CHAPTER 1

THE WEB MODULE
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

- The **model/view/controller** pattern, or **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.

![Diagram of Front Controller Pattern](image)

- 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 here’s a good start:

1. **DispatcherServlet** asks a **HandlerMapping** for a **Controller**.
2. The **Controller** does work as requested by the user, and then returns to the **DispatcherServlet** a **view**, by way of an identifying string.
3. **DispatcherServlet** then consults the **ViewResolver** to turn that string into an actual instance of a **View**.
4. The **View** is responsible for rendering the **response**.

© 2006-2012 Will Provost.
All rights reserved by Capstone Courseware, LLC.
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’ve also mentioned the **ViewResolver** as a 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.

```xml
<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. You must make sure that the Spring Web JARs are available to your web application.
  - These include, at a minimum, the Core JARs we discussed in the previous module (beans, core, context, etc.), plus:
    ```
    org.springframework.web -3.1.2.RELEASE.jar
    org.springframework.web.servlet-3.1.2.RELEASE.jar
    ```
    - You can include these in your `WEB-INF/lib` directory.
    - Or you can pre-install them on the server.
    - We use the JARs in place (`c:/Capstone/Tools/Spring3.1/dist`) as part of our compile path, but pre-install them on the server, which speeds up our build-and-deploy process a bit.
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.
The Evolution of Spring MVC

- Going back through Spring 2.5 and 2.0 and earlier, the Spring MVC framework has grown up quite a lot.

- The Spring 2.0 approach to MVC was straightforward, highly flexible, and elegant in its own way:
  - Controllers, views, handler mappings, view resolvers, and many other supporting objects would be configured in the web application context, as ordinary beans.
  - Then a rich hierarchy of classes grew out from each major concept: Controller, View, HandlerMapping, etc.
  - The developer would configure a graph of objects of these types, configure them, and connect them to the dispatcher servlet.

- The downside was felt in a code-intensive approach that seemed to require a lot of XML and a lot of Java, sometimes to do relatively straightforward things.

- Starting with Spring 2.5 and more emphatically in Spring 3.x, the existing system is simplified by focusing on a new handler-mapping approach that relies on annotated controller classes.
  - We use @Controller, @RequestMapping, and a handful of other annotations to bind HTTP requests to Java methods.
  - This results in much less overall coding.
  - It’s important to understand though that the full infrastructure described thus far is still in force: it’s just that a particularly slick handler mapping detects controllers, and the Controller classes are deprecated in favor of these annotated classes.
@Controller and @RequestMapping

- An MVC-specific component stereotype @Controller will be detected by the web application context.
  - It’s important to remember the magic incantation that makes the context scan for component stereotypes:
    `<context:component-scan base-package="..." />`
  - Then place this annotation on your web controller classes.

- Unless told otherwise, DispatcherServlet will create and use a RequestMappingHandlerMapping.
  - This class will further scan your beans for a second key annotation, which is @RequestMapping.
  - You can use this annotation on a class, to set certain common elements for all methods on the class.
  - Then, it must be used on each method that you want to be called in response to an HTTP request. You can qualify what requests apply to which methods, based on URL, URL patterns, HTTP method, request parameters, and a few other criteria.

- Again, the full Spring MVC API is still available, and has its uses.
  - The Controller classes that were so central to Spring-2.0 development are largely forgotten.
  - The similarly critical command objects are still in play, but they feel much more like ordinary JavaBeans – nothing special!
  - Infrastructure such as validators and data binders is still configured, but it is driven more by annotations and sometimes by Java reflection.
Tools and Environment

- 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 7 should be set up on your system – typically in c:\Java7.
  - Additional tools in subdirectories of c:\Capstone\Tools, including Ant1.8 and Spring3.1.

- For this module we also need a web server: that’s Tomcat 7.

- The SetEnvironment script in the Admin directory is the same as in the previous module – but you should still run it, since it adjusts your CC_MODULE variable which will be needed in upcoming builds.
  - Remember to edit this file for your JAVA_HOME location if this is anything other than c:\Java7.

- Next, run a script called SetupTomcat: this copies necessary JAR files into Tomcat’s lib directory so that they will be in the class path when needed by our applications.
  - This is how we put the Spring JARs in place.
  - We also copy in the Commons Logging JAR, and the JSTL 1.2 JARs since many of our JSPs use JSTL.
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\Tomcat7.0.

- This root has 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.

```xml
<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
  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`. 
Eclipse

- The lab software supports Ant builds and testing using a standalone browser such as Internet Explorer or FireFox.
- An optional overlay of files provides an Eclipse workspace for integrated development and testing.
- If you want to work in Eclipse, run it now from the environment we have already set up.
  - When Eclipse runs Tomcat, it will pass this environment on, and Tomcat will need the `CC_MODULE` variable to find certain resources in some of our later projects.
  - For a workspace location enter:
    `c:\Capstone\SpringWeb\Eclipse`
- You’ll see a tree of working sets and projects in the Project Explorer.
  - There is one project for each example, demonstration, and lab exercise mentioned in this coursebook.
  - The book refers to projects by their file paths, relative to `c:/Capstone/SpringWeb` – so for example `Examples/Cars/Step3`.
  - The corresponding Eclipse project names will use underscores instead of slashes: `Examples_Cars_Step3`.
  - All projects start out closed, and build automatically is set, so it’s good to keep the number of open projects to a minimum.
- To test in Eclipse, you can generally just Run on Server.
  - The projects are predefined with links to the Spring JARs.
A Minimal Spring Web Application

- 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/MVC.
  - 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

1. Build the starter application with **ant**, and test it by pointing your favorite browser at this URL:

   http://localhost:8080/Ellipsoid

   "Elliptical Math"

   This application will classify an ellipsoid as defined by its three "semi-axis" lengths A, B and C, and will calculate the volume of the ellipsoid.

   - Semi-axis A length: 2
   - Semi-axis B length: 2
   - Semi-axis C length: 1
   - Compute

   Or from Eclipse just open the Demos_MVC project and **Run on Server**. You will be asked at this point (and from time to time as you test various exercises) if you want to run on the predefined server configuration, if you want to restart the server, etc.

   Then you’ll see the above web page in an internal viewer.
A Minimal Spring Web Application

2. Enter values as shown on the previous page, and click **Compute**. See the results page:

## Elliptical Math

- **Semi-axis A length:** 2.0
- **Semi-axis B length:** 2.0
- **Semi-axis C length:** 1.0
- **Volume:** 16.755160819145562
- **Class of ellipse:** Oblate spheroid
- **Definition:** A "squashed" ellipsoid with a single axis of symmetric rotation that is shorter than the other two axes (which are equal).
A Minimal Spring Web Application

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`.

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

5. Change the URL mapping as well. We could route this specific request URL to the Spring framework, but it’s typical to map a wildcard pattern to allow Spring to handle a wide range of requests:

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

6. Copy the file `Ellipsoid-servlet.xml` from the main project directory to `docroot/WEB-INF`. This will give the `DispatcherServlet` a configuration to read when it starts up – though there isn’t much to it:

   ```
   <beans xmlns="..." ... >
   <context:component-scan base-package="cc.math" />
   </beans>
   ```
A Minimal Spring Web Application

7. Create a new class `cc.math.EllipsoidController`, and establish it as an MVC controller:

   ```java
   package cc.math;
   
   import org.springframework.stereotype.Controller;
   @Controller
   public class EllipsoidController {
   }
   ```

8. Give your class a request-handling method and bind it to a specific request URL:

   ```java
   import org.springframework.stereotype.Controller;
   import org.springframework.web.bind.annotation.RequestMapping;
   @Controller
   public class EllipsoidController {
     @RequestMapping ("/Compute.do")
     public String process (Ellipsoid ellipsoid) {
       System.out.println ("** Controller invoked **");
       return "results.jsp";
     }
   }
   ```

   - This method automatically parses request parameters into corresponding properties on an `Ellipsoid` instance – this takes the place of the typical `getParameter` and parse-parameter code seen in the `EllipsoidServlet` that we are retiring.
   - The method indicates a desired view directly by its URL.
9. Now, in `docroot/form.jsp`, change the `action` attribute of the input form to invoke Spring and your new controller:

```
<form action="Compute.do" method="post">
```

10. Test again, at the same URL, and get the same results.

- You can prove that your new code is in force by looking at the server console or log output – instead of the “** Servlet invoked **” output, your “** Controller invoked **” line will appear.
- There will also be a good bit of unmistakable logging output from the Spring framework itself.

- **Why is this better than what we had already?**
  - Compare your controller code to the servlet code – that should give you a good sense of the advantages, even for a tiny application such as this.
  - We are getting all the typical servlet processing, almost automatically, by declaring a controller and a handling method, identifying a **command object** in the method signature, and identifying the view to which we should forward when done.
  - Though we’re not taking advantage of it in this initial demo, we also have the full power of the Spring container and bean configuration at our fingertips now – so connections to services, persistence components, etc., will be much cleaner.
  - And there are many more capabilities in the MVC framework that we will investigate in the next few chapters, including custom data bindings, validation, and more sophisticated request mappings.

- **This is the intermediate answer in Examples/Ellipsoid/Step2.**
Let’s explore a bit further ...

11. In docroot/WEB-INF/Ellipsoid-servlet.xml, define a non-default view resolver that will map core view names to JSPs:

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

12. Now you can refer to the desired view by a more abstract name in your controller:

```java
@RequestMapping ("/Compute.do")
public String process (Ellipsoid ellipsoid) {
    return "results.jsp";
}
```

13. Test this and see no difference in functionality:
   - This may not seem like a big win, and in itself it isn’t. But it’s a simple proof of an important feature, which is the ability to represent outcomes of requested work, rather than explicit resource locations.
   - A view resolver translates the outcome to the resource, and different mapping strategies can be applied over time, making for much more maintainable code.

• This is the intermediate answer in Examples/Ellipsoid/Step3.
14. Let’s take the view-resolver experiment a bit farther: remove the `InternalResourceViewResolver` and replace it with a resolver that maps the outcome string to a Spring bean by name.

```xml
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver" />
```

- Then provide three options:

```xml
<bean
   id="form"
   class="org.springframework.web.servlet.view.InternalResourceView"
>
   <constructor-arg value="/form.jsp" />
</bean>

<bean
   id="normal"
   class="org.springframework.web.servlet.view.InternalResourceView"
>
   <constructor-arg value="/results.jsp" />
</bean>

<bean
   id="sphere"
   class="org.springframework.web.servlet.view.InternalResourceView"
>
   <constructor-arg value="/sphere.jsp" />
</bean>
```

- Note that some bean names are the same as the JSP names, but not all of them; it’s up to us to make those connections when we use this resolution strategy.
A Minimal Spring Web Application

15. Give this more of a workout by changing your controller code as shown below:

```java
@RequestMapping (/"Compute.do")
public String process (Ellipsoid ellipsoid) {
    return ellipsoid.getType ().equals ("Sphere")
        ? "sphere"
        : "normal";
}
```

16. Test one last time. For the most part you'll see the same old results. But if you enter the same values for all three semi-axis lengths, you get a different page entirely:

- The controller is able to report the **logical** difference between these two types of outcomes – or whatever is important about the work that it might be asked to do.
- The view resolver binds that information to a **physical** resource in the web application – or even to a Java class that can produce HTML dynamically.

- This is the final answer in Examples/Ellipsoid/Step4.
Autowiring in the DispatcherServlet

- **DispatcherServlet** finds its handler mapping, view resolver, and other strategy 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.¹
- 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”.

¹ Though it is possible to make it a bean and to configure it as such, using Servlets 3 and Spring MVC Java-based configuration rather than XML-based configuration. This is not an approach we will pursue in this course, but see Section 16.14 of the Spring reference for more on this.
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**.

```java
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 View Interface

- While the **Controller** part of the API is deprecated, **View** is still alive and kicking!
- 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

- Translating, or resolving, the controller’s view identifier to an actual View object is the responsibility of a configured ViewResolver:

```java
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 quickly when you’re writing tens or scores or hundreds of controllers.
Wholesale Spring

**Suggested time: 60 minutes**

In this lab you will begin an implementation of a web application over the Wholesale domain objects that you configured in some lab exercises in the earlier module on the Spring Core. This provides an opportunity to build a Spring MVC application from scratch. The domain model is largely intact from those earlier exercises – 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.

Then you will improve on the initial response page, which just shows the total sales figure after the fulfillment engine is done processing. We have a good XSLT transform already defined that can produce HTML from the XML sales report. All you need to do is integrate this XSLT into a Spring request/response cycle.

And 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.
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 roles genuinely 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.

The annotation-based approach common in Spring 2.5 and 3.x does not alter this architecture; it merely simplifies what had become one of the more labor-intensive parts of it by facilitating more direct bindings between HTTP and Java methods in controller classes.
In this lab you will begin an implementation of a web application over the Wholesale domain objects that you configured in some lab exercises in the earlier module on the Spring Core. This provides an opportunity to build a Spring MVC application from scratch. The domain model is largely intact from those earlier exercises – 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.

Then you will improve on the initial response page, which just shows the total sales figure after the fulfillment engine is done processing. We have a good XSLT transform already defined that can produce HTML from the XML sales report. All you need to do is integrate this XSLT into a Spring request/response cycle.

And 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/Lab1

**Backup of starter code:** Examples/Wholesale/Step1

**Answer folder(s):** Examples/Wholesale/Step2 (intermediate) Examples/Wholesale/Step3 (final)

**Files:**
- docroot/WEB-INF/web.xml
- docroot/WEB-INF/Wholesale-servlet.xml
- src/cc/sales/web/ProcessOrders.java
- src/cc/sales/Fulfillment.java
**Wholesale Spring**

**LAB 1**

**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. Create a `Wholesale-servlet.xml` configuration file; best approach is to copy this from `Examples/Ellipsoid/Step3`, and rename it. Remove everything but the `<context:component-scan />` element and the `InternalResourceViewResolver` bean definition.

If you didn’t do the Wholesale exercises in the previous module (or maybe didn’t do them all that recently), take a moment to review the domain model. A summary diagram is shown below.

![Domain Model Diagram](image)

For our purposes the interesting classes are the `Fulfillment` engine and the `OrderDAO`. For the web application, there is a new DAO implementation that knows how to find records of standing orders for the wholesaler – which we call “feeds” – by scanning a given root directory. See `src/cc/sales/db/OrderDAOImplFile.java`. This class can give a list of feed filenames, and can load or save a given feed as represented by a `ListOfBatches`.

The `Fulfillment` class holds the logic for processing a given feed – as when a customer calls to request a repeat of their standing order. The logic focuses on calculating the invoice price for that order, including consideration of a configurable tax policy. It produces a report of the completed order as a file – which it saves in its own configurable root directory – and returns the total cost.
3. You’ll configure beans of these types now. Start by defining a bean whose class is `cc.sales.MASalesTax`. You won’t need an id or name for it as we’ll use autowiring.

4. Now define a bean of type `cc.sales.Fulfillment`. Set its autowire policy to “constructor”.

5. Then set one constructor argument explicitly – this is the root directory for saving sales reports once the application gets running:

   ```xml
   <bean class="cc.sales.Fulfillment" autowire="constructor" >
     <constructor-arg
       index="1"
       value="${env.CC_MODULE}/Examples/Wholesale/DB/TotalSales"
     />
   </bean>
   ``

   As you may have guessed the class knows how to translate the expression “${env.CC_MODULE}” into an absolute path.


7. Define the homeBase property on this bean to be

   `${env.CC_MODULE}/Examples/Wholesale/DB/Feeds`

8. You’re going to do some autowiring in your annotated controller, so add the `<context:annotation-config/>` element to your context as well.

9. Now you’re ready to create your controller class, `cc.sales.web.ProcessOrders`. Make it a `@Controller` and give the class itself a `@RequestMapping` annotation, with a value of “/Process/Process.do”. (The JSPs and now our controller request URLs are segregated by major function; we’ll be in the /Process space for this lab and then look at /Manage URLs and control logic in later chapters.)

10. Give your controller a field `engine`, of type `Fulfillment`.

11. Create a `setEngine` method for this field, and annotate it as `@Autowired`.

12. Do the same thing for a field `database` of type `OrderDAO`.

13. Create a method `onSetup` that takes no parameters and returns a `ModelAndView`. We haven’t looked at this class yet, but we’ll see plenty of it in the next chapter. It is basically a nice briefcase in which to hand over both model and view information, and has many overloaded constructors to allow for different view and model definitions.

14. Annotate your method as shown below. This too is new – we are saying that we only want this method to be called for an HTTP GET at the URL defined in the class-level annotation:

   ```java
   @RequestMapping (method=RequestMethod.GET)  
   public ModelAndView onSetup ()
   ```
15. Implement the method to return a new `ModelAndView` object, passing three arguments to the constructor: the view identifier “process”, the model attribute name “database”, and your `database` field.

16. Open `index.html` and see that one of its hyperlinks is to your Process.do URL. So you will see that HTTP GET immediately and your controller will be active.

17. Test your application at this point. When you visit the following URL (or just Run on Server in Eclipse) you will see that index page:

   `http://localhost:8080/Wholesale`

---

**Wholesale Order Processing**

**Define feeds**

Edit existing feeds or add new ones.

**Process orders**

Process one or more order feeds in a batch, and view sales report.
Click Process Orders and your controller will be invoked. This will serve up a second page, with a list of available feeds from which to choose:

You can see the logic for this page in `docroot/process.jsp` – and notice that it is creating the HTML `<option>`s by iterating over the list given by the expression `${database.list}`. That is where the JSP consumes the model attribute “database” that you provide in your `ModelAndView` object.

You can also see that this page will HTTP POST a selection of one or more feed-definition filenames back to the Process.do URL.

18. Add a second request-handling method to your controller. Call this one `onSubmit`; it too will return a `ModelAndView`, and it can take a `ListOfBatches` as a command parameter – call this `orders`. This type aligns nicely with the HTML form being submitted, as it has a `feeds` property that is ready to bind zero to many strings as feed-definition filenames.

19. In this method, call `orders.getBatches`, passing `database`, to get a list of lists of `Order` objects: one `List<Order>` corresponding to each filename chosen by the user. (The cardinality can get confusing here. What we’ve been calling a standing order or “feed” is represented as a list of `Orders`, each of which is a product, price, and quantity. Since the user can select multiple feeds for processing in one shot, we need a `List<List<Order>>`.)

20. Now call `engine.fulfill`, passing that list of lists of orders as the first of two arguments. The second argument is a unique stamp for the resulting report filename, and is expected to be a timestamp. To derive this, call `System.currentTimeMillis`, and convert the result to a string.

21. `engine.fulfill` returns a total sales figure, as a `double`. Create and return a new `ModelAndView` object, with a view name of “processed” and a model attribute named “totalSales” with this value.
22. Test again, and you should now be able to choose one or more feeds from the list and click DXX to see a page with the total amount for invoicing.

You can also check Examples/Wholesale/DB for a filename of the form TotalSales_NNNNNNNNNN.xml, as produced by the Fulfillment bean.

This is the intermediate answer in Step2.

23. Now you’re going to migrate to a more complete response page by taking advantage of the XSLT transforms already defined for this application. First, you’re going to need access to the XML report produced by the Fulfillment engine. Open that class and change the signature of the fulfill method so that it returns a String.

24. At the bottom of the method, instead of the sales number, just return filename as assembled earlier in the method.

25. Now, at the top of your onSubmit method, define a local variable view of type org.springframework.web.servlet.view.xslt.XsltView.

26. This view type is going to need to know a few things so that it can produce the right HTTP response content. First, call setUrl, passing “Process/SalesReport.xsl” as the location of the XSLT transform for it to load.

27. The view will need a reference to our application context so that it can load this and possibly other resources. So we need to derive a reference to that context object. Make your class implement org.springframework.context.ApplicationContextAware, and implement the setApplicationContext method to initialize a new field context.

28. In onSubmit, call view.setApplicationContext and pass this context object in.

29. Create a Reader variable called in, and initialize it to a new BufferedReader, based on a new FileReader, which in turn is based on the results of your call to engine.fulfill – that is, you can copy that whole method invocation and the two argument expressions in it into this new code.

30. Now, when you build your ModelAndView, don’t bother with the “processed” JSP at all; instead, pass view as the first argument to the constructor. You don’t need to pass the sales figure as a model attribute anymore. One quirk of XsltView is that, while it takes the transform definition as a configurable property, it looks for its source XML document as a model attribute called “sourceKey”. So pass “sourceKey” and your in variable as the second and third arguments to the ModelAndView constructor.
31. Test again, and you will see the complete HTML output of the XSLT transform as applied to the XML report that is generated on the fly based on the user’s choice of sales feed.

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macoun</td>
<td>$56.50</td>
<td>6</td>
<td>$358.49</td>
</tr>
<tr>
<td>Winesap</td>
<td>$40.00</td>
<td>1</td>
<td>$42.30</td>
</tr>
<tr>
<td>Golden delicious</td>
<td>$55.00</td>
<td>2</td>
<td>$116.32</td>
</tr>
<tr>
<td><strong>Total sales</strong></td>
<td></td>
<td></td>
<td><strong>$517.12</strong></td>
</tr>
</tbody>
</table>

This is the final answer in Step3.