-model.

The Front Controller Pattern

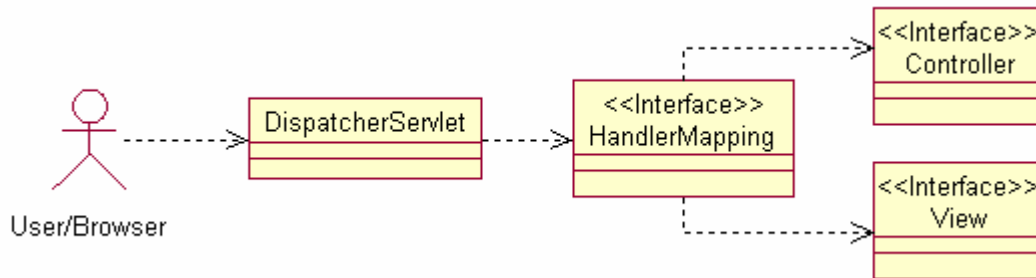
- A very popular Java EE design pattern is the **Front Controller**.
- This pattern recognizes the need for consistent pre-processing shared by many different request handlers – especially once they’ve been separated out according to MVC.
- This calls for a single controller at the front of the process – hence the pattern name – that can carry out the common pre-processing.



- This front controller is almost always linked to an **application controller**, which is responsible for dispatching to individual controllers, based on request URI or parameters, session attributes, or other variables.
 - Thus there is a **demultiplexing** of multiple request URIs to a front controller, and the application controller **re-multiplexes** to keep the control paths separate.

The DispatcherServlet Class

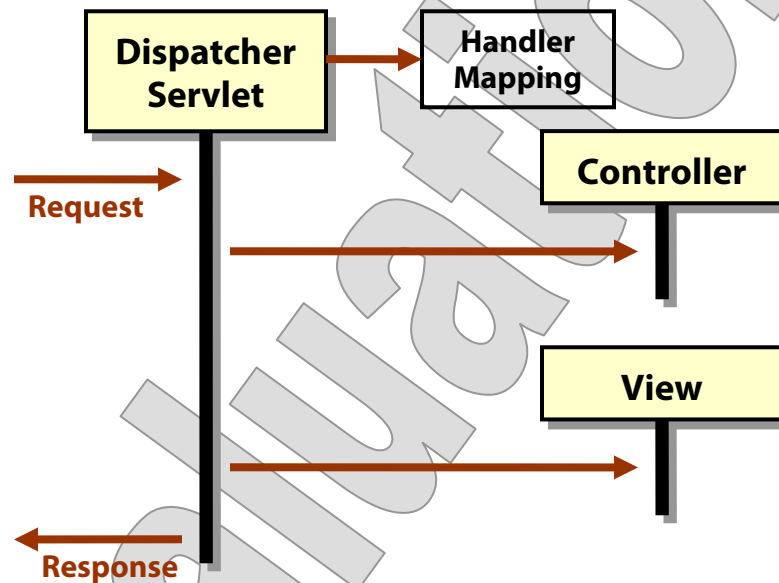
- The entry point to the entire Spring Web module is the **DispatcherServlet**, which is often configured as the one and only servlet in a Spring web application.
 - Find this and most of the key Spring Web types in **org.springframework.web.servlet**, or subpackages thereof.
 - This servlet handles all control requests to the application, and then relies on a **HandlerMapping** implementation to dispatch to individual controllers. Does this diagram look familiar?



- There is not much public interface to show for this class.
- It handles HTTP requests via template methods **doService** and **processRequest**, which are called from its base class' implementations of **doGet**, **doPost**, etc.
- What's most interesting about **DispatcherServlet** is all the dependencies that don't show up as public methods.
 - It uses Spring IoC **autowiring by type** to find most of its delegate objects; we'll see more of this throughout the chapter.
 - It is also configurable through a few servlet initialization parameters – that is, via **web.xml**.

A Spring Request/Response Cycle

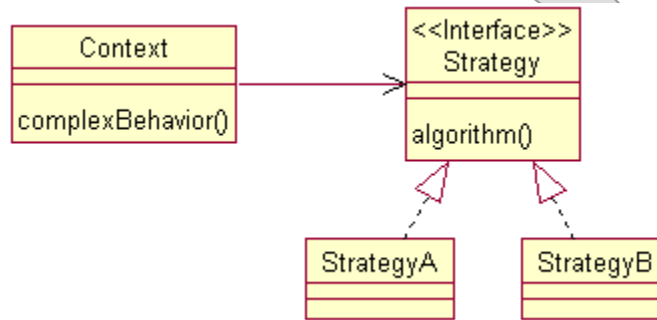
- So already we're getting an idea of the request/response cycle as implemented by the Spring Web module.
- We don't have the whole story yet, but we know this much:



- **DispatcherServlet** asks a **HandlerMapping** for a **Controller** and a **View**.
- It invokes the controller, and requested work is done there.
- It renders the view and hands it back to the user as the HTTP response.

The Strategy Pattern

- The **Strategy** design pattern is a basic but often overlooked technique for factoring out pieces of a complex algorithm.



- The **Context** object (with its remarkably apt name for what we're doing) has a complicated job to do.
 - It could implement it, whole, but that would make for terrible maintenance characteristics.
 - It could define a big pile of virtual methods – **onThis** and **onThat** – allowing subclasses to hook into its process and customize it.
 - This is in fact the **Template Method** pattern, and it's useful but it has its limits, especially since each unique set of customizations would require a fresh subclass.
- **Strategy calls for a separate interface for each piece of the larger process that can be made reasonably discrete.**
 - Then subtypes can implement the strategy and plug in to the main processor.
- **Does this sound familiar?**

JavaBeans as Web Components

- Spring's Web MVC is stuffed full of Strategies.
- We've just seen one: the **HandlerMapping**, which can be implemented several different ways without even building your own subclass.
- **Controller** and **View** are strategies in themselves, at least the way Spring encapsulates them.
 - Most MVC implementations take a similar approach, but it's probably not accurate to say that Strategy is baked into MVC by definition.
- Indeed, Spring gets tremendous mileage out of this one pattern, factoring nearly all of the job of HTTP request handling into a handful of key roles and then allowing each of them to be played by a different actor.
 - We'll soon see the **ViewResolver** as another top-level strategy.
 - There will be more to come ...

Configuring DispatcherServlet

- Install a Spring application by the simple act of declaring the **DispatcherServlet** in **web.xml** and mapping some or all of your request URLs to it.

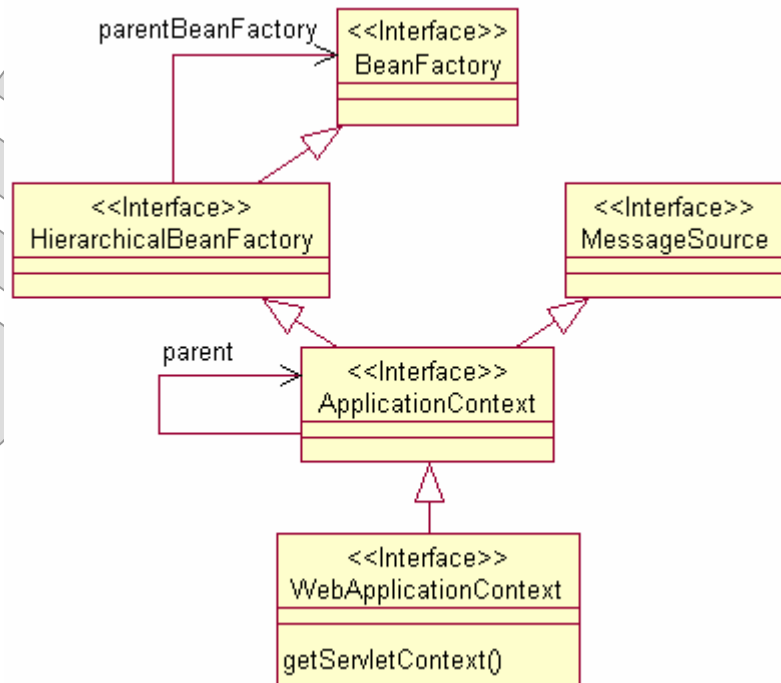
```
<servlet>
  <servlet-name>MyApplication</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>MyApplication</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

- Well, all right, there's a little more to do ...
 - Include at least **spring.jar** in your **WEB-INF/lib** directory.
 - If you want to use Spring's JSP custom tag library, include the **spring.tld** in **WEB-INF**.
- With this in place, everything else you do will be in the “Spring domain,” so to speak, and the starting point for all such tasks is the Spring IoC container, which is specialized for web applications into a **web application context**.

Web Application Contexts

- The Spring Web module relies heavily on the Core module, in particular on IoC containers.
- Every Spring web application has at least one **web application context**, which brings several of the behaviors we've already seen into a central position in the framework:
 - A web application context is a **bean factory** – so there's our primary IoC container capability.
 - It is also a **message source** – so we have internationalization
 - As an **application context**, it is **hierarchical**, meaning that a complex application can be organized into a tree or list of related modules.
 - By itself it adds the definition of a well-known name for a **root context** for the application, and our primary connection to the **servlet context**.
- Part of learning to develop in Spring is rethinking how you do familiar things – many of which you can do directly with Spring objects instead of requiring a path to a Servlets object.



Designing Web Applications

- With or without Spring, Web application development will benefit from a consistent design process.
- Java development in general uses **object-oriented analysis and design (OOAD)** techniques:
 - To understand and to decompose problems (analysis)
 - To express solutions in the abstract (design)
 - To reduce these designs to practice (implementation)
- OOAD excels where languages and component models are rigorously object-oriented themselves.
- The primary challenge for Web applications is to confront the presentation tier, which exists at the fringes of an otherwise object-oriented J2EE platform.
 - **HTTP** and **HTML** are far from object-oriented!
 - **JavaScript** is a pseudo-object-oriented language that does some halfhearted encapsulation but does not fully support the features necessary to iterative OO development.
 - Servlets are, fundamentally, an adaptive layer between these non-OO elements and pure, object-friendly Java.
- Also, where class and component design will often focus on **static analysis** - glibly, the **nouns** in the system - the presentation tier is much more about behavior - **dynamic analysis** of the **verbs**.

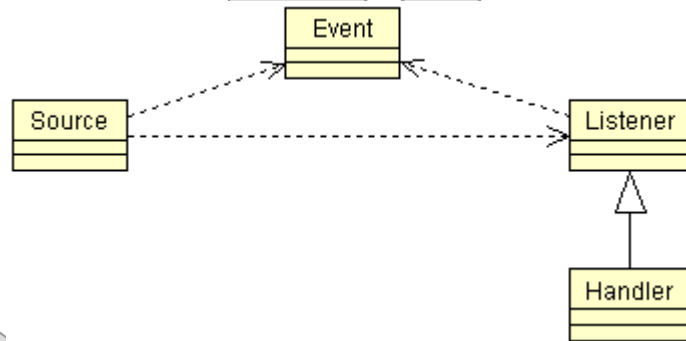
Unified Modeling Language

- In OOAD practice, consensus has evolved around the **Unified Modeling Language**, or **UML**, as the common notation for both static and dynamic analysis.

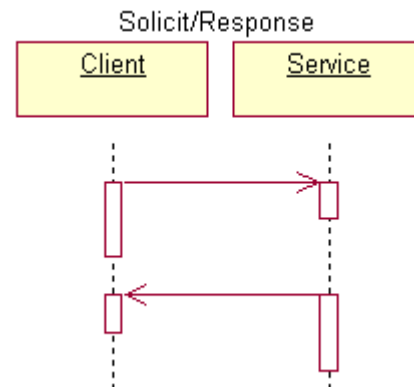
- UML **class diagrams**

clearly express encapsulations, semantics, and interdependencies.

This is most apt for a Spring model.



- **Interaction and sequence diagrams** illustrate behavior in specific scenarios, showing sequences of events and chains of triggered behavior.



- **None of this quite suits presentation-tier needs, however.**
 - We'd like to show things like **flows** between views and controllers, and especially for Spring it would be good to formally denote things like handler mappings and view resolution.
 - Traditional flowcharts might be a more apt solution than UML, but in fact these would be too detailed - too low-level.

A Web Control-Flow Notation

- For this course, we'll adopt a simple, non-standard notation that focuses on four elements:

Compute

- **Controllers** – the name is the bean name, or perhaps a method on the controller class

Results.jsp

- **Views** – the name is the page filename

Submit →

- **Handler mappings** – request URI to Controller; the label suggests a button caption or link text

failure →

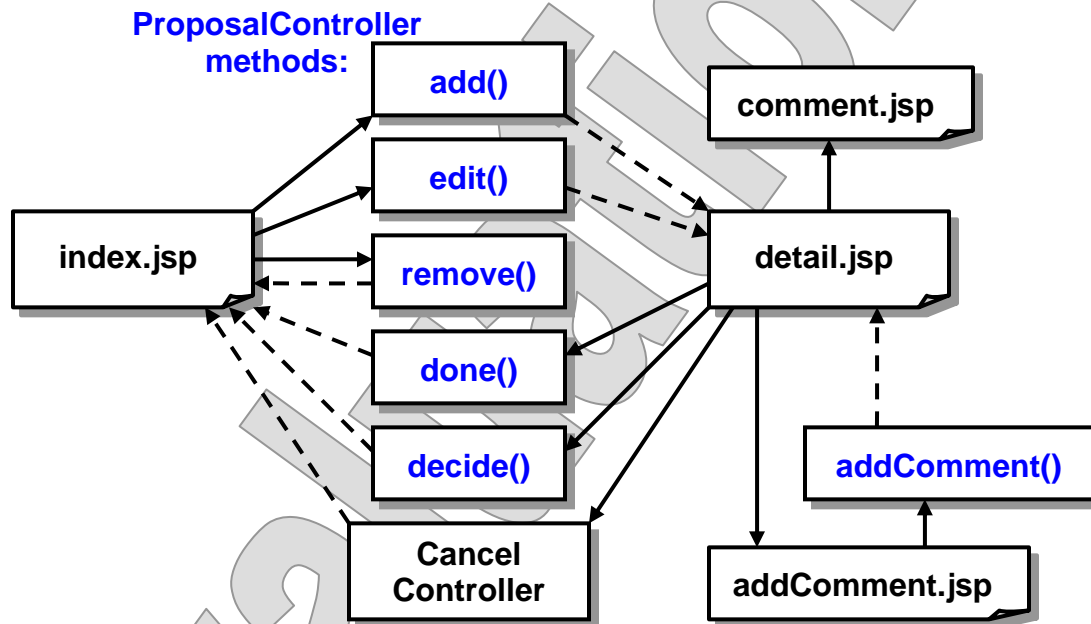
- **View results** – Controller to View; the label is the name of the page or the view itself

- With these four elements (and a good deal of improvisation), we can illustrate moderately complex website designs and individual request/response flows.
- As needed, controller and view components may be integrated with UML class diagrams, to show dependencies on model components.

The LandUse Application

EXAMPLE

- Here is the LandUse application, last seen in Chapter 1 but diagrammed here as it will be working after some later exercises:

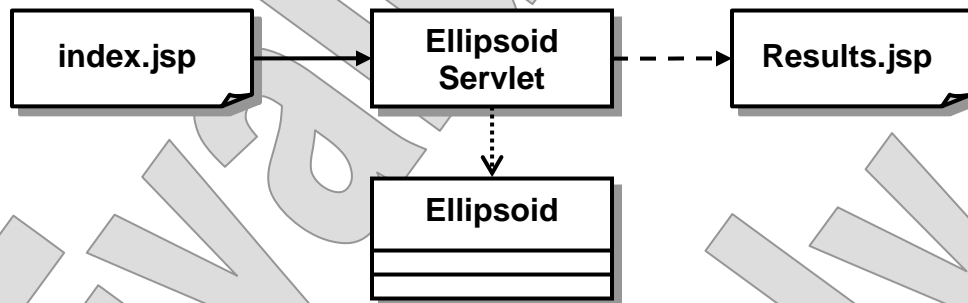


- LandUse relies heavily on a specific Spring controller, the **MultiActionController**, which dispatches requests to individual methods on a delegate class – in this case **ProposalController**.
- So we bend the notation a bit to support that – and more often than not there will be some such accommodation, but the basic job of communicating the layout and flow of the application is done.

A Minimal Spring Web Application

DEMO

- As a next step in getting familiar with Spring, we'll carry out the process of refactoring an existing, simple web application.
 - The Ellipsoid application begins its life as a traditional servlet-and-JSP web application, with a JavaBean to capture useful state information and share it between components.
 - We'll gradually replace the standard Java EE workings with Spring components, and learn some new concepts along the way.
- We'll work in **Demos\SpringApp**.
 - The completed demo is in **Examples\Ellipsoid\Step4**.
- Review the layout and code for the starter application.



- **index.jsp** presents an HTML form that gathers three dimensions of a three-dimensional ellipsoid and places a request.
- **EllipsoidServlet** handles the request by creating and populating a JavaBean, **Ellipsoid**, with request parameters. It publishes the bean at request scope and forwards to **Results.jsp**.
- **Results.jsp** reads out the information in the JavaBean, including the request parameters and additional calculated properties: volume, classification, and description.

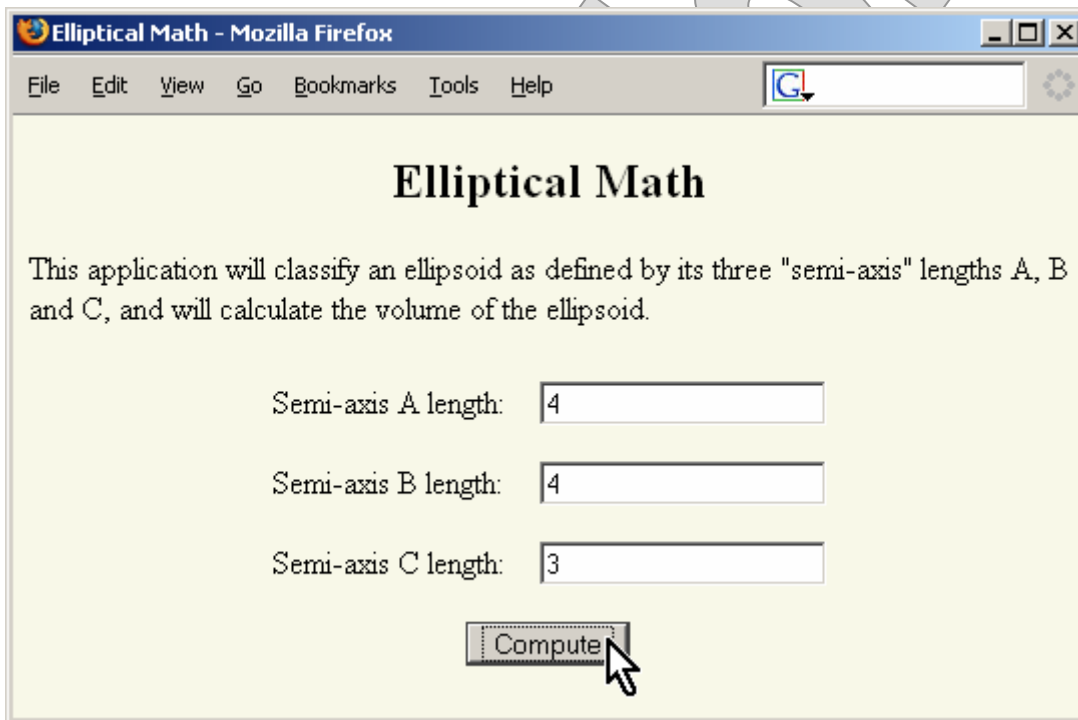
A Minimal Spring Web Application

DEMO

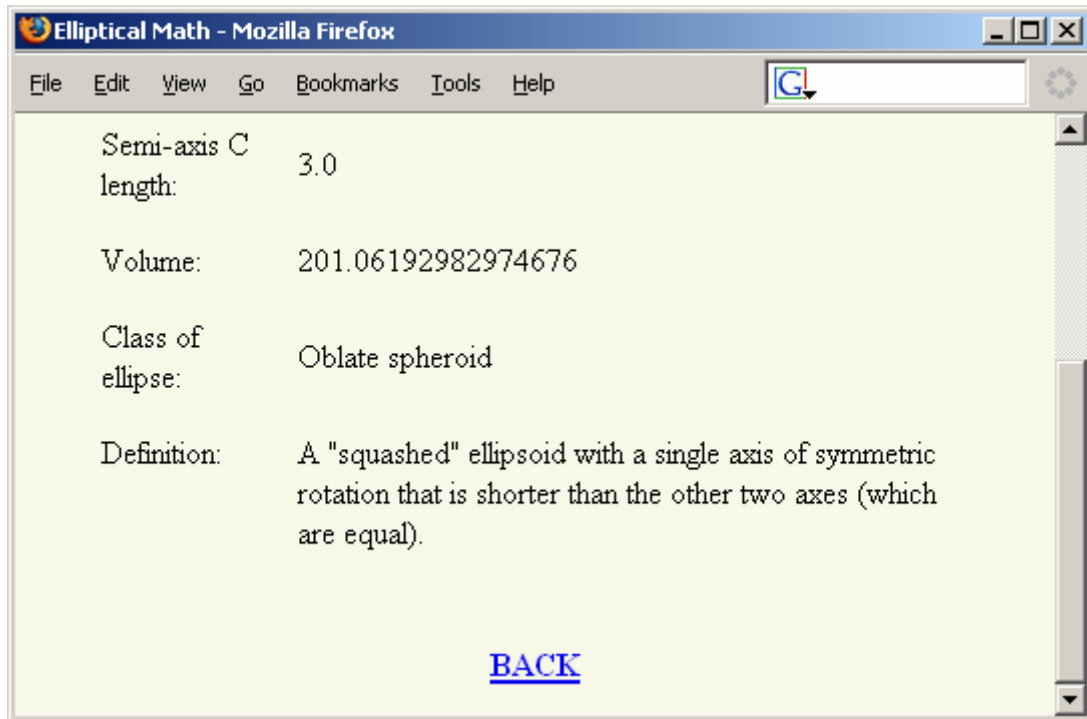
1. Build the starter application and test it out:

ant

`http://localhost:8080/Ellipsoid`



A Minimal Spring Web Application

DEMO

2. Job one is to put Spring in place, so let's start by opening **docroot/WEB-INF/web.xml**.
3. Replace the mapping to **EllipsoidServlet** with a mapping to the **DispatcherServlet**. (The URL pattern can stay, since it's already set to ***.do**, and we really only have one request path anyway.)

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

A Minimal Spring Web Application

DEMO

4. This one change puts the incoming request on a completely different track, and brings new files into play that are already completed. Look at **docroot/WEB-INF/Ellipsoid-servlet.xml** for starters.
 - This is the primary context declaration for the web application – it plays the role of **web.xml** for a Spring application, using the vocabulary of the Spring beans configurations we've been working with so far.
 - Note that the name of the file is based on the declared name of the servlet in **web.xml**.

```
<bean
  class="org.springframework.web.servlet.handler
    .SimpleUrlHandlerMapping"
>
  <property name="mappings">
    <props>
      <prop key="/Compute.do" >
        EllipsoidController</prop>
      </props>
    </property>
  </bean>

<bean id="EllipsoidController"
  class="cc.math.EllipsoidController" />
```

- This application's handler mapping uses an explicit map of URL keys and values that are interpreted as bean names.
- The only controller is **EllipsoidController** and this is mapped to the incoming **/Compute.do**.

A Minimal Spring Web Application

DEMO

5. Open `src/cc/math/EllipsoidController.java` and see the controller code. So far, this is just barely changed from the code that's in the servlet. The only difference is in how the controller moves the request along to a view, creating a **ModelAndView** object and handing that back to the dispatcher servlet.

```
public class EllipsoidController
    implements Controller
{
    public ModelAndView handleRequest
        (HttpServletRequest request,
         HttpServletResponse response)
        throws Exception
    {
        Ellipsoid delegate = new Ellipsoid ();
        delegate.setA (Double.parseDouble
            (request.getParameter ("a")));
        delegate.setB (Double.parseDouble
            (request.getParameter ("b")));
        delegate.setC (Double.parseDouble
            (request.getParameter ("c")));
        request.setAttribute ("ellipsoid", delegate);
        return new ModelAndView
            (new InternalResourceView ("Results.jsp"));
    }
}
```

- This **ModelAndView** is constructed to aggregate a prepared **InternalResourceView**, which is view implementation that simply wraps a browser-addressable resource within the application.
6. Build, deploy, and test again to see that the application's behavior is unchanged. (This is the version in the example **Step2**.)

A Minimal Spring Web Application

DEMO

- Spring's authors have an unusual take on MVC, when it comes time to serve up the view: they suggest that the controller should decide on a view, and populate that view with a model.
 - That is, the controller “creates a model” for the view.
 - Traditionally, web MVC applications have treated the controller and view more as (perhaps unequal) partners, letting the view find the model information it would need, just as the controller would go to find the model for itself.
 - The approaches are not so different in this case: the servlet was posting a bean at request scope, and this is just what Spring will do – when you pass the bean and name to a **ModelAndView** constructor.
 - It will take more than one key/value pair, too.
7. To get with the Spring program a bit more fully, remove the call to **request.setAttribute**, and use the **ModelAndView** constructor to pass the bean along:
- ```
return new ModelAndView
 (new InternalResourceView ("Results.jsp"),
 "ellipsoid", delegate);
```
8. Build and retest to see that this also works.

## A Minimal Spring Web Application

DEMO

9. Now let's start taking advantage of a **view resolver**. See the rest of the context configuration file, which declares a **BeanNameViewResolver**. This class converts a requested view name to a **View**-implementing bean of a name that matches that view name.
10. See also the three beans defined to wrap three JSPs in the application, giving them simple names "Form", "NormalResult" and "SphereResult".
11. Modify the controller to choose between normal and sphere results – represented now as strings, not **View** objects – based on the results of **delegate.getType**:

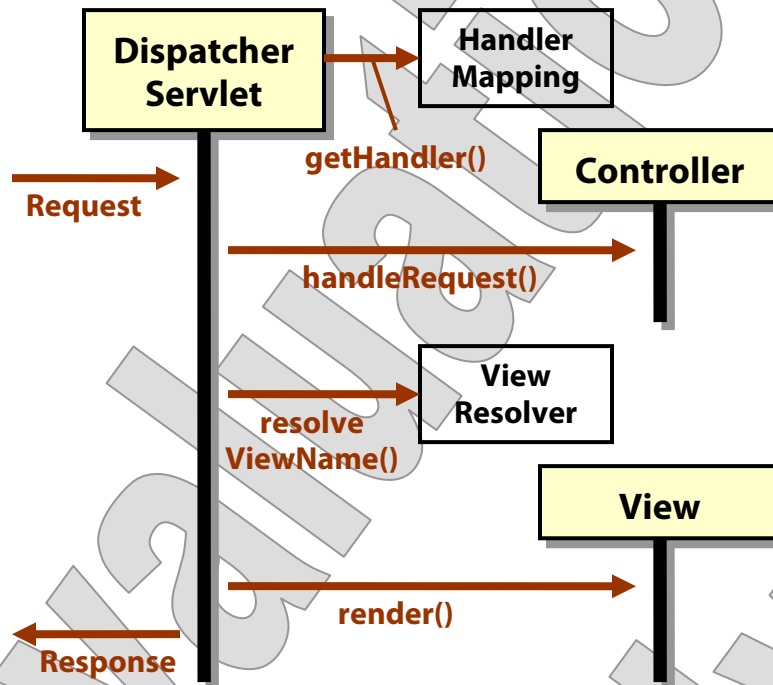
```
return new ModelAndView
 ((delegate.getType ().equals ("Sphere")
 ? "SphereResult"
 : "NormalResult"), "ellipsoid", delegate);
```

12. Build and test one last time. Now the view resolver is consulted with the string returned by the controller as part of the **ModelAndView** object.

## A Minimal Spring Web Application

**DEMO**

- With a few code changes, we've put the main body of Spring MVC to work for the application – here's our request-handling diagram again, with some new details gleaned from this demonstration:



- So **HandlerMapping** and **ViewResolver** are the next-level decision points after the servlet: one finds controllers and one finds views.
- Have you started wondering: how did the servlet find these two key objects?

## Autowiring in the DispatcherServlet

---

- The answer harks back to our study of Spring IoC in the previous chapter: **DispatcherServlet** finds these two delegates through beans autowiring by type.
- This isn't obvious, for the simple reason that the servlet isn't declared as a bean in the configuration itself.
- Also, it does what a declared bean could not do, which is choose to autowire by type, by name, or not at all, at a property level, rather than for the object as a whole.
- See the javadoc for this class for more on which delegates are found by what means – but an incomplete list, including several concepts we've yet to study, is here:
  - A **HandlerMapping** is wired by type
  - A **ViewResolver** is wired by type
  - A **MessageSource** is wired by the name “messageSource” – actually this is the web application context, not the servlet itself, doing the matching
  - A **HandlerExceptionResolver** is wired by type
  - A **MultipartResolver** is wired by the name “multipartResolver”
  - A **LocaleResolver** is wired by the name “localeResolver”
  - A **ThemeResolver** is wired by the name “themeResolver”

## Many Frameworks in One

---

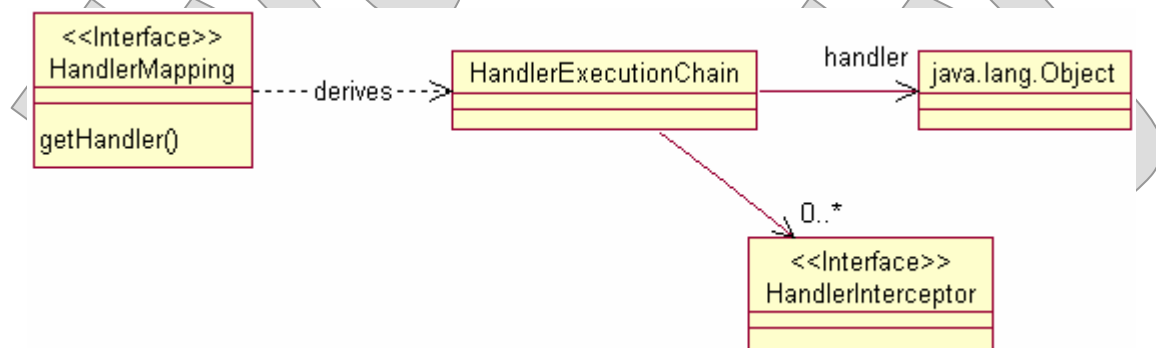
- **Spring is so sophisticated and finely decomposed that it is almost a meta-framework: that is, a means of developing your own favorite way of building web applications.**
  - **C++** was sometimes called a **meta-language**, for similar reasons.
  - **XML** is known as a meta-language for data, because within its well-considered grammar there is room to define virtually any useful data vocabulary.
  - **Spring** has that same feel: rather than worrying about having a way to solve a particular problem, the Spring developer is more likely to struggle with choosing between the many solutions available.
- **Over the rest of this chapter we'll tour a few of the major strategy delegates of the dispatcher servlet, and see a few common options under each one:**
  - Handler mappings
  - Controllers
  - View resolvers
- **Then we'll wrap up with a discussion of best practice and move on to this chapter's lab exercise.**

## The HandlerMapping Interface

- We've already seen the most important job that **HandlerMapping** does, which is help the dispatcher servlet decide on a controller for a given request.
- There is (how many times will we say this?) more to the story.
- The full responsibility of a **HandlerMapping** is to derive a **HandlerExecutionChain**.

```
public interface HandlerMapping
{
 public HandlerExecutionChain
 getHandler (HttpServletRequest request);
}
```

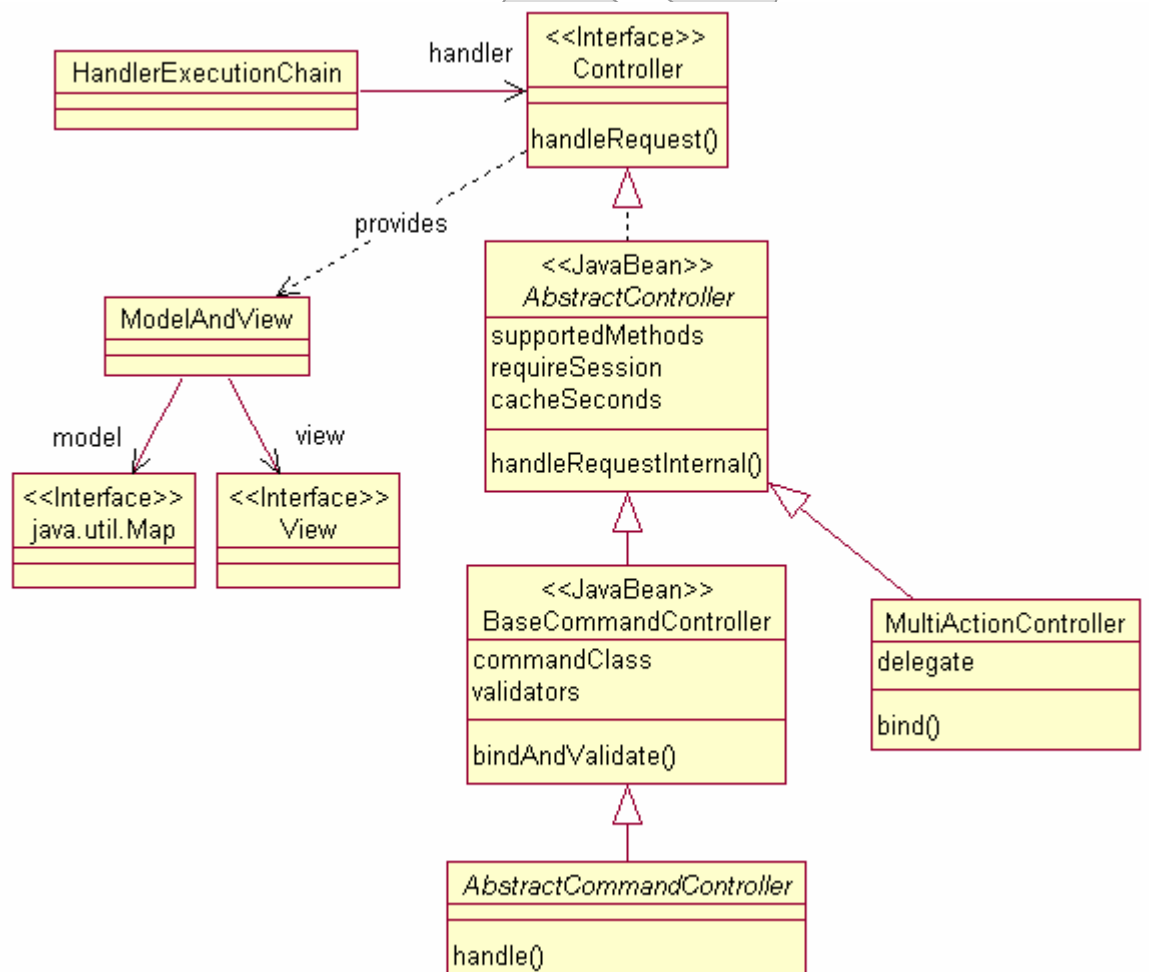
- This, in turn, navigates to one handler and any number of **HandlerInterceptors**.
- (We're still in `org.springframework.web.servlet` for all three of these types.)



- Interceptors implement the **Intercepting Filter** pattern for Spring; they are loosely analogous to servlet **Filters**.
  - We'll consider interceptors in more depth later in the course.

## The Controller Interface

- It all starts with the **HandlerMapping ...** but most of the real action is in the **Controller**.
- Though any object can technically be a Spring request handler, for HTTP requests all controllers will be implementations of the **Controller** interface.
  - All the controller types below are from the package **org.springframework.web.servlet.mvc**:



## Controller Responsibilities

---

- The Spring controller has an outsized role compared to the model and view: it really manages the remainder of the request-handling process.
- The basic job of a controller is to carry out the requested work and to serve up a view and a map of objects which the dispatcher servlet should make available to that view during its rendering.
- **AbstractController** is a convenient base type for controllers playing just this simple role.
- Other subtypes define – and then meet – additional responsibilities:
  - **MultiActionController** does additional dispatching to a delegate object, with additional strategy choices for deciding what methods to call for what request URIs and query strings.
  - **AbstractCommandController** formalizes the use of a **command object**: creating this JavaBean during request handling, binding request parameters to it, calling configured validators, and then making it available to controller and view components.
  - **AbstractFormController** goes further and encapsulates some of the concepts of the HTML form itself, managing form input, processing, and even redirecting flow back to the form when errors occur.
- While we survey **HandlerMapping** and **ViewResolver** choices in this chapter, we defer considering these different controller types until we can spend more time with them in the next two chapters.

## The ModelAndView Class

---

- **ModelAndView** is a simple aggregation of a **View** object and a “model” – which in this context means a map of keys and objects that might be useful to view rendering, and not the overall state model of the MVC application.

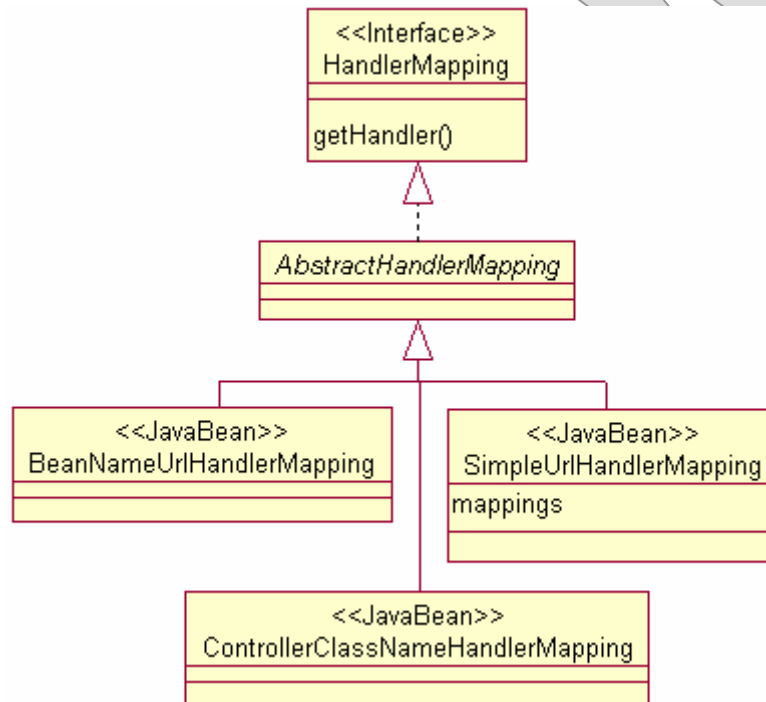
```
public class ModelAndView extends Object
{
 public ModelAndView (View view);
 public ModelAndView (View view, Map model);
 public ModelAndView (View view,
 String oneKey, Object oneValue);
 public ModelAndView (String viewName);
 public ModelAndView (String viewName, Map model);
 public ModelAndView (String viewName,
 String oneKey, Object oneValue);
 ...
}
```

- Overloads of its constructor allow for various usages:
  - Provide a **View** instance or a name to be passed to a **ViewResolver**.
  - Provide no model, a single key/value pair (surprisingly useful and common), or a full-fledged map.
- Its public methods are mostly called by framework code, so we won't delve into those too deeply.

## HandlerMapping Options

**EXAMPLE**

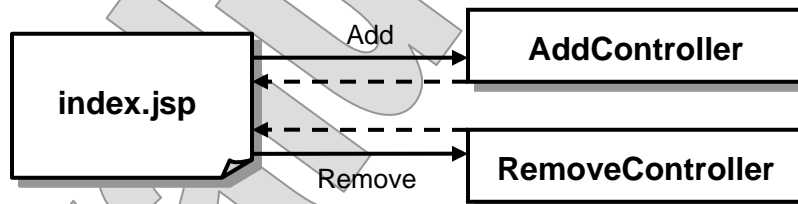
- There are many implementations of most strategies defined for Spring, and **HandlerMapping** is no exception:



## HandlerMapping Options

**EXAMPLE**

- In **Examples/Cookies**, we'll consider three variants of the same web application, each of which uses a different handler mapping.
- Subdirectories are
  - `BeanNameUrl`
  - `ControllerClassName`
  - `SimpleUrl/Step1`
- All three follow a layout and flow like this, with slight variations in tactics:



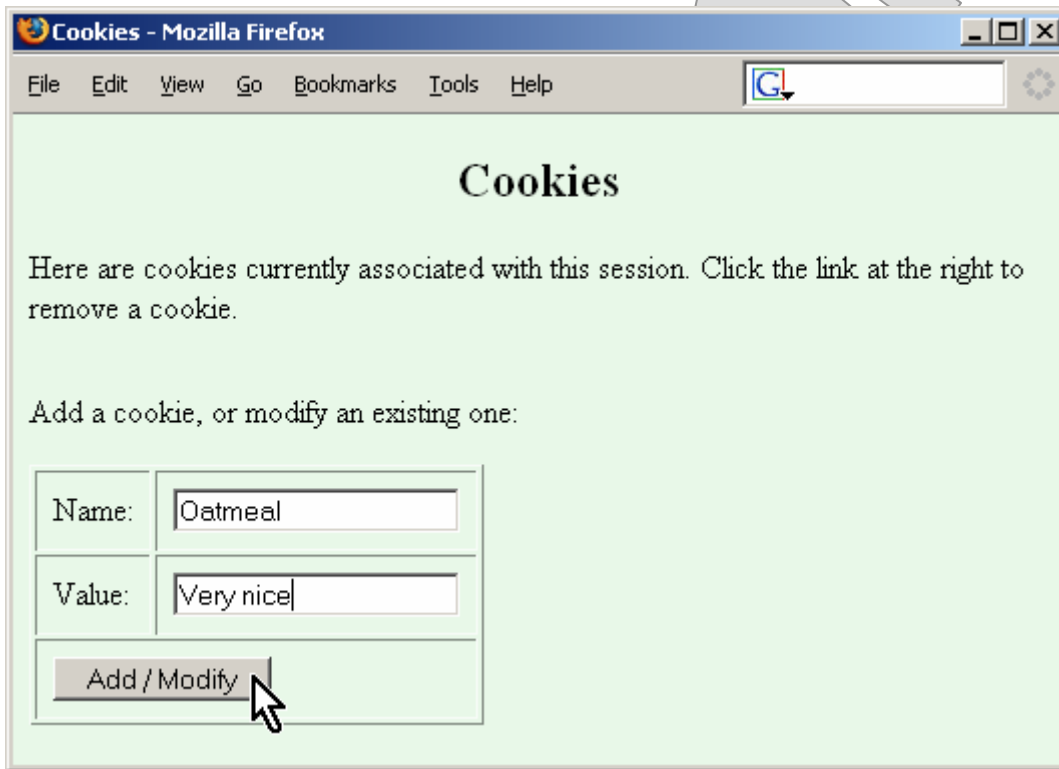
- We'll run just one of them – trust us, they all work! – but review relevant code and differences between all three.

## HandlerMapping Options

**EXAMPLE**

- Build and test the application from the **BeanNameUrl** directory.

`http://localhost:8080/Cookies`



## HandlerMapping Options

**EXAMPLE**

**Cookies**

Here are cookies currently associated with this session. Click the link at the right to remove a cookie.

|            |                                  |                        |
|------------|----------------------------------|------------------------|
| JSESSIONID | E7DCE04FDFCC96EA7B75F3CB1832CF0E | <a href="#">Remove</a> |
|------------|----------------------------------|------------------------|

- Hmm. The session-ID cookie appears automatically, so that’s right
  - but where’s my Oatmeal cookie?
- **Try reloading – ah, that works:**

**Cookies**

Here are cookies currently associated with this session. Click the link at the right to remove a cookie.

|            |                                  |                        |
|------------|----------------------------------|------------------------|
| JSESSIONID | E7DCE04FDFCC96EA7B75F3CB1832CF0E | <a href="#">Remove</a> |
| Oatmeal    | Very nice                        | <a href="#">Remove</a> |

- We’ll come back later to the question of why we need to do this reload – and then we’ll fix it!
- **Meanwhile, you can try out removing and editing cookies as well.**

## HandlerMapping Options

**EXAMPLE**

- Now let's look over the application code.
- You'll see the usual declarations in **web.xml**, now with the servlet name "Cookies".
- Therefore there is a **Cookies-servlet.xml** with the Spring Web beans declared in it.
  - Because the controllers in this example build their own **View** objects, there is no need for a view resolver, and so the context configuration is very simple:

```
<beans>
 <bean
 class="org.springframework.web.servlet.handler
 .BeanNameUrlHandlerMapping"
 />
 <bean name="/Add.do"
 class="cc.web.AddController" />
 <bean name="/Remove.do"
 class="cc.web.RemoveController" />
</beans>
```

- The handler mapping declared here is **BeanNameUrlHandlerMapping**, which maps the request URL (minus the static stuff of host, port, and application context) to the name of a bean that implements **Controller**.
  - Notice the need to shift from **id** to **name** for controller beans in this system: **id** values have to be legal XML names, which rules out the essential slash character

## HandlerMapping Options

**EXAMPLE**

- In the **ControllerClassName** version of the application, the configuration file is even simpler – even if the mapping class name keeps getting longer!

```
<beans>
 <bean class=
 "org.springframework.web.servlet.mvc.support
 .ControllerClassNameHandlerMapping"
 />
 <bean class="cc.web.AddController" />
 <bean class="cc.web.RemoveController" />
</beans>
```

- Using **ControllerClassNameHandlerMapping**, the controller beans are distinguished entirely by their types.
  - So neither **id** nor **name** is necessary.
  - This is not the same thing as autowiring, by the way – though the effect is pretty much the same.
- Notice that the request URLs for this version will not be the same as in the last one: **/add.do** and **/remove.do** become simply **add** and **remove**.
  - This mapping strips away package name and a standard (yet optional) “Controller” suffix from the full class name, and also demotes the first letter to make it fit with the camelCase URLs that are expected.
  - Thus does **add** match **cc.web.AddController**.

## HandlerMapping Options

**EXAMPLE**

- Finally, consider the **SimpleUrl/Step1** version of the application:
  - This uses one of the most common and powerful mapping strategies: **SimpleUrlHandlerMapping**, which, rather than trying to find automatic matches by naming convention, defines all mappings explicitly in a map:

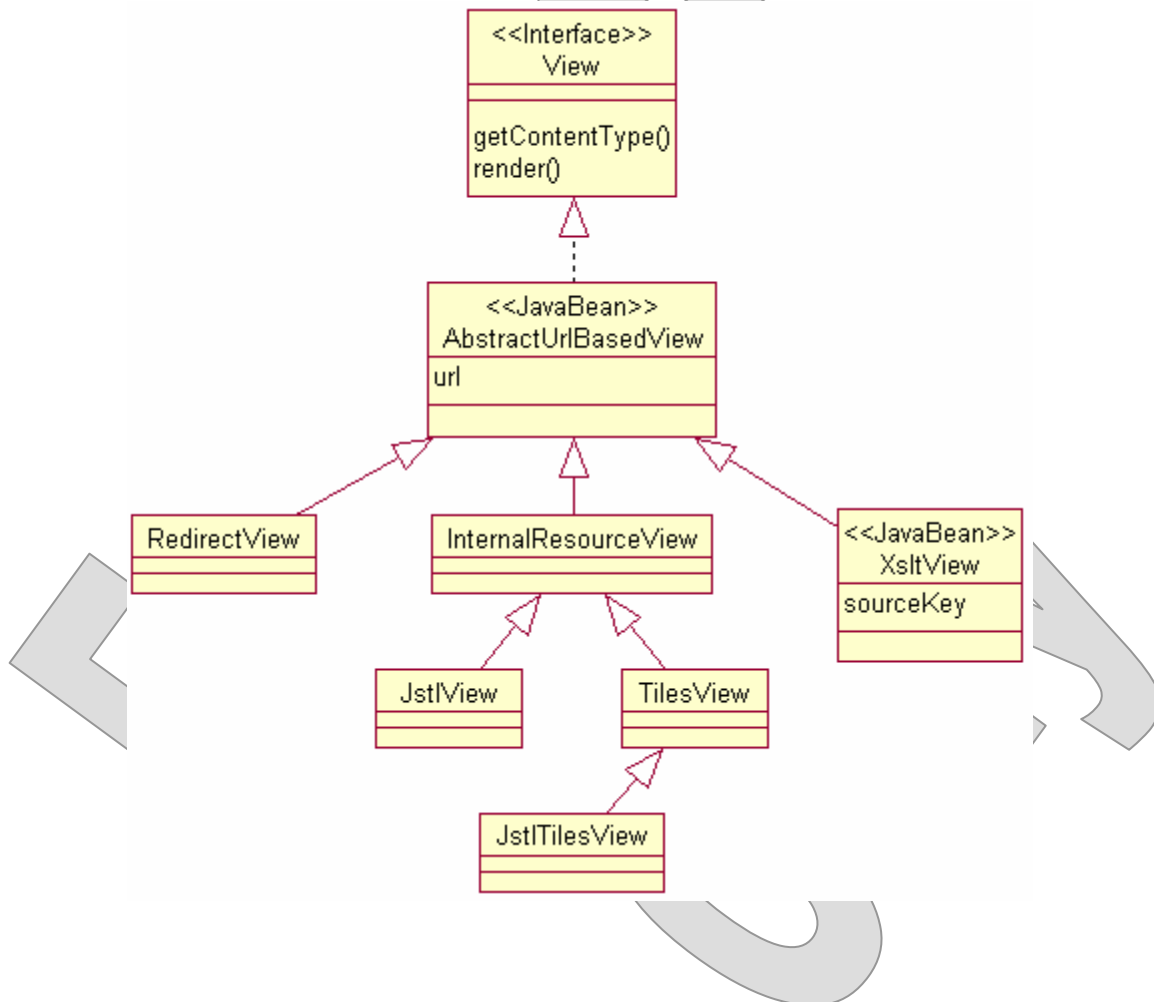
```
<beans>
 <bean class=
 "org.springframework.web.servlet.handler
 .SimpleUrlHandlerMapping"
 >
 <property name="mappings" >
 <props>
 <prop key="/Add.do" >add</prop>
 <prop key="/Remove.do" >remove</prop>
 </props>
 </property>
 </bean>
 <bean id="add" class="cc.web.AddController" />
 <bean id="remove"
 class="cc.web.RemoveController" />
</beans>
```

- This mapping demands the most effort, but is almost infinitely flexible since any number of URLs can be mapped to controllers – which may or may not have any naming characteristics in common with those URLs.
- And we're back to **/Add.do** and **/Remove.do** for request URLs – though now these are completely unconstrained, and don't have to match any Java types or bean names to make things work.

## The View Interface

---

- A **View** is simply a component that can render a response in the appropriate content type.
- There are over two dozen view types implemented in Spring.
- Just a handful support most Spring development – here are some of the most common types:



## The ViewResolver Interface

---

- Once a controller has done its work, it will report back to the dispatcher servlet with a **ModelAndView** object.
- For complex applications, this will usually carry the name of a desired view for the servlet to render to the HTTP response.
- Translating, or **resolving**, this name to an actual **View** object is the responsibility of a configured **ViewResolver**:

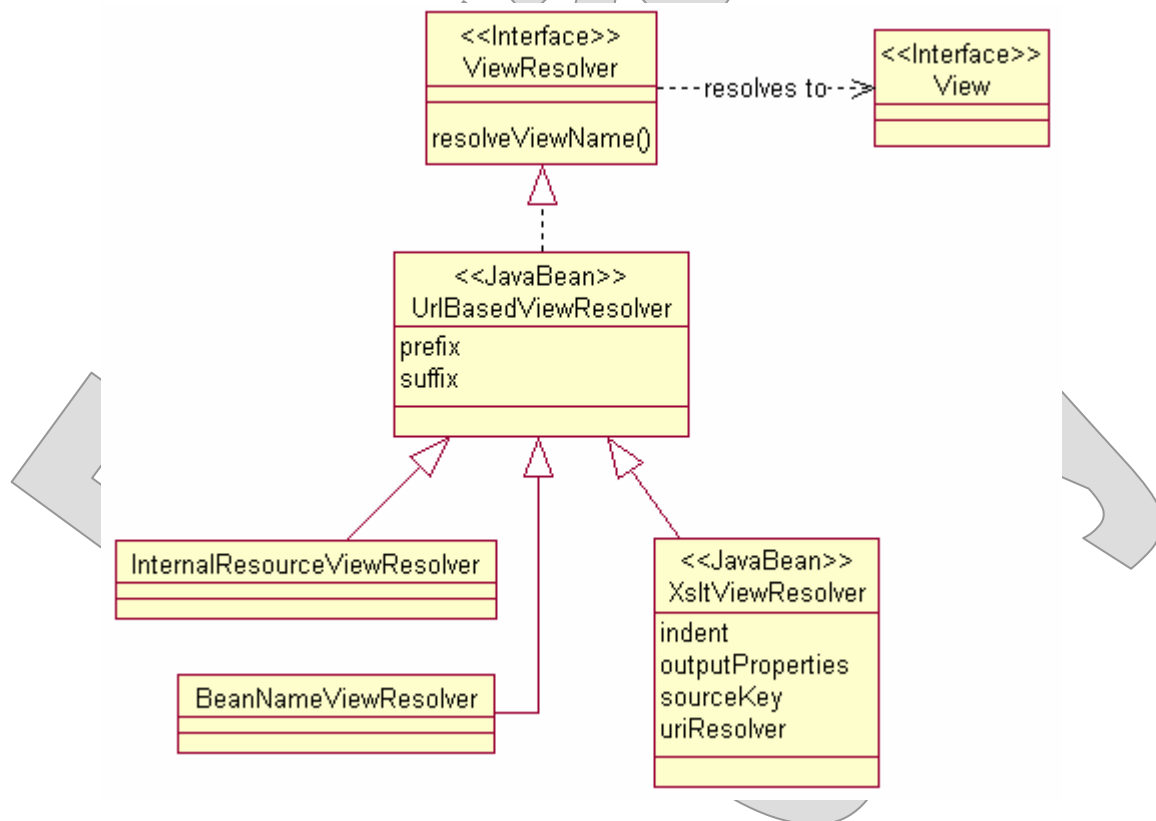
```
public interface ViewResolver
{
 public View resolveViewName
 (String name, Locale locale);
}
```

- The one method on this interface, **resolveViewName**, illustrates the simplicity of the job description.
- It also points up one of the more compelling reasons to use view resolvers, which is the built-in internationalization support.
- If a controller builds or finds its own view, any i18n support will have to come from the controller's own logic – and that gets old pretty quick when you're writing tens or scores or hundreds of controllers.

## ViewResolver Options

**EXAMPLE**

- We'll revisit three variants on the LandUse application to see more of the options for choosing and managing views.
- Under the base directory **Examples/LandUse/ViewResolvers**, see web application projects in the following subdirectories:
  - **InternalResource**
  - **None**
  - **Xml**



## ViewResolver Options

**EXAMPLE**

- First, look at the **Xml** version.
  - If you want a refresher on how this application works, build and test it now, and/or refer back to Chapter 1 for screenshots, etc.
- See **docroot/WEB-INF/LandUse-Servlet.xml** – there is just one bean declaration that interests us here:

```
<bean class="org.springframework.web.servlet.view
 .XmlViewResolver" >
 <property name="location"
 value="/WEB-INF/LandUse-views.xml" />
</bean>
```

- A nice feature of the **XmlViewResolver** is that it breaks its mappings from name to view instance out to a separate XML file.
  - See **docroot/WEB-INF/LandUse-views.xml**:

```
<bean
 id="summary"
 class="org.springframework.web.servlet.view
 .InternalResourceView"
 >
 <constructor-arg value="index.jsp" />
</bean>
<alias name="summary" alias="doneEditing" />
<alias name="summary" alias="confirmDecision" />
```

- Notice, too, a good application for Spring bean **aliases**: Spring Web derives a useful many-to-one relationship by affording an alias to any distinct path that resolves to the same view.
- This in turn makes for flexibility in future implementation.

## ViewResolver Options

**EXAMPLE**

- Another option you've already seen, which is no view resolver at all, just controllers conjuring their own view objects and handing them back to the servlet.
- Then, probably the most common view resolution strategy is still one of the truly simple ones: **InternalResourceViewResolver** implements a one-to-one mapping from view name to internal resource.
- That's not to say that the view name equals the resource address.
  - The resolver defines static **prefix** and **suffix** to translate, for instance, "detail" into "/MyPages/detail.jsp".

## Chaining ViewResolvers

---

- For more complex applications, often there is no one strategy that covers all bases.
  - Well, **XmlViewResolver** can handle pretty much anything, but it's code-intensive and prone to its own maintenance issues.
  - The more automatic policies generally reduce flexibility and load more decision-making onto individual controllers than is desirable. Consider that even **InternalResourceViewResolver** can only support one page type – that is, one standard suffix.
- Spring allows that multiple view resolvers can be organized into a chain, such that each gets a crack at the URL.
  - Since resolvers are autowired by type, they must declare their ordering, via an **order** attribute whose value is an ordinal integer.
  - The first non-null value returned by a resolver is used as the view.
  - Not all resolvers can play at any position, though. Especially, **InternalResourceViewResolver** will never return null; it will just return an **InternalResourceView** object that will later fail to load its resource.
- Generally, this system is about halfway broken.
- Where multiple, overlapping strategies are required, it's usually best to take the bull by the horns and implement a master **ViewResolver** of your own – which is simple enough to do.

## Redirecting to a View

DEMO

- Another useful tactic in resolving views for a Spring application is choosing to **redirect** to a view, instead of forwarding to it.
- Forwards are generally preferred to redirects, for simple performance reasons: they don't incur the cost of an additional network round-trip.
- But redirecting may be useful in certain cases:
  - When one controller delegates to another, and POSTed form data is involved: this can result in unintentional absorption of request parameters that were not intended for the delegate controller.
  - When the user may be prone to submit form data more than once: a redirect will be resolved by the browser using HTTP GET.
- We'll do it for a different reason, which is specific to the Cookies application.
- We've seen that actions to add, edit, and remove cookies succeed, but are not reflected in the immediately returned view.
  - This is a quirk of HTTP requests and responses where we're taking data that actually goes into the response header and showing it on the page – that is, in the response body.
  - An add-cookie request, for example, will cause the cookie header to be added to the response; then **index.jsp** will read the request headers to see what cookies the browser supplied.
  - But of course these headers are stale; they don't reflect the results of the current action.

## Redirecting to a View

DEMO

- **Triggering a redirect neatly solves the problem.**
    - The subsequent HTTP GET from the browser will echo the up-to-date cookie information from our initial response, allowing the JSP to show the exact state of the system.
  - **Do your work in Demos/Redirect.**
    - The completed demo is in **Examples\Cookies\SimpleUrl\Step2**.
1. Open **src/cc/web/AddController.java**, and at the end of the **handleRequest** method, replace the **InternalResourceView** with a **RedirectView**. Add a second argument **true** to the call – this clarifies that the URL value is still to be interpreted relative to the application context (which might not have been the case with a redirect where it's always true for a forward).

```
return new ModelAndView
 (new RedirectView ("/index.jsp", true));
```

2. This class lives in the same package, so change the import of **InternalResourceView** to import **RedirectView** as well.

```
import
 org.springframework.web.servlet.view.RedirectView;
```

3. Open **src/cc/web/RemoveController.java** and make the same change there.
4. Build and test the application. You should see that all three actions (add and edit being two distinct actions from the user's perspective) now give immediate, accurate feedback.

## The redirect: Prefix

---

- The Cookies application does not use a view resolver.
  - This is by design, as we wanted to keep this example of handler mappings as simple as possible.
- But use of a view resolver is generally the best practice.
- How can controllers trigger redirects without instantiating **RedirectView** by themselves?
- Spring view resolvers recognize a special prefix that looks like a URL protocol – this is **redirect:** and when it is encountered at the beginning of a view name, it will be stripped out and will trigger a redirect to the remaining URL.
- This preserves the inversion of control inherent in using view names from controllers when building **ModelAndView** return values.
  - Though we haven't seen it (and won't in this course), another, more demanding best practice is to **inject the view names** as dependencies into the controller.
  - This approach realizes a total decoupling of a controller's runtime decisions about view service from any absolute resource locations.
  - **redirect:** is especially compelling when this practice is undertaken.

## Settling on a Practice

---

- So we have a big pile of options for handler mappings, controllers, and view resolvers.
  - And the few options we've seen in this chapter just scratch the surface: there are more than a dozen viable strategies for view resolution; fewer for handler mapping; and about that many again for controllers.
  - And we haven't even considered custom subtypes.
- Having read over the menu, it does become necessary to make some selections, if one wants to eat.
- Following are some recommendations – and we will use these in later course examples.
- The **SimpleUrlHandlerMapping** gives good flexibility without becoming onerous in ongoing development and maintenance.
- Though **XmlViewResolver** comes highly recommended by the Spring crew, it seems that **InternalResourceViewResolver** is more popular.
  - It links view names to resource paths rather tightly.
  - But the view names can be derived in some other, more dynamic way – after all, Spring IoC and dependency injection should be just the ticket for these sorts of problems.
  - **InternalResourceViewResolver** is a great way to get started at low cost, knowing that dependency injection can be used at a later stage of development to restore some needed flexibility.

## Developing Spring Web Applications

---

- How can a developer or team get started on a new Spring application most smoothly?
- Generally the process runs like this:
  1. Establish the stock shell for Spring applications, meaning a general-purpose mapping to **DispatcherServlet** and a context configuration file.
  2. Define at least a starter message bundle – we'll look at this more closely later, but the basic task is to drop a file such as **messages.properties** in **WEB-INF/classes**.
  3. Declare **stock handler-mapping** and **view-resolver** beans of your choice, so that you have a way of getting individual MVC cycles moving.
  4. Set up a parallel tree of **JUnit test cases** for your controllers and other significant actors. We won't have time to look at JUnit testing in this course, but testability is one of Spring's strong suits, so you should definitely take advantage of it.
  5. Now, page by page and controller by controller, you can start rolling out **individual JSPs and Java classes**, usually testing as you work through a given use case.
- Not considered here are general choices of basic infrastructure such as make utility (ahem, use Ant), IDE (Eclipse, but who's counting), and version control repository.

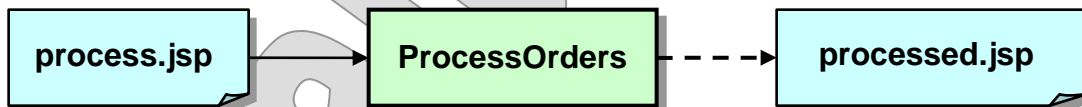
## Wholesale Spring

**LAB 4A**

### Suggested time: 60 minutes

In this lab you will implement a partial version of the Wholesale application, now retooled for the web. This will provide some challenging exercise in building Spring applications from scratch; it's not a refactoring exercise, as we've had plenty of those by now. In general this case study will offer opportunities to build new functionality "the right way" right off the bat.

The domain model is largely intact from earlier chapters – a few tweaks – and for this exercise you'll implement a simple page flow that will demonstrate end-to-end connectivity in processing prepared sales feeds at the direction of the operator.



Detailed instructions are found at the end of the chapter.

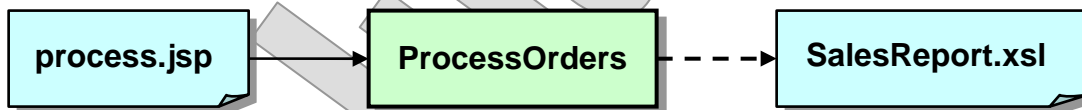
## Integrating XSLT

LAB 4B

### Suggested time: 30 minutes

In this lab you will improve on the response provided by the Wholesale application after it processes a batch of orders. We have a good XSLT transform already defined that can produce HTML from the XML sales report. What we need to do is integrate this XSLT into a Spring request/response cycle.

It turns out there's a **View** class for that! You'll instantiate **XsltView** and inform it with the XML source and XSLT transform locations, resulting in a modified page flow:



Detailed instructions are found at the end of the chapter.

## SUMMARY

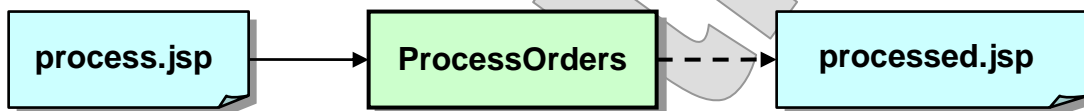
- **The Spring Web module is meant to simplify the development of complex web applications – but it is quite a complex system in and of itself.**
- **Still, there is an elegance to the kernel of the module: the request-handling process carried out by the dispatcher servlet.**
  - It's extrapolated from MVC, with **Controller** and **View** interfaces at the heart of the system.
  - Each of these actors is chosen by an agent: **HandlerMapping** for **Controller**, **ViewResolver** for **View**.
  - Each of these four roles is plugged in to the dispatcher servlet via the Strategy design pattern.
  - Each has multiple subtypes, which can be mixed, matched, combined, and extended.
- **The whole system sits on top of one or more Spring IoC containers.**
  - Bean configurations, autowiring, dependencies, and collections – all these Core techniques are now folded into the declarative side of Spring Web development.

# Wholesale Spring

## LAB 4A

In this lab you will implement a partial version of the Wholesale application, now retooled for the web. This will provide some challenging exercise in building Spring applications from scratch; it's not a refactoring exercise, as we've had plenty of those by now. In general this case study will offer opportunities to build new functionality "the right way" right off the bat.

The domain model is largely intact from earlier chapters – a few tweaks – and for this exercise you'll implement a simple page flow that will demonstrate end-to-end connectivity in processing prepared sales feeds at the direction of the operator.



### Lab workspace:

Labs/Lab4A

### Backup of starter code:

Examples/Wholesale/Web/Step1

### Answer folder(s):

Examples/Wholesale/Web/Step2

### Files:

docroot/WEB-INF/web.xml

docroot/WEB-INF/Wholesale-servlet.xml

src/cc/sales/web/ProcessOrders.java (to be created)

### Instructions:

1. As always, let's start with the deployment descriptor. Open **web.xml** and declare the **DispatcherServlet** with the name "Wholesale". Declare a mapping to this servlet for all requests of the pattern **\*.do**.
2. Open **Wholesale-servlet.xml**, and see that many of the bean definitions from previous exercises have been carried over to this starter code. Now that we're in a web context – and might be deployed to many different directories in different scenarios – both the fulfillment engine and the orders DAO require some flexibility in their paths for persistent files. This is the meaning of the **\${env.CC\_HOME}** phrases you see in these configurations: the syntax is borrowed from Ant, and the Java classes themselves resolve these phrases to the actual value of the **CC\_HOME** environment variable at runtime.
3. Declare a **SimpleUrlHandlerMapping** for your application. Set up one mapping, from the URL **/processOrders.do** to the bean name "processOrders".

**Wholesale Spring****LAB 4A**

4. Define your controller bean, with that same name, and class **cc.sales.web.ProcessOrders**.
5. Declare an **InternalResourceViewResolver** with an empty prefix and a suffix of “.jsp”.
6. Declare your message bundle. The file already exists – **WEB-INF/classes/messages.properties**. We haven’t covered this process yet; the declaration you want is shown here:

```
<bean
 id="messageSource"
 class=
 "org.springframework.context.support.ResourceBundleMessageSource"
 >
 <property name="basename" value="messages" />
</bean>
```

7. Validate your context configuration file, and fix any validity errors.
8. Take a look at **docroot/process.jsp** and see what sort of request you’re about to get: the page holds a multi-select list of filenames and will submit a request parameter **feeds** for each value the user chooses.
9. Now create your controller class, in **src/cc/sales/web/ProcessOrders.java**. Make the class implement the **Controller** interface and stub out your **handleRequest** method. You’ll want to import the **cc.sales** package for general use, as well.
10. Declare private fields referring to a **Fulfillment** object and an **OrderDAO**, and define mutator methods for each, **setEngine** and **setDatabase**.
11. Implement your **handleRequest** method: start by declaring a local variable **orders** and initializing it to a new **ListOfBatches**. Then call **setFeeds** on the bean, passing the array of values retrieved by calling **request.getParameterValues** with the parameter name “feeds”.
12. Call **orders.getBatches** with the **database** fields as a parameter. This will load in and return your order data.
13. Call **engine.fulfill**, passing the result of your call to **getBatches** as the first argument. This second argument gives the resulting sales file a name unique to the calling thread; a good way to generate this is to call **System.currentTimeMillis**, and then use **Long.toString** to convert that value to a string.
14. The result of this call to **fulfill** is a **double** representing total sales. Create a **ModelAndView**, passing three arguments: “processed” as your base view name, “totalSales” as a model key, and this total-sales value from the **fulfill** call.
15. Return this **ModelAndView** from your method.
16. Okay, what do you think – are you ready to build and test?

**Wholesale Spring****LAB 4A**

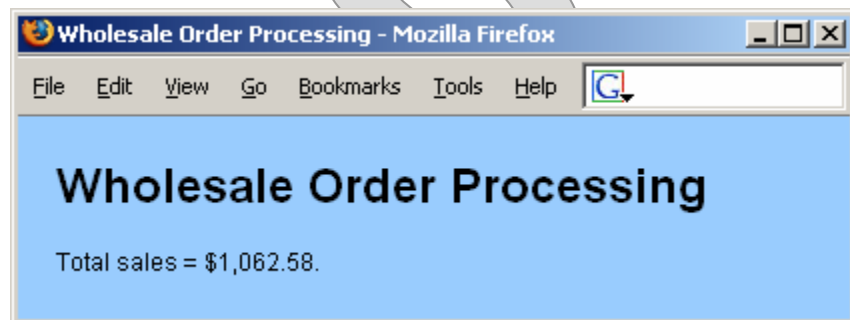
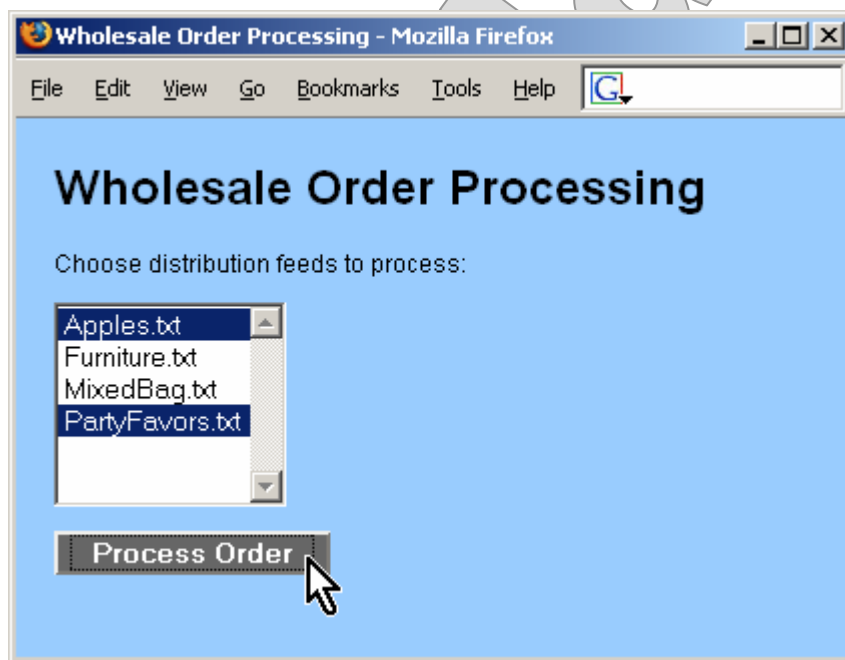
17. There is one thing missing: how will your **engine** and **database** dependencies be satisfied?

You could explicitly configure values for each property in the context configuration. But this is a good opportunity to use autowiring, since both properties are of distinctive types that should be preserved as singletons. Simply set **autowire** to “byType” on your “processOrder” bean.

18. Now, do build and test. The **ant** command will trigger the whole build process, and will deploy the application to Tomcat. If Tomcat is not running, start it, either before or after your Ant build.

19. Everything should now be in place; if you visit the following URL in your browser you should be able to select one or more sales feeds, click **Process**, and see the simple confirmation page in your browser.

`http://localhost:8080/Wholesale/process.jsp`



# Integrating XSLT

LAB 4B

In this lab you will improve on the response provided by the Wholesale application after it processes a batch of orders. We have a good XSLT transform already defined that can produce HTML from the XML sales report. What we need to do is integrate this XSLT into a Spring request/response cycle.

It turns out there's a **View** class for that! You'll instantiate **XsltView** and inform it with the XML source and XSLT transform locations, resulting in a modified page flow:



**Lab workspace:** Labs/Lab4B  
**Backup of starter code:** Examples/Wholesale/Web/Step2  
**Answer folder(s):** Examples/Wholesale/Web/Step3  
**Files:** docroot/SalesReport.xsl  
 src/cc/sales/Fulfillment.java  
 src/cc/sales/web/ProcessOrders.java

## Instructions:

1. The first step in getting the XSLT transformation activated is to make the source XML available through the controller. Right now the **Fulfillment.fulfill** method returns the raw sales number as a **double**. Refactor this: change the method to return the output filename instead: this means changing the method signature and returning **filename** instead of **totalSales**.
2. In **ProcessOrders.handleRequest**, after initializing the **orders** variable, declare a new local variable **view** and initialize it to a new **XsltView**. (Import this from **org.springframework.web.servlet.view.xslt**.)
3. Call **view.setUrl**, passing "SalesReport.xsl".

**Integrating XSLT****LAB 4B**

4. XSLT views need visibility to the application context, in a way that simple internal-resource views do not. (XSLT has broader general requirements than just loading a single file – there are at least two files to load, and both the source document and the transform can call for additional resources, including the use of relative paths.) So you’ll first need to declare a web context listener, back in **web.xml**:

```
<listener>
 <listener-class>
 org.springframework.web.context.ContextLoaderListener
 </listener-class>
</listener>
```

This is stock code for a Spring **web.xml** file, right along with the servlet and mapping.

5. Now get the context reference, using Spring utility classes – here’s the incantation:

```
view.setApplicationContext
 (WebApplicationContextUtils.getRequiredWebApplicationContext
 (request.getSession ().getServletContext ()));
```

You’ll need to import

**org.springframework.web.context.support.WebApplicationContextUtils.**

6. Remove the wrapping of a new **ModelAndView** around the call to **engine.fulfill** – but leave that method call intact, including its arguments.
7. Now that this method returns a filename, you can instead pass it to the constructor for a new **FileReader**. Pass that reader to a new **BufferedReader**, and use that to initialize a new local variable **in**.
8. Now you’re ready to return a **ModelAndView**. You have a **view**, which is primed with the XSLT document’s URL, and an application context to load other resources as necessary. This view will look in the model map for one of a few object types, one of which is a **Reader**, and finding that object it will treat it as the source for the transform. So: create a new **ModelAndView**, passing **view**, “sourceKey”, and **in** as the constructor arguments, and return that new object.
9. Now you have a new page flow, as your controller is ignoring the view resolver and returning an XSLT-based view:

**Integrating XSLT****LAB 4B**

10. Build and test: you should see your form submission followed directly by the formatted HTML report:

The image shows two screenshots of a web browser (Mozilla Firefox) demonstrating the workflow of an XSLT application. The first screenshot shows the 'Wholesale Order Processing' form with a list of distribution feeds and a 'Process Order' button. The second screenshot shows the resulting 'Sales Report' table.

**Wholesale Order Processing - Mozilla Firefox**

Choose distribution feeds to process:

- Apples.txt
- Furniture.txt
- MixedBag.txt
- PartyFavors.txt

**Process Order**

**Sales Report - Mozilla Firefox**

Product	Price	Quantity	Total
Candles	\$2.25	20	\$47.59
Baldwin	\$50.20	4	\$212.35
Winesap	\$48.00	6	\$304.56
Macoun	\$56.50	6	\$358.49
Horns	\$.75	80	\$63.45
Hats	\$1.50	48	\$76.14
<b>Total sales</b>			<b>\$1,062.58</b>

[HOME](#)