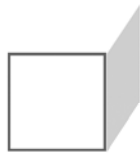
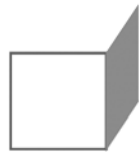
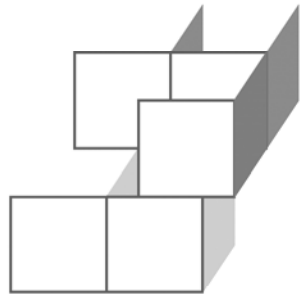




CHAPTER 1
THE WEB MODULE



OBJECTIVES

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

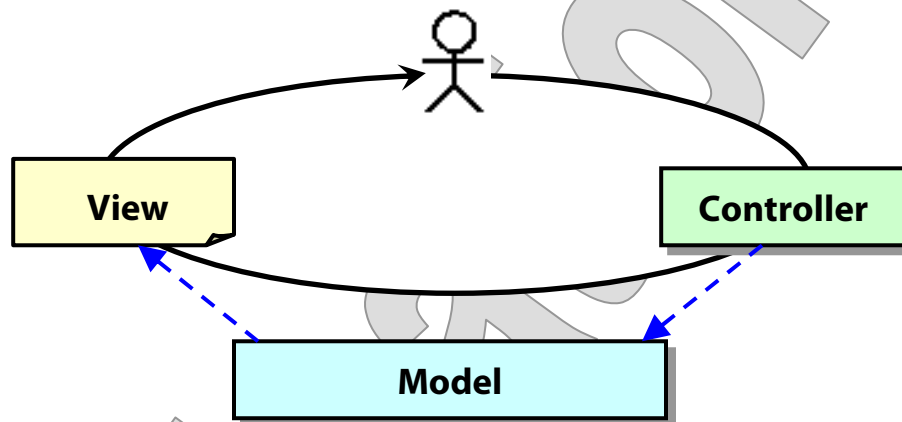
- Identify best practices and design patterns that have emerged in the years since Java EE was born, and explain how Spring facilitates these practices and patterns.
- Describe the lifecycle of an HTTP request/response roundtrip through a Spring web application: what components are involved in handling the request, carrying out work, presenting the next page, and handling errors.
- Refactor a traditional Java EE web application to use Spring.
- Describe the roles of key strategies in the Spring MVC cycle:
 - **HandlerMapping**
 - **Controller**
 - **ModelAndView**
 - **ViewResolver**
 - **View**

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.**
 - How should a servlet **choose the JSP** to present the next page – without hard-coding JSP locations in servlet code?
 - How can servlets and JSPs **share and propagate** information, especially that related to user input, and establish common access to business logic, including validation rules?

The Model/View/Controller Pattern

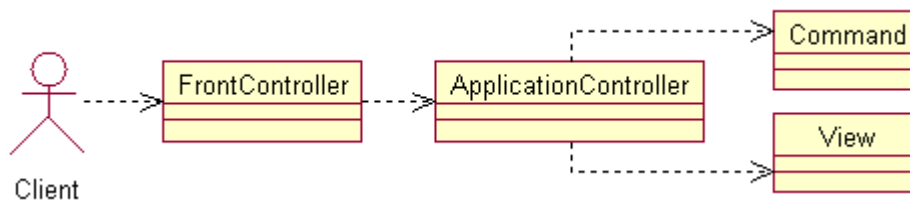
- As introduced in Chapter 1, MVC is a way of organizing any system – we’ll apply it to web applications specifically – into major roles **model**, **view**, and **controller**.



- 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

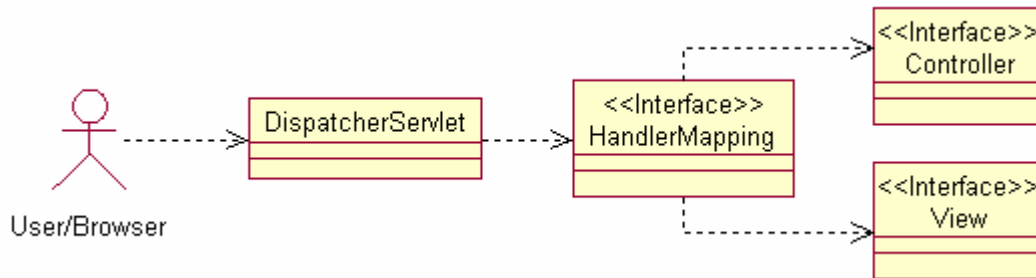
- A very popular Java EE design pattern is the **Front Controller**.
- This pattern recognizes the need for consistent pre-processing shared by many different request handlers – especially once they've been separated out according to MVC.
- This calls for a single controller at the front of the process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

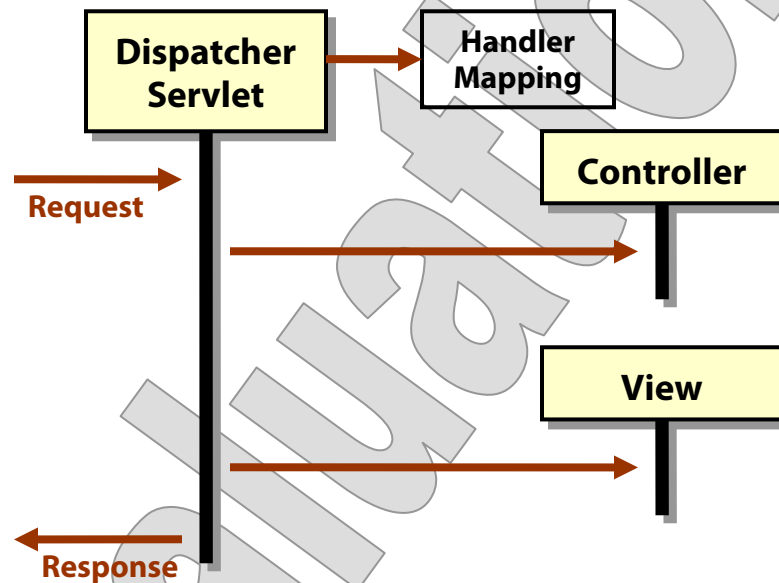
- The entry point to the entire Spring Web module is the **DispatcherServlet**, which is often configured as the one and only servlet in a Spring web application.
 - Find this and most of the key Spring Web types in **org.springframework.web.servlet**, or subpackages thereof.
 - This servlet handles 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

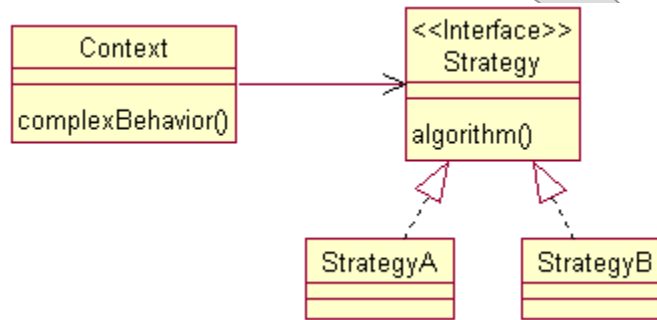
- So already we're getting an idea of the request/response cycle as implemented by the Spring Web module.
- We don't have the whole story yet, but we know this much:



- **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.

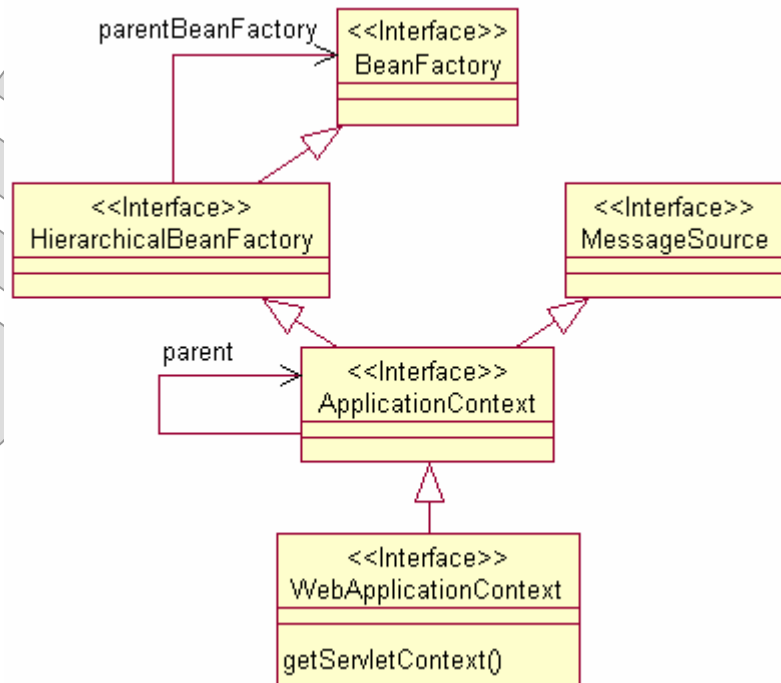
```
<servlet>
  <servlet-name>MyApplication</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

```
<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 – or include a series of more fine-grained “module” JARs, which is what we'll do in our builds.
 - 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:
 - Remember that 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.



Environment and Setup

- Check now to be sure that your system is set up to support the build-and-test structure for our hands-on exercises.
- Much of this has already been proven in the course of the hands-on exercises in the previous module:
 - JDK 6 should be set up on your system – typically in **c:\Java6**.
 - Additional tools in subdirectories of **c:\Capstone\Tools**, including **Ant1.6** and **Spring2.5**.
 - Your executable **path** includes the Java and Ant **bin** directories.
- For this module we also need a web server: that's **Tomcat 6**.
- Change your **CC_MODULE** variable to **c:\Capstone\SpringWeb**.

Tomcat

- To host our exercise applications, we'll be using the **Tomcat** web server from Apache.
 - Tomcat is a fully-functioning production Web server, and as an open-source tool is also a tremendous resource for any Web-application or HTTP development.
 - Tomcat is the reference implementation for servlets and JSP.
- Tomcat is already installed and runnable on your machine, in **c:\Capstone\Tools\Tomcat6.0**.
- This root has at the following subdirectories, among others:
 - **bin** holds the binaries, including scripts you will use to start and shut down the server.
 - **conf** holds configuration files, including a main file **server.xml** and a directory tree of XML fragment files per Web application.
- To start Tomcat, run **startup.bat** from the **bin** directory.
 - For this course, be sure to start Tomcat from the environment described on the previous page, including the **CC_MODULE** variable.
 - Applications that we deploy to Tomcat need to use this value to find their resources.
- You can start Tomcat now, and leave it running for most of this course. To shut it down, run **shutdown.bat**.



Code Organization

- Web-application projects use Ant, which will carry out a more complex set of tasks.

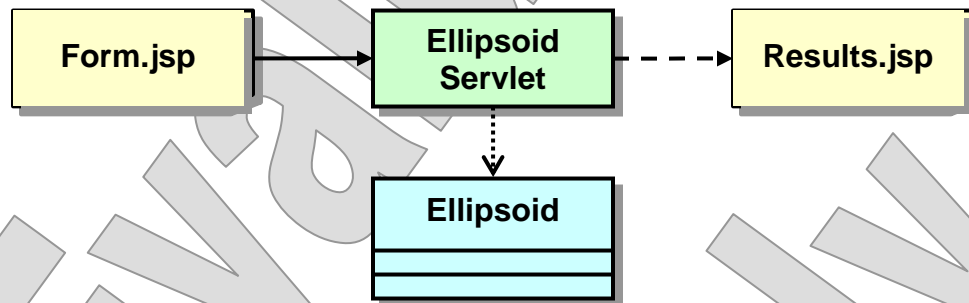
```
<root> (e.g. Examples/Ellipsoid/Step5)
  build.properties
  build.xml
  doc (as in standalone projects)
  docroot
    index.jsp
    Form.jsp
    Results.jsp
    Sphere.jsp
  classes
  tags
  WEB-INF
    Ellipsoid-servlet.xml
    web.xml
  META-INF
    application.xml
  src
    cc
      math
        Ellipsoid.java
        EllipsoidController.java
```

- In addition to the **src** directory, these projects include a **docroot** directory that holds web files such as HTML pages, JSPs, the **web.xml** deployment descriptor, supporting JARs, and tag files.
 - There will often be supporting data files as well.
- **Build and deploy to Tomcat with a simple command: ant.**

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.



- **Form.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

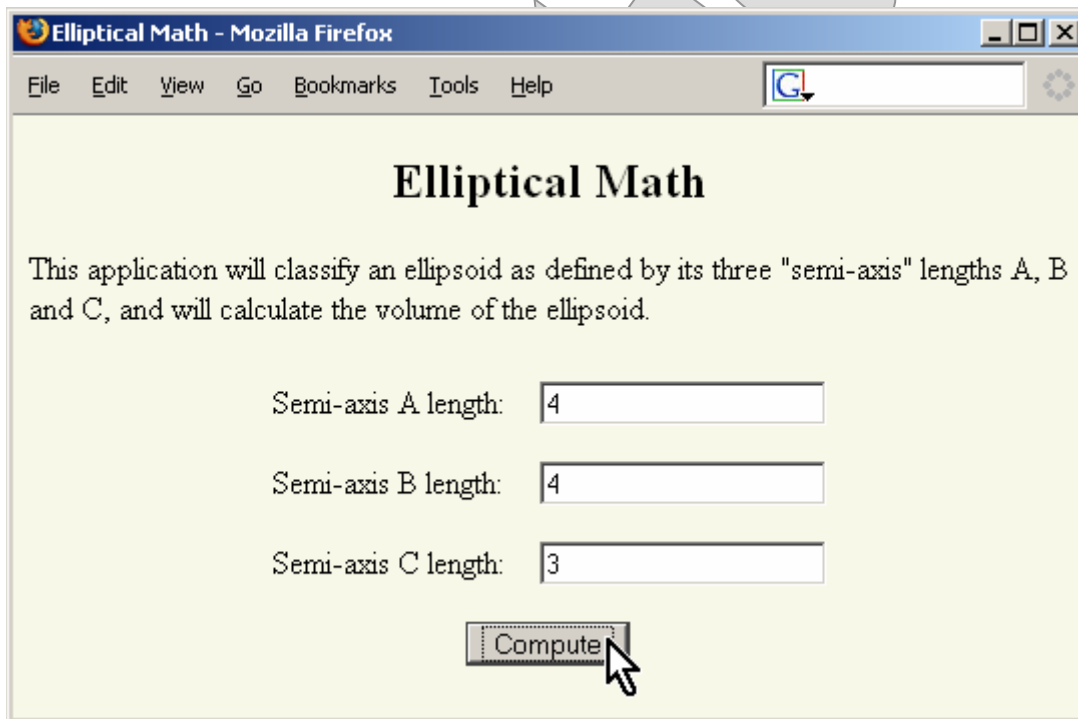
DEMO

1. Build the starter application ...

ant

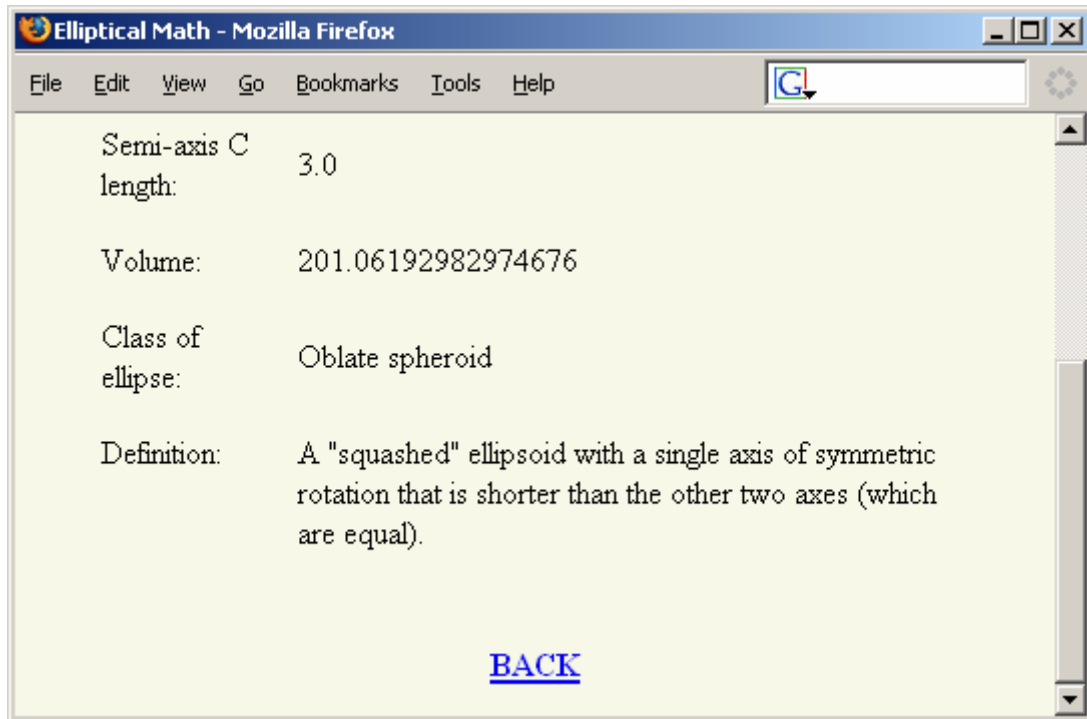
2. ... and test it out:

http://localhost:8080/Ellipsoid



A Minimal Spring Web Application

DEMO



3. Job one is to put Spring in place, so let's start by opening **docroot/WEB-INF/web.xml**.
4. 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

5. 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 name="/Compute.do"  
      class="cc.math.EllipsoidController" />
```

- This application uses Spring's default **handler mapping**, the **BeanNameUrlHandlerMapping**, which simply seeks a bean whose name is the request URL.
- That bean is the controller, and will be invoked to handle the request.
- Other strategies are possible, and we'll broaden our view of this function in the next chapter.

A Minimal Spring Web Application

DEMO

6. Open `src/cc/math/EllipsoidController.java` and see the controller code. So far, the `handleRequest` method doesn't do much:

```
public class EllipsoidController
    implements Controller
{
    public ModelAndView handleRequest
        (HttpServletRequest request,
         HttpServletResponse response)
        throws Exception
    {
        System.out.println ("**Controller invoked.**");
        return new ModelAndView
            (new InternalResourceView ("Results.jsp"));
    }
}
```

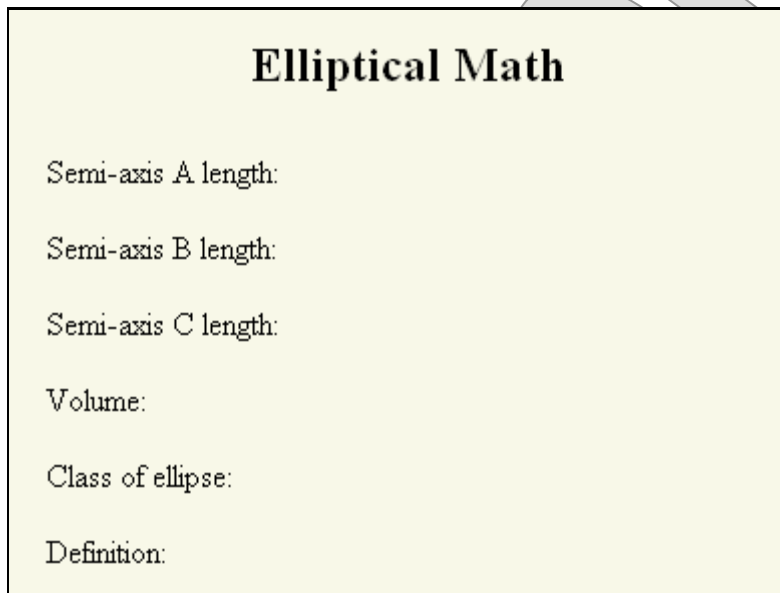
- This **ModelAndView** is constructed to aggregate a prepared **InternalResourceView**, which is view implementation that simply wraps a browser-addressable resource within the application.

A Minimal Spring Web Application

DEMO

7. Build, deploy, and test again to see two things.

- The results page does appear, but it shows no values, only the static labels:



- We can see that the controller was invoked by looking in the Tomcat console:

```
INFO: Servlet 'Ellipsoid' configured successfully  
***** Controller invoked. *****
```

(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.
8. Copy code from the servlet source file to the controller to create an **Ellipsoid** object and to populate it using request parameters:

```
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")));
    return new ModelAndView
        (new InternalResourceView ("Results.jsp"));
}
```

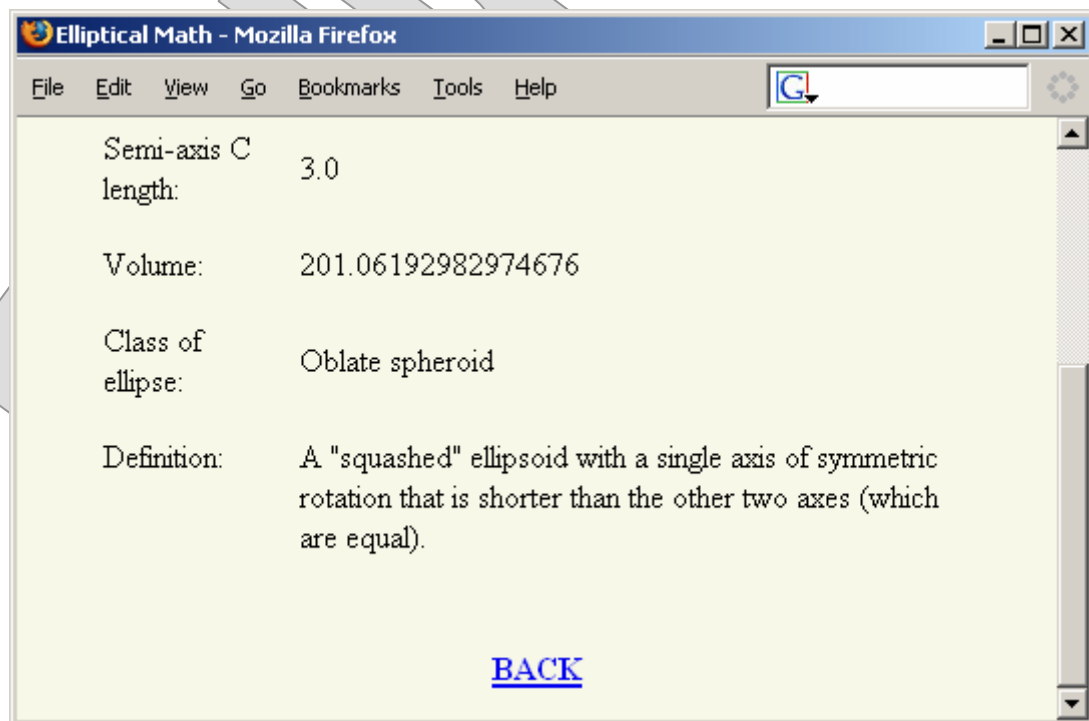
A Minimal Spring Web Application

DEMO

9. Still, so far our **ModelAndView** is really just a view. Add model information using a different constructor overload:

```
return new ModelAndView  
    (new InternalResourceView ("Results.jsp"),  
     "ellipsoid", delegate);
```

- **ModelAndView** can take a prepared **Map** with multiple keys and values, too, but we'll use this convenience constructor here.
10. Build and retest to see that you again have correct functionality: the controller is now creating a bean, piping input to the bean, and making the bean itself available to the view so that the expressions in **Results.jsp** can read out the results.



(This is the version in the example **Step3**.)

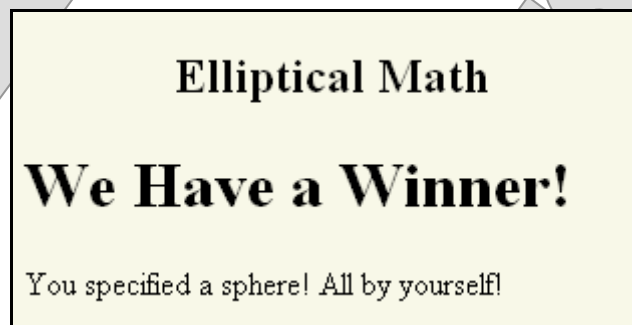
A Minimal Spring Web Application

DEMO

11. 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.
12. See also the three beans defined to wrap three JSPs in the application, giving them simple names "Form", "NormalResult" and "SphereResult".
13. 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);
```

14. 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. Try parameters you were using before, and then try using the same number for all three semi-axis lengths ...

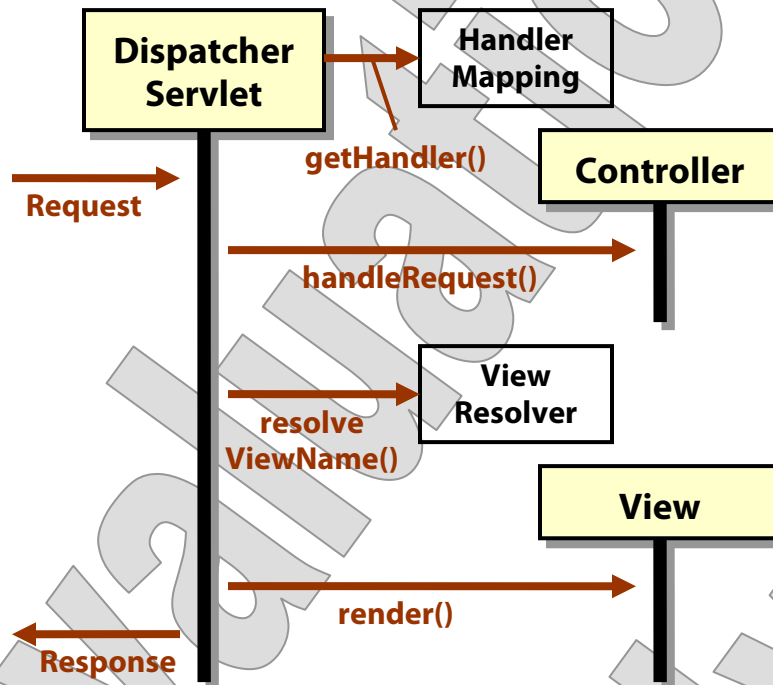


(This is the final version in the example **Step4**.)

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.
 - Our approach relies on the names of our Spring beans, for both controllers and views.
 - But both of these strategies can be customized – in fact you may be surprised at how many options there are – and we'll investigate that in the following chapter.
- Meanwhile, have you started wondering: how did the servlet find these two key objects?

Autowiring in the DispatcherServlet

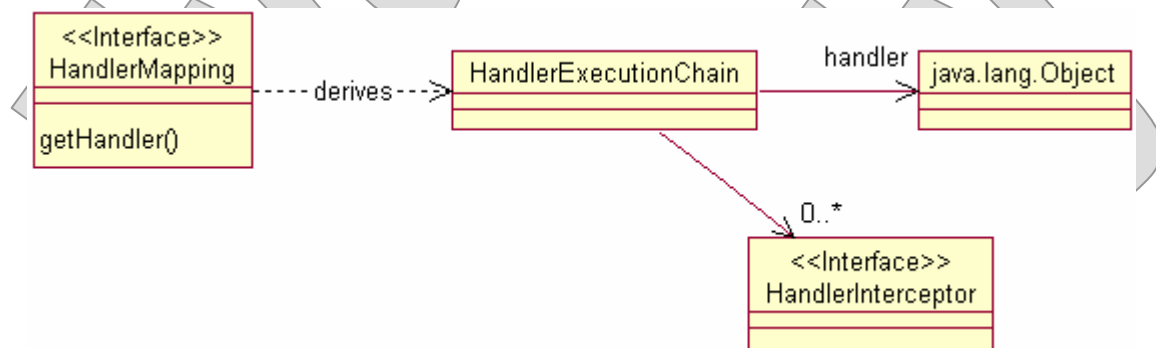
- The answer harks back to our study of Spring IoC in the previous chapter: **DispatcherServlet** finds these two delegates through beans autowiring by type.
- This isn't obvious, for the simple reason that the servlet isn't declared as a bean in the configuration itself.
- Also, it does what a declared 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:
 - One or more **HandlerMappings** are wired by type – as are **HandlerAdapters**, which allow request handlers of various types and method signatures to plug into HTTP request handling.
 - One or more **ViewResolvers** are 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”.

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);
}
```

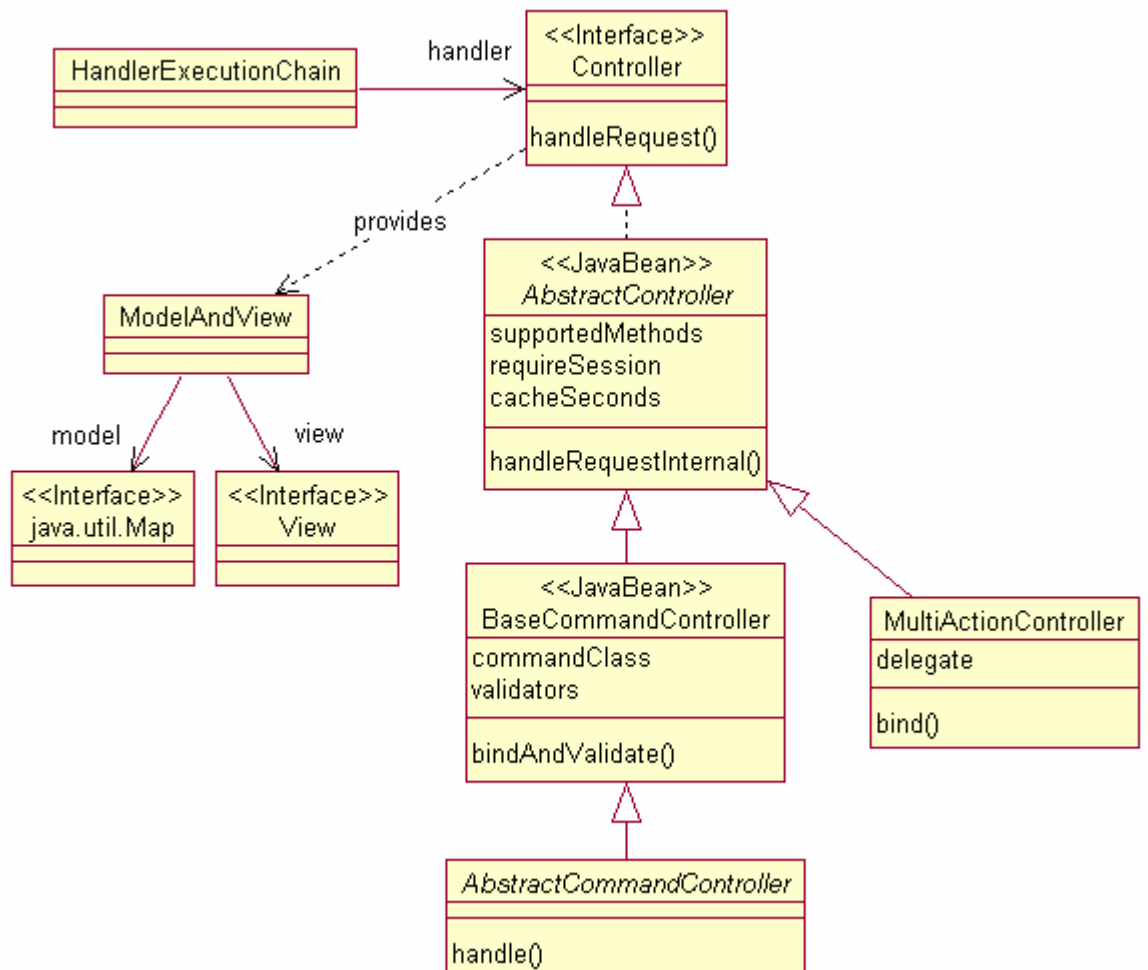
- This, in turn, navigates to one handler and any number of **HandlerInterceptors**.
- (We're still in `org.springframework.web.servlet` for all three of these types.)



- 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.
- We'll consider these different controller types in two later chapters.

The ModelAndView Class

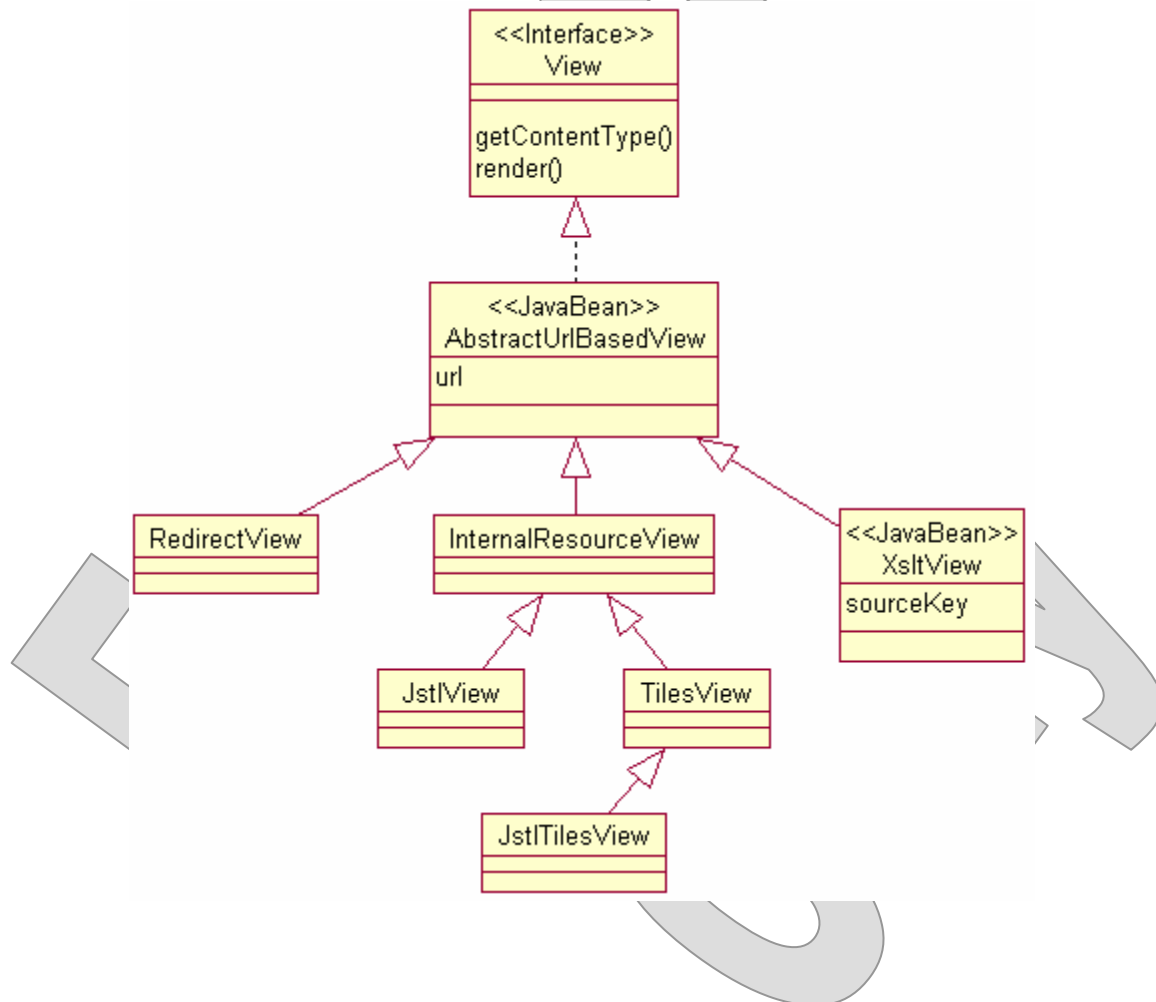
- **ModelAndView** is a simple aggregation of a **View** object and a “model” – which in this context means a map of keys and objects that might be useful to view rendering, and not the overall state model of the MVC application.

```
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.

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**:

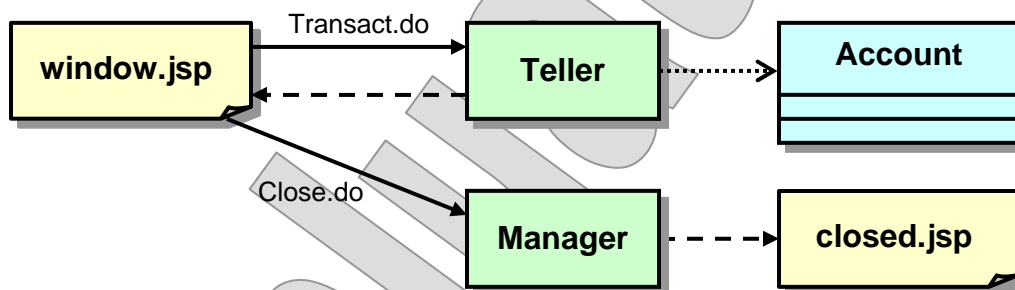
```
public interface ViewResolver
{
    public View resolveViewName
        (String name, Locale locale);
}
```

- 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.

Online Banking

EXAMPLE

- Let's use another example application as a way of reviewing the control paths that make Spring's MVC work.
 - We'll trace the flow of control all the way from the receipt of the incoming HTTP request around to the rendering of the response.
- See **Examples/Bank** for a two-page web application that allows the user to make transactions on a preconfigured bank account:



- From **window.jsp** the user actually has three options:
 - **Deposit** and **Withdraw** buttons request the **Transact.do** URL.
 - A link to **close this account** requests **Close.do**.

Online Banking

EXAMPLE

- Let's build and run the application first – so we know what it does, before asking how it does it.
 - Run **ant** and visit the following URL:
`http://localhost:8080/Bank`

Online Banking

Current account balance is 500.0.

Amount:

You may [close this account](#) at any time.

- Make a deposit ...

Online Banking

Current account balance is 598.5.

Amount:

You may [close this account](#) at any time.

- ... and close the account:

Online Banking

The account has been closed.

Online Banking

EXAMPLE

- Consider a request to make a deposit – what happens from the moment the user clicks the **Deposit** button?

1. The server receives an HTTP request for this resource:

`http://localhost:8080/Bank/Transact.do`

2. This is fielded by the web container, which (a) determines the correct web application by searching its index of deployed applications and their context URLs, and (b) consults the **web.xml** file for a possible servlet mapping.

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

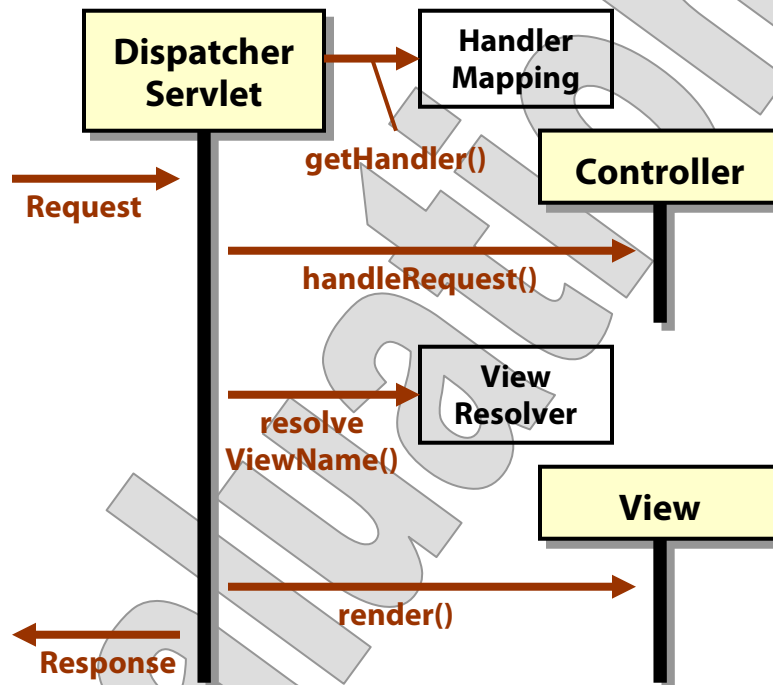
<servlet-mapping>
  <servlet-name>Bank</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

3. The application says all requests ending in **.do** go to the Spring **DispatcherServlet**, and the container dispatches the request there and waits for request handling and response rendering.
4. The **DispatcherServlet** has been initialized based on the servlet name given by **web.xml** and so knows to configure itself from a file **Bank-servlet.xml**.

Online Banking

EXAMPLE

5. Based on those bean declarations, it will now carry out the process diagrammed earlier:



6. So, first it looks for a **HandlerMapping**, and finds one:

```

<bean
  class="org.springframework.web.servlet.handler
    .SimpleUrlHandlerMapping"
  >
  <property name="mappings">
    <props>
      <prop key="/Approach.do" >Teller</prop>
      <prop key="/Transact.do" >Teller</prop>
      <prop key="/Close.do" >Manager</prop>
    </props>
  </property>
</bean>
  
```

Online Banking

EXAMPLE

7. The **SimpleUrlHandlerMapping** tells the servlet that a request to **Transact.do** should be handled by an instance of the **Teller** bean:

```
<bean id="Teller" class="cc.bank.Teller"
      autowire="byType" />
```

8. Thus comes a call to the **handleRequest** method on **cc.bank.Teller**. This singleton bean has a dependency on a bank account (specifically, Spring sees the method **setAccount**) auto-wired to the only **Account** instance in the configuration:

```
<bean class="cc.bank.CheckingAccount" >
  <constructor-arg type="double" value="500" />
</bean>
```

9. **handleRequest** then looks at request parameters and either deposits or withdraws the requested amount:

```
double amount = Double.parseDouble (amountParam);
if (request.getParameter ("deposit") != null)
  account.deposit (amount);
if (request.getParameter ("withdraw") != null)
  account.withdraw (amount);
```

10. Then it returns instructions to the **DispatcherServlet** in the form of a **ModelAndView** object that says two things: (a) put a bean called “account” at request scope so the view can see it, and (b) resolve to a view identified as “window”.

```
return new ModelAndView
  ("window", "account", account);
```

Online Banking

EXAMPLE

11. The **DispatcherServlet** finds (actually, has already found) its **ViewResolver** in the configuration:

```
<bean
  class="org.springframework.web.servlet.view
    .InternalResourceViewResolver"
>
  <property name="prefix" value="" />
  <property name="suffix" value=".jsp" />
</bean>
```

12. The **InternalResourceViewResolver** maps the view name to the location of an internal resource: the **window.jsp**.

13. It wraps this resource in an **InternalResourceView** object and hands that back to the **DispatcherServlet**.

14. The **DispatcherServlet** then asks this view object to **render** the HTTP response – this occurs as usual for a JSP, no real Spring intervention in the process – and then returns control to the web container.

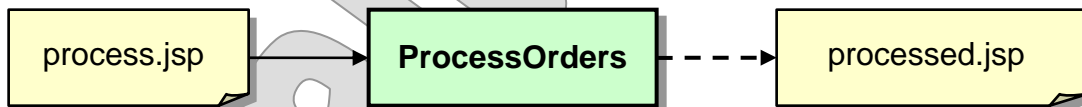
Wholesale Spring

LAB 1A

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.

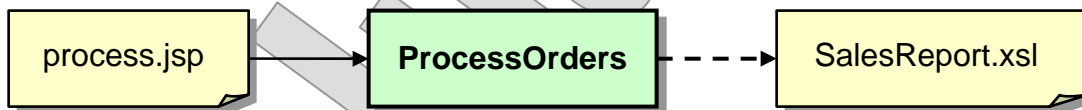
Integrating XSLT

LAB 1B

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

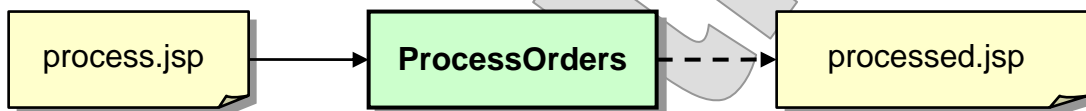
- **The Spring Web module is meant to simplify the development of complex web applications – but it is quite a complex system in and of itself.**
- **Still, there is an elegance to the kernel of the module: the request-handling process carried out by the dispatcher servlet.**
 - It's extrapolated from 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 1A

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/Lab1A

Backup of starter code: Examples/Wholesale/Step1

Answer folder(s): Examples/Wholesale/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_MODULE}** 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_MODULE** 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 1A**

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’ll dig into message and resource bundles later in the course; meanwhile, 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?

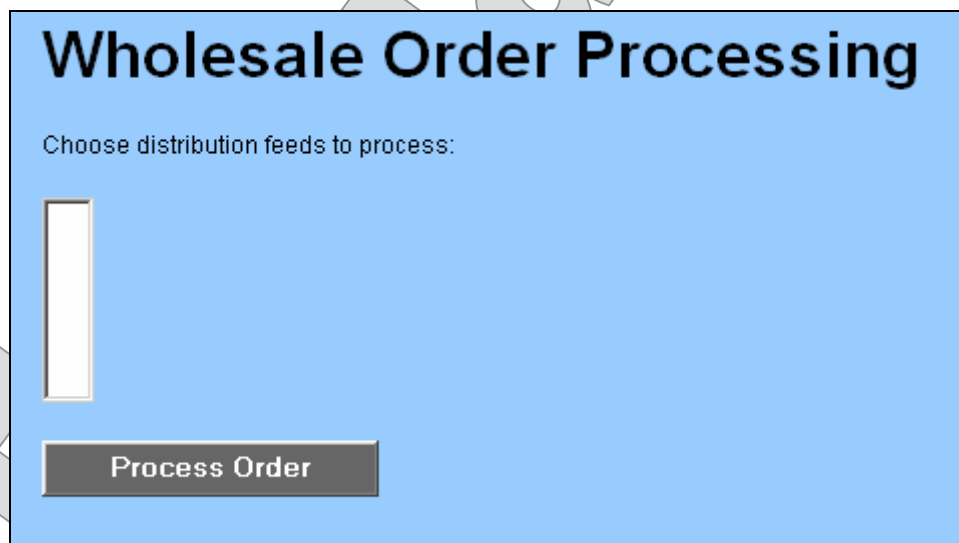
Wholesale Spring**LAB 1A**

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>



Wholesale Order Processing

Choose distribution feeds to process:

Process Order

Okay ... not quite everything!

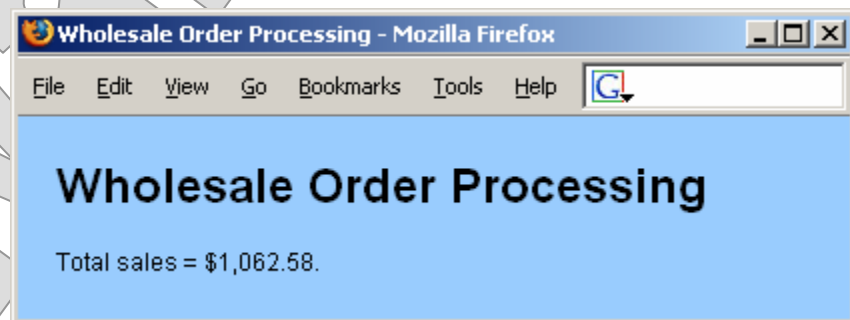
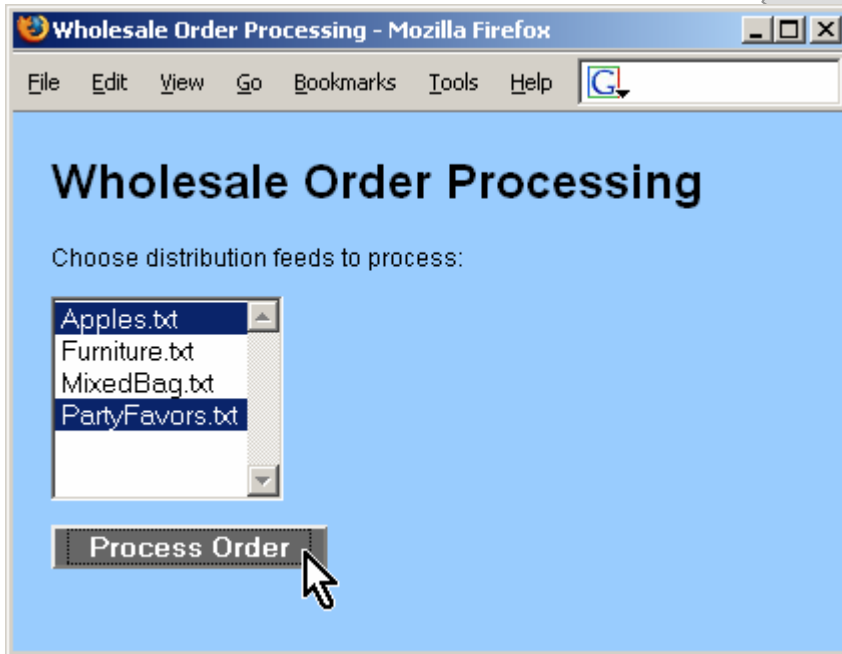
20. There’s a set of files that this application reads as its persistent data, and it doesn’t exist yet. Run a special Ant target to make this happen:

ant build-DB

Copying 4 files to C:\Capstone\SpringWeb\Examples\Wholesale\DB

Wholesale Spring**LAB 1A**

You'll now see a new directory `c:\Capstone\SpringWeb\Examples\Wholesale\DB ...` and so will the application. Refresh the browser, choose from the list, and try processing that "feed" of distribution orders:

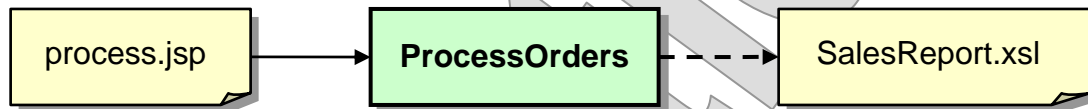


Integrating XSLT

LAB 1B

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/Lab1B
Backup of starter code: Examples/Wholesale/Step2
Answer folder(s): Examples/Wholesale/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 1B

- 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.

- 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.

- Remove the wrapping of a new **ModelAndView** around the call to **engine.fulfill** – but leave that method call intact, including its arguments.
- 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**.
- 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.
- Now you have a new page flow, as your controller is ignoring the view resolver and returning an XSLT-based view:

Integrating XSLT**LAB 1B**

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' as a formatted HTML 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
Total sales			\$1,062.58

[HOME](#)