Labs

Wiring Dependencies using XML

In this lab you will learn to use the Spring container to inject dependencies. Dependency Injection can be configured using XML or annotations. In this lab you will use XML.

Step 1 - Creating a project
The labs are available in Mercurial. Open a terminal and enter the following commands to open the project.

```
hg clone http://bitbucket.org/paulbakker/spring
cd spring
hg update 0
```

Open STS, choose a workspace location, and import the project as an existing Maven project. You should now have two project: lab1 and parent.

All Spring dependencies are included in the POM file of the parent project. This is the preferred way to work with Spring dependencies, so explore the POM file to understand how the project is configured.

First add a Spring Bean Configuration File using the context menu in STS. Name the configuration file “applicationContext.xml”. Add a new class with a main method. In the main method you will bootstrap the Spring container from code. To bootstrap Spring you’ll have to create an ApplicationContext. Use the following code to do so.

```
ApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("applicationContext.xml");
```

Step 2 - Adding a service
Create a new Java interface “PrinterService” with a single method:

```
void print(String message);
```

Create a new class “ConsolePrinter” and implement the PrinterService interface. Now add a bean definition in the applicationContext.xml:

```
 <bean id="printer" class="lab1.ConsolePrinter"/>
```

Test your configuration by looking up the printer bean in the applicationContext in the main class:

```
ApplicationContext applicationContext =
      new ClassPathXmlApplicationContext("applicationContext.xml");
PrinterService printer =
      applicationContext.getBean(PrinterService.class);
printer.print("Hallo!");
```

Step 3 - Wiring dependencies
Now you'll add more beans that will depend on each other to see the different forms of dependency injection. Start by adding a new class "SetterDI" to the project. Add a property (private field and setter method) of type PrinterService to the class. This dependency must be wired by the Spring container, you'll do that in a minute. Also add a method "sayHello" to the class that uses the PrinterService to print a message to the console.

Now add the new class to the Spring configuration as you did in the previous step. Test the project by looking up the setterDI bean from the main method of the project and calling the sayHello method. This should result in a *NullPointerException* because the PrinterService is not wired to the Hello bean yet.

To wire the PrinterService dependency you will use the most basic form of dependency injection by configuring the dependency in XML. Change your bean definition as follows:

```
<bean id="setterDI" class="lab1.SetterDI">
      <property name="printer" ref="printer"/>
</bean>
```

This is *setter injection.* Spring will use the setPrinter method to inject the printer bean into the hello bean.

Step 4 - Constructor injection
Instead of using *setter injection* you could use *constructor injection*. Add a new class to the project named "ConstructorDI". Add a private field "printer" of type PrinterService but omit get/set methods. Add a constructor with an argument of type PrinterService to the class and initialize the printer field using this argument. Again, add a method "sayHello" that uses the printer.

Add a bean declaration to the configuration for the new class. To wire the PrinterService dependency you'll now have to use a *constructor-arg*.

```
<bean id="constructorDI" class="lab1.ConstructorDI">
      <constructor-arg ref="printer"/>
</bean>
```

Test the new class again by looking it up in the container from the main method.

Step 5 - AutoWiring
In this step you'll try out AutoWiring of dependencies. AutoWiring can be enabled on a whole applicationContext, but in this case you will enable AutoWiring for a single bean. Add a new class "AutoWireDI" to the project and add a private field of type PrinterService, a setter method for this field and a sayHello method.

Add the bean configuration to the applicationContext. Instead of wiring the dependency to the PrinterService explicitly you will autowire this dependency using the following configuration:

```
<bean id="autowiringDI" class="lab1.AutoWireDI" autowire="byType"/>
```

Add code to the main method to test the new bean. Also try out the different modes of autowiring (byType, byName etc.).

Step 6 - Factory methods
In this step you will use a factory method to initialize a bean. Create a new class "FactoryMethodInstantiation" to the project. Add a private no-arg constructor so you can be sure Spring will not call this constructor. Also add two private fields, a PrinterService and a String.

Now add a factory method. A factory method must be static and return an instance of the class it's part of. The factory method should have two arguments, a PrinterService instance and a String. Make sure the method looks as follows:

```
public static FactoryMethodInstantiation createInstance(
                           PrinterService printer, String value) {
     FactoryMethodInstantiation instance =
          new FactoryMethodInstantiation();
     instance.printer = printer;
     instance.value = value;
     return instance;
}
```

Add a printValue method to the class that uses the printer instance to print the value field to the console.

Configure the new bean in the application context. To pass arguments to a factory method you'll have to use constructor-args (although it's not really a constructor).

```
<bean id="factoryMethodInstantiation"
class="lab1.FactoryMethodInstantiation"
     factory-method="createInstance">
     <constructor-arg index="0" ref="printer" />
     <constructor-arg index="1" value="MyTest" />
</bean>
```

Test the bean again from the project's main method.

Step 7 - Factory beans
The factory approach can be taken a step further by implementing the factory-method on a separate factory-bean instead of in the class itself. This is similar to the distinction of the Factory Method and Abstract Factory pattern. You can also use factory beans to instantiate and fill lists of data for example. That's what you'll do in this step.

Add a new class "NameFactory" to the project. Add a non-static method createNameList to the class that returns a java.util.List of names.

For this approach you'll need two bean configurations in the applicationContext. One of the bean that represents the list of names, and one for the factory.

```
<bean id="nameFactory" class="lab1.NameFactory"/>

<bean id="names" factory-bean="nameFactory"
      factory-method="createNameList"/>
```

Test the configuration by looking up the names bean from the main method and print it to the console.

Step 8 - Scopes
So far you have only used the default singleton scope. To understand the difference between singleton and prototype scope, and to understand method injection you'll experiment with those in this step. First of all create a new class "HitCounter" to the project and add a field "hits" of type int to it. Also add a void method "increment" that increments the hits field. Configure the class as a bean in the applicationContext.

Add another class "Hitter" to the project. This class should have a dependency to the HitCounter class. Add a void method "hit" to the class that calls the HitCounter's increment method. Now add *two* bean definitions for the Hitter class (meaning two instances of the same class). The configuration should now be as follows.

```
<bean id="hitCounter" class="lab1.HitCounter"/>

<bean id="hitter1" class="lab1.Hitter">
      <property name="counter" ref="hitCounter"/>
</bean>

<bean id="hitter2" class="lab1.Hitter">
      <property name="counter" ref="hitCounter"/>
</bean>
```

Retrieve both hitter1 and hitter2 from the applicationContext in the project's main method. Call the hit method twice on both hitters. Call and print the getHits on hitter1. You should see 4 hits, because at this point the HitCounter is shared (it's a singleton) by the two hitters.

Change the scope of the hitCounter bean to prototype. Test again. How many hits do you see? Each bean that references the hitCounter gets it’s own instance now.

What if you would want a new hitCounter instance on each call to the hitter’s hit method? This would not be very useful in this example, but it’s interesting anyway. You could use method injection for this. Add a new class “DynamicHitter” to the project and give it an abstract method “getHitCounter”, a “hit” and “getHits” method.

```
public abstract class DynamicHitter {
     public void hit() {
          getHitCounter().increment();
     }

     public int getHits() {
          return getHitCounter().getHits();
     }

     public abstract HitCounter getHitCounter();
}
```

The abstract method will be implemented at runtime by Spring. To enable this you’ll need some extra configuration in the bean definition.

```
<bean id="hitter3" class="lab1.DynamicHitter">
     <lookup-method name="getHitCounter" bean="hitCounter"/>
</bean>
```

Test the new bean by calling the hit method multiple times and asking for getHits after that. The result should be 0 hits, because a HitCounter is created by Spring each time the getHitCounter method is called.

Wiring Dependencies using annotations

In this lab you will learn to use Dependency Injection using the different annotations available in the Spring framework. You will implement a simple MovieLister class that uses multiple MovieCatalogs to list movies.

Step 1 - Create the project

Create a new Maven module in STS. Choose simple-project in the wizard. Add a applicationContext.xml file. There are two steps in enabling an annotation based approach: Enable annotation configuration and enable component scanning. Component scanning scans the classpath for @Component annotated classes that will be added to the context. Annotation configuration is in the “context” namespace. The full configuration file is as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:context="http://www.springframework.org/schema/context"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-
context-3.0.xsd">

     <context:annotation-config />
     <context:component-scan base-package="lab2"/>
</beans>
```

Create a Tester class with a main method that bootstraps the Spring container.

Step 2 - A simple movie lister

You’ll start with some basic annotation based dependency injection. First create a new class MovieLister. Add a void method listMovies to the class. Annotate the class with @Component to make it a Spring bean.

Add an interface MovieCatalog with a single method getCatalog that returns a List<String>. Implement the interface in a class GeneralMovieCatalog and return a few movies and annotate this class too with @Component. Go back to the MovieLister class. Add a private field of type MovieCatalog, and use this field to print the list of movies returned by the getCatalog movie. To wire the dependency to the MovieCatalog you can use the @AutoWire annotation.

```
@Autowired
private MovieCatalog movieCatalog;
```

This is dependency injection by type, so there may only be one implementation of the MovieCatalog interface. Test if the dependency is wired correctly.

Step 3 - Qualifiers
Create another implementation of the MovieCatalog interface named ActionMovieCatalog. When you try to start the application again the container should give an error telling that there is no unique bean of type MovieCatalog. The container can't know which of the two implementations of the MovieCatalog interface should be injected. By using qualifiers you can bind a specific implementation class to an injection point, while still programming to interfaces at the Java level (it's still easy to unit test).

You'll create a fully type-safe qualifier instead of relying on Strings. While this is slightly more work, it can prevent errors. Create a new annotion named Action and annotate it as a Qualifier.

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Action { }
```

Add the newly created @Action both on the ActionMovieCatalog class definition and the field where you want to inject the movie catalog.

```
@Autowired
private MovieCatalog movieCatalog;
```

Test the application again. Only action movies should be listed now.

Step 4 - Multiple catalogs
In some cases you want to get all implementation of a certain interface. You can do that by injecting into a List or Map of that interface.

Add another field to the MovieLister that contains all movie catalogs including their bean names:

```
@Autowired
private Map<String, MovieCatalog> allCatalogs;

public void listMovies() {
     for(String key : allCatalogs.keySet()) {
          System.out.println(key);
          for(String title : allCatalogs.get(key).getMovieCatalog()) {
               System.out.println("-" + title);
          }
     }
}
```

AOP

In this lab you will learn AOP within Spring. You can use the project from the previous lab as a starting point. There are two types of AspectJ usage. You can use Spring AOP or full AspectJ. To use full AspectJ you would need to introduce the AspecJ compiler/weaver into your build. This is not the case for Spring AOP, but the possibilities are more limited. For example, only Spring beans can be advised. In many cases you will not need the more advanced options of AspectJ, and Spring AOP is a much easier in that case. For this lab you'll only use Spring AOP.

Step 1 - A trace log aspect

One of the most obvious cases for using AOP is trace logging. It's easy to log before and after each method call without changing any existing code. First you'll have to enable annotation based AOP in your Spring configuration.

```
<aop:aspectj-autoproxy/>
```

Now create a new class called TraceLogAspect and annotate it with @Aspect and @Component. The latter annotation is necessary because Spring will now scan the classpath for aspects, so a class has to be a Spring bean before it's recognized as an aspect.

A tracelog can easily be implemented using an @Around advice. Add a method called trace to the Aspect class and annotate it with an inline pointcut.

```
@Around("execution(public * *(..))")
public Object trace(ProceedingJoinPoint jp) throws Throwable {
      long curTime = System.currentTimeMillis();
      Object result = jp.proceed();
      long time = System.currentTimeMillis() - curTime;
      System.out.println(
            jp.getStaticPart().getSignature().getName() + " " + time);
      return result;
}
```

Run the application again and see if the Aspect works.

Step 2 - Retry after exception

Sometimes services you depend upon can be unreliable. In some cases it's useful to "just try again" after a failure. This is easy to implement using AOP. Add a new method to one of the classes in the project, and add code that throws an exception at random (use Math.random for example). Implement an aspect that catches the exception, write a log to the console, and try the action again until it succeeds.

Using JPA

In this lab you will start using JPA. First by using JPA standalone to explore the API and later integrated with Spring.

Step 1 - Project Configuration and first test
Open a terminal and enter the following commands:

```
cd spring
hg update jpa
```

Import the newly created project “webshop” in STS. The project already contains some classes and a persistence.xml. Open the persistence.xml and check the settings. Now open the Book class. It contains an id and a name, but is not an Entity yet. Now open the StandaloneJpaTest in the test folder. This test class already contains a test method that checks if a Book can be persisted. Try to run it, it should fail at this moment. The reason the test fails is because Book is not an Entity yet. Fix Book so that the test passes.

Step 2 - Finding books
Add a new test method to test retrieval of a book using the find method. The tables are recreated before every test, so you can be sure to have a clean database in every test. This does mean the table is empty! Insert some books to be able to find books. Your test could look as follows.

```
@Test
    public void testFind() throws Exception {
        createTestBooks();

        Book book = em.find(Book.class, 1L);
        assertNotNull(book);
        assertThat(book.getTitle(), is("Angels and demons"));
    }

    private void createTestBooks() {
        em.getTransaction().begin();
        em.persist(new Book("Angels and demons"));
        em.persist(new Book("Digital Fortress"));
        em.persist(new Book("The Da Vinci code"));
        em.persist(new Book("The Lost Symbol"));
        em.persist(new Book("Deception Point"));
        em.getTransaction().commit();

        em.close();
        em = emf.createEntityManager();
    }
```

Step 3 - Editing managed books
Create another test that test if managed books can be edited. Always use a new EntityManager and transaction to test for database changes!

Step 4 - Editing detached books
Retrieve a book using the find method and detach it by closing the EntityManager. Try if changes to the instance or not persisted to the database. Now use the merge method to synchronize the changes with the database.

Step 5 - Listing books
Write a simple query to select all books.

Step 6 - Mapping books
Add the following fields to Book:
-Date releaseDate (Date in the database)
-Enum Category
-String summary (should be a longtext in the datase)

Also make the title unique.

Add mapping configuration to make sure that the correct datatypes are used in the database.

Step 7 - Deleting books
Use the remove method to remove a book.

Step 8 - One to One Promotion
Create a new class Promotion. A promotion can be a temporarily lowered price for a limited amount of time. Add the following properties.
-String description
-BigDecimal newPrice
-Date beginDate
-Date endDate

Map a bi-directional relationship between Book and Promotion. Test if saving fetching from both sides work. Use cascading to be able to save a new book with a new promotion without persisting the promotion explicitly. Write tests to test bi-directional behavior and to test cascading remove.

Step 9 - One to Many reviews
Create a new class Review and map a bi-directional one-to-many relation with Book. Add the following properties to Review.
-String reviewerName
-Date reviewDate
-int rating
-String text

Test adding, cascading, lazy loading and bi-directional behavior. Also test cascading remove. Create a separate test to test a join fetch query for explicit eager loading.

Step 10 - Many to Many authors
Create a new class Author and configure a bi-directional Many-to-Many relation with Book with cascading persist. Test if persisting and retrieval works correctly.

Step 11 - Inheritance and ElementCollections
Create a new class EBook and a new enum EBookFormat. The EBookFormat represents formats such as EPUB and PDF. An EBook is a subclass of Book and contains a List of available EBookFormats. To map the List you'll need an @ElementCollection mapping. Test if EBooks can be persisted correctly.

Step 12 - Details embeddable
Create a new class Details with the following properties.
-int pages
-int isbn10
-int isbn13
-String language

Make Details @Embeddable and add a reference to it in Book. Make sure you initialize an empty details by default in Book, otherwise it's impossible to save a book without Details. Create a test to check if Details are persisted correctly on a Book.

Step 13 - Query for books written by more than one author
Write and test a JPQL query to retrieve books that are written by at least two authors. Don't forget to insert some test data first!

Step 14 - Query for the average rating for each Author
Write and test a JPQL query to retrieve the average rating for each author. The results should be returned as a value object Rating.

```
public class Rating {
    private String authorName;
    private double avgRating;

    public Rating(String authorName, double avgRating) {
        this.authorName = authorName;
        this.avgRating = avgRating;
    }

    public String getAuthorName() {
        return authorName;
    }

    public double getAvgRating() {
        return avgRating;
    }
}
```

Don't forget to add some authors and ratings before executing the query!

Step 15 - Criteria API
Write and test a method that accepts a title and author name filter. If both arguments are empty, all books should be returned, including books without an author. If both are non-empty, an AND filter should be used and if only one argument is non-empty only that field should be filtered. The method should use the Type Safe Criteria API to build this dynamic query. Order results ascending by title.

Step 16 - Bean validation
Add bean validation to the Book class to make sure books always have a title. Write a test that expects a ConstraintViolationException.

Step 17 - Custom validator
Write a new bean validator annotation and implementation class to disallow certain words in a title. The annotation should be configurable:

```
@NotExplicit(filter = {"sex", "drugs", "rock & roll"})
private String title;
```

Spring JPA

Now that you have some experience with JPA you can integrate with Spring. First update the project with Mercurial in a terminal.

```
cd spring
hg update spring_jpa
```

Besides the persistence.xml configuration you'll also need some Spring configuration now. First of all you'll need a database connection. Open dao-context.xml and add the following configuration.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
   <property name="driverClassName" value="${jdbc.driverClassName}"/>
   <property name="url" value="${jdbc.url}"/>
   <property name="username" value="${jdbc.username}"/>
   <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="classpath:jdbc.properties"/>
```

The property-placeholder reads the jdbc.properties file for the actual connection properties.

Now create a transaction manager that can be used with JPA.

```
<bean id="transactionManager" class="...JpaTransactionManager">
     <property name="entityManagerFactory" ref="myEmf" />
</bean>

<bean id="myEmf" class="...LocalContainerEntityManagerFactoryBean">
     <property name="dataSource" ref="dataSource" />
</bean>
```

That's all configuration needed for JPA/Hibernate. The EntityManagerFactory could now be used to create EntityManager instances, or the @PersistenceContext annotation can be used to let Spring create shared transaction scoped EntityManagers.

There's still a problem though. With the current configuration the transaction manager configured for JPA is not available for JDBC. This means it's not possible to run JDBC and JPA code in the same transaction. That also means you can't use countRowsInTable in your tests for example. With some additional configuration you can enable JDBC to use JPA transactions. You'll need to configure a JpaDialect for this to work. The full configuration is as follows.

```
<bean id="transactionManager" class="...JpaTransactionManager">
     <property name="entityManagerFactory" ref="myEmf" />
</bean>

<bean id="jpaDialect" class="...HibernateJpaDialect"/>

<bean id="myEmf" class="...LocalContainerEntityManagerFactoryBean">
     <property name="dataSource" ref="dataSource" />
     <property name="jpaDialect" ref="jpaDialect"/>
</bean>
```

Step 2 - Implement a JPA Dao
You'll create a dao for managing books. The Book entity from the previous lab can be used again.

Implement the BookDao interface completely. When working with JPA you'll need an EntityManager. Spring can inject a transaction scoped EntityManager into a bean. Based on the EntityManager you can easily query for Entities. Don't forget to make the dao implementation a Spring bean and transactional.

```
@PersistenceContext
EntityManager em;

@Override
@SuppressWarnings("unchecked")
@Transactional(readOnly=true)
public List<Book> listBooks() {
     Query q = em.createQuery("select b from Book b");
     return q.getResultList();
}
```

Step 3 - Testing the Dao
Create a new Test Case class for the BookDao. Extend from `AbstractTransactionalJUnit4SpringContextTests` to bootstrap the container and activate transaction management in the tests.

```
@ContextConfiguration("classpath:dao-context.xml")
public class JpaBookDaoImplTest extends
AbstractTransactionalJUnit4SpringContextTests {
```

First you need some test data. Create another class TestDataInserter in the test folder. Add a method that adds a few books using JPA. This method should first delete all existing books. Now there is a small problem. If the test data is re-inserted before each test the generated ids are incremented each test. You can prevent this by resetting the index before each test.

```
private void removeBooks() {
    em.createQuery("delete from Book b").executeUpdate();
    simpleJdbcTemplate.update("ALTER TABLE book AUTO_INCREMENT = 1");
}
```

Now write a test for each DAO method. Remember that the transaction is rolled back after each test so you should not see any changes in the database. That also means that some JPA operations, such as a remove, should be flushed explicitly to execute. This can be done in the test, the production code should not change.

```
@Test
public void testRemoveBookById() throws Exception {
   bookDao.removeBook(1);
   em.flush();
   assertThat(countRowsInTable("book"), is(4));
}
```

Step 4 - Simple JDBC Template
Create a new class BookStatsDao. This class uses a SimpleJdbcTemplate to do some “advanced” queries. Add a method to count books. Remember to use JDBC, not JPA! Test the method in a new test method.

Controllers and pages

In this lab you will learn to use Spring 3 style annotation based controllers. You can extend the project of the previous lab.

Step 1 - Configure the project

The project already has a Maven webapp structure, but there's no Spring Web MVC configuration yet. You'll first have to configure the Spring Dispatcher Servlet in the web.xml file. The Dispatcher Servlet acts as a front controller in a Spring web project.

By default you'll need a Spring context configuration file called [servlet name]-servlet.xml, so in the following example you'll need a spring-servlet.xml file. It is common to have all the web related configuration in a separate context (configuration file) then the middle-tier configuration. To initialize another context (e.g. the context of the previous lab) you need to add a context-param and listener to the web.xml.

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:dao-context.xml</param-value>
</context-param>

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

<servlet>
      <servlet-name>spring</servlet-name>
      <servlet-class>
           org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
         <param-name>contextConfigLocation</param-name>
          <param-value>classpath:spring-servlet.xml</param-value>
       </init-param>
      <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
      <servlet-name>spring</servlet-name>
      <url-pattern>/spring/*</url-pattern>
</servlet-mapping>
```

Now create a new configuration file in the main/resources folder named spring-servlet.xml. This context will only keep configuration related to the web application. The example configuration enables annotation configuration, component scanning and it configures a view resolver. As you can see, JSP files will be placed in the WEB-INF/jsp folder.

```
<context:component-scan base-package="webshop.controllers" />
<context:annotation-config />

<bean id="jspViewResolver"

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

Step 2 - Creating a controller
Create a new class webshop.controllers.BookController. Annotate the class @Controller and @RequestMapping(“books”). Add a method listBooks that returns a ModelAndView and annotate it @RequestMapping(method=RequestMethod.GET). This method will be executed when a user visits the URL /spring/books.

Use the BookCatalog from the previous lab to retrieve a list of books and add it to the ModelAndView. Set the viewName property of the ModelAndView to books/index. Now create a new JSP file WEB-INF/jsp/books/index.jsp. Iterate over the list of books to display them.

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
  <head><title>Books</title></head>
  <body>
    <table>
        <tr>
            <th>Title</th>
        </tr>

       <c:forEach items="${books}" var="book">
        <tr>
            <td>${book.title}</td>
        </tr>
       </c:forEach>
        </table>
  </body>
</html>
```

Test if the list of books is rendered correctly.

Step 3 - Lazy loading with JPA
Dealing with lazy loading can be quite a difficulty in a web application. You should already have a many-to-many relationship between book and author.

Try to render a list of authors for every book on the page you just created. So within the forEach loop for books add the following:

```
 <c:forEach items="${book.authors}" var="author">
     <li>${author.name}</li>
</c:forEach>
```

Try reloading the page. It breaks on a LazyInitializationException! The problem is that the authors list should be lazily fetched, but at the time the JSP is rendering, the JPA entity manager is closed already. There are two solutions for this problem, using JPQL to fetch a certain collection eagerly or keep the entity manager open until the view is rendered. The latter solution requires less work, but has a technical downside. The disadvantage is that this will not work when the transactional layer and the view layer are running on separate JVMs. A second, more conceptual, disadvantage of this approach is that it's easy to accidentally create a view with the n+1 select problem. This is solved by carefully monitoring the queries going to the database and tuning fetching strategies whenever necessary. For this lab you'll configure entity managers to be kept open the entire request. Add the following configuration to spring-servlet.xml.

```
<mvc:annotation-driven/>

<mvc:interceptors>
     <bean class=”...OpenEntityManagerInViewInterceptor">
         <property name="entityManagerFactory" ref="myEmf"/>
     </bean>
</mvc:interceptors>
```

This installs an interceptor that binds an EntityManager to the thread that handles the request. Transaction managers will pick this up automatically. Try reloading the page again. There should be no errors.

Step 4 - Adding a form
Add a new JSP page jsp/books/edit.jsp that contains a form using the Spring's form tag library. At this point, only add a form:input field for the book's title.

Add a new method to the BookController that displays the new page when a user requests /books/edit. This method will only display the form, not handle it's submit. Within the method, instantiate a new book and add it to the model. Test if the form can be displayed correctly.

Step 5 - Handling form submits
Add another method to the controller that accepts POST requests, and a ModelAttribute argument containing the posted book.

```
@RequestMapping(value = "edit", method = RequestMethod.POST)
public String saveBook(Book book) {
```

Write code to store the new book in the database using the Hibernate Dao. The method should return a String. This String should be a view name. Use the redirect keyword within the String to send a redirect to the view.

```
return "redirect:/spring/books";
```

Step 6 - Bean Validation
Spring integrates nicely with the Bean Validation API. If a Bean Validation implementation, such as Hibernate Validator, is found on the classpath it's enabled by default when the mvc:annotation-config is enabled.

First add some validation annotations to the Book entity. For example, use the @NotEmpty annotation on the title field to make sure a book always has a title. Now change the controller to validate the book after form submission.

```
@RequestMapping(value = "edit", method = RequestMethod.POST)
public String saveBook(@Valid Book book, BindingResult result) {
    if (result.hasErrors()) {
        return "books/edit";
    } else {
        bookCatalog.saveBook(book);
        return "redirect:/spring/books";
    }
}
```

Check if the form is re-displayed when an empty title is submitted. Of course there should be some error messages telling the user that there is a problem. You can display an error message for a specific field on a command object using the form:errors tag.

```
<form:errors path="title"/>
```

It would also be useful to show a common message like "there are validation errors" on top of the page. You can use the spring:hasBindErrors tag for this.

Step 7 - Handling Ajax requests
Spring MVC doesn't include any Ajax support. With the RESTful Web Service support it's easy to use any JavaScript library however. First, you'll need a JavaScript library. Any JavaScript library would do, but for this lab you'll use jQuery. Download jQuery and import the library on your page.

```
<script type="text/javascript"
src="/path/to/jquery.js"></script>
```

Spring MVC makes it easy to support both *Content Centric* Ajax and *Data Cenric* Ajax. Content Centric Ajax is easier but less flexible. In this lab you will use Ajax to filter the list of books while typing. Add a text field on the page and the following jQuery code.

```
<script type="text/javascript">
    $(function() {
        $("#filter").keyup(function() {
             $("#booktable").load("books/booktable",
                {"filter": $("#filter").val()}
             );
        });
    });
</script>
```

Refactor the books table to a separate include file. This makes it possible to use the include file as the view for the Ajax call. Implement the new controller method and test the filtering.

RESTful Web Services

Building RESTful Web Services in Spring is very similar to creating web applications. The only difference is the type of content sent to the client. In this lab you'll implement a simple service for listing and adding books.

Step 1 - JAXB mapping

The Book service will support representations in XML. This mean you'll have to tell how a Book should be serialized to XML. A number of XML libraries are supported by Spring, in this lab you'll use JAXB2. First open the Book class and annotate it with @XmlRootElement. Because the service also need to return a list of books you'll need to create a new class BookList and annotate this class @XmlRootElement too. Add a property List<Book> to it.

```
@XmlRootElement
public class BookList {
    private List<Book> books;

    public BookList() {
    }

    public BookList(List<Book> books) {
        this.books = books;
    }

    public List<Book> getBooks() {
        return books;
    }

    public void setBooks(List<Book> books) {
        this.books = books;
    }
}
```

Step 2 - Implementing the controller

Code for implementing RESTful services is very similar to normal controller code. There are different strategies in Spring to select a controller method to handle a request.

1. Content negotiation

You can use the same method to output HTML and XML depending on the requested content-type by the client. Clients can use the Accept header to specify which content-type they want to receive. The controller method returns a Model objects just like normal, but the Model will be rendered according the content-type using a ContentNegotiatingViewResolver. You only have to write the controller code once to support a number of content-types, but this is also the downside. In many cases you want to execute different code for generating different content-types.

2. Use the headers property in the @RequestMapping annotations

The headers property can be used to select a controller method on the value of a given header property. If you would have two methods, one annotated @RequestMapping (headers = “accept=application/xml”) and one @RequestMapping(headers = “accept=text/html”) the correct method would be chosen depending on the accept header. This is a very good approach, the only problem is that it might return the wrong content type for some browsers because the accept header can’t be influenced here.

3. Use a URL extension
While not completely RESTful this is a much used approach. Each supported content-type gets it’s own URL, for example /books.xml. The accept header is not used to select the right controller method. This is the approach best fit for this lab.

Add a new method to the controller that is selected for /books.xml and returns a BookList instance. To serialize model objects to the client you have to annotate the return type @ResponseBody.

```
@RequestMapping(method = RequestMethod.GET, value = "books.xml")
public @ResponseBody BookList listBooksXml() {
    List<Book> books = bookCatalog.listBooks();
    return new BookList(books);
}
```

Step 3 - Posting new books
Implement a method to save new books using a POST. Use the Firefox plugin Poster to send a POST to the server.

Step 4 - Creating a client
Create a new project with a simple main class that uses the RestTemplate to use the Book Service.

Spring Security

In this lab you will secure the web application using Spring Security. While Spring Security is not part of the core Spring framework, it's used in many Spring web projects.

Step 1 - Setting up Spring Security
You'll first have to configure the Spring Security filter in web.xml. It's common to place all security configuration in a separate configuration file, so this file should also be added to the list of configuration files in the context-params.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml
        /WEB-INF/applicationContext-security.xml
    </param-value>
</context-param>

<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>
```

Step 2 - Security configuration
There is a basic configuration file available at the root of the project. Copy the contents of security-starterconfig.xml to a new file WEB-INF/applicationContext-security.xml. This file enables security with an in-memory authentication provider. All pages starting with /spring require a logged in user. /spring/books/edit requires a user with ROLE_SUPERVISOR. Start the application and test if it works as expected.

Step 3 - Setting up a user database
The in-memory authentication provider is not very useful for real application. With a little extra work Spring Security can use a database to store users and roles. There's a database script "security-tables.sql". Use it to create the required tables in the MySQL database. The tables follow the standard Spring Security model, but can be customized if necessary. Add a user to the user table. The password should be hashed using md5. It's also required to give the user at least one role.

Step 4 - Configuring the user database
Replace the existing authentication-manager with the following configuration.

```
<authentication-manager>
    <authentication-provider >
        <password-encoder hash="md5"/>
        <jdbc-user-service data-source-ref="dataSource" />
    </authentication-provider>
</authentication-manager>
```

Test the application again.

Step 5 - Securing page elements

The application is now secure, but not very user friendly. When logged in as a normal user you'll still see the "new book" link, but you can't access it. It would be better to not render the link if the user isn't authorized for the page. Spring Security provides a JSP tag library for this. To hide the "new book" link you could for example use the following tag.

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

<sec:authorize url="/spring/books/edit">
    <a href="books/edit">New book</a>
</sec:authorize>
```

JMX

In this lab you will a simple JMX management interface.

Step 1 - Add the JMX bean
Create a new class and annotate it @Component to make it a Spring bean. Add some methods to display information about books. For example a method that returns the number of books in the database.

Step 2 - Configure the JMX exporter
Spring can use the JMX server which is by default available on Tomcat. The only thing you have to configure is a JMX exporter.

```
<bean id="exporter"
 class="org.springframework.jmx.export.MBeanExporter" lazy-init="false">
    <property name="beans">
        <map>
            <entry key="bean:name=bookManager"
                    value-ref="jmxBookManager"/>
        </map>
    </property>
</bean>
```

Step 4 - Using JConsole
Open a command prompt and type jconsole. Connect to the catalina process and test your MBean on the MBeans tab.

JMS

In this lab you will connect the Spring web application to an ActiveMQ JMS queue.

Step 1 - Start ActiveMQ
Apache ActiveMQ is a much used JMS provider. Start it by opening a command prompt and type bin/activemq. After starting you can explore the management console at http://localhost:8161/admin.

Step 2 - Configure a connection factory
Spring needs a ConnectionFactory implementation to connect to a JMS queue. Depending on the environment you're running in there are a number of ways to create a ConnectionFactory. For this lab you'll use an ActiveMQ implementation directly.

```
<bean id="connectionFactory"
      class="org.apache.activemq.pool.PooledConnectionFactory"
      destroy-method="stop">
    <property name="connectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL">
                <value>tcp://localhost:61616</value>
            </property>
        </bean>
    </property>
</bean>
```

Step 3 - Implementing a MessageListener
Create a new class that implements MessageListener and annotate it @Component to make it a Spring bean. Implement the onMessage method. You can cast the Message to a TextMessage to work with simple text based messages which are very common.

Step 4 - Configure the MessageListener
The last step is to connect the MessageListener implementation you just created to the queue. You can do that with the following configuration.

```
<jms:listener-container>
      <jms:listener destination="queue.releases" ref="bookJmsListener" />
</jms:listener-container>
```

Test the receiving of messages by sending messages to the queue from ActiveMQ's admin console.

Scheduling tasks

In this lab you will learn to use the scheduling functionality provided by Spring.

Step 1 - Implement a task

A task can just be implemented as a POJO. Create a new class and make it a Spring bean. Give it a method that should be scheduled.

Step 2 - Configure the scheduler

Using the task namespace it’s very easy to implement a task scheduler. You’ll need a scheduler and a scheduled-tasks configuration.

```
<task:scheduler id="scheduler" pool-size="10"/>

<task:scheduled-tasks scheduler="scheduler">
    <task:scheduled ref="pingJob" method="ping" cron="*/5 * * * * *"/>
</task:scheduled-tasks>
```