Table of Contents

Chapter 1 - Introduction to Spring Boot

Objectives

Key objectives of this chapter

- Overview of Spring Boot
- Using Spring Boot for building microservices
- Examples of using Spring Boot

1.1 What is Spring Boot?

- Spring Boot (*http://projects.spring.io/spring-boot/)* is a project within the Spring IO Platform (*https://spring.io/platform*)
- Developed in response to Spring Platform Request SPR-9888 "Improved support for 'containerless' web application architectures"
- Inspired by the DropWizard Java framework *(http://www.dropwizard.io/)*
- The main focus of Spring Boot is on facilitating a fast-path creation of stand-alone web applications packaged as executable JAR files with minimum configuration
- An excellent choice for creating microservices
- Released for general availability in April 2014
  - (*https://spring.io/blog/2014/04/01/spring-boot-1-0-ga-released*)

1.2 Spring Boot Main Features

- Spring Boot offers web developers the following features:
  - Ability to create WAR-less stand-alone web applications that you can run from command line
  - Embedded web container that is bootstrapped from the **`public static void main`** method of your web application module:
    - Tomcat servlet container is default; you have options to plug in Jetty or Undertow containers instead
  - No, or minimum, configuration
    - Spring Boot relies on Spring MVC annotations (more on annotations later ...) for configuration
  - Build-in production-ready features for run-time metrics collection, health checks, and externalized configuration

1.3 Spring Boot on the PaaS

- PaaS (Platform as a Service) clouds offer robust, scalable, and cost-efficient run-time environments that can be used with minimum operational involvement
- There are two popular choices for deploying Spring Boot applications:
  - Cloud Foundry *(https://www.cloudfoundry.org/)*
  - Heroku (*https://www.heroku.com/*)
  - **Note**: Google App Engine PaaS has not advanced beyond the Servlet 2.5 API, so you won't be able to deploy a Spring Application there without some modifications
    - Spring Boot uses Servlet API version 3.1
- Spring Boot is being financed by Pivotal Software Inc. which also, in partnership with VMware, sponsor Cloud Foundry
  - While Cloud Foundry is OSS available for free, Pivotal offers a commercial version of it called Pivotal Cloud Foundry

1.4 Understanding Java Annotations

- Annotations in Java are syntactic metadata that is baked right into Java source code; you can annotate Java classes, methods, variables and parameters
- Basically, an annotation is a kind of label that is processed during a compilation stage by annotation processors when the code or configuration associated with the annotation is injected in the resulting Java class file, or some additional operations associated with the annotation are performed
- An annotation name is prefixed with an '@' character
- **Note:** Should you require to change annotation-based configuration in your application, you would need to do it at source level and then re-build your application from scratch

1.5 Spring MVC Annotations

- Spring Boot leverages Spring MVC (Model/View/Controller) annotations instead of XML-based configuration
  - Additional REST annotations, like *@RestController* have been added with Spring 4.0
- The main Spring MVC annotations are:
  - **@Controller / @RestController** – annotate a Java class as an HTTP end-point
  - **@RequestMapping** – annotate a method to configure with URL path / HTTP verb the method responds to
  - **@RequestParam** - named request parameter extracted from client HTTP request

1.6 Example of Spring MVC-based RESTful Web Service

- The following code snippet shows some of the more important artifacts of Spring MVC annotations (with REST-related Spring 4.0 extensions) that you can apply to your Java class
- **Note:** Additional steps to provision and configure a web container to run this module on are required

```
// … required imports are omitted
@RestController
public class EchoController {
    @RequestMapping("/echoservice", method=GET)
    public String echoback(@RequestParam(value="id") String echo) {
        return echo;
    }
}
```

- The above annotated Java class, when deployed as a Spring MVC module, will echo back any *echo* message send with an HTTP GET request to this URL:

`http(s)://<Deployment-specific>`**`/echoservice?id=hey`**

1.7 Spring Booting Your RESTful Web Service

- Spring Boot allows you to build production-grade web application without the hassle of provisioning, setting up, and configuring the web container
- A Spring Boot application is a regular executable JAR file that includes the required infrastructure components and your compiled class that must have the **`public static void main`** method which is called by the Java VM on application submission
  - The **`public static void main`** method references the **`SpringApplication.run`** method that loads and activates Spring annotation processors, provisions the default Tomcat servlet container and deploys the Spring Boot-annotated modules on it

1.8 Spring Boot Skeletal Application Example

- The following code is a complete Spring Boot application based on the Spring MVC REST controller from a couple of slides back

```
// … required imports are omitted
@SpringBootApplication
@RestController
public class EchoController {
    @RequestMapping("/echoservice", method=GET)
    public String echoback(@RequestParam(value="id") String echo) {
        return echo;
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(EchoController.class, args);
    }
}
```

- You run Spring Boot applications from command line as a regular executable JAR file:

```
java -jar Your_Spring_Boot_App.jar
```

- The default port the Spring Boot embedded web container starts listening to is 8080
  - To change the default port, you need to pass a **server.port** System property or specify it in the Spring's **application.properties** file

1.9 Converting a Spring Boot Application to a WAR File

- In some scenarios, users want to have a Sprint Boot runnable JAR file converted into a WAR file
- For those situations, Spring Boot provides two plug-ins:
  - **spring-boot-gradle-plugin** for the Gradle build system (*https://gradle.org/*)
  - **spring-boot-maven-plugin** for the Maven build system (*https://maven.apache.org/*)
- For more details on Spring Boot JAR to WAR conversion, visit *https://spring.io/guides/gs/convert-jar-to-war/*

1.10 Externalized Configuration

- Lets you deploy the same Spring Boot artifact (jar file) in different environments - config is drawn from the environment
- Properties are pulled from (note - this is an incomplete list)
  - OS Environment variable SPRING_APPLICATION_JSON
  - Java System properties
  - OS Environment variables
  - Application property files - application.properties or application.yml
    - in a '/config' folder in current directory
    - in the current directory
    - in a classpath '/config' package
    - in the classpath root
    - In a folder pointed to by 'spring.config.location' on the command line.

1.11 Starters

- Spring Boot auto-configures based on what it finds in the classpath
- So, the set of modules is determined by what's in the 'pom.xml'
- The project provides a number of 'starter' artifacts that pull in the correct dependencies for a given technology
- Some examples:
  - spring-boot-starter-web
  - spring-boot-starter-jdbc
  - spring-boot-starter-amqp
- You just need to include these artifacts in the 'pom.xml' to configure those technologies.

1.12 The 'pom.xml' File

- Assuming you're building with Apache Maven, the 'pom.xml' file describes all the artifacts and build tools that go into producing a delivered artifact.
- Generally, use the 'starter parent':

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.2.RELEASE</version>
</parent>
```

- It defines all the dependency management and core dependencies for a Spring Boot application

1.13 Spring Boot Maven Plugin

- The Maven plugin can package the project as an executable jar file, that includes all the dependencies

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

1.14 Summary

- Spring Boot eliminates many of the headaches related to provisioning, setting up and configuring a web server by offering a framework for running WAR-less web applications using embedded web containers
- Spring Boot favors annotation-based configuration over XML-based one
  - Any change to such a configuration (e.g. a change to the path a REST-enabled method responds to), would require changes at source level and recompilation of the project
- Spring Boot leverages much of the work done in Spring MVC

Chapter 2 - Spring MVC

Objectives

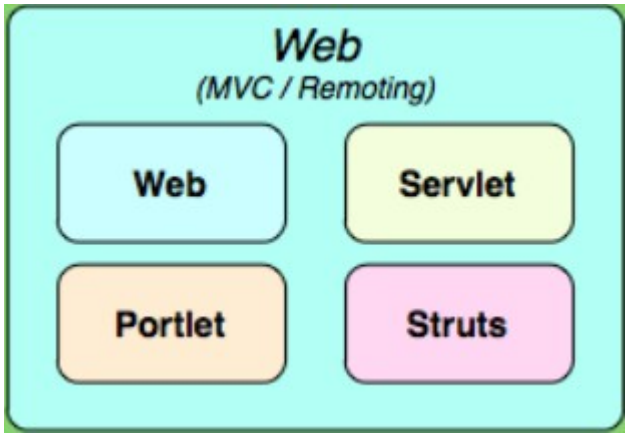
Key objectives of this chapter:

- Overview of Spring MVC
- Mapping Web Requests
- Controller Handler Methods
- View Resolution
- Form Tag Library

2.1 Spring MVC

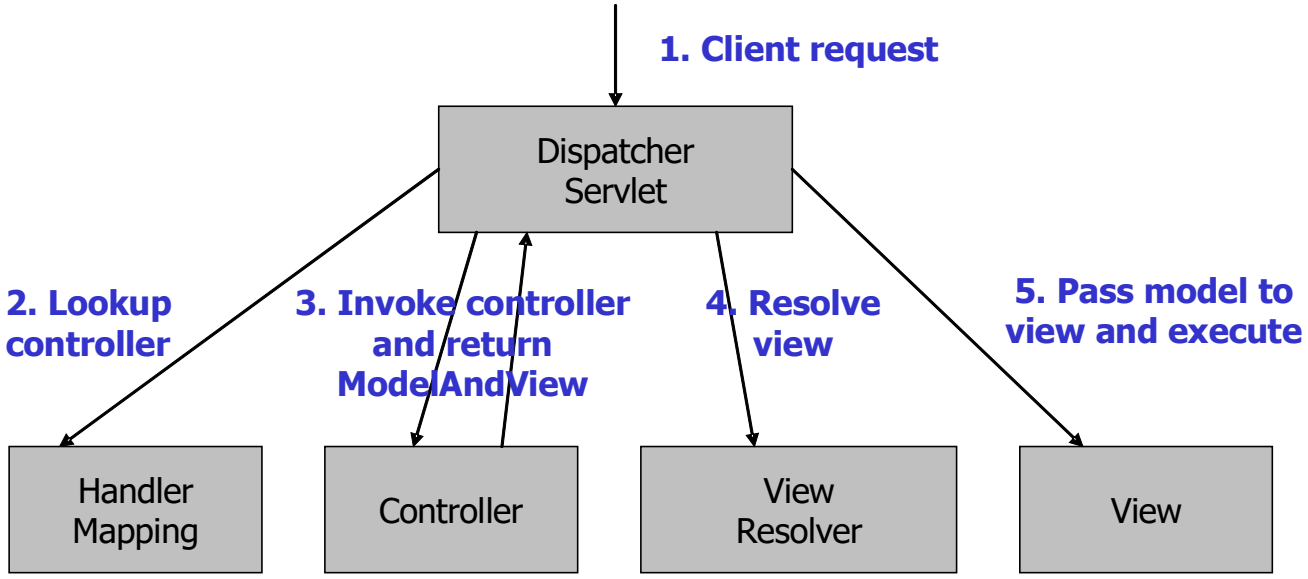
- Spring provides a MVC (Model/View/Controller) web application framework
- Addresses state management, workflow, validation, etc
- Spring MVC framework is modular, allowing various components to be changed easily
- The "Controllers" and "Handler Mappings" of Spring MVC underwent major changes in Spring 3 and later
  - Now annotations are used heavily for these components
  - The implementations of the Spring MVC 'Controller' interface that used to be the parent class of application controllers are now deprecated in favor of annotation controllers
- Spring 3 also introduced an 'mvc' namespace for configuration files to simplify Spring MVC configuration
- Spring 4 added some different view technologies

2.2 Spring Web Modules



- The Spring 'Web' module has core web support classes
  - It is required by other modules
- The 'Servlet' module has Spring MVC and other Servlet support classes
- There are also 'Portlet' and 'Struts' modules for integrating those types of web applications with Spring
- Spring 2.5 had broken the Spring MVC classes into a separate distribution
  - It was the first step to the modular packaging of Spring 3

2.3 Spring MVC Components



2.4 DispatcherServlet

- Configure DispatcherServlet in web.xml

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
    org.springframework…DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>*.request</url-pattern>
</servlet-mapping>
```

2.5 Context Loaders

- A context loader is used to load configuration files
- Use a servlet context listener

```
<listener>
      <listener-class>org…ContextLoaderListener
      </listener-class>
</listener>
```

- Use context-param to specify file locations

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring-services.xml,
               /WEB-INF/spring-daos.xml</param-value>
</context-param>
```

2.6 Spring MVC Example

- To demonstrate each part of a Spring MVC web application, we will develop an application that displays "Hello World"
- The steps are:
  - Develop the controller
  - Configure Spring MVC controllers
  - Configure a view resolver
  - Develop the view definition

2.7 Spring MVC Example

- Develop the controller

```
@Controller
public class HelloController  {
      @RequestMapping("/hello")
  public ModelAndView sayHello(
      @RequestParam("userMessage") String message) {
      return new ModelAndView(
        "helloView", "helloMessage", message);
  }
}
```

- The method above is invoked by the request:

```
<context root>/hello.request?userMessage=Howdy
```

- Configure Spring MVC to pick up controllers
  - This is actually very easy to tell Spring MVC to use annotations

```
<mvc:annotation-driven />
<context:component-scan base-package="com.webage.web" />
```

2.8 Spring MVC Example

- Configure a view resolver

```
<bean id="vres" class="org…InternalResourceViewResolver">
  <property name="prefix"><value>/WEB-INF/jsp/</value>
  </property>
  <property name="suffix"><value>.jsp</value>
  </property>
</bean>
```

- Create the JSP (/WEB-INF/jsp/helloView.jsp)

```
<html>
  <head><title>Hello World Example</title></head>
  <body>
    Here is the message: <b>${helloMessage}</b>
  </body>
</html>
```

2.9 Spring MVC Mapping of Requests

- Prior to Spring 3 you would use various "Handler Mapping" beans in Spring configuration files to tell Spring how to determine how the incoming requests would map to controllers
- Now in Spring 3 you simply need to use @RequestMapping annotations on methods that are activated by the <mvc:annotation-driven> configuration
  - This is much simpler and easier to configure how individual methods will be invoked
- You can also use a <mvc:view-controller> element in the configuration to map a request directly to a view without going to a controller first
  - This is very common for the "home page" of an application

```
<mvc:view-controller path="/" view-name="home"/>
```

2.10 Advanced @RequestMapping

- You can have the @RequestMapping annotation at the class level and the method level to combine how a common pattern is used for a set of requests
  - This can even include @PathVariable declarations or the type of HTTP request

```
@Controller
@RequestMapping("/appointments")
public class AppointmentController {
      // gets all appointments
      @RequestMapping(method = RequestMethod.GET)
      public List<Appointment> getAll() { .. }

      // gets only appointments for a specific date
      @RequestMapping(value="/{day}",
            method = RequestMethod.GET)
      public List<Appointment> getForDay(
            @PathVariable Date day) { .. }
```

2.11 Composed Request Mappings

- Spring 4 added "composed" annotations that extend from @RequestMapping
  - @GetMapping
  - @PutMapping
  - @PostMapping
  - @DeleteMapping
- They add in the appropriate 'method=...', so you don't need to type it.

2.12 Spring MVC Annotation Controllers

- With Spring MVC 3 the "controller" classes do not need to extend or implement Spring-specific classes
  - All that is needed is the @Controller annotation
- This means that methods within these classes that are meant to handle requests have a number of options for the method signature
  - What is used depends on what the method needs to do to handle the request
- There are a number of options for "handler methods":
  - Method parameters
  - Method return types

2.13 Controller Handler Method Parameters

- Some of the most common parameters to controller handler methods are:
  - Domain classes from the application
    - Spring can initialize this object matching properties of the object to request parameter names

```
@RequestMapping("/addCustomer")
public String addNewCustomer(Customer customerToAdd) { ..
```

  - Parameters annotated with @PathVariable, @RequestParam, or @CookieValue

```
@RequestMapping("/display/{purchaseId}")
public String displayPurchase(@PathVariable int purchaseId)
```

  - A Map, Spring 'Model', or Spring 'ModelMap' object which can have objects added for the view that will be rendered

```
@RequestMapping("/edit/{purchaseId}")
public String editPurchase(Model model,
      @PathVariable int purchaseId) {
      Purchase toEdit =
            purchaseService.findPurchaseById(purchaseId);
      model.addAttribute("purchase", toEdit);
```

2.14 Controller Handler Method Return Types

- Some of the most common return types of controller handler methods are:
  - A Spring 'ModelAndView' object constructed in the method

```
public ModelAndView listUsers() {
      List<User> allUsers = ...;
      ModelAndView mav = new ModelAndView("userList");
      mav.addObject("users", allUsers);
      return mav;
```

  - A String with the view name

```
public String listUsers(Model model) {
      List<User> allUsers = ...;
      model.addObject("users", allUsers);
      return "userList";
```

  - A Map, Spring 'Model' object, or application domain class

```
public Map<String, Object> listUsers() {
      List<User> allUsers = ...;
      ModelMap map = new ModelMap()
      map.addAttribute("users", allUsers);
      return map;
```

2.15 View Resolution

- Spring uses a ViewResolver to map the logical view name returned by a controller to a View bean
- The View bean is responsible for rendering output to the client
- Implementations of ViewResolver include:
  - InternalResourceViewResolver (discussed earlier)
  - BeanNameViewResolver
  - XmlViewResolver
  - ResourceBundleViewResolver

2.16 InternalResourceViewResolver

- Maps a logical view name to a View object that delegates to a template in the web context
- A common choice of template is a JSP
- It attaches a prefix and suffix to the logical name to create a path to the template file

```
<bean id="viewResolver"
      class="org…InternalResourceViewResolver">
  <property name="prefix"><value>/WEB-INF/jsp/</value>
  </property>
  <property name="suffix"><value>.jsp</value>
  </property>
</bean>
```

2.17 BeanNameViewResolver

- Maps a logical view name to a bean with the same name
- Declare the BeanNameViewResolver to enable resolution by bean names

```
<bean id="beanNameViewResolver" class="org…BeanNameViewResolver"/>
```

- Useful for non-template output such as MS Excel or PDF views

```
<bean id="orderReport" class="com…OrderExcelView">
  …
</bean>
```

2.18 XmlViewResolver

- Works similar to BeanNameViewResolver but it checks a separate XML file for the bean definitions

```
<bean id="xmlViewResolver" class="org…XmlViewResolver">
  <property name="location">
    <value>/WEB-INF/views/views.xml</value>
  </property>
</bean>
```

- Helps keep configuration files modular by putting View beans in a separate file

2.19 ResourceBundleViewResolver

- Use a ResourceBundleViewResolver to resolve to a view based on the user's locale

```
<bean id="resourceBundleViewResolver" class="org…
ResourceBundleViewResolver">
  <property name="basename">
    <value>view</value>
  </property>
</bean>
```

- View definitions are stored in properties files
- The basename property defines the fixed portion of the properties file name
  - E.g. view_en_CA.properties for Canadian English

2.20 ResourceBundleViewResolver

- Example:
  - view_en_US.properties

```
orderDisplay.class=com…OrderPdfView
```

  - view_en_CA.properties

```
orderDisplay.class=org.springframework…JstlView
orderDisplay.url=/WEB-INF/jsp/orderEnCA.jsp
```

  - view.properties (default)

```
orderDisplay.class=org.springframework…JstlView
orderDisplay.url=/WEB-INF/jsp/orderDefault.jsp
```

2.21 Using Multiple View Resolvers

- You can use multiple view resolvers
- Use the "order" property to set a priority search order

```
<bean id="beanNameViewResolver" class="org…BeanNameViewResolver">
  <property name="order">
    <value>1</value>
  </property>
</bean>
<bean id="vres" class="org…InternalResourceViewResolver">
  <property name="prefix"><value>/WEB-INF/jsp/</value>
  </property>
  <property name="suffix"><value>.jsp</value>
  </property>
  <property name="order">
    <value>2</value>
  </property>
</bean>
```

2.22 Spring Form Tags

- Spring 2.x introduced form tags that allow you to bind form fields to command objects in Spring MVC
- Helps to avoid using verbose HTML tags with manual binding of names and values
- JSPs require the following taglib:

```
<@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
```

- Tag classes and tld files are in the Spring 'Servlet' module with the rest of Spring MVC

2.23 form and input Tags

- Example where command bean is called customer and has name and phoneNumber properties:

```
<form:form commandName="customer"
    action="saveCustomer.request">
  Name:
  <form:input path="name"/><br/>
  Phone:
  <form:input path="phoneNumber"/><br/>
  <input type="submit"
    value="Save Changes"/>
</form:form>
```

2.24 password and hidden Tags

- The password and hidden tags create an HTML input element of type password and hidden, respectively:

```
<form:form commandName="login"
     action="login.request">
   Password:
   <form:password path="password"/><br/>
   <form:hidden path="userId"/><br/>
   ...
</form:form>
```

2.25 checkbox Tag

- The checkbox tag creates an HTML input element of type checkbox:

```
<form:form commandName="user"
     action="saveUser.request">
   Allow access?
   <form:checkbox
        path="accessAllowed"/><br/>
   ...
</form:form>
```

- The property is of type boolean or Boolean

2.26 checkbox Tag

- The property can also be an array or Collection:

```
<form:form commandName="user"
     action="saveUser.request">
   User Groups<br/>
   Finance: <form:checkbox
      path="userGroups" value="FINANCE"/><br/>
   Regulatory: <form:checkbox
      path="userGroups" value="REGULATORY"/><br/>
   Legal: <form:checkbox
      path="userGroups" value="LEGAL"/><br/>
   ...
</form:form>
```

- The array or Collection will have one element for each checked checkbox

2.27 radiobutton Tag

- The property behind radiobuttons will be set to the value of the radiobutton that is checked:

```
<form:form commandName="paintConfig"
    action="saveConfig.request">
  Red: <form:radiobutton
    path="color" value="R"/> <br/>
  Green: <form:radiobutton
    path="color" value="G"/> <br/>
  Blue: <form:radiobutton
    path="color" value="B"/> <br/>
   ...
</form:form>
```

2.28 textarea Tag

- The textarea tag creates an HTML textarea with a number of rows and columns
- The property behind a textarea is typically a String:

```
<form:form commandName="order"
     action="saveOrder.request">
   Description:<br/>
   <form:textarea path="descr"
      rows="5" cols="25"/>
   ...
</form:form>
```

2.29 select Tag

- The select tag creates an HTML select
- The property will be set to the selected item:

```
<form:form commandName="order"
     action="saveOrder.request">
   Product:
   <form:select path="product"
      items="${productList}"/>
   ...
</form:form>
```

2.30 option Tag

- Alternatively, use nested option tags instead of the items attribute:

```
<form:form commandName="order"
     action="saveOrder.request" >
   Product:
   <form:select path="product">
     <form:option value="BIKE" label="Bike"/>
     <form:option value="SOFTBALL"
         label="Softball"/>
     <form:option value="CLUB" label="Club"/>
   </form:select>
   ...
</form:form>
```

2.31 options Tag

- Alternatively, use nested options tag:

```
<form:form commandName="order"
     action="saveOrder.request">
   Product:
   <form:select path="product">
     <form:option value="-"
       label="--Select Product--"/>
     <form:options items="${productList}"
       itemValue="id" itemLabel="name"/>
   </form:select>
   ...
</form:form>
```

2.32 errors Tag

- The errors tag displays error messages in an HTML span tag
- E.g. Display field-level errors for the name property:

```
<form:form commandName="customer"
    action="saveCustomer.request">
  Name:
  <form:input path="name"/><br/>
  <form:errors path="name"/><br/>
  ...
</form:form>
```

- Display all errors by setting path="*"

2.33 Spring Boot Considerations

- Spring Boot uses an embedded web server
- Embedded web servers don't do well with JSP
- Spring Boot includes auto-configuration for
  - FreeMarker
  - Groovy
  - Thymeleaf
  - Velocity (although this is deprecated)
  - Mustache
- Put templates in 'src/main/resources/templates'

2.34 Summary

- Spring MVC provides a rich framework for web applications
- Spring 3 modified the definition of "Controllers" and "Handler Mapping" quite a bit with the use of annotations like @Controller and @RequestMapping
- The Spring MVC implementations of the Controller interface are deprecated and Spring MVC applications should use the new annotation-based approach
- The view resolution and Spring form tags are still the same as defined in Spring 2.x
- With Spring MVC many of the common use cases for applications are simple although you still have access to low-level API if needed

Chapter 3 - Overview of Spring Database Integration

Objectives

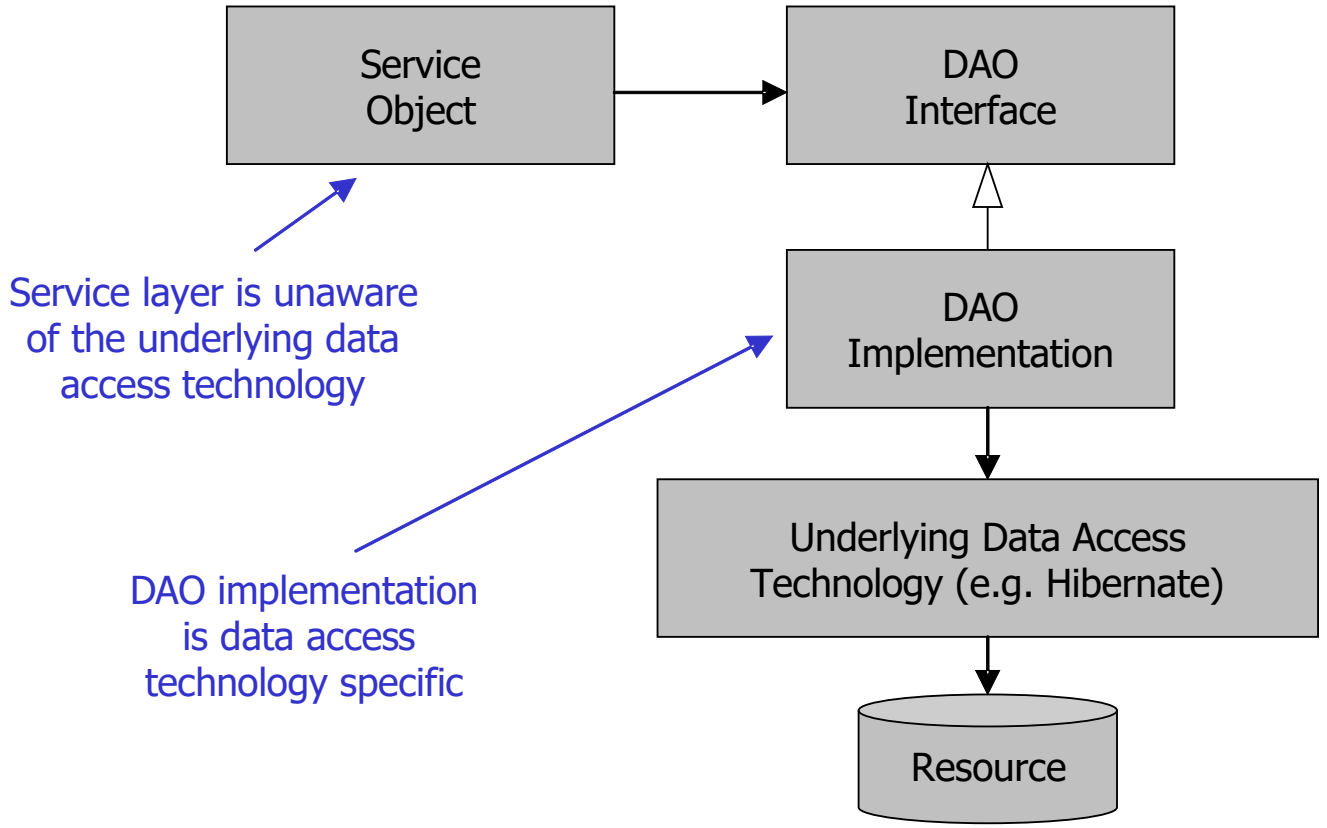
Key objectives of this chapter:

- DAO Support in Spring
- Various data access technologies supported by Spring

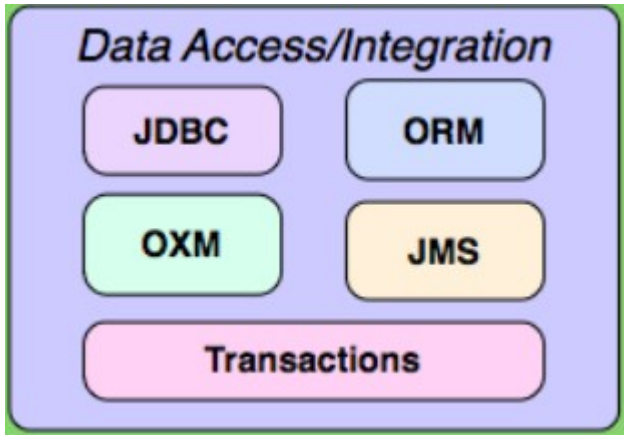
3.1 DAO Support in Spring

- A data access object (DAO) is a mechanism to read and write data from a database
- DAOs use underlying data access technologies such as JDBC or an ORM framework like Hibernate
- Spring DAO support is designed so it is easy to switch from one data access technology to another
  - E.g. JDBC, Hibernate, JDO, etc

3.2 DAO Support in Spring



3.3 Spring Data Access Modules



- Spring has 2 modules providing support for various data access options
  - JDBC module
  - ORM module
    - The ORM module depends on the JDBC module
- Both modules depend on the Spring Transaction module
  - Data access is transactional in nature but Spring supports this with a separate module

3.4 Spring JDBC Module

- The Spring JDBC module provides a framework and supporting classes for simplifying JDBC code
  - It is a good fit for applications that already use direct JDBC access and have not migrated to some Object Relational Mapping framework (ORM)
- The Spring JDBC module also has classes for DataSource access
  - This includes DataSource implementations that could be used in testing or when running an application outside a Java EE server
  - This also includes support for an embedded database which could be used in testing
  - The DataSource support from this module is used even when an application is using the Spring ORM module which is why the ORM module depends on the JDBC module
- Much of the classes from this module used directly in applications are some form of JdbcTemplate class

3.5 Spring ORM Module

- The Spring ORM module has supporting classes for the following ORM frameworks:
  - Hibernate
  - Java Persistence API (JPA)
    - This is the relatively new Java standard way to do persistence
  - Java Data Objects (JDO)
  - iBATIS SQL maps
- An application would typically use only one of these frameworks although the Spring ORM module contains support for all four
  - For applications not already using one of these frameworks the JPA standard is generally the default choice since it is a Java standard

3.6 Spring ORM Module

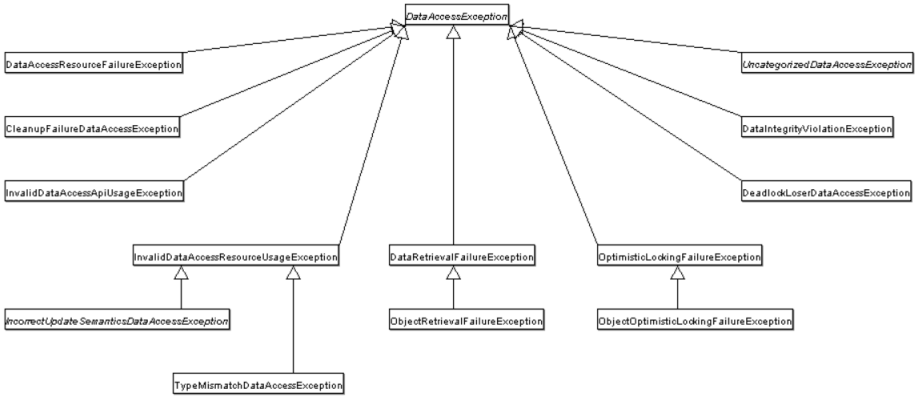
- Code written using one of the ORM frameworks supported by Spring typically uses the persistence framework directly
  - Spring provides supporting classes to integrate other Spring features like transactions with the ORM framework but these are often not seen in the code of the application itself
  - Spring configuration files is the area that largely contains the integration with the ORM framework
  - Spring has various XXXTemplate classes, like HibernateTemplate, but these generally should NOT be used as Spring 3 can properly integrate when the application uses the ORM API directly

3.7 DataAccessException

- The Spring DAO framework insulates higher layers from technology-specific exceptions
  - E.g. SQLException and HibernateException
- Instead, it throws DataAccessException
  - Root cause is still available using getCause()
- This "Exception translation" is one of the key benefits of Spring data access support
  - This keeps higher layers decoupled from lower level technologies
- DataAccessException is a RuntimeException
  - RuntimeExceptions are "unchecked" which means they do not need to be caught in order for the code to compile
  - This is important because it means that application code is not required to catch these Exceptions since nothing can often be done about them anyway
    - Why add a bunch of code to catch Exceptions when you aren't doing anything except ignoring the Exception?

3.8 DataAccessException

- Spring provides a hierarchy of DataAccessException subclasses representing different types of exceptions
  - E.g. DataRetrievalFailureException – Data could not be retrieved
- Spring converts data access technology errors to subclasses in this hierarchy
  - Spring understands some database specific error codes and ORM specific exceptions
- Spring exceptions are more descriptive



3.9 @Repository Annotation

- The DataAccessException translation described previously can easily be added to a DAO implementation by adding the @Repository annotation
- This annotation is one of the Spring "stereotype" annotations for declaring Spring components with an annotation
- This annotation is the only code that needs to be added to a class to enable the Exception translation functionality

```
@Repository
public class PurchaseDAOJDBCImpl implements PurchaseDAO {
```

- You would only need the 'component-scan' configuration in the Spring XML configuration to pick up this annotation as a Spring component

```
<context:component-scan base-package="com.webage" />
```

3.10 Using DataSources

- DAOs require DataSource objects in order to access the database
- DataSources are configured as beans
  - They are often then injected into Spring components to use or linked to other Spring configuration of JDBC or ORM components
- JNDI – Use jndi:lookup

```
<jndi:lookup id="dataSource"
      jndi-name="java:comp/env/jdbc/myDS"/>
```

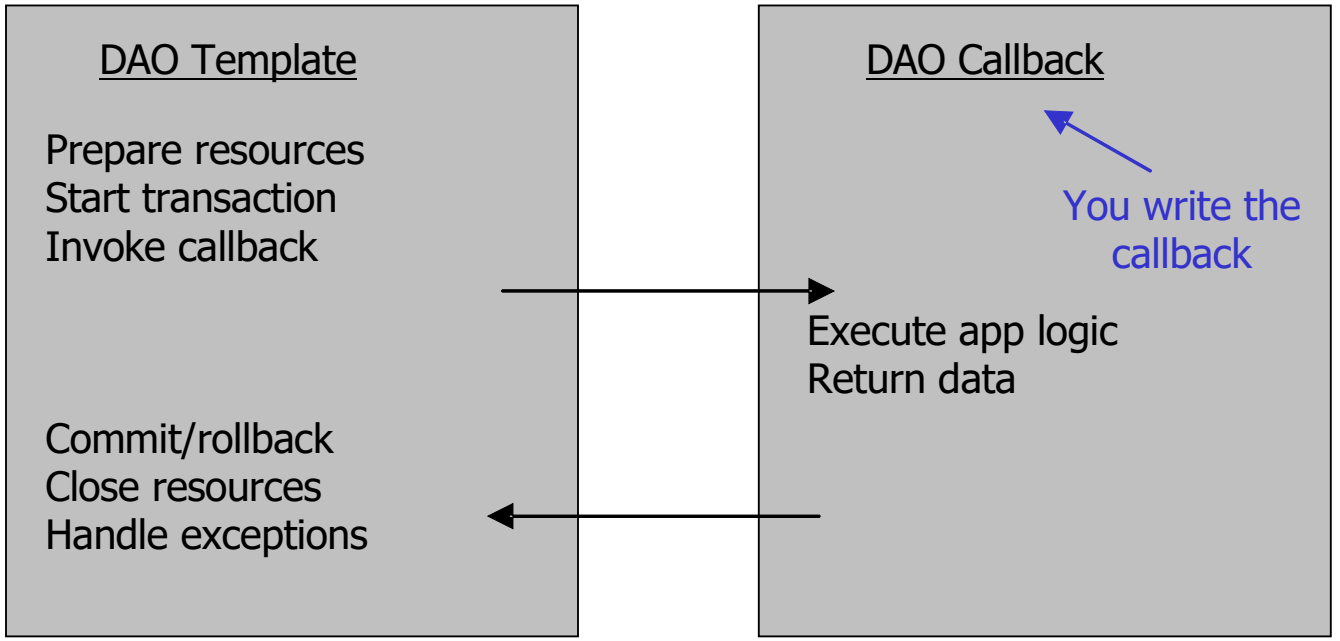
- The Spring DriverManagerDataSource is provided for testing

```
<bean id="dataSource" class=
"org.s..f.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName"
            value="${dataSource.driverClassName}" />
      <property name="url" value="${dataSource.url}" />
</bean>
```

3.11 DAO Templates

- Data access logic often consists mostly of infrastructural code
  - Resource management, exception handling, connection management, etc
- Spring uses the template method pattern to provide the infrastructural logic for you
- The developer focuses on creating the application-specific logic using a "DAO callback"

3.12 DAO Templates and Callbacks



3.13 ORM Tool Support in Spring

- ORM (object/relational mapping) frameworks provide functionality over JDBC such as:
  - Lazy loading, caching, cascading updates, etc
- Spring provides integration support for several ORM frameworks including:
  - Hibernate, JDO, iBATIS SQL Maps, Apache OJB
- Spring also provides services on top of these frameworks including:
  - Transaction management
  - Exception handling
  - Template classes
  - Resource management

3.14 Summary

- Spring provides modules to simplify data access code
  - These modules are some of the most used features of Spring
- No matter what technology is used Spring provides a common architecture so it is easy to switch data access technologies with minimum impact on the rest of an application
- The Spring @Repository annotation should be used on DAO components to enable Spring's data Exception translation

Chapter 4 - Using Spring with JPA or Hibernate

Objectives

Key objectives of this chapter:

- Using Spring with JPA
- Using Spring with Hibernate

4.1 Spring ORM

- Besides basic JDBC support, Spring also provides support for ORM, or Object Relational Mapping
  - ORM is the ability of mapping values of Java objects in memory to tables and columns in a database and having the environment automatically synchronize the two
- Spring provides an ORM module that provides this support

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>org.springframework.orm</artifactId>
</dependency>
```

- The ORM frameworks Spring supports are:
  - Java Persistence (JPA)
  - Hibernate
  - Java Data Objects (JDO)
- Although Spring does support the "template" approach like with JDBC, you are often using the ORM framework natively and using Spring just to bootstrap it into the environment

4.2 Benefits of Using Spring with ORM

- Simply using an ORM framework can simplify an application considerably
  - You can simply indicate what is persisted and let the framework calculate the actual SQL code to accomplish that
- Using Spring's ORM support in addition to an ORM framework provides the following benefits:
  - Spring can provide the configuration for the ORM framework as Spring configuration which can make it easy to swap out different configurations for testing, production, etc
  - Spring can translate the data access exceptions of the ORM framework into a standardized set of Spring exceptions
  - Spring can provide access to the ORM managed resources avoiding many common issues when used without Spring
  - The ORM support of Spring also integrates with the transactional support so that transactional behavior of a Spring application is consistent and the ORM framework is integrated into this correctly

4.3 Spring @Repository

- One of the Spring configuration stereotype annotations is specifically designed for Spring components in an application that work with data access
  - This is the @Repository annotation
- By using this annotation (and the appropriate annotation scanning XML configuration) you can register the Java class as a Spring component and enable the Spring data access exception translation
  - Java class:

```
@Repository
public class ProductDaoImpl implements ProductDao {
```

  - Spring configuration:

```
<context:component-scan ... />
<context:annotation-config />
```

4.4 Using JPA with Spring

- The use of JPA is based on the JPA 'EntityManager'
  - This is the interface used to persist or query data managed by JPA
  - Java classes that are mapped by JPA to database tables are 'Entities'

```
@Entity
public class Product { ...
```

- The EntityManager is often injected into a Spring component so that code in the rest of the Spring component can use JPA directly

```
public class PurchaseDAOJPAImpl implements PurchaseDAO {
      @PersistenceContext
      private EntityManager em;
```

  - The @PersistenceContext annotation is a standard JPA annotation that is used for injection provided the Spring <context:annotation-config /> element is in your Spring configuration
- Spring 3 can be used with JPA 2.0 as long as the JPA provider being used also supports JPA 2.0
  - Hibernate 3.5+ would support JPA 2.0

4.5 Configure Spring JPA EntityManagerFactory

- The primary support for JPA from Spring is in the configuration of a JPA 'EntityManagerFactory'
- There are three ways to configure this depending on the environment you are running the Spring application in
  - Obtain EntityManagerFactory from JNDI
    - This is done when the application runs on a Java EE 5 server that already provides support for JPA
  - LocalContainerEntityManagerFactoryBean
    - Uses Spring to fully manage and configure the JPA EntityManager providing the most options when in a web container like Tomcat or in a stand-alone Java application
  - LocalEntityManagerFactoryBean
    - Bootstraps JPA in a Java SE environment but doesn't allow configuration from Spring of the settings of the EntityManager

4.6 Using JNDI to Lookup JPA EntityManagerFactory

- If you are running a Spring application that uses JPA in a Java EE 5+ server the server will detect the JPA configuration (persistence.xml) and register the JPA EntityManager in JNDI
- The only thing you need to do in the Spring configuration is to use the JNDI lookup mechanism in Spring to register the proper JNDI name to lookup
  - This JNDI name may be different on different servers

```
<jee:jndi-lookup id="entityManagerFactory"
    jndi-name="jpa/BankAccountJPA"/>
```

- All of the configuration of the JPA EntityManager, like what DataSource it connects to, is in the JPA persistence.xml configuration using standard JPA syntax

```
<persistence ...>
<persistence-unit name="BankAccountJPA" transaction-type="JTA">
      <jta-data-source>java:jdbc/BankDS</jta-data-source>
      ... other configuration
</persistence-unit>
</persistence>
```

4.7 LocalContainerEntityManagerFactoryBean

- This configuration option is used in environments like Tomcat, where the server does not provide it's own JPA support or for stand-alone applications
- This is useful if full control over JPA configuration is needed and some of the JPA configuration will be linked to other Spring configuration elements
  - In this case the JPA persistence configuration (persistence.xml) will likely be minimal and more configuration is provided in Spring

```
<bean id="entityManagerFactory"
class="org...jpa.LocalContainerEntityManagerFactoryBean">
      <property name="jpaVendorAdapter">
            <bean
            class="org...jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="dataSource" ref="dataSource" />
      <property name="jpaProperties">
            <props>
                  ...
            </props>
      </property>
</bean>
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/OEData" />
```

4.8 LocalEntityManagerFactoryBean

- The final option is to use the Spring LocalEntityManagerFactoryBean to configure a JPA EntityManager factory
- Typically though this would only be used outside of a Java EE 5+ server and when your persistence.xml already contains all required configuration
  - LocalContainerEntityManagerFactoryBean provides much more flexibility and can be used in the same Tomcat or Java SE environments

```
<bean id="entityManagerFactory" class=
    "org.s-f.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName"
     value="myPersistenceUnit"/>
</bean>
```

4.9 Application JPA Code

- Although Spring is providing the JPA configuration the actual code that works with persisted data can use only the JPA API

```
public class PurchaseDAOJPAImpl implements PurchaseDAO {
	@PersistenceContext
	private EntityManager em;

	public void savePurchase(Purchase purchase) {
		if (getPurchase(purchase.getId()) == null) {
			em.persist(purchase);
		} else {
			em.merge(purchase);
		}
	}

	public List<Purchase> getAllPurchases() {
		Query q = em.createQuery("select p from Purchase p");
		List<Purchase> results = (List<Purchase>)
			q.getResultList();

		return results;
	}
```

4.10 Hibernate

- Hibernate is a popular full-featured open-source persistence framework
  - Spring 3.0 supports Hibernate 3.2 and later
- It uses XML config files to describe how objects are mapped to a relational database
  - Hibernate can also support Java annotations although if this approach is taken it is generally suggested to use the JPA standard approach instead
- Example persistent object:

```
public class Order {
  private Integer id;
  private String customerName;
  public Integer getId() {return id;}
  public void setId(Integer id) {this.id=id;}
  …
}
```

4.11 Hibernate

- Example Order.hbm.xml file:

```
<hibernate-mapping>
  <class name="com…Order" table="order">
    <id name="id" column="id">
      <generator class="assigned"/>
    </id>
    <property name="customerName"
      column="cust_name"/>
  </class>
</hibernate-mapping>
```

4.12 Hibernate Session Factory

- The Hibernate SessionFactory reads the mapping files and is used to get Hibernate Session objects
- Use the Session objects to access the database
- Example (without using Spring):

```
Session session = sessionFactory.openSession();
Order order=(Order)session.load(Order.class, id);
session.close();
```

4.13 Spring LocalSessionFactoryBean

- Use Springs LocalSessionFactoryBean to configure a Hibernate session factory

```
<bean id="mySessionFactory"
    class="org…LocalSessionFactoryBean">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
  <property name="mappingDirectoryLocations">
    <list>
      <value>classpath:/com/webage/order</value>
    </list>
  </property>
</bean>
```

4.14 Application Hibernate Code

- By using Spring XML configuration for Hibernate properties and injection of the Hibernate SessionFactory into the Spring component you could achieve the goal of not having ANY Spring API in your application code

```
public class ProductDaoImpl implements ProductDao {
      private SessionFactory sessionFactory;

      public void setSessionFactory(SessionFactory
            sessionFactory) {
                  this.sessionFactory = sessionFactory;
      }

      public Product findProductByID(int id) {
            return (Product)this.sessionFactory.
                  getCurrentSession().get(Product.class, id);
            }
}
```

  - With the following Spring configuration

```
<bean id="myProductDao" class="product.ProductDaoImpl">
      <property name="sessionFactory" ref="mySessionFactory"/>
</bean>
```

4.15 "Classic" Spring ORM Usage

- The preferred way to use ORM frameworks with Spring is described in the lecture up to this point
- Some older Spring applications though may still use the older ways of doing ORM which used more of the XXXTemplate and XXXCallback approach to using ORM
  - This style is described in the rest of the chapter
  - This style was also more similar to the JDBCTemplate approach of Spring JDBC so it seemed more "familiar" at the time
- The problem with this style is that it has Spring API classes used in the application code creating a dependence on Spring (besides for just configuration)
  - Since it is now possible to write fully functional ORM code using only the API of the ORM framework (JPA, Hibernate, etc) and use Spring only for the configuration that approach is preferred

4.16 Spring JpaTemplate

- Use Springs JpaTemplate to access JPA

```
<bean id="jpaTemplate"
    class="org…JpaTemplate">
  <property name="entityManagerFactory">
    <ref bean="entityManagerFactory"/>
  </property>
</bean>
…
<bean id="orderDao"
    class="com…OrderDaoJpaImpl">
  <property name="jpaTemplate">
    <ref bean="jpaTemplate"/>
  </property>
</bean>
```

4.17 Spring JpaCallback

- Use Springs JpaCallback to provide application specific logic

```
public List getOrders(final Integer prod) {
  return (List)jpaTemplate.execute(
    new JpaCallback() {
      public Object doInJpa(EntityManager mgr)
          throws PersistenceException {
        Query q = mgr.createQuery("from Order…");
        return q.execute(prod);
      }
    });
}
```

- Exception handling, resource management, etc is handled by JpaTemplate

4.18 JpaTemplate Convenience Features

- JpaTemplate has several convenience methods for common requests
  - E.g. find method (using JPA query language)

```
List list = jpaTemplate.find("from Order");
```

- Spring also provides the JpaDaoSupport convenience class
  - This base class provides get and setEntityManagerFactory and getJpaTemplate methods

4.19 Spring HibernateTemplate

- Use Springs HibernateTemplate to access Hibernate

```
<bean id="hibernateTemplate"
    class="org…HibernateTemplate">
  <property name="sessionFactory">
    <ref bean="sessionFactory"/>
  </property>
</bean>
…
<bean id="orderDao"
    class="com…OrderDaoHibernateImpl">
  <property name="hibernateTemplate">
    <ref bean="hibernateTemplate"/>
  </property>
</bean>
```

4.20 Spring HibernateCallback

- Use Springs HibernateCallback to provide application specific logic

```
public Order getOrder(final Integer id) {
  return (Order)hibernateTemplate.execute(
    new HibernateCallback() {
      public Object doInHibernate(Session session)
          throws HibernateException {
        return session.load(Order.class, id);
      }
    });
}
```

- Exception handling, resource management, etc is handled by HibernateTemplate

4.21 HibernateTemplate Convenience Methods

- HibernateTemplate has several convenience methods for common requests
- load method

```
Order order =   (Order)hibernateTemplate.load(Order.class, id);
```

- update method

```
hibernateTemplate.update(order);
```

- find method (using HQL)

```
List list = hibernateTemplate.find(
  "from Order order where order.cust_name = ?",
  custName, Hibernate.STRING);
```

4.22 Spring Boot Considerations

- Spring Boot will automatically configure an embedded database if there's one available on the classspath
  - H2
  - HSQL
  - Apache Derby
- At startup time, it runs the scripts 'schema.sql' and 'data.sql' from the resources root.
- Override by configuring 'spring.datasource.*'

4.23 Spring Data JPA Repositories

- Spring Data JPA repositories are interfaces that you define to access data
- Spring Data reads the name of the methods on your interface and automatically creates queries to implement them

```
public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

- e.g. 'findByNameAndCountryAllIgnoringCase' provides enough information (along with the type definitions) to synthesize a query

4.24 Summary

- Spring provides exceptional support for various ORM frameworks
- The preferred approach in modern Spring applications is:
  - Use the JPA API to create standards-based persistence code
  - Use a JPA provider library (like Hibernate) to provide the JPA support
  - Use Spring to configure JPA and link to other Spring configuration or the environment in a Java EE 5+ server
- Direct use of the Hibernate API in application code is still possible

Chapter 5 - Spring REST Services

Objectives

Key objectives of this chapter

- A Basic introduction to REST-style services
- Using Spring MVC to implement REST services
- Combining the JAX-RS standard with Spring

5.1 Many Flavors of Services

- Web Services come in all shapes and sizes
  - XML-based services (SOAP, XML-RPC, RSS / ATOM Feeds)
  - HTTP services (REST, JSON, standard GET / POST)
  - Other services (FTP, SMTP)
- While SOAP is the most common style of service, increasingly organizations are utilizing REST for certain scenarios
  - REpresentational State Transfer (REST), first introduced by Roy Fielding (co-founder of the Apache Software Foundation and co-author of HTTP and URI RFCs)
  - REST emphasizes the importance of resources, expressed as URIs
  - Used extensively by Amazon, Google, Yahoo, Flickr, and others

5.2 Understanding REST

- REST applies the traditional, well-known architecture of the Web to Web Services
  - Everything is a resource
  - Each URI is treated as a distinct resource and is addressable and accessible using an application or Web browser
  - URIs can be bookmarked and even cached
- Leverages HTTP for working with resources
  - GET – Retrieve a representation of a resource. Does not modify the server state. A GET should have no side effects on the server side.
  - DELETE – Remove a representation of a resource
  - POST – Create or update a representation of a resource
  - PUT – Update a representation of a resource

5.3 RESTful Services

- A RESTful Web service services as the interface to one or more resource collections.
- There are three essential elements to any RESTful service
  - Resource Address – expressed as a URI
  - Representation Format – a known MIME type such as TXT or XML, common data formats include JSON, RSS / ATOM, and plain text
  - Resource Operations – a list of supported HTTP methods (GET, POST, PUT, DELETE)

5.4 REST Resource Examples

- GET /checkflightstatus/AA1215 HTTP/1.1
  - Check the flight status for American Airlines flight #1215
- POST /checkflightstatus HTTP/1.1
  - Upload a new flight status by sending an XML document that conforms to a previously defined XML Schema
  - Response is a “201 Created” and a new URI

```
201 Created
Content-Location: /checkflightstatus/AA1215
```

- PUT /checkflightstatus/AA1215 HTTP/1.1
  - Update an existing resource representation
- DELETE /checkflightstatus/AA1215 HTTP/1.1
  - Delete the resource representation

5.5 REST vs SOAP

- REST
  - Ideal for use in Web-centric environments, especially as a part of Web Oriented Architecture (WOA) and Web 2.0
  - Takes advantage of existing HTTP tools, techniques, & skills
  - Little standardization and general lack of support regarding enterprise-grade demands (security, transactions, etc.)
- SOAP
  - Supports robust and standardized security, policy management, addressing, transactions, etc.
  - Tools and industry best practices for SOA and WS assume SOAP as the message protocol

5.6 REST Services With Spring MVC

- The Spring MVC web framework provides support for implementing REST services
  - The core of this support is provided by the @RequestMapping annotation within Spring MVC @Controller classes
- Unlike other Spring MVC web applications you do not use "View Handlers" as the return type from a Spring MVC handler method is often used as the body of the response directly

5.7 Spring MVC @RequestMapping with REST

- The primary support for REST services in Spring MVC is the @RequestMapping annotation in a @Controller class
- The annotation can be used at the class level and method level
  - If used both places the settings are combined
- The primary attribute used is the 'value' which becomes the path that the handler method is mapped to

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {
```

- It is also very common in REST services to use the 'method' attribute to map a particular method to a specific HTTP method

```
@RequestMapping(value="/appointments",
      method = RequestMethod.GET)
public AppointmentList getAllAppointments() {
```

- It is also very common to use the @PathVariable annotation to get data from the request URL directly

```
@RequestMapping(value="/owners/{ownerId}",
      method=RequestMethod.GET)
public PetList findAllPets(@PathVariable int ownerId) {
```

5.8 Working With the Request Body and Response Body

- With REST services it is common to have the entire request body processed as incoming data or to return data as the body of the response
- With Spring MVC you can use the @RequestBody and @ResponseBody annotations on method parameters or the method return type
  - The @ResponseBody annotation is easiest to have on the method since a method can only have one return type
  - The classes used as method parameters and return types should be used with JAXB to allow conversion between Java and XML

```
@RequestMapping(method=RequestMethod.POST)
@ResponseBody
public QuoteResponse getQuote(@RequestBody QuoteRequest request) {
```

5.9 @RestController Annotation

- Spring 4 added the @RestController annotation
- It combines @Controller, @ResponseBody, @RequestBody
- So now you just need one annotation on a REST controller class.

5.10 Implementing JAX-RS Services and Spring

- JAX-RS is the official Java specification for defining REST services in Java
  - Similar to Spring MVC this is done mainly with JAX-RS annotations although these annotations are different
- Spring MVC is NOT a JAX-RS implementation
- It is possible though to use standard JAX-RS code and implementation to define the REST service itself and simply inject Spring components into the JAX-RS service class
  - This would provide for more "standard" code for the REST service but allow the other features of Spring to be used as well
- The best way to do this would be to use the SpringBeanAutowiringSupport class from within a @PostConstruct method of the JAX-RS class which contains @Autowired Spring components

```
public class QuoteService {
@Autowired
private QuoteGenerator generator;
@PostConstruct
public void initSpringComponents() {
      SpringBeanAutowiringSupport.
            processInjectionBasedOnCurrentContext(this);
}
```

5.11 JAX-RS Annotations

- JAX-RS uses different annotations from Spring MVC
- @Path annotation for linking classes or methods to the request path that will map to them

```
@Path("/quotes")
public class QuoteService {
```

- Separate annotations for mapping to the various HTTP methods

```
@GET  @POST @PUT  @DELETE
```

- You can use a @PathParam annotation with a method parameter to extract data from the URL

```
@Path("/thisResource/{resourceId}")
public String getResourceById(@PathParam("resourceId") String id)
{ … }
```

5.12 Java Clients Using RestTemplate

- The most common types of clients for REST services are JavaScript and AJAX clients
- It is perhaps common also to need to have Java code communicate with REST services as a client
- Although REST service requests and responses are fairly simple using basic Java APIs like the java.net package would be difficult and low-level
- JAX-RS 1.x does not provide a client API although JAX-RS 2.0 will
- Spring provides the RestTemplate class which has a number of convenience methods for sending requests to REST services
  - These REST services do not need to be implemented using Spring MVC or JAX-RS
- To use the RestTemplate class in a Java client you would still need to include the Spring MVC module in the application

5.13 RestTemplate Methods

- There are many different methods and these are generally provided based on the HTTP method that will be sent
  - The parameters and return type differ depending on if the request body will contain data from an object and if the response body will contain data coming back
  - All take a String for URL and a variable argument Object... parameter for parameters in the URL
- DELETE methods are probably the simplest

```
void delete(String url, Object... urlVariables)
```

- GET takes no request body but returns an object for the response body

```
<T> getForObject(String url, Class<T> responseType, Object...
urlVariables)
```

- PUT takes a request body but returns nothing

```
void put(String url, Object request, Object... urlVariables)
```

- POST takes a request body and can return a response body

```
<T> postForObject(String url, Object request, Class<T> responseType,
Object... uriVariables)
```

5.14 Summary

- Spring MVC provides basic support for implementing REST services
- Spring could also be used behind standard JAX-RS REST services

Chapter 6 - Spring Security

Objectives

Key objectives of this chapter

- Overview of Spring Security
- Configuring Spring Security
- Defining security restrictions for an application
- Customizing Spring Security form login, logout, and HTTPS redirection
- Using various authentication sources for user information

6.1 Securing Web Applications with Spring Security 3.0

- Spring Security (formerly known as Acegi) is a framework extending the traditional JEE Java Authentication and Authorization Service (JAAS)
- It can work by itself on top of any Servlet-based technology
  - It does however continue to use Spring core to configure itself
- It can integrate with many back-end technologies
  - Support for OpenID, CAS, LDAP, Database
- It uses a servlet-filter to control access to all Web requests
- It can also integrate with AOP to filter method access
  - This gives you method-level security without having to actually use EJB

6.2 Spring Security 3.0

- Because it is based on a servlet-filter, it can also work with SOAP based Web Services, RESTful Services, any kind of Web Remoting, and Portlets
- It can even be integrated with non-Spring web frameworks such as Struts, Seam, and ColdFusion
- Single Sign On (SSO) can be integrated through CAS, the Central Authentication Service from JA-SIG
  - This gives us access to authenticate against X.509 Certificates, OpenID (supported by Google, Facebook, Yahoo, and many others), and LDAP
  - WS-Security and WS-Trust are built on top of these
- It can integrate into WebFlow
- There's support for it in SpringSource Tool Suite

6.3 Authentication and Authorization

- Authentication answers the question “Who are you?”
  - Includes a User Registry of known user credentials
  - Includes an Authentication Mechanism for comparing the user credentials with the User Registry
  - Spring Security can be configured to authenticate users using various means or to accept the authentication that has been done by an external mechanism
- Authorization answers the question “What can you do?”
  - Once a valid user has been identified, a decision can be made about allowing the user to perform the requested function
  - Spring Security can handle the authorization decision
    - Sometimes this may be very fine-grained. For example, allowing a user to delete their own data but not the data of other users

6.4 Programmatic v Declarative Security

- Programmatic security allows us to make fine grained security decisions but requires writing the security code within our application
  - The security rules being applied may be obscured by the code being used to enforce them
- Whenever possible, we would prefer to declare the rules for access and have a framework like Spring Security enforce those rules
  - This allows us to focus on the security rules themselves and not writing the code to implement them
- With Spring Security we have a DSL for security that enables us to declare the kinds of rules we would have had to code before
  - It also enables us to use EL in our declarations which gives us a lot of flexibility
  - This can include contextual information like time of access, number of items in a shopping cart, number of previous orders, etc.

6.5 Getting Spring Security from Maven

- Spring 3.0 split many different packages into different modules available from Maven so you can use just what you need
- The following will almost always be used
  - Core – Core classes
  - Config – XML namespace configuration
  - Web – filters and web-security infrastructure
- The following will be used if the appropriate features are required
  - JSP Taglibs
  - LDAP – LDAP authentication and provisioning
  - ACL – Specialized domain object ACL implementation
  - CAS – Support for JA-SIG.org Central Authentication Support
  - OpenID – 'OpenID for Java' web authentication support

6.6 Spring Security Configuration

- Spring Security is configured as a Servlet filter in the web.xml of the application
  - It is important that the name of the filter is '**springSecurityFilterChain**' as this will match up with internal Spring beans used by Spring Security

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- You would also normally configure the name of Spring configuration files in web.xml with the Spring ContextLoaderListener
  - This is done just like other Spring web applications

6.7 Spring Security Settings in Spring Configuration

- Spring Security is configured in a Spring configuration file
- Technically there are two ways to configure Spring Security
  - Manually create Spring <bean> definitions and know what Spring Security classes to link them to and the names of properties to set
  - Use the Spring Security XML namespace in your configuration
- The second option is MUCH easier because:
  - There is much less chance of errors
  - XML editors can use code completion for the XML elements available and the attributes you can set on each
  - The syntax is much more concise
  - You can concentrate on the settings without needing to know the details of which Spring Security class implements that feature
- To use the Spring Security XML namespace in your configuration file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:security=
"http://www.springframework.org/schema/security" ...
```

6.8 Basic Web Security Configuration

- The basic element of the Spring Security web configuration is <http>
  - This element has an 'auto-config' attribute that when set to true does a number of things
    - Enables generation of a form for form-based login
    - Enables a logout service
    - Enables HTTP basic authentication (browser dialog prompt) although the form-based will still be the default
- You would also generally add a child <intercept-url> element to indicate a URL that should be restricted to a list of user roles

```
<http auto-config='true'>
      <intercept-url pattern="/**" access="ROLE_USER" />
</http>
```

- The above configuration would restrict every page in the application to the "USER" role defined in the separately configured "authentication manager"

6.9 Granting Anonymous Access to Pages and Resources

- For users that have not logged in with a unique user you can setup anonymous access
- The access attribute value of 'IS_AUTHENTICATED_ANONYMOUSLY' on an <intercept-url> element indicates the user does not need to be authenticated

```
<intercept-url pattern="/login.jsp*"
access="IS_AUTHENTICATED_ANONYMOUSLY"/>
```

- Anonymous authentication is configured automatically with the <http> configuration but you can customize it with the <anonymous> element
  - **username** – change the username associated with the "anonymous" login. The default is 'anonymousUser'
  - **enabled** – The default is 'true' so you generally only use this attribute with a value of 'false' to disable anonymous login
  - **granted-authority** – Defaults to 'ROLE_ANONYMOUS' which could be used directly in access lists or you can customize the value

```
<anonymous username="guest" enabled="true"
      granted-authority="ROLE_GUEST" />
```

- You can also exclude resources from the Spring Security filters altogether although this should not be used for resources that might require other Security features

```
<intercept-url pattern="/css/**" filters="none"/>
```

6.10 Requiring Encrypted HTTPS Communication

- It is a common security requirement that certain pages are only sent to the user's browser over encrypted HTTPS channels
  - The minimum is usually the login process although other pages could be restricted to HTTPS as well for things like submitting credit cards
- To do this with Spring Security you add the 'requires-channel' attribute to the <intercept-url> element
- The values can be 'https', 'http', or 'any' although 'https' is the one you will likely use most

```
<intercept-url ... requires-channel="https" />
```

- Since Spring Security may be redirecting HTTP requests to use HTTPS you may need to provide the values of these ports for your server using the <port-mappings> element

```
<http>
...
      <port-mappings>
            <port-mapping http="9080" https="9443"/>
      </port-mappings>
</http>
```

6.11 Customizing Form-based Login

- Although Spring Security can automatically generate a login form it is often desired to provide a custom form that matches the rest of the application
- You can customize the <form-login> element to control this
  - **login-page** – the URL of the custom login page. If this value is not set the generated page will use a URL of '/spring_security_login'
  - **authentication-failure-url** – The URL to redirect to if authentication fails. If not set the value '/spring_security_login?login_error' is used
  - **default-target-url** – The redirect URL after successful authentication if a previous request for a secure URL had not already been submitted. The default is '/' which will redirect to the root of the application
  - **always-use-default-target** – Always redirect to the default target
  - **login-processing-url** – The URL a custom login form should submit to process the login. Defaults to '/j_spring_security_check'

```
<http> ...
      <form-login login-page="/login.jsp"
      default-target-url="/welcome"
      authentication-failure-url="/login.jsp?error=true"/>
</http>
```

6.12 Custom Login Page

- If you supply the 'login-page' attribute of the <form-login> element the page you supply must have a few elements
  - POST to '/j_spring_security_check' or a custom URL set with the 'login-processing-url' attribute
  - Username parameter of 'j_username'
  - Password parameter of 'j_password'
  - Display authentication failure messages

```
<html><body> ...
<c:if test="${not empty param.error}">
      <font color="red">Error:
      <c:out value=
      "${SPRING_SECURITY_LAST_EXCEPTION.message}" />.
      </font>
</c:if>
<form action="j_spring_security_check" method="POST">
User Name: <input type="text" name="j_username"><br />
Password: <input type="password" name="j_password"><br />
<input type="submit" value="LogIn"><br />
</form></body></html>
```

6.13 Configure Logout

- You can also customize the logout service by using the <logout> element and various attributes
  - **logout-url** – This is the URL that will cause the user to logout. This defaults to '/j_spring_security_logout' and would be used for a "logout" link on the pages of the application

```
<a href="<c:url value="/logout" />">Logout</a>
```

  - **logout-success-url** – The URL to redirect to after logout. Defaults to '/'
  - **invalidate-session** – Indicates if the HttpSession should also be invalidated upon logout. The default is 'true' so this is generally only set if you need to disable this for some reason

```
<http> ...
      <logout logout-success-url="/welcome"
      logout-url="/logout" />
</http>
```

6.14 Session Management

- Spring can control a few situations with the user session
  - This is done with properties of the <session-management> element
- The 'invalid-session-url' will indicate what URL to redirect to when a user session times out
- The <concurrency-control> child element can be used to prevent concurrent logins for the same user (from different computers or browsers)

```
<http> ...
      <session-management
      invalid-session-url="/sessionTimeout.htm" >
            <concurrency-control max-sessions="1"
            error-if-maximum-exceeded="true" />
      </session-management>
</http>
```

- The concurrency control requires a Servlet listener in the web.xml

```
<listener>
<listener-class>
org.s-f.security.web.session.HttpSessionEventPublisher
</listener-class>
</listener>
```

6.15 Selectively Display Links in a JSP

- Spring Security has a module that will let you add custom JSP tags to leverage some of the features of Spring Security for JSP pages
- After adding the JSP Taglibs module to the project add the following to a page

```
<%@ taglib prefix="sec"
   uri="http://www.springframework.org/security/tags" %>
```

- The most common tag is the <authorize> tag which will display the content of the tag if the user has permissions determined by various attributes
  - **url** – This can be the URL that will be tested to see if the user has access

```
<sec:authorize url="/admin">
<a href="/admin”>Administration</a>
</sec:authorize>
```

  - **access** – This can be a security expression that will check various rules

```
<sec:authorize access="hasRole('supervisor')">
```

    - To use these expressions you must also enable them in the <http>

```
<http use-expressions="true">
```

- You can also use the <authentication> tag to get access to details of the authenticated user or the <accesscontrollist> tag to work with Spring Security ACL for domain objects

6.16 Method Level Security

- Spring Security has various ways of protecting methods called in Java code
- AOP Pointcuts using Spring AOP pointcut expressions

```
<global-method-security>
   <protect-pointcut expression=
   "execution(* com.mycompany.*Service.*(..))"
   access="ROLE_USER"/>
</global-method-security>
```

- Annotations on the methods or classes to be protected
  - JSR 250 annotations like @RolesAllowed("ADMIN") can be enabled by

```
<global-method-security jsr250-annotations="enabled" />
```

  - Spring @Secured("ROLE_ADMIN") can be enabled by

```
<global-method-security secured-annotations="enabled" />
```

  - Spring @PreAuthorize("hasRole('ROLE_ADMIN')") which can use security expressions can be enabled by

```
<global-method-security pre-post-annotations="enabled" />
```

- The <global-method-security> element is a separate element in the security configuration and is not a child of something like <http>

6.17 Authentication Manager

- The <http> and <global-method-security> elements simply define what the security restrictions are
  - They do not define the users, their passwords, and what authorities they are granted
- The <authentication-manager> element and the child <authentication-provider> element(s) provide user information
  - You can use multiple <authentication-provider> elements and they will be checked in the declared order to authenticate the user
- One way to define an <authentication-provider> is to configure a list of users directly in the security configuration

```
<authentication-manager>
<authentication-provider>
<user-service>
      <user name="jsmith" password="john"
       authorities="ROLE_USER, ROLE_ADMIN" />
      <user name="jdoe" password="jane"
       authorities="ROLE_USER" />
</user-service>
</authentication-manager>
</authentication-provider>
```

6.18 Using Database User Authentication

- You can obtain user details from tables in a database with the <jdbc-user-service> element
  - This will need a reference to a Spring Data Source bean configuration

```
<authentication-manager>
<authentication-provider>
<jdbc-user-service data-source-ref="securityDatabase" />
</authentication-provider>
</authentication-manager>
```

- If you do not want to use the database schema expected by Spring Security you can customize the queries used and map the information in your own database to what Spring Security expects

```
<jdbc-user-service data-source-ref="securityDatabase"
      users-by-username-query="SELECT username, password,
      'true' as enabled FROM member WHERE username=?"
      authorities-by-username-query="SELECT
      member.username, member_role.role as authority
      FROM member, member_role WHERE member.username=?
      AND member.id=member_role.member_id"
/>
```

6.19 LDAP Authentication

- It is common to have an LDAP server that stores user data for an entire organization
- The first step in using this with Spring Security is to configure how Spring Security will connect to the LDAP server with the <ldap-server> element

```
<ldap-server id="ldapServer" url="ldap://localhost:389"
manager-dn="cn=Directory Admin" manager-password="ldap" />
```

- You can also use a "embedded" LDAP server in a test environment by not providing the 'url' attribute and instead providing ldif files to load

```
<ldap-server id="ldapServer" root="dc=mycompany,dc=com"
ldif="classpath:users.ldif" />
```

- You would then configure a <ldap-user-service> as a child of <authentication-provider> with a reference to the server configuration and can customize how the LDAP searches are performed

```
<authentication-provider>
<ldap-user-service server-ref="ldapServer"
 user-search-filter="(uid={0})"
 user-search-base="ou=people"
 group-search-filter="member={0}"
 group-search-base="ou=groups" />
</authentication-provider>
```

6.20 Encoding Passwords

- Storing passwords in clear text can still be a security risk
  - Encoding or "masking" the passwords can provide extra security
- A <password-encoded> element can be part of an <authentication-provider> definition
  - The **hash** attribute indicates the hashing mechanism and can contain values like 'md5' and 'sha' which are two common mechanisms

```
<authentication-provider>
      <password-encoder hash="md5" />
      ... user-service configuration
</authentication-provider>
```

- To make the encoding stronger you can use a <salt-source> child element to indicate a 'user-property' of the UserDetails object or a 'system-wide' value that will be used as part of the encoding mechanism

```
<password-encoder hash="sha">
      <salt-source user-property="username"/>
</password-encoder>
```

6.21 Using an External Authentication Provider

- Although the default for Spring Security is to perform authentication and authorization it is possible to work with an authenticated identity provided by an external system
  - This is often the case when there is an established standard for authenticating users and this is then passed to the application
- To do this in Spring Security you use some configuration of the "pre-authentication" part of Spring Security
  - The goal of this configuration is to turn the authentication information provided by the external system into user information recognized by Spring Security
- There are several common ways to implement this pre-authorization with various samples from Spring Security downloads
  - Getting user details from HTTP request headers using the RequestHeaderAuthenticationFilter
    - This is common with Siteminder
  - The J2eePreAuthenticatedProcessingFilter will extract the userPrincipal property of the request set by a Java Enterprise server authentication
  - Using X.509 client certificate authentication

6.22 Summary

- Spring Security has many features that simplify securing web applications
- Making use of many of these features only requires configuration in a Spring configuration file
- Spring Security can work with many different sources of user and permission information

Chapter 7 - Spring JMS

Objectives

Key objectives of this chapter

- Spring JMS
- JmsTemplate
- Async Message Receiver, Message-Driven POJOs(MDPs)
- Transaction management
- Producer Configuration
- Consumer Configuration
- Message-Driven POJO's Async receiver Configuration

7.1 Spring JMS

- Spring provides a Template based solution (JmsTemplate class) to simplify the JMS API code development
- JmsTemplate class can be used for sending messages and synchronously receive messages
- For Asynchronous message receiving, just like JEE MDBS, Spring uses Message-Driven POJOs (MDPs)
- Spring 3.x supports JMS 1.1 API, JMS 1.0.2 is deprecated
- JMS 1.1 provides domain-independent API, that means messages can be sent and received to/from different model Queue or Topic using the same Session object
- For different JMS API, Spring provide two sets of template classes, JmsTemplate102 and JmsTemplate for JMS1.0.2 and JMS1.1 respectively
- This template converts JMSException into org.springframework.jms.JmsException

7.2 JmsTemplate

- The JmsTemplate class helps to obtain and release the JMS resources
- To send messages, simply call send method which take two parameters, destination and MessageCreator
- MessageCreator object is usually implemented as an anonymous class
- MessageCreator interface has only one method createMessage() to be implemented.
- JmsTemplate objects are thread-safe and keeps the reference to ConnectionFactory
- Define template on the configuration file

```
<bean id="jmsTemplate"
        class="org. springframework. jms.core. JmsTemplate">
        <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

7.3 Connection and Destination

- Spring provided SingleConnectionFactory which is a implementation of JMS ConnectionFactory
- Calls to createConnection() on ConnectionFactory return a Connection object
- JmsTemplate can be configured with a default destination via the property defaultDestination
  - The default destination can be used on JmsTemplate with send and receive operations that do not refer to a specific destination

```
<bean id="jmsTemplate"
   class="org.(sf).jms.core.JmsTemplate">
   …
   <property name="connectionFactory"
      ref="connectionFactory" />
   <property name="defaultDestination"
      ref="orderDestination" />
</bean>
```

7.4 JmsTemplate Configuration

- Declare JmsTemplate, ConnectionFactory, and Destinations
  - The ConnectionFactory and Destination may depend on the JMS implementation of the environment
  - Below is shown using ActiveMQ as a JMS implementation in Tomcat
    - In a Java EE Environment you wouldlikely use <jee:jndi-lookup> to link Spring beans to JNDI lookups

```
<bean id="connectionFactory" class=
"org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
       value="tcp://localhost:61616" />
</bean>
<bean id="orderDestination"
     class="org.apache.activemq.command.ActiveMQQueue">
     <constructor-arg value="order.queue" />
</bean>
<bean id="jmsTemplate" class="org.sf.jms.core.JmsTemplate">
    <property name="connectionFactory"
       ref="connectionFactory" />
    <property name="defaultDestination"
      ref="orderDestination" />
</bean>
```

7.5 Transaction Management

- JmsTransactionManager provides support for managing transactions on a single ConnectionFactory
- Add the followings to bean configuration file (see the example below)
  - <tx:annotation-driven/>
  - JmsTransactionManager
- JmsTemplate automatically detects transaction
- JmsTemplate can be used with JtaTransactionManager (XA) for distributed transactions

7.6 Example Transaction Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
    ...>
   <tx:annotation-driven/>
    <bean id="transactionManager" class=
"org.springframework.jms.connection.JmsTransactionManager">
        <property name="connectionFactory">
            <ref bean="connectionFactory" />
        </property>
    </bean>
</beans>
```

7.7 Producer Example

- The following example illustrates how to call the send method on the JmsTemplate

```
@Component
public class OrderProducer {
   @Autowired private JmsTemplate jmsTemplate;

   public void addOrder(final Order order) {
     jmsTemplate.send(new MessageCreator() {
       public Message createMessage(Session session)
           throws JMSException {
         MapMessage m = session.createMapMessage();
         m.setLong("id", order.getId());
         m.setString("custName", order.getCustName());
         …
         return m;
       }
     });
   }
}
```

7.8 Consumer Example

- Call receive method on the JmsTemplate

```
@Component
public class OrderConsumer {
    @Autowired private JmsTemplate jmsTemplate;

    public Order receiveOrder() {
      MapMessage m = (MapMessage)jmsTemplate.receive();
      try {
        Order order = new Order();
        order.setId(m.getLong("id"));
        order.setCustName(m.getString("custName"));
        …
        return order;
      }
      catch (JMSException e) {
        throw JmsUtils.convertJmsAccessException(e);
      }
    }
}
```

7.9 Converting Messages

- You can use a MessageConverter to encapsulate the logic to convert messages to domain objects

```
public class OrderConverter implements MessageConverter {
  public Object fromMessage(Message m) {
    MapMessage mm = (MapMessage)m;
    Order order = new Order();
    try {
      order.setId(mm.getLong("id"));
      …
    }
    catch (JMSException e) {
      throw new
        MessageConversionException(e.getMessage());
    }
    return order;
  }
```

7.10 Converting Messages

- And convert domain objects to messages

```
  …
  public Message toMessage(Object object,
      Session session) throws JMSException {
    Order order = (Order)object;
    MapMessage m = session.createMapMessage();
    m.setLong("id", order.getId());
    m.setString("custName", order.getCustName());
    …
    return m;
  }
}
```

7.11 Converting Messages

- Configure JmsTemplate and MessageConverter

```
<bean id="jmsTemplate" class="org…JmsTemplate">
  …
  <property name="messageConverter">
    <ref bean="messageConverter"/>
  </property>
</bean>
<bean id="messageConverter" class="com…OrderConverter"/>
```

- The addOrder method is much simpler now

```
public void addOrder(Order order) {
  jmsTemplate.convertAndSend(order);
}
```

- The receiveOrder method is also simpler

```
public Order receiveOrder() {
  return (Order)jmsTemplate.receiveAndConvert();
}
```

7.12 Message Listener Containers

- Spring provides message-driven POJOs (MDPs) similar to Message-Driven Bean in EJB container, to receive messages
- A message listener container can be used to receive messages from a JMS message queue and drive the MessageListener that is injected into it.
  - A message listener container is the intermediary between a MDP and a messaging provider
  - This container is responsible for all threading of message reception and dispatches into the listener for processing.
- Spring provides two standard JMS message listener containers
  - SimpleMessageListenerContainer
    - Creates a fixed number of JMS sessions and consumers at startup, registers the listener using the standard JMS MessageConsumer.setMessageListener() method
    - JMS provider should perform listener callbacks
    - Does not support managed Transaction environment, like Java EE
  - DefaultMessageListenerContainer
    - It supports managed Transaction environment, like Java EE
    - Received messages are registered with XA transaction

7.13 Message-Driven POJO's Async Receiver Example

- Create a message listener class using MessageListener interface

```
public class OrderListener implements MessageListener {
    public void onMessage(Message message) {
        MapMessage msg = (MapMessage) message;
        try {
            Orderinfo info = new OrderInfo();
            info. setOrderID(msg.getString("orderID"));
            info.setDate(msg.getString("orderDate"));
            info.setDeliveryDate(msg.
             getDouble("deliveryDate"));
            displayOrderInfo(info);
        } catch (JMSException e) {
            throw JmsUtils.convertJmsAccessException(e);
        }
    }
    private void displayOrderInfo(Orderinfo info) {
        System. out. println("OrderID" + info.getOrderID()
            " received and to be shipped on "  +
            info.getDeliveryDate);
    }
}
```

7.14 Message-Driven POJO's Async Receiver Configuration

```
<bean id="connectionFactory" class=
      "org.apache.activemq.ActiveMQConnectionFactory">
   <property name="brokerURL"
      value="tcp://localhost:61616" />
</bean>
<bean id="orderListener"
      class="com.simple.OrderListener" />
<bean class=
"org.(sf).jms.listener.DefaultMessageListenerContainer">
   <property name="connectionFactory"
      ref="connectionFactory" />
   <property name="destinationName" value="order.queue" />
   <property name="messageListener" ref="orderListener" />
</bean>
```

7.15 Spring Boot Considerations

- Spring Boot will automatically configure an 'AmqpTemplate' and 'AmqpAdmin' if you use the 'spring-boot-starter-amqp' starter
- Support for RabbitMQ is built-in
- Configure using application properties:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

- To setup a listener, simply annotate a method with '@RabbitListener'

```
@RabbitListener(queues = "someQueue")
public void processMessage(String content) {
    // ...
}
```

7.16 Summary

- JMS support in Spring, how to use Spring to build message-oriented architectures.
- You learned how to configure both producer and consumer messages using a message queue.
- You learned how to build message-driven POJOs.