



# Core Spring

## Lab Instructions

Building Enterprise Applications using Spring



Pivotal

---

This Page Intentionally Left Blank

---

---

# Table of Contents

<b>Reward Dining: The Course Reference Domain .....</b>	<b>vii</b>
1. Introduction .....	vii
2. Domain Overview .....	vii
3. Reward Dining Domain Applications .....	ix
3.1. The Rewards Application .....	ix
4. Reward Dining Database Schema .....	xiii
<b>1. Introduction to Core Spring .....</b>	<b>1</b>
1.1. Introduction .....	1
1.2. Instructions .....	1
1.2.1. Getting Started with the Spring Tool Suite .....	2
1.2.2. Understanding the 'Reward Network' Application Domain and API .....	7
<b>2. Dependency Injection with JavaConfig .....</b>	<b>16</b>
2.1. Introduction .....	16
2.2. Quick Instructions .....	16
2.3. Detailed Instructions .....	19
2.3.1. Creating the application configuration .....	19
2.3.2. System testing the application with Spring and JUnit .....	23
<b>3. Configuration with annotations .....</b>	<b>27</b>
3.1. Introduction .....	27
3.2. Quick Instructions .....	27
3.3. Detailed Instructions .....	28
3.3.1. Reviewing the application .....	28
3.3.2. Dependency injection using Spring's @Autowired annotation .....	29
3.3.3. Working with Init and Destroy callbacks .....	30
<b>4. XML Dependency Injection .....</b>	<b>33</b>
4.1. Introduction .....	33
4.2. Quick Instructions .....	33
4.3. Detailed Instructions .....	33
4.3.1. First verify that everything works .....	33
4.3.2. Convert to XML configuration / Component Scanning .....	33
4.3.3. Switch to pure XML Configuration .....	34
4.3.4. Bonus - Remove Component Scanning .....	36
<b>5. Dependency Injection Best Practices .....</b>	<b>37</b>
5.1. Introduction .....	37
5.2. Quick Instructions .....	37
5.3. Detailed Instructions .....	37

5.3.1. (TODO 01) Using bean definition inheritance to reduce the amount of configuration .....	37
5.3.2. (TODO 03) Externalizing values to a Properties file .....	39
5.3.3. (TODO 04) Using the <code>&lt;import/&gt;</code> tag to combine configuration fragments .....	40
<b>6. Integration Testing with Profiles .....</b>	<b>41</b>
6.1. Introduction .....	41
6.2. Quick Instructions .....	41
6.3. Detailed Instructions .....	41
6.3.1. Refactor to use Spring's TestContext framework (TODO 01) .....	42
6.3.2. Configure Repository Implementations using Profiles .....	43
6.3.3. Switching between Development and Production Profiles .....	43
6.3.4. Optional Step - Further Refactoring .....	44
<b>7. Introducing Aspect Oriented Programming .....</b>	<b>45</b>
7.1. Introduction .....	45
7.2. Quick Instructions .....	45
7.3. Detailed Instructions .....	45
7.3.1. Creating and Testing a simple Aspect (@Before advice) .....	45
7.3.2. Performance Monitor Aspect .....	48
7.3.3. Exception Handling Aspect (Optional) .....	48
<b>8. JDBC Simplification using the JdbcTemplate .....</b>	<b>50</b>
8.1. Introduction .....	50
8.2. Quick Instructions .....	50
8.3. Detailed Instructions .....	50
8.3.1. Refactoring a repository to use JdbcTemplate .....	50
8.3.2. Using a RowMapper to create complex objects .....	51
8.3.3. Using a ResultSetExtractor to traverse a ResultSet .....	52
<b>9. Transaction Management with Spring .....</b>	<b>54</b>
9.1. Introduction .....	54
9.2. Quick Instructions .....	54
9.3. Detailed Instructions .....	54
9.3.1. Demarcating Transactional Boundaries in the Application .....	55
9.3.2. Configuring Spring's Declarative Transaction Management .....	55
9.3.3. Developing Transactional Tests .....	56
<b>10. JPA Simplification using Spring .....</b>	<b>58</b>
10.1. Introduction .....	58
10.2. Quick Instructions .....	58
10.3. Detailed Instructions .....	59
10.3.1. Using JPA in the Account module .....	60
10.3.2. Using JPA in the Restaurant module .....	62
10.3.3. Integrating JPA into the Rewards Application .....	63
10.3.4. BONUS CREDIT .....	65
<b>11. Spring MVC Essentials .....</b>	<b>67</b>

11.1. Introduction .....	67
11.2. Quick Instructions .....	67
11.3. Detailed Instructions .....	67
11.3.1. Setting up the Spring MVC infrastructure .....	68
11.3.2. Implementing another Spring MVC handler method .....	71
<b>12. Securing the Web Tier .....</b>	<b>73</b>
12.1. Introduction .....	73
12.2. Quick Instructions .....	73
12.3. Detailed Instructions .....	74
12.3.1. Setting up Spring Security in the application .....	74
12.3.2. Define the Filter class .....	74
12.3.3. Include Security Configuration in the Root Application Context .....	74
12.3.4. Configuring authentication .....	75
12.3.5. Handling unsuccessful attempts to log in .....	77
12.3.6. Managing Users and Roles .....	78
12.3.7. Using the Security Tag Library .....	80
12.3.8. Bonus question: SHA-256 encoding .....	81
<b>13. Building RESTful applications with Spring MVC .....</b>	<b>82</b>
13.1. Introduction .....	82
13.2. Quick Instructions .....	82
13.3. Detailed Instructions .....	83
13.3.1. Exposing accounts and beneficiaries as RESTful resources .....	83
<b>14. Simplifying Messaging with Spring JMS .....</b>	<b>91</b>
14.1. Introduction .....	91
14.2. Quick Instructions .....	92
14.3. Detailed Instructions .....	92
14.3.1. Providing the messaging infrastructure .....	92
14.3.2. Sending Messages with <code>JmsTemplate</code> .....	93
14.3.3. Configuring the <code>RewardNetwork</code> as a message-driven object .....	93
14.3.4. Receiving the asynchronous reply messages .....	95
14.3.5. Testing the message-based batch processor .....	95
<b>15. JMX Management of Performance Monitor .....</b>	<b>97</b>
15.1. Introduction .....	97
15.2. Quick Instructions .....	97
15.3. Detailed Instructions .....	97
15.3.1. Exposing the <code>MonitorFactory</code> via JMX .....	97
15.3.2. Exporting pre-defined MBeans .....	100
<b>16. Spring Remoting .....</b>	<b>102</b>
16.1. Introduction .....	102
16.2. Quick Instructions .....	102
16.3. Detailed Instructions .....	103
16.3.1. Remoting with RMI .....	103

16.3.2. Remoting with Spring's HttpInvoker .....	105
<b>17. Exposing SOAP Endpoints using Spring WS .....</b>	<b>107</b>
17.1. Introduction .....	107
17.2. Instructions .....	107
17.2.1. Defining the message contract .....	107
17.2.2. Generate the classes with JAXB2 .....	109
17.2.3. Exporting the <code>RewardNetwork</code> as a SOAP endpoint .....	109
17.2.4. Consuming services from a SOAP endpoint .....	111
<b>18. ORM simplification using Spring .....</b>	<b>113</b>
18.1. Introduction .....	113
18.2. Quick Instructions .....	113
18.3. Detailed Instructions .....	113
18.3.1. Using Hibernate in the Account module .....	114
18.3.2. Using Hibernate in the Restaurant module .....	116
18.3.3. Integrating Hibernate into the Rewards Application .....	117
<b>A. Spring XML Configuration Tips .....</b>	<b>120</b>
A.1. Bare-bones Bean Definitions .....	120
A.2. Bean Class Auto-Completion .....	120
A.3. Constructor Arguments Auto-Completion .....	120
A.4. Bean Properties Auto-Completion .....	121
<b>B. Eclipse Tips .....</b>	<b>122</b>
B.1. Introduction .....	122
B.2. Package Explorer View .....	122
B.3. Add Unimplemented Methods .....	123
B.4. Field Auto-Completion .....	124
B.5. Generating Constructors From Fields .....	125
B.6. Field Naming Conventions .....	126
B.7. Tasks View .....	126
B.8. Rename a File .....	126
<b>C. Using Web Tools Platform (WTP) .....</b>	<b>128</b>
C.1. Introduction .....	128
C.2. Verify and/or Install the Tomcat Server .....	128
C.3. Starting & Deploying to the Server .....	128

---

# Reward Dining: The Course Reference Domain

## 1. Introduction

The labs of the Core Spring course teach Spring in the context of a problem domain. The domain provides a real-world context for applying Spring to develop useful business applications. This document provides an overview of the domain and the applications you will be working on within it.

## 2. Domain Overview

The Domain is called Reward Dining. The idea behind it is that customers can save money every time they eat at one of the restaurants participating to the network. For example, Keith would like to save money for his children's education. Every time he dines at a restaurant participating in the network, a contribution will be made to his account which goes to his daughter Annabelle for college. See the visual illustrating this business process below:



Figure 1: Papa Keith dines at a restaurant in the reward network





Figure 2: A percentage of his dining amount goes to daughter Annabelle's college savings

### 3. Reward Dining Domain Applications

This next section provides an overview of the applications in the Reward Dining domain you will be working on in this course.

#### 3.1. The Rewards Application

The "rewards" application rewards an account for dining at a restaurant participating in the reward network. A reward takes the form of a monetary contribution to an account that is distributed among the account's beneficiaries. Here is how this application is used:

1. When they are hungry, members dine at participating restaurants using their regular credit cards.
2. Every two weeks, a file containing the dining credit card transactions made by members during that period is generated. A sample of one of these files is shown below:

AMOUNT	CREDIT_CARD_NUMBER	MERCHANT_NUMBER	DATE
100.00	1234123412341234	1234567890	12/29/2010
49.67	1234123412341234	0234567891	12/31/2010
100.00	1234123412341234	1234567890	01/01/2010
27.60	2345234523452345	3456789012	01/02/2010

3. A standalone `DiningBatchProcessor` application reads this file and submits each `Dining` record to the rewards application for processing.

### 3.1.1. Public Application Interface

The `RewardNetwork` is the central interface clients such as the `DiningBatchProcessor` use to invoke the application:

```
public interface RewardNetwork
{ RewardConfirmation rewardAccountFor(Dining dining); }
```

A `RewardNetwork` rewards an account for dining by making a monetary contribution to the account that is distributed among the account's beneficiaries. The sequence diagram below shows a client's interaction with the application illustrating this process:

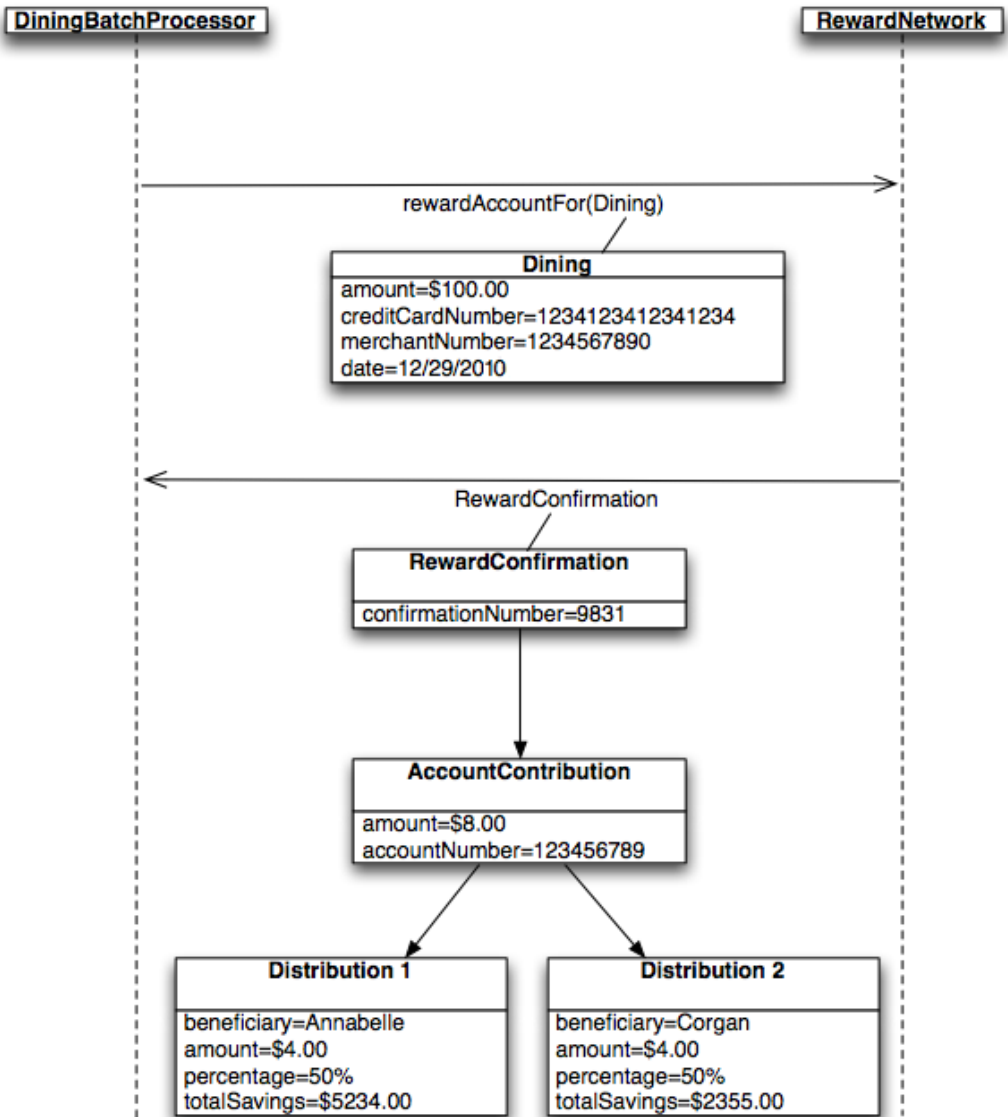


Figure 3: A client calling the `RewardNetwork` to reward an account for dining.

In this example, the account with credit card 1234123412341234 is rewarded for a \$100.00 dining at restaurant

1234567890 that took place on 12/29/2010. The confirmed reward 9831 takes the form of an \$8.00 account contribution distributed evenly among beneficiaries Annabelle and her brother Corgan.

### 3.1.2. Internal Application implementation

Internally, the `RewardNetwork` implementation delegates to domain objects to carry out a `rewardAccountFor(Dining)` transaction. Classes exist for the two central domain concepts of the application: `Account` and `Restaurant`. A `Restaurant` is responsible for calculating the benefit eligible to an account for a dining. An `Account` is responsible for distributing the benefit among its beneficiaries as a "contribution".

This flow is shown below:

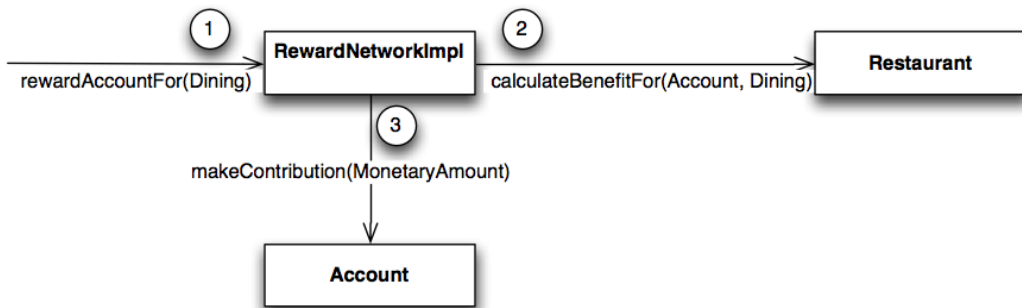


Figure 4: Objects working together to carry out the `rewardAccountFor(Dining)` use case.

The `RewardNetwork` asks the `Restaurant` to calculate how much benefit to award, then contributes that amount to the `Account`.

### 3.1.3. Supporting `RewardNetworkImpl` Services

`Account` and `restaurant` information are stored in a persistent form inside a relational database. The `RewardNetwork` implementation delegates to supporting data access services called 'Repositories' to load `Account` and `Restaurant` objects from their relational representations. An `AccountRepository` is used to find an `Account` by its credit card number. A `RestaurantRepository` is used to find a `Restaurant` by its merchant number. A `RewardRepository` is used to track confirmed reward transactions for accounting purposes.

The full `rewardAccountFor(Dining)` sequence incorporating these repositories is shown below:

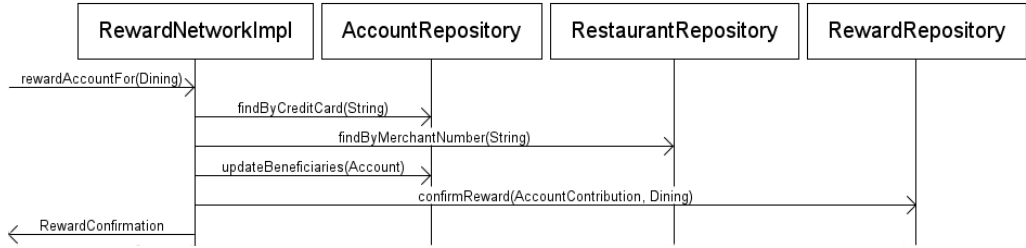


Figure 5: The complete `RewardNetworkImpl.rewardAccountForDining(Dining)` sequence

## 4. Reward Dining Database Schema

The Reward Dining applications share a database with this schema:

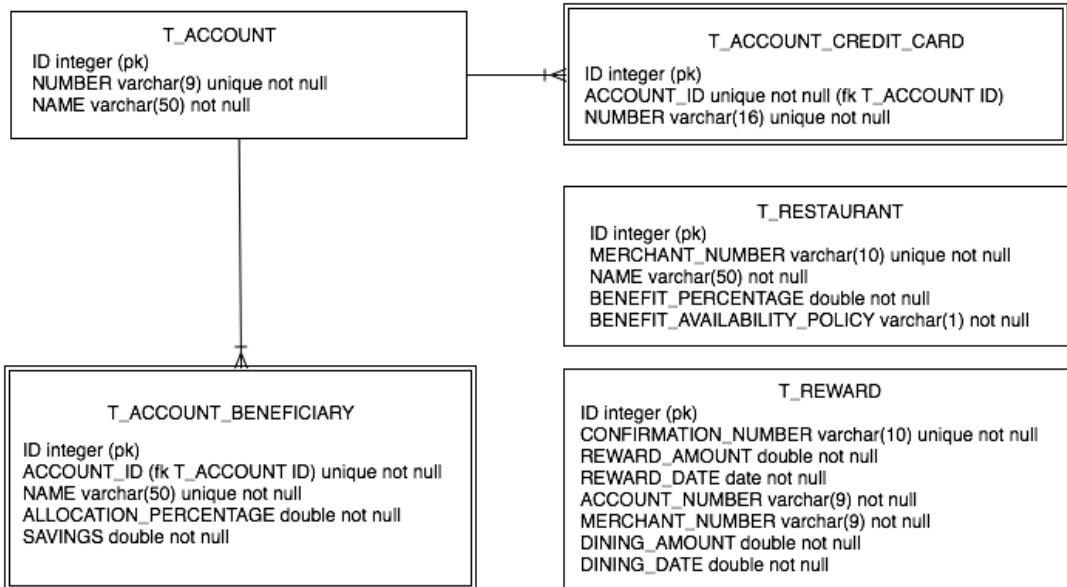


Figure 6: The Reward Dining database schema

---

# Chapter 1. Introduction to Core Spring

## 1.1. Introduction

Welcome to *Core Spring*! In this lab you'll come to understand the basic workings of the *Reward Network* reference application and you'll be introduced to the tools you'll use throughout the course.

Once you will have familiarized yourself with the tools and the application domain, you will implement and test the rewards application using Plain Old Java objects (POJOs).

At the end of the lab you will see that the application logic will be clean and not coupled with infrastructure APIs. You'll understand that you can develop and unit test your business logic without using Spring. Furthermore, what you develop in this lab will be directly runnable in a Spring environment without change.

Have fun with the steps below, and remember the goal is to get comfortable with the tools and application concepts. *If you get stuck, don't hesitate to ask for help!*



### Note

*In every lab, read to the end of each numbered section before doing anything. There are often tips and notes to help you, but they may be just over the next page or off the bottom of the screen.*

#### What you will learn:

1. Basic features of the Spring Tool Suite
2. Core *RewardNetwork* Domain and API
3. Basic interaction of the key components within the domain

Estimated time to complete: 30 minutes

## 1.2. Instructions

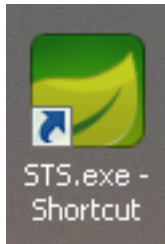
Before beginning this lab, read about the course reference domain to gain background on the rewards application.

## 1.2.1. Getting Started with the Spring Tool Suite

The Spring Tool Suite (STS) is a free IDE built on the Eclipse Platform. In this section, you will become familiar with the Tool Suite. You will also understand how the lab projects have been structured.

### 1.2.1.1. Launch the Tool Suite

Launch the Spring Tool Suite by using the shortcut link on your desktop.



**Figure 1.1. STS Desktop Icon**

After double-clicking the shortcut, you will see the STS splash image appear.



**Figure 1.2. STS Splash Image**

You will be asked to select a workspace. You should accept the default location offered. You can optionally check the box labeled *use this as the default and do not ask again*.

### 1.2.1.2. Understanding the Eclipse/STS project structure



#### Tip

If you've just opened STS, it may be still starting up. Wait several moments until the progress indicator on the bottom right finishes. When complete, you should have no red error markers within the *Package Explorer* or *Problems* views

Now that STS is up and running, you'll notice that, within the *Package Explorer* view on the left, projects are organized by *Working Sets*. Working Sets are essentially folders that contain a group of Eclipse projects. These working sets represent the various labs you will work through during this course. Notice that they all begin with a number so that the labs are organized in order as they occur in this lab guide.

### 1.2.1.3. Browse Working Sets and projects

If it is not already open, expand the *01-spring-intro* Working Set. Within you'll find two projects: *spring-intro* and *spring-intro-solution*. *spring-intro* corresponds to the start project. This pair of *start* and *solution* projects is a common pattern throughout the labs in this course.

Open the *spring-intro* project and expand its *Referenced Libraries* node. Here you'll see a number of dependencies similar to the screenshot below:



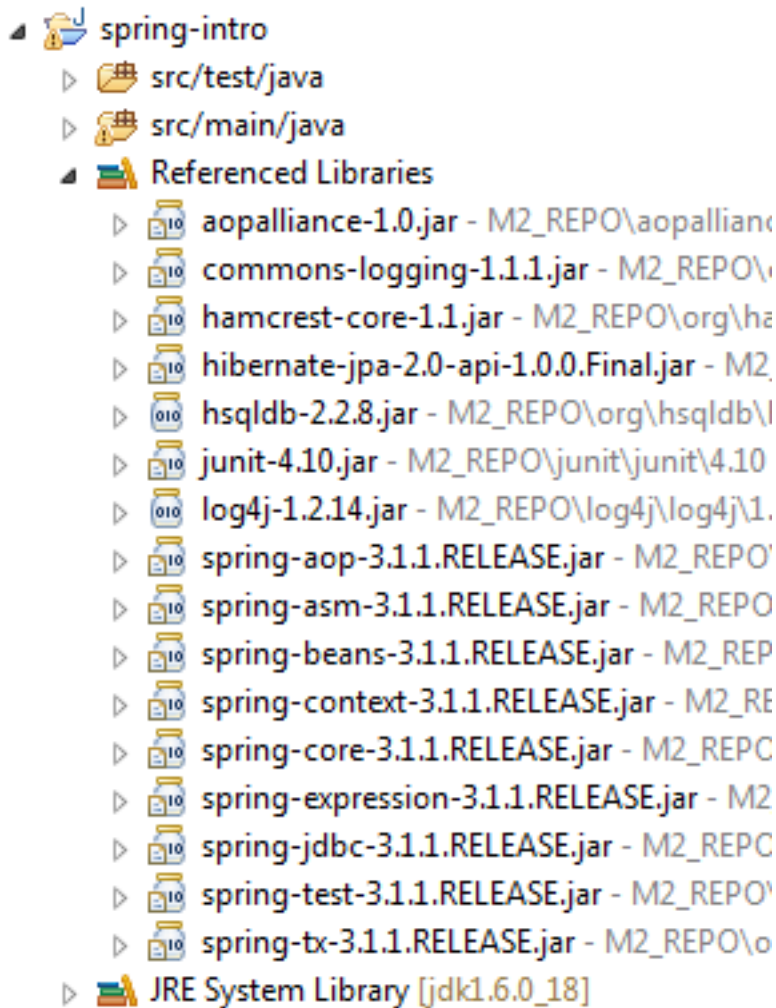


Figure 1.3. Referenced Libraries

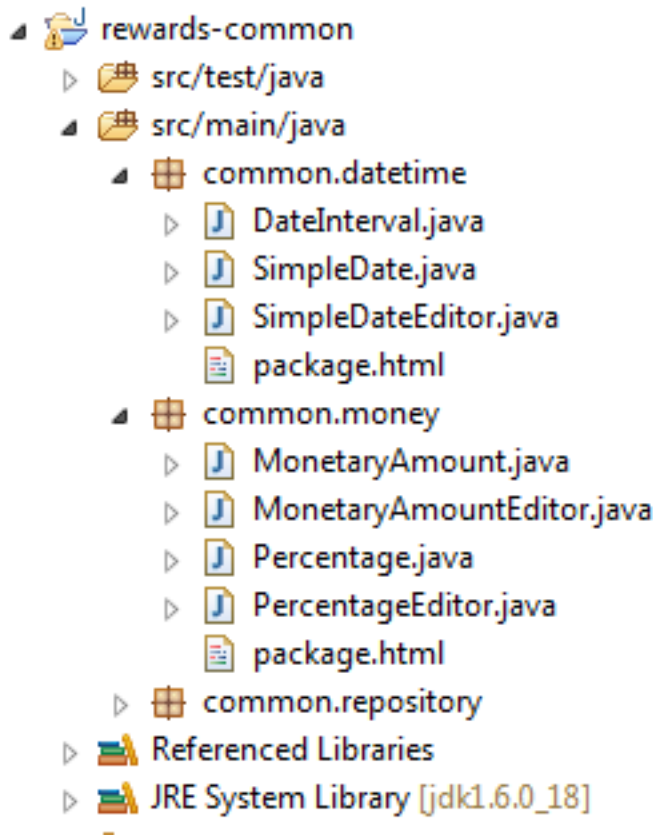


## Tip

This screenshot uses the "Hierarchical" Package Presentation view instead of the "Flat" view (the default). See the Eclipse tips section on how to toggle between the two views.

For the most part, these dependencies are straightforward and probably similar to what you're used to in your own projects. For example, there are several dependencies on Spring Framework jars, on Hibernate, DOM4J, etc.

In addition to having dependencies on a number of libraries, all lab projects also have a dependency on a common project called *rewards-common*.



**Figure 1.4.** *rewards-common* common components

This project is specific to Spring training courseware, and contains a number of types such as *MonetaryAmount*, *SimpleDate*, etc. You'll make use of these types throughout the course. Take a moment now to explore the contents of that jar and notice that if you double-click on the classes, the sources are available for you to browse.

#### 1.2.1.4. Configure the TODOs in STS

In the next labs, you will often be asked to work with TODO instructions. They are displayed in the *Tasks* view in Eclipse/STS. If not already displayed, click on *Window -> Show View -> Tasks* (be careful, *not Task List*). If you can't see the *Tasks* view, try clicking *Other ...* and looking under *General*.

By default, you see the TODOs for all the active projects in Eclipse/STS. To limit the TODOs for a specific project, execute the steps summarized in the following screenshots:

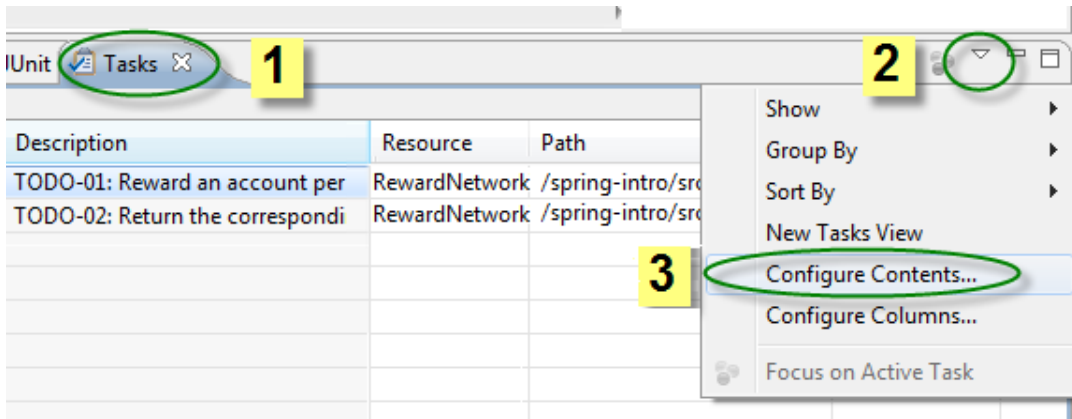
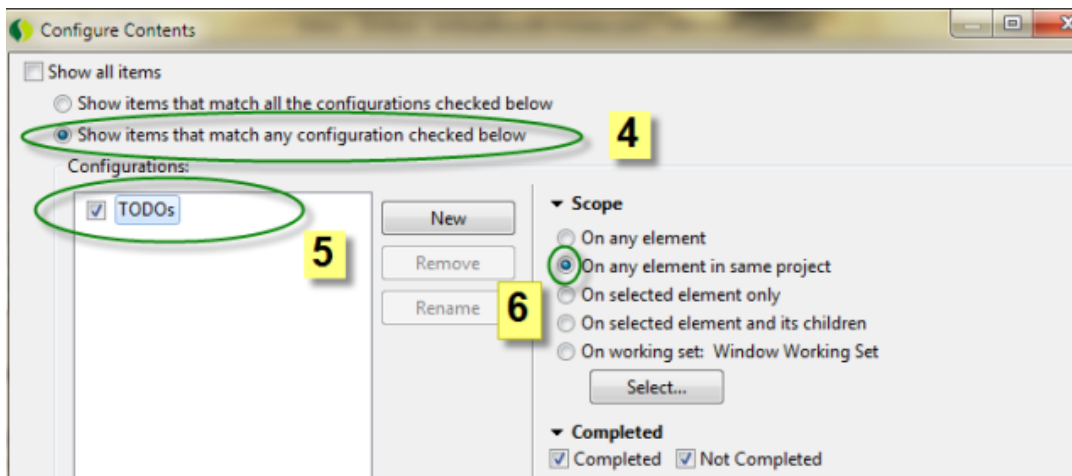


Figure 1.5. Configure TODOs



### Figure 1.6. Configure TODOs

**Caution:** It is possible, you might not be able to see the TODOs defined within the XML files. In this case, you can check the configuration in Preferences -> General -> Editors -> Structured Text Editor -> Task Tags pane. Make sure `Enable searching for Task Tags` is selected. On the `Filters` tab, verify if XML content type is selected. In case of refresh issues, you may have to uncheck it and then check it again.

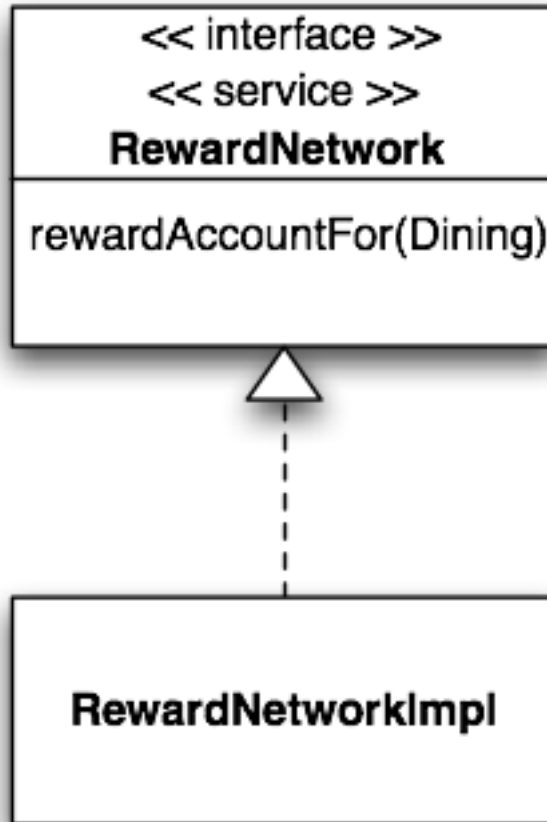
## 1.2.2. Understanding the 'Reward Network' Application Domain and API

Before you begin to use Spring to configure an application, the pieces of the application must be understood. If you haven't already done so, take a moment to review *Reward Dining: The Course Reference Domain* in the preface to this lab guide. This overview will guide you through understanding the background of the Reward Network application domain and thus provide context for the rest of the course.

The rewards application consists of several pieces that work together to reward accounts for dining at restaurants. In this lab, most of these pieces have been implemented for you. However, the central piece, the `RewardNetwork`, has not.

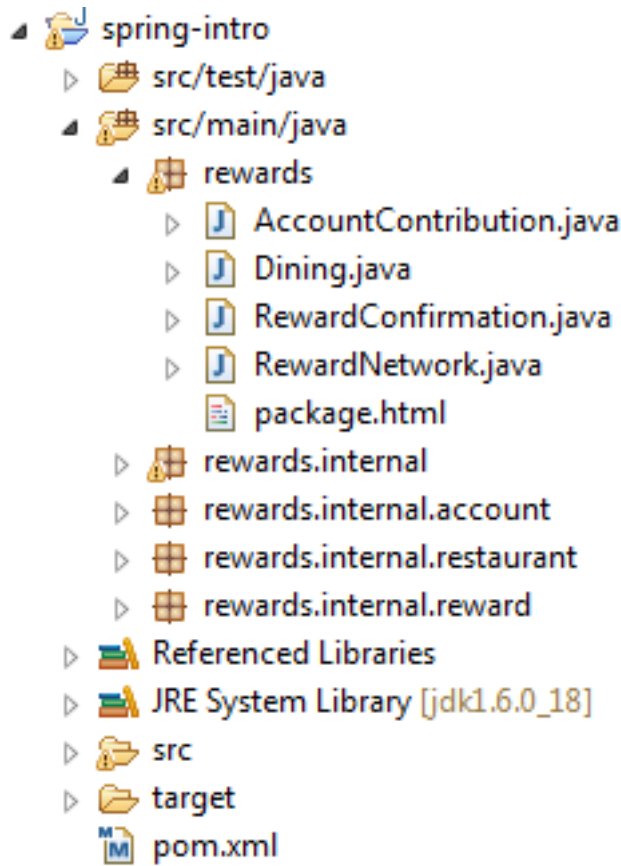
### 1.2.2.1. Review the `RewardNetwork` implementation class

The `RewardNetwork` is responsible for carrying out `rewardAccountFor(Dining)` operations. In this step you'll be working in a class that implements this interface. See the implementation class below:



**Figure 1.7.** `RewardNetworkImpl` implements the `RewardNetwork` interface

Take a look at your `spring-intro` project in STS. Navigate into the `src/main/java` source folder and you'll see the root `rewards` package. Within that package you'll find the `RewardNetwork` Java interface definition:



**Figure 1.8. The `rewards` package**

The classes inside the root `rewards` package fully define the public interface for the application, with `RewardNetwork` being the central element. Open `RewardNetwork.java` and review it.

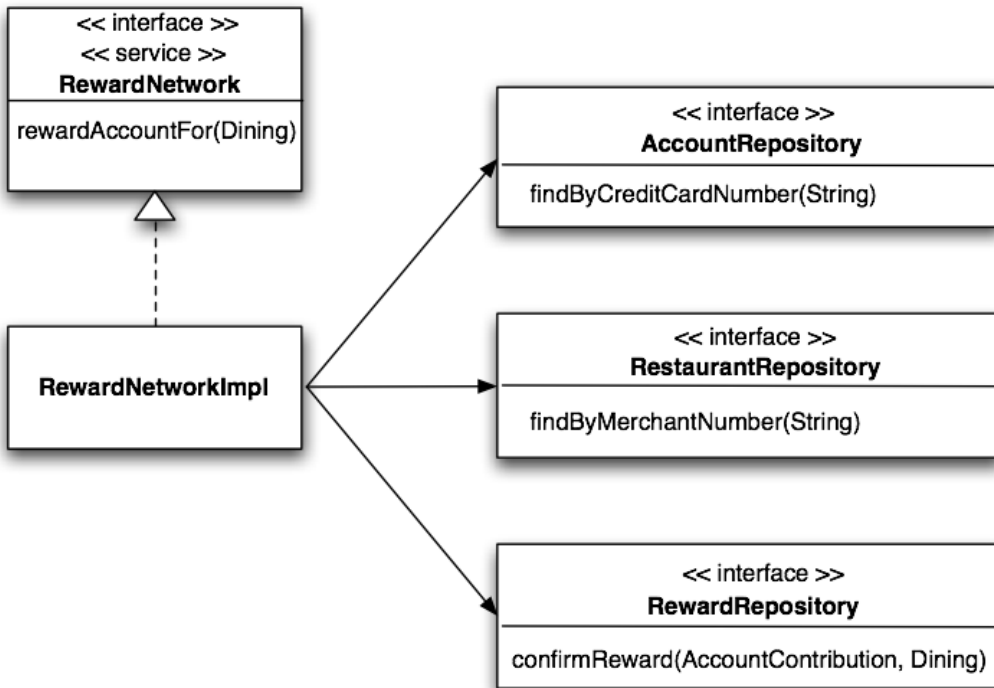
Now expand the `rewards.internal` package and open the implementation class `RewardNetworkImpl.java`.

#### **1.2.2.2. Review the `RewardNetworkImpl` configuration logic**

`RewardNetworkImpl` should rely on three supporting data access services called 'Repositories' to do its job. These include:

1. An `AccountRepository` to load `Account` objects to make benefit contributions to.
2. A `RestaurantRepository` to load `Restaurant` objects to calculate how much benefit to reward an account for dining.
3. A `RewardRepository` to track confirmed reward transactions for accounting and reporting purposes.

This relationship is shown graphically below:



**Figure 1.9.** `RewardNetworkImpl` class diagram

Locate the single constructor and notice all three dependencies are injected when the `RewardNetworkImpl` is constructed.

### 1.2.2.3. Implement the `RewardNetworkImpl` application logic

In this step you'll implement the application logic necessary to complete a `rewardAccountFor(Dining)` operation, delegating to your dependents as you go.

Start by reviewing your existing `RewardNetworkImpl` `rewardAccountFor(Dining)` implementation. As you will see, it doesn't do anything at the moment.

Inside the task view in Eclipse/STS, complete all the TODOs. Implement them as shown in Figure 1.10



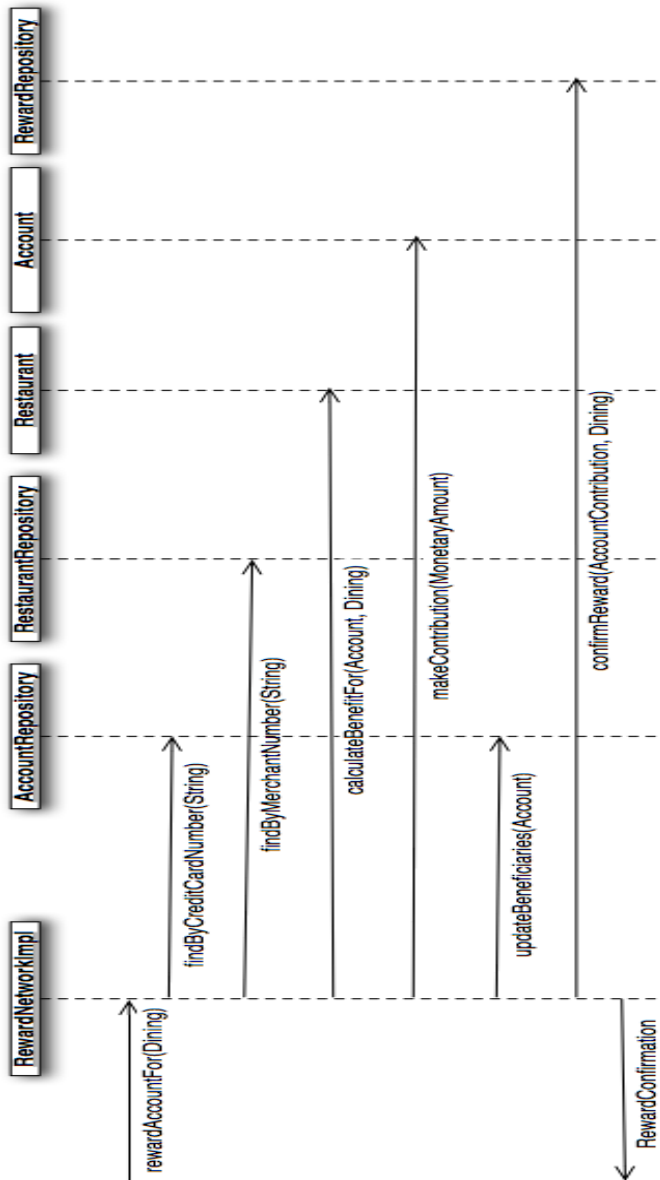


Figure 1.10. The `RewardNetworkImpl.rewardAccountFor(Dining)` sequence



## Tip

Use Eclipse's autocomplete to help you as you define each method call and variable assignment.



## Tip

You should not need to use operator `new` in your code. Everything you need is returned by the methods you use. The interaction diagram doesn't show what each call returns, but most of them return something



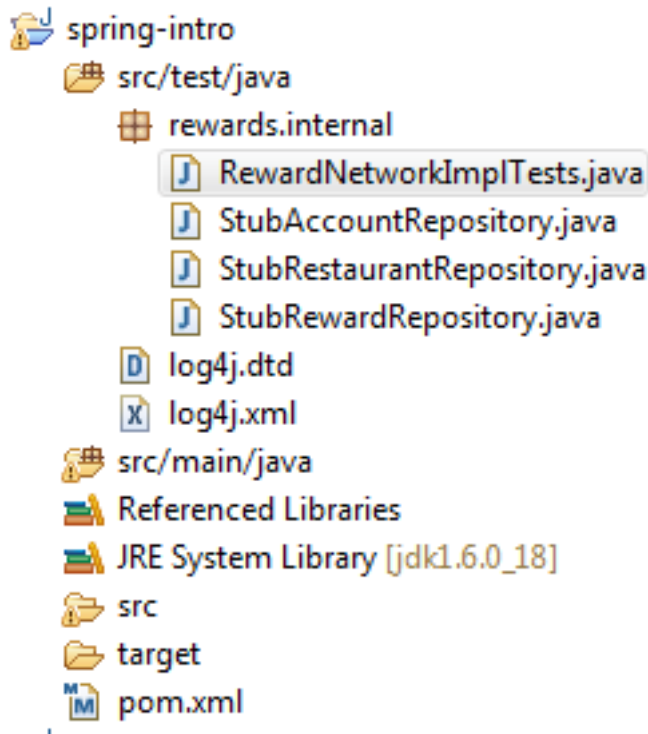
## Tip

You get the credit card and merchant numbers from the `Dining` object.

### 1.2.2.4. Unit test the `RewardNetworkImpl` application logic

How do you know the application logic you just wrote actually works? You don't, not without a test that proves it. In this step you'll review and run an automated JUnit test to verify what you just coded is correct.

Navigate into the `src/test/java` source folder and you'll see the root `rewards` package. All tests for the rewards application reside within this tree at the same level as the source they exercise. Drill down into the `rewards.internal` package and you'll see `RewardNetworkImplTests`, the JUnit test for your `RewardNetworkImpl` class.



**Figure 1.11. The `rewards` test tree**

Inside `RewardNetworkImplTests` you can notice that the `setUp()` method, 'stub' repositories have been created and injected into the `RewardNetworkImpl` class using the constructor.

Review the only test method in the class. It calls `rewardNetwork.rewardAccountFor(Dining)` and then makes assert statements to evaluate the result of calling this method. In this way the unit test is able to construct an instance of `RewardNetworkImpl` using the mock objects as dependencies and verify that the logic you implemented functions as expected.

Once you reviewed the test logic, run the test. To run, right-click on `RewardNetworkImplTests` and select *Run As -> JUnit Test*.

When you have the green bar, congratulations! You've completed this lab. You have just developed and unit tested a component of a realistic Java application, exercising application behavior successfully in a test environment inside your IDE. You used stubs to test your application logic in isolation, without involving

external dependencies such as a database or Spring. And your application logic is clean and decoupled from infrastructure APIs.

In the next lab, you'll use Spring to configure this same application from all the *real* parts, including plugging in *real* repository implementations that access a relational database.

---

# Chapter 2. Dependency Injection with JavaConfig

## 2.1. Introduction

In this lab you will gain experience using Spring to configure the completed rewards application. You'll use Spring to configure the pieces of the application, then run a top-down system test to verify application behavior.

### What you will learn:

1. The *big picture*: how Spring "fits" into the architecture of a typical Enterprise Java application
2. How to use Spring to configure plain Java objects (POJOs)
3. How to organize Spring configuration files effectively
4. How to create a Spring `ApplicationContext` and get a bean from it
5. How Spring, combined with modern development tools, facilitates a test-driven development process

### Specific subjects you will gain experience with:

1. Spring JavaConfig configuration syntax
2. Spring 3 embedded database support
3. SpringSource Tool Suite

Estimated time to complete: 45 minutes

## 2.2. Quick Instructions

If you feel you have a good understanding of the material, implement the tasks listed here. Alternatively, the next section contains more detailed step-by-step instructions. If you aren't sure at any point, each task here is also described in more detail by a corresponding section in the step-by-step instructions.

### Create the Application Configuration

## 1. Creating application config file (details)

Go to the `config` package. Select new / class, name the file `RewardsConfig` and click Finish. Note that the class does not need to extend any other classes or implement any interfaces.

Annotate the `RewardsConfig` class to mark it as a special class for providing configuration instructions. Within this class, define your four `@Bean` methods as shown below, in the 'RewardsConfig.java' box. Each method should contain the code needed to instantiate the object and set its dependencies. Since each repository has a `DataSource` property to set, and since the `DataSource` will be defined elsewhere (`TestInfrastructureConfig.java`), you will need to define a `DataSource` field / instance variable set by Spring using the `@Autowired` annotation. For consistency with the rest of the lab, give your `RewardNetworkImpl` `@Bean` method the name `rewardNetwork`.

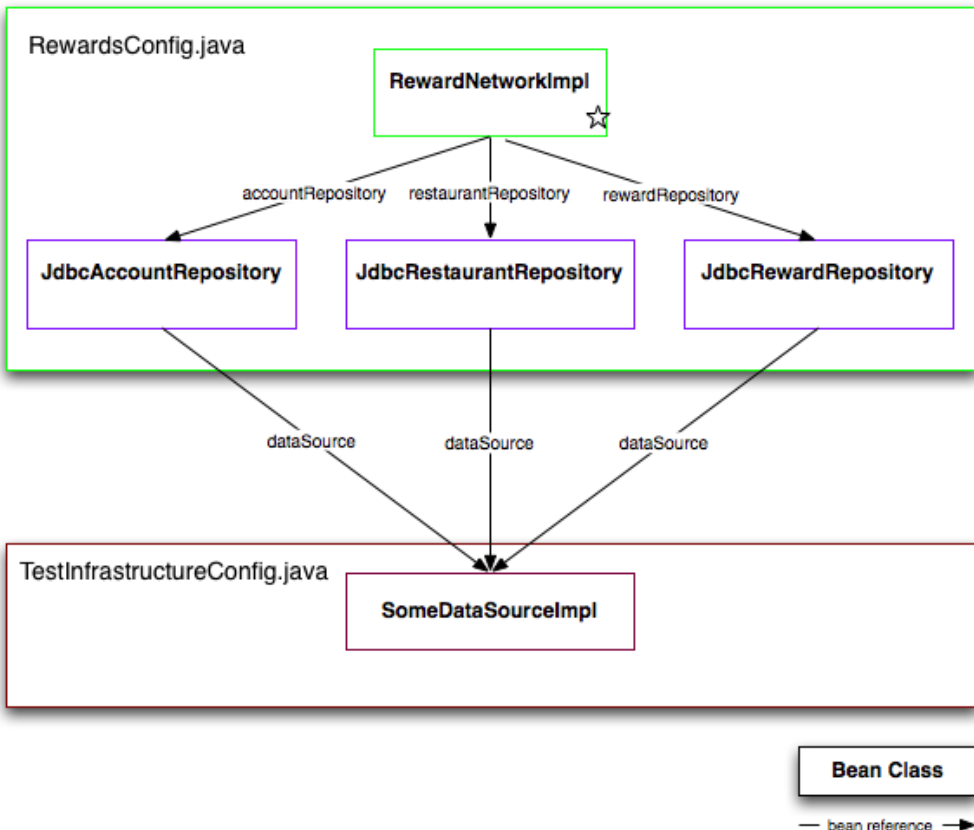


Figure 1: Application configuration

## 2. Infrastructure configuration (details)

Next review the infrastructure configuration necessary to test your application. We need a datasource for your application to use to acquire database connections in a test environment. Open `TestInfrastructureConfig.java` file and verify the datasource and database connection code.

## System testing the application with Spring and JUnit

### 1. Create the system test class (details)

Create a new JUnit test called `RewardNetworkTests` in the `rewards` package inside the `src/test/java` source folder.

### 2. Implement test setup logic (details)

Add the test setup logic required to run your system test. In this method, you will need to create an application context and configure it with the Spring Bean configuration classes you just created (`RewardsConfig.java` and `TestInfrastructureConfig.java`). In addition, you need to obtain a reference to the `rewardNetwork` bean from the application context. Assign this as a private field you can reference in your test methods.

### 3. Implement test logic (details)

Copy the unit test (the `@Test` method) from `RewardNetworkImplTests.testRewardForDining()` - we are testing the same code, but using a different setup.

### 4. Run the test

With the test setup logic implemented, you're ready to test your application. Run your new unit test. It will invoke `RewardNetwork.rewardAccountFor(Dining)` method to verify all pieces of your application work together to carry out a successful reward operation.

Congratulations the lab is finished.

### 5. Bonus: Use `@Import` (details)

Ammend the test's `@Before` method to reference only the `TestInfrastructureConfig.java` configuration class. Next, modify `TestInfrastructureConfig.java` to include the `RewardsConfig.java` using `@Import`. Re-run your test, it should still pass.

## 2.3. Detailed Instructions

Instructions for this lab are divided into two sections. In the first section, you'll use Spring to configure the pieces of the rewards application. In the second section, you'll run a system test to verify all the pieces work together to carry out application behavior successfully. Have fun!

### 2.3.1. Creating the application configuration

So far you've coded your `RewardNetworkImpl`, the central piece of this reward application. You've unit tested it and verified it works in isolation with dummy (stub) repositories. Now it is time to tie all the *real* pieces of the application together, integrating your code with supporting services that have been provided for you. In the following steps you'll use Spring to configure the complete rewards application from its parts. This includes plugging in repository implementations that use a JDBC data source to access a relational database!

Below is a configuration diagram showing the parts of the rewards application you will configure and how they should be wired together:



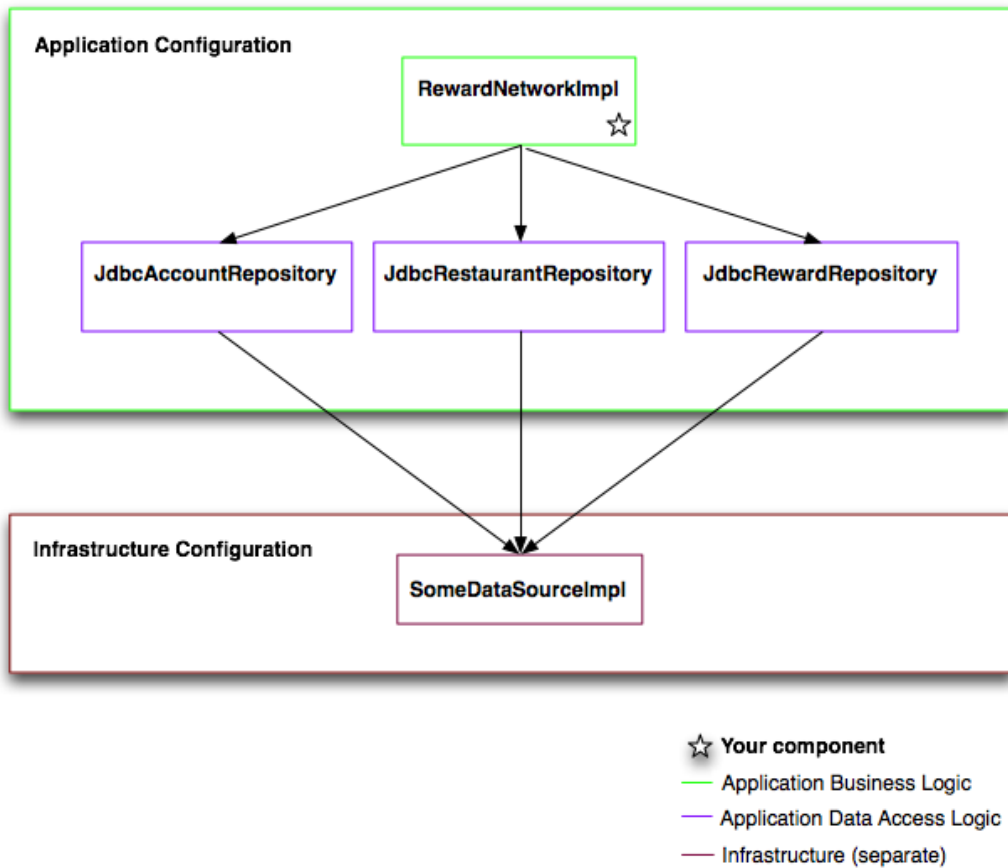


Figure 4: The rewards application configuration diagram

The diagram shows the configuration split into two categories: Application Configuration and Infrastructure Configuration. The components in the Application Configuration box are written by you and makeup the application logic. The components in the Infrastructure Configuration box are not written by you and are lower-level services used by the application. In the next few steps you'll focus on the application configuration piece. You'll define the infrastructure piece later.

In your project, you'll find your familiar `RewardNetworkImpl` in the `rewards.internal` package. You'll find each JDBC-based repository implementation it needs located within the domain packages inside the `rewards.internal` package. Each repository uses the JDBC API to execute SQL statements against a `DataSource` that is part of the application infrastructure. The `DataSource` implementation you will use is not

important at this time but will become important later.

### 2.3.1.1. Create the application configuration class

Spring configuration information is typically externalized from Java code, partitioned across one or more JavaConfig files. In this step you'll create a single JavaConfig file that tells Spring how to configure your application components.

Under the `src/main/java` folder, right-click the `config` package and select new / class, name the file `RewardsConfig` and click Finish. Note that the class does not need to extend any other classes or implement any interfaces, we will add some Spring annotations to it, however.

In the next few steps, we will add code to the `RewardsConfig` class to create the result illustrated in the 'RewardsConfig' box below:

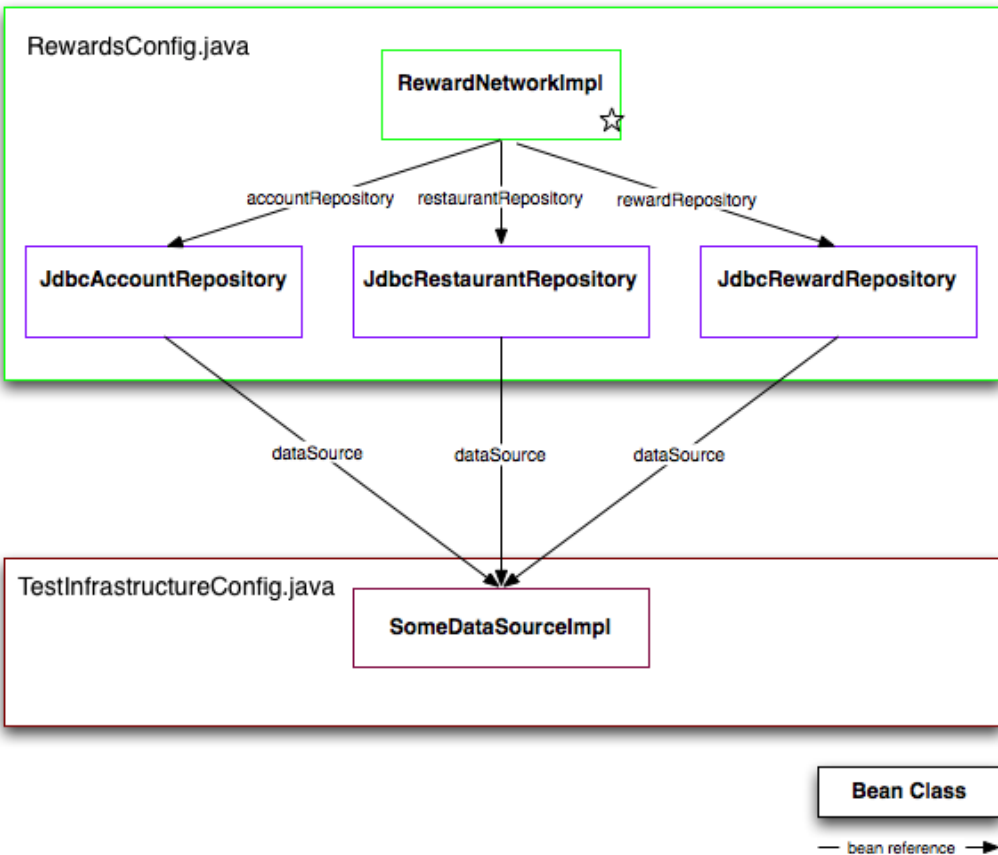


Figure 6: Application configuration

First, place a `@Configuration` annotation on the `RewardsConfig` class. This tells Spring to treat this class as a set of configuration instructions to be used when the application is starting up.

Next, within this `RewardsConfig.java` class, define four methods annotated with the `@Bean` annotation. Each method should instantiate and return one of the beans in the illustration, `accountRepository`, `restaurantRepository`, `rewardRepository`, and `rewardNetwork`. For example, you should create an `accountRepository()` method that instantiates `JdbcAccountRepository` and returns it.

Looking back at the illustration, you can see that each of the three repositories has a dependency on a `DataSource` that will be defined elsewhere. This means in each repository method we must make a call to the repository's `setDataSource()`, passing in a reference to the `DataSource`. But where will we get the `DataSource` from if it is defined in another file (like `TestInfrastructureConfig.java`)? Spring is able to populate fields on our configuration class with references to beans defined elsewhere. To do this, create a field on the class of type `DataSource` named `dataSource` and mark it with the `@Autowired` annotation. When Spring sees this, it will automatically populate this field with a reference to the `DataSource` defined in `TestInfrastructureConfig.java`, assuming both configuration files are specified at startup.

Finally, you should be aware that Spring will assign each bean and ID based on the `@Bean` method name. The instructions below will assume that the ID for the `RewardNetwork` bean is `rewardNetwork`. Therefore, for consistency with the rest of the lab, give your `RewardNetworkImpl` `@Bean` method the name `rewardNetwork`.



## Follow bean naming conventions

As you define each bean, follow bean naming conventions. The arrows in the diagram representing bean references follow the recommended naming convention.

A bean's name should describe the *service* it provides. It should not describe implementation details. For this reason, a bean's name often corresponds to its *service interface*. For example, the class `JdbcAccountRepository` implements the `AccountRepository` interface. This interface is what callers work with. By convention, then, the bean name should be `accountRepository`.



## Use Eclipse auto-completion

As you define each bean, have Eclipse auto-suggest for you. Press `Ctrl+Space` when defining a return type and Eclipse will suggest what's legal based on types available in the classpath. In-line documentation of each tag will also be displayed.

Once you have the four beans defined and referenced as shown in the diagram, move on to the next step!

### 2.3.1.2. The infrastructure configuration needed to test the application

In the previous step you visualized bean definitions for your application components. In this step we'll investigate the infrastructure configuration necessary to test your application.

To test your application, each JDBC-based repository needs a `DataSource` to work. For example, the `JdbcRestaurantRepository` needs a `DataSource` to load `Restaurant` objects by their merchant numbers from rows in the `T_RESTAURANT` table. So far, though, you have not defined any `DataSource` implementation (you can see this graphically in the graphic as the 'dataSource' references are dangling). In this step you'll see how to setup a `DataSource` in a separate configuration file in your test tree. It's in the test area, because it is only for testing - it is not the one you would use in production.

In the `src/test/java` source folder, navigate to the root `config` package. There you will find a file named `TestInfrastructureConfig.java`. Open it.

You will see that a `dataSource` is already configured for you. You don't need to make any changes to this file but you do need to understand what we have defined here for you.

Spring ships with decent support for creating `DataSources` based on in-memory databases such as H2, HSQLDB and Derby. The code you see is a quick way to create such a database.

Notice how the Builder references external files that contain SQL statements. These SQL scripts will be executed when the application starts. Both scripts are on the classpath, so you can use Spring's resource loading mechanism and prefix both of the paths with `classpath:.`. Note that the scripts will be run in the order specified (top to bottom) so the order matters in this case.

## 2.3.2. System testing the application with Spring and JUnit

In this final section you will test your rewards application with Spring and JUnit. You'll first implement the test setup logic to create a Spring `ApplicationContext` that bootstraps your application. Then you'll implement the test logic to exercise and verify application behavior.

### 2.3.2.1. Create the system test class

Start by creating a new JUnit Test Case called `RewardNetworkTests` in the `rewards` package inside the `src/test/java` source folder. Use the *New -> Other -> Java -> JUnit Test Case* wizard to help you (note that you might need to change the version of JUnit that will be used to 4):

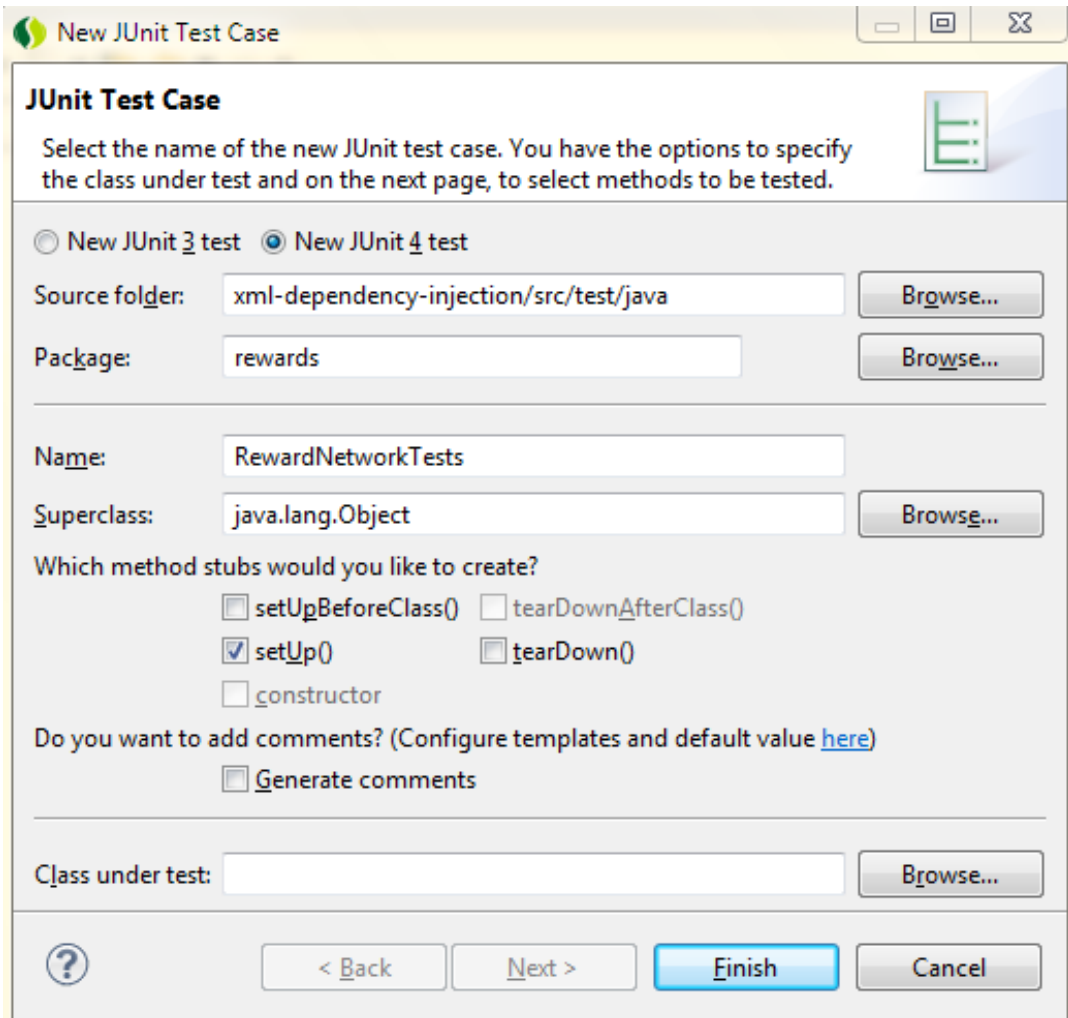


Figure 13: Creating the RewardNetworkTests TestCase using the JUnit Test Case wizard

Once you have your `RewardNetworkTests` class created, move on to the next step!

### 2.3.2.2. Implement the test setup logic

In this step you'll implement the setup logic needed to run your system test. You'll first create a `Spring ApplicationContext` that bootstraps your application, then lookup the bean you'll use to invoke the

application.

First, ensure you have a `public void setUp()` method annotated with `@org.junit.Before`. (this was done for you when you checked the `setUp()` checkbox in JUnit test case wizard)

Within `setUp()`, create a new `AnnotationConfigApplicationContext`, providing it the two configuration classes you wish to load, `RewardsConfig.class` and `TestInfrastructureConfig.class`. Doing this will bootstrap your application by having Spring create, configure, and assemble all beans defined in those two files.

Next, ask the context to get the `rewardNetwork` bean for you, which represents the entry-point into the rewards application. Assign the bean to a private field of type `RewardNetwork` you can reference from your test methods.



## Tip

Be sure to assign the reference to the `rewardNetwork` bean to a field of type `RewardNetwork` and not `RewardNetworkImpl`. A `Spring ApplicationContext` encapsulates the knowledge about which component implementations have been selected for a given environment. By working with a bean through its interface you decouple yourself from implementation complexity and volatility.



## Tip

Don't ask the context for beans "internal" to the application. The `RewardNetwork` is the application's entry-point, setting the boundary for the application defined by a easy-to-use public interface. Asking the context for an internal bean such as a repository or data source is questionable.

Now verify that Spring can successfully create your application on test `setUp`. To do this, create a public void test method called `testRewardForDining()` and annotate it with `@org.junit.Test`. Leave the method body blank for now. Then, run your test class by selecting it and accessing *Run -> Run As -> JUnit Test* from the menu bar (you may also use the *Alt + Shift + X then T* shortcut to do this). After your test runs, you should see the green bar indicating `setUp` ran without throwing any exceptions. If you see red, inspect the failure trace in the JUnit view to see what went wrong in the setup logic. Carefully inspect the stack trace- Spring error messages are usually very detailed in describing what went wrong.

Once you have the green bar, move on to the next step!

### 2.3.2.3. Implement the test logic

With the test setup logic implemented, you're ready to test your application. In this step, you'll invoke the `RewardNetwork.rewardAccountFor(Dining)` method to verify all pieces of your application work together to carry out a successful reward operation.

You will not have to write the Unit Test yourself. Have a look at `RewardNetworkImplTest.testRewardForDining()`. You can just copy and paste its content into `RewardNetworkTests.testRewardForDining()`.



## Tip

In a real life application you would not have the same content for both tests. We are making things fast here so you can focus on Spring configuration rather than spending time on writing the test itself.

You can now run your test in Eclipse. This time you may simply select the green play button on the tool bar to *Run Last Launched* (Ctrl+F11).

When you have the green bar, congratulations! You've completed this lab. You have just used Spring to configure the components of a realistic Java application and have exercised application behavior successfully in a test environment inside your IDE.

---

# Chapter 3. Configuration with annotations

## 3.1. Introduction

In this lab you will gain experience using the annotation support from Spring to configure the rewards application. You will use an existing setup and transform that to use annotations such as `@Autowired`, `@Repository` and `@Service` to configure the components of the application. You will then run a top-down system test that uses JUnit 4.

### What you will learn:

1. How to use some of Spring's dependency injection annotations such as `@Autowired`
2. The advantages and drawbacks of those annotations
3. How to implement your own bean lifecycle behaviors

### Specific subjects you will gain experience with:

1. Annotation-based dependency injection
2. How to use Spring component scanning

Estimated time to complete: 25 minutes

## 3.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> `Tasks` (*not Task List*)).

Occasionally, TODO'S defined within XML files may fail to appear in the `Tasks` view (i.e. gaps in the number sequence). To correct this, go to `Preferences -> General -> Editors -> Structured Text Editor -> Task Tags` pane. Check `Enable searching for Task Tags` and click `Clean` and `Redetect Tasks`. On the `Filters` tab, ensure XML content type is checked.

The following sequence diagram will help you to perform the TODOs for implementing the bean life cycle behaviors.



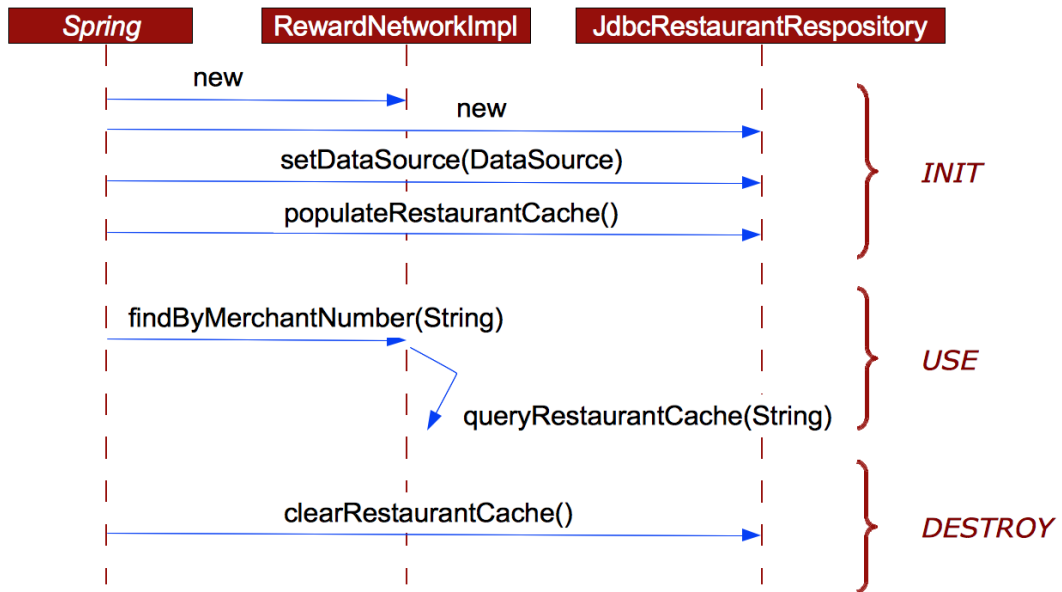


Figure 3.1.

### 3.3. Detailed Instructions

Instructions for this lab are divided into four sections. In the first section we will review an existing application configuration and make sure it works correctly. The second section you'll configure Spring's component scanning feature to create beans from the application classes automatically. In the third section, you'll implement a cache using Spring's bean lifecycle behaviours. We will use a JUnit 4 system test throughout to verify that we haven't broken anything. Have fun!

#### 3.3.1. Reviewing the application

In this lab, we are using a version of the `rewards` application that is already fully functional. It has repository implementations that are backed by JDBC and which connect to an in-memory embedded HSQLDB database. There is no transactional behavior yet, but we will learn how to define that shortly. We will then rewrite some of the application code to make use of annotations.

##### 3.3.1.1. First verify that everything works

(TODO-01) The project features an integration test that verifies the system's behavior. It's called `RewardNetworkTests` and lives in the `rewards` package. Run this test by right-clicking on it and selecting 'Run As...' followed by 'JUnit Test'. The test should run successfully.

Now open the application configuration (use `CTRL+SHIFT+T` [Windows] or `Command+SHIFT+T` [Mac] to quickly navigate to it) called `RewardsConfig.java` and review the `@Beans` that wires up all the dependencies. As you can see, we're using constructor arguments.

Remember that the infrastructure components (the `DataSource` for example) are located in a separate application configuration class. If you navigate back to the test you will see that the `setUp()` method specifies the `TestInfrastructureConfig.java` infrastructure configuration file.

### 3.3.2. Dependency injection using Spring's `@Autowired` annotation

So you've fully reviewed the entire application and you've seen nothing out of the ordinary. We're now going to wire the individual components. In `RewardsConfig.java`, remove the `@Bean` methods for all beans (TODO 02). Also remove the `@Autowired DataSource`. In other, the class should contain no methods and no variables.

Try re-running the test. It should fail now. Spring has no idea how to inject the dependencies anymore, since you have removed the configuration directive. Next, we'll start adding configuration metadata using stereotype annotations and the `@Autowired` annotation.

Open the `RewardNetworkImpl` class and annotate it with one of the available stereotypes. It is definitely not a repository or controller, so we should use `@Component` or `@Service` (`@Service` is probably more descriptive here.) Also annotate the constructor with `@Autowired` (TODO 03).

Now open up `JdbcRewardRepository` and annotate it with a stereotype annotation. Since it is a repository class, the `@Repository` annotation is the obvious choice here. Mark the `setDataSource()` method with that same `@Autowired` annotation (TODO 04). This will tell Spring to inject the setter with a instance of a bean matching the `DataSource` type.

Open the `JdbcAccountRepository` class, annotate it as a `@Repository`, and annotate the `setDataSource()` method with `@Autowired` (TODO 05).

Annotate the `JdbcRestaurantRepository` class with `@Autowired`. But this time we will use the `@Autowired` annotation on the constructor instead of a setter (TODO 06). If you take a look at the constructor you will see why, it calls a `populateRestaurantCache` method, and this method requires a reference to the `DataSource` in order to access the DB.

Although our classes are now properly annotated, we still have to tell Spring to search through our Java classes to find the annotated classes and carry out the configuration. To do this, open `RewardsConfig.java` and add the `@ComponentScan("rewards")` annotation (TODO 07). This annotation turns on a feature called component

scanning which looks for all classes annotated with annotations such as `@Component`, `@Repository` OR `@Service` and creates Spring beans from those classes. It also enables detection of the dependency injection annotations.

Once you've added this, save all your changes and re-run the test and see that it passes.

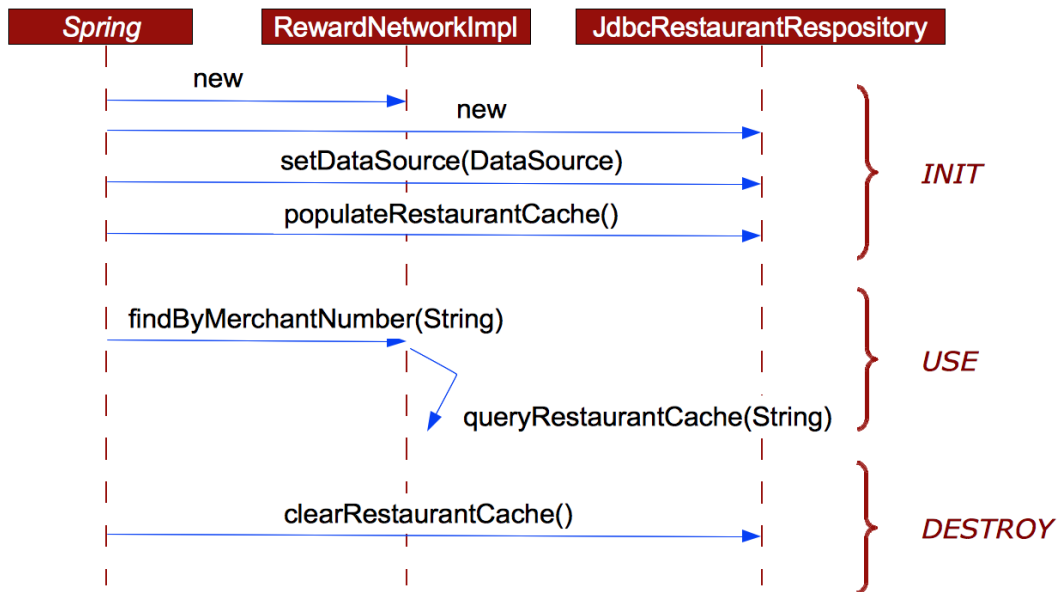
### 3.3.3. Working with Init and Destroy callbacks

In the reward dining domain, restaurant data is read often but rarely changes. You can browse `JdbcRestaurantRepository` and see that it has been implemented using a simple cache. Restaurant objects are cached to improve performance (see methods `populateRestaurantCache` and `clearRestaurantCache` for more details).

The cache works as follows:

1. When `JdbcRestaurantRepository` is initialized it eagerly populates its cache by loading all restaurants from its `DataSource`.
2. Each time a finder method is called, it simply queries Restaurant objects from its cache.
3. When the repository is destroyed, the cache should be cleared to release memory.

The full sequence is shown below.



**Figure 3.2. The `JdbcRestaurantRepository` life-cycle**

### 3.3.3.1. Initialization

Open `JdbcRestaurantRepository` in the `rewards.internal.restaurant` package. Notice that we are using the constructor to inject the dependency. You can run the test `RewardNetworkTests` and see that it works well as it is now.

However, what if you had decided to use `setter injection` instead of `constructor injection`? It is interesting to understand what happens then.

Change the dependency injection style from constructor injection to setter injection (TODO 08): Move the `@Autowired` from the constructor to the `setDataSource` method. Now, execute `RewardNetworkTests` to verify. It should fail and you should see a `NullPointerException`. Why did the test fail? Investigate the classpath to see if you can determine the root cause.

Inside `JdbcRestaurantRepository`, the default constructor is now used by Spring instead of the alternate constructor. This means the `populateRestaurantCache()` is never called. Moving this method to the default constructor will not address the issue as it requires the `datasource` to be set first. Instead, we need to cause `populateRestaurantCache()` to be executed after all initialization is complete.

Scroll to the `populateRestaurantCache` method and add a `@PostConstruct` annotation to cause Spring to call this method during the initialization phase of the lifecycle (TODO 09). You can also remove the `populateRestaurantCache()` call from the constructor if you like. Re-run the test now and it should pass.

### 3.3.3.2. Destroy

(TODO-10) Your test seems to run fine, let us now have a closer look. Open `JdbcRestaurantRepository` and add a breakpoint inside `clearRestaurantCache`. Re-run `RewardNetworkTests` in debug mode. As you can see, the method `clearRestaurantCache` is not called, which means that your cache is never cleared. As shown in your student notes, make needed changes to notify the Spring container that `clearRestaurantCache` should be called at shutdown (TODO 15).

You can then run your test one more time in debug mode. Unfortunately, it seems that your destroy callback still isn't called.

(TODO-11) Destroy callbacks are only called when the `ApplicationContext` is closed properly. Let's go back into `RewardNetworkTests`. A good way to close the `ApplicationContext` automatically is to register a shutdown hook with the JVM. A shutdown hook causes code to be executed by the JVM when the JVM itself is closing. Since we usually want a Spring application to run for the full lifetime of the JVM in which it is running, this is a good choice. Uncomment the line of code that calls `context.registerShutdownHook()`.

Run `RewardNetworkTests` in debug mode one more time. You should then stop into the breakpoint, which means that your destroy callback has been called properly.



## Tip

Later in this course, you will learn that there is a more elegant way to work with JUnit: Using the `@ContextConfiguration` annotation, Spring's `ApplicationContext` can actually be opened and closed automatically so you do not have to do it by hand.

When this is done, you've completed this section! Your repository is being successfully integrated into your application, and Spring is correctly issuing the lifecycle callbacks to populate and clear your cache. Good job!

---

# Chapter 4. XML Dependency Injection

## 4.1. Introduction

**What you will learn:**

1. Configuring a Spring application using classic XML configuration.
2. How to use XML namespaces

Estimated time to complete: 30 minutes

## 4.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> `Tasks` (*not Task List*)).

Occasionally, TODO'S defined within XML files may fail to appear in the `Tasks` view (i.e. gaps in the number sequence). To correct this, go to `Preferences -> General -> Editors -> Structured Text Editor -> Task Tags` pane. Check `Enable searching for Task Tags` and click `Clean` and `Redetect Tasks`. On the `Filters` tab, ensure XML content type is checked.

## 4.3. Detailed Instructions

### 4.3.1. First verify that everything works

(TODO-01) The project features an integration test that verifies the system's behavior. It's called `RewardNetworkTests` and lives in the `rewards` package. Run this test by right-clicking on it and selecting 'Run As...' followed by 'JUnit Test'. The test should run successfully.

This test illustrates a great advantage of using automated tests to verify that the refactoring our application is successful. We will run this test again after we make changes to the application to verify that our system functions the same as it did originally.

### 4.3.2. Convert to XML configuration / Component Scanning

(TODO 02) Open the `rewards-config.xml` file located in the `config` folder. This will serve as our main application configuration file, and will replace the configuration instructions currently in the `RewardsConfig.java` class.

Our first step will be to add the context namespace to this configuration file. This can be done manually via copy / paste, but STS provides a quicker alternative. On the bottom of the editor you should see a "Namespaces" tab. Within this tab you will see a set of checkboxes, each one represents a namespace that you can add to the XML root element. Check the "context" box (you may be prompted that this will add an element to your configuration file, which is exactly what we want, so click OK). Return to the "Source" tab and note that the XML namespace "context" has been added to your XML root element. This means you can now take advantage of the context: namespace.

(TODO 03) Now that we have added the context: namespace, we can add the element to do component scanning. If you enter "context:" and press Ctrl-space, the editor will prompt you for the possible entries in the context namespace that can be used. Select "context:component-scan".

Within the component-scan tag it is important to set the "base-package" element; this tells Spring which packages and sub-packages should be included in the scanning process. If you look at the `RewardsConfig.java` class, you can see the value presently used by the `JavaConfiguration`: "rewards". Use this same value.

(TODO 04) At this point, we have an XML configuration file equivalent of our `RewardsConfig.java` class. However, our test class is still coded to load the Java configuration files instead. We need to switch this to a variant of the `ApplicationContext` that takes XML files as parameters: `ClassPathXmlApplicationContext`.

Replace the `AnnotationConfigApplicationContext` with `ClassPathXmlApplicationContext`. For parameters, you will need to provide the paths to the two configuration files for our application. The first is the file you just modified, `rewards-config.xml` located in the `config` folder. The second is an existing XML file already prepared for you, `test-infrastructure-config.xml` located in the `rewards` folder. Note that both of these are classpath resources, so you should prefix both Strings with `classpath:` prefixes to tell Spring to look on the classpath for these resources.

Once you have finished this modification, save all your work and rerun the `RewardNetworkTests`. The test should pass at this point. If it does not, take a look at the test output to see if you can determine why. The most likely issue is the file/path literals of the configuration files.

### 4.3.3. Switch to pure XML Configuration

At this point, we are using Annotation-based configuration to define the application components (`RewardNetwork` and the three repositories) and XML configuration to define the `DataSource`. In this next section, we will demonstrate how to use a 100% XML configuration. Return to `rewards-config.xml` and perform the following.

(TODO 05) Define a bean element to instantiate the `JdbcAccountRepository`. It is good practice to give beans an ID using the "id" attribute, so give this bean the ID "accountRepository", or any other ID you like. Use the "class" attribute to specify the fully-qualified classname of what we want to instantiate, the `JdbcAccountRepository`. STS provides a great feature here to quickly determine the packaging: within the class attribute value type "JAR" (all caps) and press Ctrl-C. STS will prompt you with all known classes that match the camel-case pattern. Simply choose `JdbcAccountRepository` from the list.

Next, within the bean element start and end tags, place a property sub-element to set the `dataSource` property. The autocomplete feature is very useful here, using it you can discover that the "name" of the property we want to set is called "dataSource". We want to set this to a "ref" to another bean named "dataSource". Note that this other bean is defined elsewhere, so in this case the autocomplete feature can't help us. Also note that the editor may give you a warning that this bean is unknown for the same reason; you can safely ignore this warning for now.

(TODO 06) Define a bean element to instantiate the `JdbcRestaurantRepository`. The procedure is exactly the same as the last step, except you should select a different ID value (suggest: "restaurantRepository") and the fully-qualified classname.

Also, this class has a special method within it that must be called at startup time in order to pre-populate its cache. The method is named `populateRestaurantCache` and you should use the `init-method` attribute to specify it.

(TODO 07) Define a bean element to instantiate the `JdbcRewardRepository`. The procedure is exactly the same as the previous two steps, except a different ID should be used (suggest: "rewardRepository") and the fully-qualified classname should be different. Note there is no need for any `init-method` on this bean.

(TODO 08) Define a bean element to instantiate the `RewardNetworkImpl`. The ID for this bean should be "rewardNetwork" to allow our existing test code to work. This bean has three constructor arguments that must be populated: an `AccountRepository`, a `RestaurantRepository`, and a `RewardRepository`. These happen to be the beans defined in the previous three steps so use the `constructor-arg` sub-elements with `ref` attributes to specify these dependencies.

Now that we have defined XML bean definitions for our beans, we can remove the annotations on the classes themselves:

(TODO 09) Open `RewardNetworkImpl` and remove the `@Service` and `@Autowired` annotations.

(TODO 10) Open `JdbcAccountRepository` and remove the `@Repository` and `@Autowired` annotations.

(TODO 11) Open `JdbcRestaurantRepository` and remove the `@Repository` and `@Autowired` annotations.

(TODO 12) While in the `JdbcRestaurantRepository` remove the `@PostConstruct` annotation from the `populateRestaurantCache` method. Our XML configuration instructions will ensure that this method is called



during startup.

(TODO 13) Open `JdbcRewardRepository` and remove the `@Repository` and `@Autowired` annotations.

At this point, we have removed all of the annotation-based configuration. Save all your work, and re-run the `RewardNetworkTests`. It should pass - Spring is now using XML-based bean definitions. Congratulations, you have completed this lab.

#### **4.3.4. Bonus - Remove Component Scanning**

(TODO 14) Now that we are using XML configuration and have removed all the stereotype and DI annotations, is there any reason for the component-scanning element to remain? Remove this element and rerun the test, It should pass. You can also experiment with removing the `RewardsConfig` and `TestInfrastructureConfig` classes since they are no longer used.

---

# Chapter 5. Dependency Injection Best Practices

## 5.1. Introduction

**What you will learn:**

1. Techniques for reducing the amount of Spring configuration code
2. How to import XML namespaces
3. How to apply custom configuration behaviors to objects created by Spring

**Specific subjects you will gain experience with:**

1. Bean Definition Inheritance
2. Importing Configuration Files

Estimated time to complete: 30 minutes

## 5.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)).

Occasionally, TODO'S defined within XML files may fail to appear in the `Tasks` view (i.e. gaps in the number sequence). To correct this, go to `Preferences -> General -> Editors -> Structured Text Editor -> Task Tags` pane. Check `Enable searching for Task Tags` and click `Clean` and `Redetect Tasks`. On the `Filters` tab, ensure XML content type is checked.

## 5.3. Detailed Instructions

### 5.3.1. (TODO 01) Using bean definition inheritance to reduce the amount of configuration

Spring provides several features that help you reduce the amount of application configuration code. In this section you'll gain experience with one of them called *bean definition inheritance*.

Inside the `rewards` package, open the test class called `RewardsNetworkTests`. As you can see, the instruction `new ClassPathXmlApplicationContext("")` currently does not have any argument. You can add 2 files: `test-infrastructure-config.xml` and `application-config.xml`. Keep in mind that you should include the file paths starting from inside the classpath root folders (`src/test/java` and `src/main/java`). When done, run `RewardsNetworkTests` to make sure the file paths are correct.

Bean definition inheritance is useful when you have several beans that should be configured the same way. It lets you define the shared configuration once, then have each bean inherits from it.

In the `rewards` application, there is a case where bean definition inheritance makes sense. Recall there are three JDBC-based repositories, and each repository needs the same `dataSource`. In this section, you'll use bean definition inheritance to reduce the amount of repository configuration code.

#### 5.3.1.1. (TODO 02) Define the `abstractJdbcRepository` bean

In this step, you'll define an abstract bean that centralizes the configuration of the `dataSource` needed by each JDBC-based repository.

Inside `src/main/java` within the `rewards.internal` package open `application-config.xml`. Note how the property tag instructing Spring to set the `dataSource` is currently duplicated for each repository.

Now in `application-config.xml`, create an abstract bean named `abstractJdbcRepository` that centralizes the `dataSource` configuration.

#### 5.3.1.2. Update each repository bean definition

In `application-config.xml`, update each repository bean so it extends from your `abstractJdbcRepository` bean definition, then clean up the bean definition to remove the duplication (TODO 02).



### Tip

You can now in-line the bean tag defining each repository to save a line of code.

#### 5.3.1.3. Re-run the `RewardNetwork` system test

In this last step you will re-run your `RewardNetworkTests` to verify your configuration changes don't break your application.

Run `RewardNetworkTests` located within the `rewards` package of your test tree. When you see green, you have just verified your application still works with your changes and you've completed this section.

### 5.3.2. (TODO 03) Externalizing values to a Properties file

In this section, you'll gain experience with using the `<context:property-placeholder>` element. Specifically, you will move the configuration of your embedded-database from `test-infrastructure-config.xml` into a `.properties` file, then declare a `<context:property-placeholder>` element to apply the configuration. By doing this, you'll make it easier for administrators to safely change your configuration.

#### 5.3.2.1. Create the `.properties` file

In this step you'll create a properties file that externalizes the configuration of your `dataSource` factory bean.

Within the `rewards` package inside `src/test/java` create a file named `testdb.properties`. Add the following properties:

```
schemaLocation=classpath:rewards/testdb/schema.sql
testDataLocation=classpath:rewards/testdb/test-data.sql
```

Notice how these values match the current script values of the embedded `dataSource` in `test-infrastructure-config.xml`.

#### 5.3.2.2. Replace static property values with `${placeholders}`

In this step you will replace the static property values in your `test-infrastructure-config.xml` with placeholders.

In `test-infrastructure-config.xml`, replace each property value configured for your embedded-database with a placeholder. The placeholder name should match the respective property name in your properties file.

Once you have done this, run your `RewardNetworkTests` to see if anything broke. You should see the red bar indicating the placeholders are not yet being replaced with valid values. One more step left to complete...

#### 5.3.2.3. Declare a `<context:property-placeholder>` element

In this step you will declare a `<context:property-placeholder>` element that will replace each placeholder with a value from your properties file.

In `test-infrastructure-config.xml`, declare an instance of the `<context:property-placeholder>` element. Set its `location` attribute to point to your properties file. Remember that this configuration will be automatically detected by Spring and called before any other bean is created. No other configuration is necessary.

Now re-run your `RewardNetworkTests`. You should see the green bar indicating your placeholders are being replaced with valid values.



## Tip

Even if you get green on your first attempt, try experimenting with some failure scenarios. For example, try misspelling a placeholder, property name, or property value and see what happens.

### 5.3.3. (TODO 04) Using the `<import/>` tag to combine configuration fragments

Using the `<import/>` tag is often a good idea when working with multiple configuration files. In this section you will refactor your configurations to use this tag and see the strengths of this technique.

#### 5.3.3.1. Refactor the system test configuration

Open `RewardNetworkTests`. Note how all the configuration files required to run the system test are listed in this file. Now suppose you added another configuration file. You would have to update your test code to accommodate this change. Now consider a production web environment. In that environment you'd also have to update your `web.xml` file any time the structure of your application configuration changed.

The `import` tag allows you to create a single 'master' configuration file for each environment that imports everything else. This technique can simplify the code needed to bootstrap your application and better insulate you from changes in your application configuration structure.

In this step you will refactor your system test to include a single 'master' configuration file that imports everything else.

Now use the `<import/>` tag to import the application configuration. Re-run `RewardNetworkTests` to verify your configuration changes don't break your application. When you see green, you have verified your application works with your improved configuration design. Good Job!



## Tip

You might consider renaming the `test-infrastructure-config.xml` file to `system-test-config.xml`. Such naming indicates that this file fully defines the configuration needed to run the system test.

---

# Chapter 6. Integration Testing with Profiles

## 6.1. Introduction

In this lab you will refactor the `RewardNetworkTests` using Spring's system test support library to simplify and improve the performance of your system. You will then use Spring profiles to define multiple tests using different implementations of the `AccountRepository`, `RestaurantRepository` and `RewardRepository` for different environments.

### What you will learn:

1. The recommended way of system testing an application configured by Spring
2. How to write multiple test scenarios

### Specific subjects you will gain experience with:

1. JUnit
2. Spring's `TestContext` framework
3. Spring Profiles

Estimated time to complete: 30 minutes

## 6.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> `Tasks` (*not Task List*)).

Occasionally, TODO'S defined within XML files may fail to appear in the `Tasks` view (i.e. gaps in the number sequence). To correct this, go to `Preferences -> General -> Editors -> Structured Text Editor -> Task Tags` pane. Check `Enable searching for Task Tags` and click `Clean` and `Redetect Tasks`. On the `Filters` tab, ensure XML content type is checked.

## 6.3. Detailed Instructions

### 6.3.1. Refactor to use Spring's TestContext framework (TODO 01)

In `rewards.RewardNetworkTests` we setup our test-environment using Spring in the `@Before setUp` method. Instead we are going to use Spring's test-runner. Comment out the `@Before` method (highlight the method and use `Ctrl+Shift+C`). Now run the test. You will get a Red bar because the `rewardNetwork` field is null.

One of the central components in the TestContext framework is the `SpringJUnit4ClassRunner`. Your next step is to tell JUnit to run your test with it and then refactor your test as necessary to work with it. You will need to add 3 annotations just like the example in the notes. If you are not sure how to do this - refer to the tips below.



#### Tip

In `RewardNetworkTests` add an `@RunWith` annotation passing in `SpringJUnit4ClassRunner.class` as the default property. Be sure to use `Ctrl+Space` to get code completion of this long class name (for example, by typing `SpJ` and then pressing `Ctrl+Space`)

Now annotate the test with `@ContextConfiguration`. Set the `locations` property of the annotation to the same Spring Bean file you used in your `setUp` method. Spring's test runner will automatically create (and cache) an `ApplicationContext` for you.

One more change left to make: annotate the `rewardNetwork` field with `@Autowired`.

Now when you run your test the test runner's setup logic will use *auto-wiring* on your test class to set values from the `ApplicationContext`. This means your `rewardNetwork` will be assigned to the `RewardNetwork` bean from the context automatically.

Re-run your test in Eclipse and verify you get a green bar. If so, the `rewardNetwork` field is being set properly for you. If you don't see green, try to figure out where the problem lies. If you can't figure it out, ask the instructor to help you find the issue.



#### Note

When you have the green bar, you've successfully reconfigured the rewards integration test, and at the same time simplified your system test by leveraging Spring's test support. In addition, the performance of your system test has potentially improved as the `ApplicationContext` is now created once per test case run (and cached) instead of once per test method. This test only has one method so it doesn't make any difference here.

We can clear up what we no longer need by deleting the `context` field and removing the `@Before` and `@After` methods.

Rerun the test and check that "Clearing restaurant cache" appears on the console - this means the `@PreDestroy` method is still being invoked by Spring.

### 6.3.2. Configure Repository Implementations using Profiles

We are now going to modify the test to use different repository implementations - either Stubs or using JDBC.

First we are going to use the stub repositories in `/src/test/java/rewards/internal`. We need to make them Spring beans by annotating them as repository components. Follow `TODO 02` and annotate the stub classes with `@Repository`.

If you run `RewardNetworkTests` again, it should fail because you have multiple beans of the same type. To fix this we introduce two profiles. The stub repositories will belong to the "stub" profile and the existing repositories to the "jdbc" profile.

Follow all the `TODO 03` steps and use the `@Profile` annotation to put all the repositories in this project into their correct profile - there are 6 repository classes to annotate in total.

Finally annotate the `RewardNetworkTests` class with `@ActiveProfiles` to make "stub" the active profile. Rerun the test - it should work now. Check the console to see that the stub repository implementations are being used. Notice that the embedded database is also being created even though we don't use it. We will fix this soon.

Switch the active-profile to "jdbc" instead (`TODO 04`). Rerun the test - it should still work. Check the console again to see that the JDBC repository implementations are being used.

### 6.3.3. Switching between Development and Production Profiles

Profiles allow different configurations for different environments such as development, testing, QA (Quality Assurance), UAT (User Acceptance Testing), production and so forth. In the last step we will introduce two new profiles: "jdbc-dev" and "jdbc-production". In both cases we will be using the JDBC implementations of our repositories so two profiles will need to be active at once.

The difference between development and production is typically different infrastructure. In this case we are going to swap between an in-memory test database and the "real" database defined as a JNDI resource.

Modify `TestInfrastructureDevConfig.java` so that all the beans are members of the profile called "jdbc-dev" (`TODO 05`).

Does `RewardNetworkTests` still run OK? Why not?

Fix the test by adding the "jdbc-dev" profile to the `@ActiveProfiles` annotation in `RewardNetworkTests` (`TODO 06`). Remember you will need to retain the "jdbc" profile as well. Rerun the test - it should work again.



We have already setup the production `dataSource` for you using a JNDI lookup (see `TODO 07`). We have used a standalone JNDI implementation - normally JNDI would be provided by your JEE container (such as Tomcat or tc Server).

Change the active profile of `RewardNetworkTests` from "jdbc-dev" to "jdbc-production". Rerun the test, it should still work. To see what has changed, look at the console and you will see logging from an extra bean called `SimpleJndiHelper`. Switch the profile back to "jdbc-dev" and rerun. Check the console and note that the `SimpleJndiHelper` is no longer used.

### 6.3.4. Optional Step - Further Refactoring

When no class or XML file is specified, `@ContextConfiguration` will look for an inner class marked with `@Configuration` (If none is found it will also look for an XML file name of `<Classname>-context.xml`). Since the `TestInfrastructureConfig` class is so small anyway, copy the entire class definition, including annotations, to an inner class within the test class. Then remove the configuration class reference from the `@ContextConfiguration` annotation (no property in the brackets). This is an example of convention over configuration. Does the test still run?

---

# Chapter 7. Introducing Aspect Oriented Programming

## 7.1. Introduction

In this lab you will gain experience with aspect oriented programming (AOP) using the Spring AOP framework. You'll add cross-cutting behavior to the rewards application and visualize it.

### What you will learn:

1. How to write an aspect and weave it into your application

Estimated time to complete: 35 minutes

## 7.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)).

## 7.3. Detailed Instructions

### 7.3.1. Creating and Testing a simple Aspect (@Before advice)

Up until now you have used Spring to configure and test your main-line application logic. Real-world enterprise applications also demand supporting services that cut across your main-line logic. An example would be logging: there may be many places in your application where you need to log data for monitoring. Historically, this may have lead to copying-and-pasting code, or entangling your application code with infrastructure. Today, you turn to aspect oriented programming (AOP). In the following steps you will create an aspect to monitor your application's data access performance.

#### 7.3.1.1. Create an aspect

In this step you will create a simple logging aspect. First you will define the logging behavior, then the rules about where the behavior should be applied. You'll use the `@Aspect` style.

(TODO-01) The definition of the aspect has already been started for you. Find it in the `rewards.internal.aspects` package.

Open the `LoggingAspect.java` file and you'll see several TODOs for you to complete. First, complete the step by annotating the `LoggingAspect` class with the `@Aspect` annotation. That will indicate this class is an aspect that contains cross-cutting behavior called "advice" that should be woven into your application.

The `@Aspect` annotation marks the class as an aspect, but it is still not a Spring bean. Component scanning can be very effective for aspects, so mark this class with the `@Component` annotation. This object requires constructor injection, so mark the constructor with an `@Autowired` annotation. We will see where this dependency comes from and turn on the actual component scanning in a later step.

(TODO-02) We aren't interested in monitoring *every* method of your application, though, only a subset. At this stage, you're only interested in monitoring the `find*` methods in your repositories, the objects responsible for data access in the application.

Therefore, define a pointcut expression that match all the `find*` methods (such as `findByCreditCard()`) in the `AccountRepository`, `RestaurantRepository`, or `RewardRepository` interfaces. Use `@Before` advice, and implement the `implLogging()` method that takes a `JoinPoint` object as a parameter, and logs information about the target objects invoked during the application execution.

### 7.3.1.2. Configure Spring to weave the aspect into the application

(TODO-03) Now that your aspect has been defined, you will create the Spring configuration needed to weave it into your application.

inside `config/AspectsConfig.java`, add an annotation to scan for components **ONLY** in the `rewards.internal.aspect` package. This will cause your `LoggingAspect` to be detected and deployed as a Spring bean.

(TODO-04) Next, add the `@EnableAspectJAutoProxy` tag to this file. This instructs Spring to process beans that have the `@Aspect` annotation by weaving them into the application using the proxy pattern. This weaving behavior is shown graphically below:

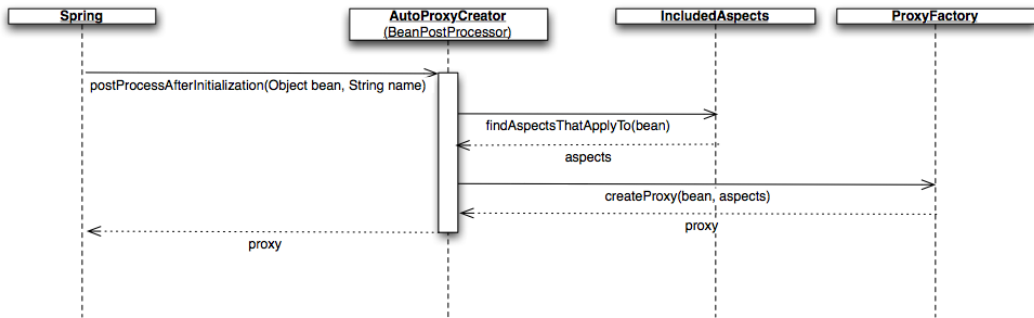


Figure 1: Spring's auto proxy creator weaving an aspect into the application using the proxy pattern

Figure 2 shows the internal structure of a created proxy and what happens when it is invoked:

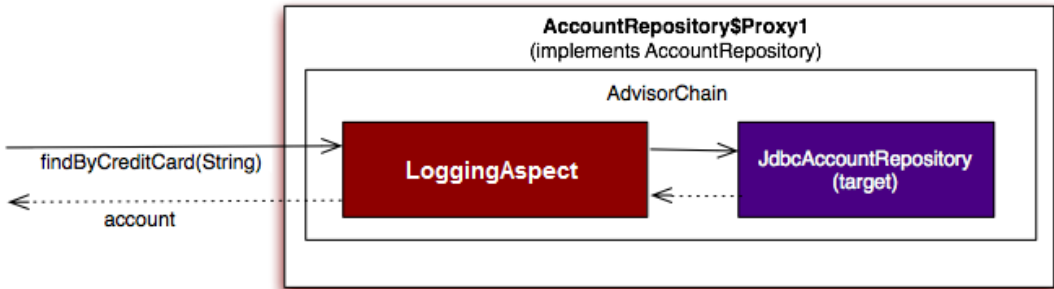


Figure 2: A proxy that applies logging behaviour to a `JdbcAccountRepository`

When you have your aspect defined as a Spring bean along with the `autoproxy` annotation, move on to the next step!

### 7.3.1.3. Test the Aspect Implementation

To see this aspect in action, plug it into the application's system test configuration. To do that, simply adjust the `@Import` to include `AspectsConfig.class` in the `SystemTestConfig.java` configuration class (TODO-05).

After the configuration file has been added, run `LoggingAspectTest` in STS and watch the console.

```

INFO : rewards.internal.aspects.LoggingAspect -
'Before' Advice implementation - class rewards.internal.account.JdbcAccountRepository;
Executing before findByCreditCard() method
    
```

When you see the logging output, your aspect is being applied. Move on to the next step!

### 7.3.2. Performance Monitor Aspect

You will implement an `Around Advice` which logs the time spent in each of your `update repository` methods.

- Modify the `LoggingAspect` class, and implement the `monitor(ProceedingJoinPoint)` method. The method should start a monitor, proceed with the repository invocation, stop the monitor after the invocation returns, and log a report.
- (TODO-06) Specify `@Around` advice for the `monitor` method. Define a pointcut expression that matches all the `update*` methods (such as `JdbcAccountRepository.updateBeneficiaries(...)` on the `AccountRepository`, `RestaurantRepository`, or `RewardRepository` interfaces.
- (TODO-07) Now in `monitor(ProceedingJoinPoint)` method, notice the `Monitor` start and stop logic has already been written for you. What has not been written is the logic to proceed with the target method invocation after the watch is started. Complete this step by adding the `proceed` call.



#### Tip

Remember, the call to `repositoryMethod.proceed()` returns the target method's return value. Make sure to return that value out, otherwise you may change the value returned by a repository!

- (TODO-08) Once you've added the `proceed` call, run the `RewardNetworkTests` class in the test tree. If you can see relevant logging information in the console, your monitoring behavior has been implemented correctly.

### 7.3.3. Exception Handling Aspect (Optional)

Create an exception handling aspect as follows:

- (TODO-09) Modify the `DBExceptionHandlerAspect` class by implementing the method `implExceptionHandler(Exception e)` to report an exception.
- Specify `@AfterThrowing` advice on top of this method. Define a pointcut expression that matches all the methods in the three repositories (regardless of the method names).
- (TODO-10) Although this class is presently marked as an `@Aspect`, it isn't defined as a `@Component`, and therefore it is not picked up when component scanning. Change this by simply adding a `@Component` annotation to the top of the class.

After the configuration has been added, run `DBExceptionHandlerAspectTests` in Eclipse and watch the console. If you can see relevant logging information in the console, your exception handling behavior has been implemented correctly.

Congratulations, you've completed the lab!

---

# Chapter 8. JDBC Simplification using the JdbcTemplate

## 8.1. Introduction

In this lab you will gain experience with Spring's JDBC simplification. You will use a `JdbcTemplate` to execute SQL statements with JDBC.

### What you will learn:

1. How to retrieve data with JDBC
2. How to insert or update data with JDBC

### Specific subjects you will gain experience with:

1. The `JdbcTemplate` class
2. The `RowMapper` interface
3. The `ResultSetExtractor` interface

Estimated time to complete: 45 minutes

## 8.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)).

## 8.3. Detailed Instructions

### 8.3.1. Refactoring a repository to use JdbcTemplate

The goal for this lab is to refactor the existing JDBC based repositories from their standard try, catch, finally, try, catch paradigm to using Spring's `JdbcTemplate`. The first repository to refactor will be the

`JdbcRewardRepository`. This repository is the easiest to refactor and will serve to illustrate some of the key features available because of Spring's simplification.

#### 8.3.1.1. Use `JdbcTemplate` in a test to verify insertion

(TODO-01) Before making any changes to `JdbcRewardRepository`, let's first ensure the existing functionality works by implementing a test. Open `JdbcRewardRepositoryTests` in the `rewards.internal.reward` package and notice the `getRewardCount()` method. In this method use the `JdbcTemplate` included in the test fixture to query for the number of rows in the `T_REWARD` table and return it.

(TODO-02) In the same class, find the `verifyRewardInserted(RewardConfirmation, Dining)` method. In this method, use the `JdbcTemplate` to query for a map of all values in the `T_REWARD` table based on the `confirmationNumber` of the `RewardConfirmation`. The column name to use for the `confirmationNumber` in the where clause is `CONFIRMATION_NUMBER`.

Finally run the test class. When you have the green bar, move on to the next step.

#### 8.3.1.2. Refactor `JdbcRewardRepository` to use `JdbcTemplate`

(TODO-03) We are now going to refactor an existing Repository class so it can use the `JdbcTemplate`. To start find the `JdbcRewardRepository` in the `rewards.internal.reward` package. Open the class and add a private field to it of type `JdbcTemplate`. In the constructor, instantiate the `JdbcTemplate` and assign it to the field you just created.

Next refactor the `nextConfirmationNumber()` method to use the `JdbcTemplate`. This refactoring is a good candidate for using the `queryForObject(String, Class<T>, Object...)` method.



### Tip

The `Object...` means a variable argument list in Java5. A variable argument list allows you to append an arbitrary number of arguments to a method invocation, including zero.

Next refactor the `confirmReward(AccountContribution, Dining)` method to use the template. This refactoring is a good candidate for using the `update(String, Object...)` method.

Once you have completed these changes, run the test class again (`JdbcRewardRepositoryTests`) to ensure these changes work as expected. When you have the green bar, move on to the next step.

### 8.3.2. Using a `RowMapper` to create complex objects

#### 8.3.2.1. Use a `RowMapper` to create `Restaurant` objects



(TODO-04) In many cases, you'll want to return complex objects from calls to the database. To do this you'll need to tell the `JdbcTemplate` how to map a single `ResultSet` row to an object. In this step, you'll refactor `JdbcRestaurantRepository` using a `RowMapper` to create `Restaurant` objects.

Before making any changes, run the `JdbcRestaurantRepositoryTests` class to ensure that the existing implementation functions correctly. When you have the green bar, move on to the next step.

Next, find the `JdbcRestaurantRepository` in the `rewards.internal.restaurant` package. Open this class and again modify it so that it has a `JdbcTemplate` field.

Next create a private inner class called `RestaurantRowMapper` that implements the `RowMapper` interface. Note that this interface has a generic type parameter that should be populated in the implementation. If you've implemented the interface correctly, the class and method declarations should look like Figure 1. The implementation of the `mapRow(ResultSet, int)` method should delegate to the `mapRestaurant(ResultSet)` method.

```
private class RestaurantRowMapper implements RowMapper<Restaurant> {  
  
    public Restaurant mapRow(ResultSet rs, int rowNum) throws SQLException {
```

Figure 1: `RestaurantRowMapper` class and method declarations

Next refactor the `findByMerchantNumber(String)` method to use the template. This refactoring is a good candidate for using the `queryForObject(String, RowMapper<T>, Object...)` method.

Finally run the `JdbcRestaurantRepositoryTests` class. When you have the green bar, move on to the next step.

### 8.3.3. Using a `ResultSetExtractor` to traverse a `ResultSet`

#### 8.3.3.1. Use a `ResultSetExtractor` to traverse a `ResultSet` for creating `Account` objects

(TODO-05) Sometimes when doing complex joins in a query you'll need to have access to an entire result set instead of just a single row of a result set to build a complex object. To do this you'll need to tell the `JdbcTemplate` that you'd like full control over `ResultSet` extraction. In this step you'll refactor `JdbcAccountRepository` using a `ResultSetExtractor` to create `Account` objects.

Before making any changes run the `JdbcAccountRepositoryTests` class to ensure the existing implementation functions properly. When you have the green bar, move on.

Next find the `JdbcAccountRepository` in the `rewards.internal.account` package. Open this class and again modify it so that it has a field of type `JdbcTemplate`.

In this repository there are two different methods that need to be refactored. Start by refactoring the `updateBeneficiaries(Account)` method to use the `JdbcTemplate`. This refactoring is very similar to the one that you did earlier for the `JdbcRewardRepository`.

Next create a private inner class called `AccountExtractor` that implements the `ResultSetExtractor` interface. Note that this interface also has a generic type parameter that should be populated. The implementation of the `extractData(ResultSet)` method should delegate to the `mapAccount(ResultSet)` method.

Next refactor the `findByCreditCard(String)` method to use the template. This refactoring is a good candidate for using the `query(String, ResultSetExtractor<T>, Object...)` method.

Finally run the `JdbcAccountRepositoryTests` class. When you have the green bar, you've completed the lab!



### Tip

Note that all three repositories still have a `DataSource` field. Now that you're using the constructor to instantiate the `JdbcTemplate`, you do not need the `DataSource` field anymore. For completeness' sake, you can remove the `DataSource` fields if you like.

---

# Chapter 9. Transaction Management with Spring

## 9.1. Introduction

In this lab you will gain experience with using Spring's declarative transaction management to open a transaction on entry to the application layer and participate in that transaction during all data access. You will use the `@Transactional` annotation to denote what methods need to be decorated with transactionality.

### What you will learn:

1. How to identify where to apply transactionality
2. How to apply transactionality to a method

### Specific subjects you will gain experience with:

1. The `@Transactional` annotation
2. The `PlatformTransactionManager` interface
3. The `<tx:annotation-driven/>` bean definition
4. Using transactional integration tests

Estimated time to complete: 25 minutes

## 9.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)).

## 9.3. Detailed Instructions

The goal of this lab is to declaratively add transactionality to the rewards application. The lab will be divided into two parts. In the first part you will add transactionality to the application and visually verify that your test case opens a single transaction for the entire use-case. In the second section you will experiment with some of the settings for transaction management and see what outcomes they produce.

### 9.3.1. Demarcating Transactional Boundaries in the Application

Spring offers a number of ways to configure transactions in an application. In this lab we're going to use a strategy that leverages annotations to identify where transactionality should be applied and what configuration to use.

#### 9.3.1.1. Add `@Transactional` annotation

(TODO-01) Find and open the `RewardNetworkImpl` class in the `rewards.internal` package. In that class locate the `rewardAccountFor(Dining)` method and add an `@Transactional` annotation to it. Adding the annotation will identify this method as a place to apply transactional semantics at runtime.

TODO-02 Next we need to configure the platform transaction manager. Navigate to the `SystemTestConfig` configuration class and wire up a `DataSourceTransactionManager`. Remember to set the `dataSource` property on this bean definition.

(TODO-03) Finally, find and open the `RewardsConfig.java` file in the `config` package. In this class you'll need to tell the container to look for the `@Transactional` annotation you just placed on the `RewardNetworkImpl` class. To do this add a `@EnableTransactionManagement` annotation.

#### 9.3.1.2. Verify transactional behavior

Verify that your transaction declarations are working correctly by running the `RewardNetworkTests` class from the `src/test/java` source folder. You should see output that looks like below. The important thing to note is that only a single connection is acquired and a single transaction is created.

```
... Acquired Connection [org.hsqldb.jdbc.  
    jdbcConnection@59fb21] for JDBC transaction  
... Switching JDBC Connection [org.hsqldb.jdbc.  
    jdbcConnection@59fb21] to manual commit  
... Initiating transaction commit  
... Committing JDBC transaction on Connection [org.hsqldb.  
    jdbc.jdbcConnection@59fb21]  
... Releasing JDBC Connection [org.hsqldb.jdbc.  
    jdbcConnection@59fb21] after transaction
```

If your test completes successfully and you've verified that only a single connection and transaction are used, you've completed this section. Move on to the next one.

### 9.3.2. Configuring Spring's Declarative Transaction Management

Setting up Spring's declarative transaction management is pretty easy if you're just using the default propagation setting (`Propagation.REQUIRED`). However, there are cases when you may want to suspend an existing transaction and force a certain section of code to run within a *new* transaction. In this section, you will

adjust the configuration of your reward network transaction in order to experiment with `Propagation.REQUIRES_NEW`.

### 9.3.2.1. Modify Propagation Behavior

(TODO-04) Find and open `RewardNetworkPropagationTests` from the `rewards` package in the `src/test/java` source folder. Take a look at the test in the class. This test does a simple verification of data in the database, but also does a bit of transaction management. The test opens a transaction at the beginning, (using the `transactionManager.getTransaction(...)` call). Next, it executes `rewardAccountFor(Dining)`, then rolls back the transaction, and finally tests to see if data has been correctly inserted into the database. Now run the test class with JUnit. You'll see that the test has failed because the rollback removed all data from the database, including the data that was created by the `rewardAccountFor(Dining)` method.

(TODO-05) The `rewardAccountFor(Dining)` was created with a propagation level of `Propagation.REQUIRED` which means that it *will participate in any transaction that already exists*. When the manually created transaction was rolled back it destroyed the data from the `@Transactional` method. In real life, it actually would generally be appropriate for this method to be marked as `Propagation.REQUIRED`, with the test being considered inappropriate, but this affords us a chance to test the results of changing the propagation settings.

Find and open `RewardNetworkImpl` and override the default propagation behavior with `Propagation.REQUIRES_NEW`. Run the `RewardNetworkPropagationTests`. If you get the green bar, you have verified that the test's transaction was suspended and the `rewardAccountFor(Dining)` method executed in its own transaction. You've completed this section. Move on to the next one.

## 9.3.3. Developing Transactional Tests

When dealing with persistent data in a test scenario, it can be very expensive to ensure that preconditions are met before executing a test case. In addition to being expensive, it can also be error prone with later tests inadvertently depending on the effects of earlier tests. In this section you'll learn about some of the support classes Spring provides for helping with these issues.

### 9.3.3.1. Use `@Transactional` to isolate test cases

First, back out your propagation changes from the previous section (change the propagation back to `Propagation.REQUIRED` instead of `Propagation.REQUIRES_NEW`. This is the appropriate propagation setting for this method.

(TODO-06) Find and open `RewardNetworkSideEffectTests` from the `rewards` package in the `src/test/java` source folder. Take a look at the two tests in the class. You'll notice that they simply call the `rewardAccountFor(Dining)` method, pass in some data, and verify that the data was recorded properly. Now run the test class with JUnit. You'll see that the second test method failed with an error that Annabelle's savings was 8.0, when 4.0 was expected. The reason we see this is because the data committed from the first test case

has violated the preconditions for the second test case.

The good news is that Spring has a facility that can help you to avoid this corruption of test data in a `DataSource`. You can simply annotate your test methods, or even your test class itself to apply to all methods, with `@Transactional`: this wraps each test case in its own transaction and rolls back that transaction when the test case is finished. The effect of this is that data is never committed to the tables and therefore, the database is in its original state for the start of the next test case. Now annotate the `RewardNetworkSideEffectTests` class with `@Transactional`. Run the test again and notice that there is now a green bar. Because the changes made by the first test were rolled back, the second test got the results it expected.

Congratulations, you're done with the lab!

---

# Chapter 10. JPA Simplification using Spring

## 10.1. Introduction

In this lab you will implement the repositories of the rewards application with the Java Persistence API (JPA). You'll configure JPA to map database rows to objects, use JPA APIs to query objects, and write tests to verify mapping behavior.

### What you will learn:

1. How to configure domain objects with annotations to map these to relational structures
2. How to use the JPA APIs to query objects
3. How to configure JPA in a Spring environment
4. How to test JPA-based repositories

### Specific subjects you will gain experience with:

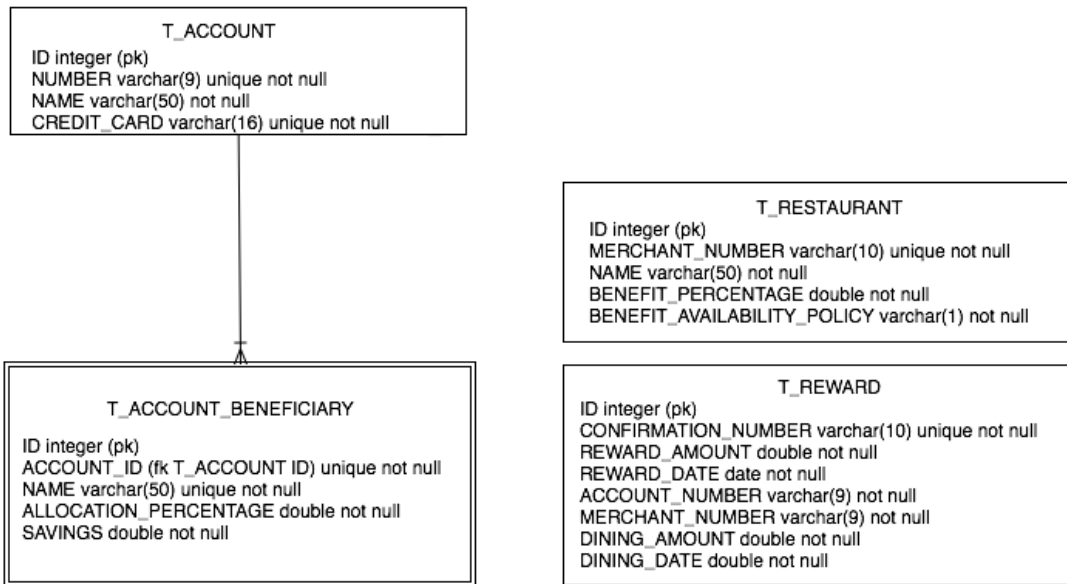
1. JPA Annotations
2. `EntityManagerFactory` and `EntityManager`
3. `LocalContainerEntityManagerFactoryBean`

Estimated time to complete: 60 minutes

## 10.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)).

Refer to `schema.sql` for help with table and column names.



The database schema for this lab, showing the credit card number as part of the account table.

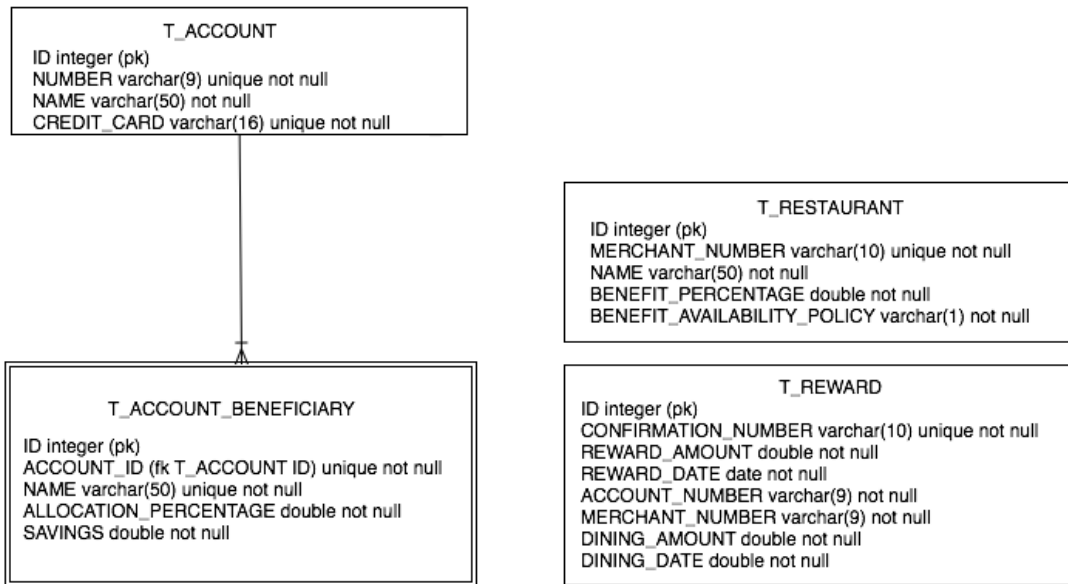
**Figure 10.1. Rewards Database Schema**

## 10.3. Detailed Instructions

The instructions for this lab are organized into three sections. The first two sections focus on using JPA within a *domain module* of the application. The first addresses the `Account` module, and the second addresses the `Restaurant` module. In each of these sections, you'll map that module's domain classes using JPA annotations, implement a JPA-based repository, and unit test your repository to verify JPA mapping behavior. In the third and final section, you'll integrate JPA (with Hibernate) into the application configuration and run a top-down system test to verify application behavior.

In the `Account` module there has been a design change in the underlying database that is relevant to the JPA mapping. The design team for the reward network has determined that each account only ever has one credit card number. The schema has been modified to reflect the modeling change (see Figure 10.2 below), and you can use this fact to simplify the JPA mapping of the `Account` entity.





The database schema for this lab, showing the credit card number as part of the account table.

**Figure 10.2. Rewards Database Schema**

## 10.3.1. Using JPA in the Account module

### 10.3.1.1. Configure the `Account` mapping annotations

Recall the `Account` entity represents a member account (a diner) in the reward network that can make contributions to its beneficiaries. In this step, you'll configure the JPA mapping annotations that map the `Account` object graph to the database.

1. Open `Account` class in the `rewards.internal.account` package. As you can see this class doesn't have any annotations in it. Let's add JPA annotations to specify the object-to-relational mapping rules.
2. (TODO-01) Add an `Entity` annotation on this class to specify that this class will be loaded from the database using the JPA implementation. As a default JPA treats the class name as the table name (in this case it would be `ACCOUNT`), you need to add a `Table` annotation to specify the table to use - in this case our database table is `T_ACCOUNT`.

3. (TODO-02) Every entity needs a primary key so it can be managed by JPA. Every table in this lab uses auto-generated numeric keys. A long integer `entityId` field has been added to each classes to be mapped. Annotate the `entityId` as the `Id` for this class.
4. We need to tell JPA to let the database automatically generate each primary key for us. Add a `GeneratedValue` annotation to the `entityId`.
5. (TODO-03) By default JPA uses the field name as the column name to map a field into a database table. Because the column name doesn't match with the field name, we need to override it desired field name.
6. Complete the mapping for the remaining `Account` fields - the `number`, `name`, `beneficiaries`, and `creditCardNumber` properties. Use the reward dining database schema in Figure 1 to help you.



## Tip

Since an `Account` can have many beneficiaries, its `beneficiaries` property is a collection. Map this property as a Java `Set` with a `OneToMany` annotation. The foreign key column in the beneficiary table is `ACCOUNT_ID`.

When you have finished mapping the aggregate `Account` entity, move on to mapping of its `Beneficiary` associate. Recall that an `Account` distributes contributions to its beneficiaries based on an allocation percentage.

1. (TODO-04) Annotate the `Beneficiary` class as a JPA entity and specify its table.
2. Add the mappings for the `entityId` and `name` fields - refer back to what you have already done for `Account`.
3. (TODO-05) Finally map the `savings`, and `allocationPercentage` fields.



## Tip

Note the beneficiary `savings` and `allocationPercentage` fields are of the custom types `MonetaryAmount` and `Percentage` respectively. Out of the box, JPA does not know how to map these custom types to database columns. It is possible to define custom getters and setters (used only by JPA) to do the conversion. However there is a simpler way - using `@Embedded`

Both `MonetaryAmount` and `Percentage` have a single data-member called `value`. This needs to be mapped to the correct column in the `Beneficiaries` table. This involves using the `@AttributeOverride` annotation. You must map the field name `value` to the column for `savings` and `allocationPercentage` respectively.

When you have completed mapping the `Account` and `Beneficiary` classes, move on to the next step!

#### 10.3.1.2. Implement `JpaAccountRepository`

(TODO-06) You just defined the metadata JPA needs to map rows in the account tables to an account object graph. Now you will implement the data access logic to query `Account` objects.

Open `JpaAccountRepository` and implement the `findByCreditCard(String)` method.

You have to execute an JPQL statement to find the account associated with the credit card. You achieve this by using the transactional `EntityManager` that has already been defined for your. Then, use the `createQuery(String)` method to execute an JPQL query that selects the `Account` where the `creditCardNumber` field matches the input parameter.

When you have completed the `findByCreditCard(String)` implementation, move on to the next step!

#### 10.3.1.3. Test `JpaAccountRepository`

To ensure that you implemented the `JpaAccountRepository` properly you need to test it. Run the `JpaAccountRepositoryTests` class in the `src/test/java` source folder. When you get the green bar, your repository works indicating your account object-to-relational mappings are correct. Move on to the next section!



### Note

Have you noticed that `AccountRepository` has changed from previous labs? Specifically, the `updateBeneficiaries(Account)` method has been removed because it is no longer needed with an ORM capable of transparent persistence. Changes made to the `Beneficiaries` of an account will automatically be persisted to the database when the transaction is committed. Explicit updates of persistent domain objects are no longer necessary.

## 10.3.2. Using JPA in the Restaurant module

### 10.3.2.1. Configure the `Restaurant` mapping

Recall the `Restaurant` entity represents a merchant in the reward network that calculates how much benefit to reward to an account for dining. In this step, you'll configure the JPA mapping annotations that maps the `Restaurant` object graph to the database.

1. (TODO-07) Open the `Restaurant` in the `rewards.internal.restaurant` package. We will configure all

object-to-relational mapping rules using annotations inside this class.

2. Like the `Account` module, we need to mark this class as an entity, define its table and define its `entityId` field as an auto-generated primary key. Don't forget to use a `Column` annotation to specify the target column in the database for this field.
3. Complete the mapping for the remaining `Restaurant` fields: `number`, `name` and `benefitPercentage`. Like the `Beneficiary` mapping, the percentage is a custom type and needs mapping differently. Use the schema in Figure 1 to help you.



### Tip

You will need to use the `@Embedded` and `@AttributeOverride` annotations again.

There is no need to map the `benefitAvailabilityPolicy` - it has been done for you.

When you have completed the `Restaurant` mapping, move on to the next step!

#### 10.3.2.2. Implement `JpaRestaurantRepository`

(TODO-08) You just defined the metadata JPA needs to map rows in the `T_RESTAURANT` table to a `Restaurant` object graph. Now you will implement the data access logic to query `Restaurant` objects.

Open `JpaRestaurantRepository` and implement the `findByMerchantNumber(String)` method.



### Tip

Use the `createQuery(String)` method to find the `Restaurant`.

#### 10.3.2.3. Test `JpaRestaurantRepository`

Now run the `JpaRestaurantRepositoryTests` class in the `src/test/java` source folder. When you get the green bar your repository implementation works. Move on to the next section!

### 10.3.3. Integrating JPA into the Rewards Application

Now that you have tested your JPA based repositories, you'll add them to the overall application configuration. In this section you'll update the application configuration as well as the system test configuration. Then, you'll run your system test to verify the application works!

### 10.3.3.1. Define the JPA configuration for the application

We need to setup our Spring configuration.

1. (TODO-09) Open `RewardsConfig.java` in the `config` package. In this file, define beans for the `JpaAccountRepository` and the `JpaRestaurantRepository`.
2. In contrast to the previous JDBC based implementations, these classes don't need any external dependencies. Spring automatically recognizes the `PersistenceContext` annotation inside `JpaAccountRepository` & `JpaRestaurantRepository` - check `setPersistenceContext(EntityManager)` in each class.
3. Now to setup the Entity Manager Factory. There are three steps.
  - a. (TODO-10) Open `SystemTestConfig.java` in the `src/test/java` source folder. In this file define a factory (as a Spring bean) to create the `EntityManagerFactory`. The factory bean's class is `LocalContainerEntityManagerFactoryBean`. You can name the bean whatever you like, though `entityManagerFactory` is probably the most descriptive.
  - b. Set the `dataSource` and `jpaVendorAdapter` properties appropriately. The `jpaVendorAdapter` tells Spring which JPA implementation will be used to create an `EntityManagerFactory` instance. Use the class `HibernateJpaVendorAdapter` to define the `jpaVendorAdapter` property.
  - c. You can set additional JPA implementation specific configuration properties by setting the `jpaProperties` property. During development it is very useful to have Hibernate output the SQL that it is passing to the database. The two properties to pass in are `hibernate.show_sql=true` to output the SQL and `hibernate.format_sql=true` to make it readable. Note that in some cases a property may be available in the general `jpaProperties` AND in the specific `HibernateJpaVendorAdapter`.
4. (TODO-11) Finally, define a `transactionManager` bean so the Reward Network can drive transactions using JPA APIs. Use the `JpaTransactionManager` implementation. Note that this class requires an `entityManagerFactory`, so we will need to obtain one from the previously defined bean. Fortunately, Spring makes this very easy; simply define the bean method with a parameter of type `EntityManagerFactory`. When Spring calls this method at startup time, it will find a reference by calling the `getObject()` method on the `LocalContainerEntityManagerFactoryBean` defined earlier. By letting Spring handle this (instead of calling the method directly), you are allowing it to take care of any lifecycle requirements.

### 10.3.3.2. Run the application system test

Interfaces define a contract for behavior and abstract away implementation details. Plugging in JPA-based

implementations of the repository interfaces should not change the overall application behavior. To verify this, find and run the `RewardNetworkTests` class. Even though the repositories underneath have changed to use JPA, this test should still run.

If you get a green bar, the application is now running successfully with JPA for object persistence!

Congratulations, you have completed the lab!

### 10.3.4. BONUS CREDIT

If you finish early, here are a couple of bonus tasks.

#### 10.3.4.1. Type-safe Query

(TODO-12) Redo the query you wrote in `JpaAccountRepository` to use the Criteria Query API instead.

You will need to use the `Account` meta-data class `Account_`. For convenience, it has been provided for you in the same package as `Account`. No extra build steps are required (normally it would be auto-generated).

#### 10.3.4.2. Mapping Benefit Availability Policy

(TODO-13) There are two possible benefit availability policies - Always and Never. Currently the policy for the `Restaurant` is transient (so it is not mapped) and hard-wired to `AlwaysAvailable.INSTANCE` - every restaurant will always reward a diner. Suppose a restaurant leaves the scheme, we need to stop rewards.

The policy is held as a code in the database: A for Always and N for Never. The following shows you another way to map a column value to a Java object. Instead of using `@Embedded` which wouldn't work in this case, what we need to do is run some code when the object is restored. The algorithm is:

```
If the policy code in the database is "A", set the policy to AlwaysAvailable, else if it is
"N" to NeverAvailable, else raise an exception for any other code.
```

To do this we use the JPA 2 `@Access` annotation to define setters and getters that only the database will use - this is termed Property access. It is a bit different to the default Property access Hibernate has always used by default - that used the *same* getters and setters as everyone else. Having custom getters and setters that only the database uses is more flexible and avoids unintentional side-effects.

The accessors you need have already been written. All you have to do is enable them. Go to the bottom of the `Restaurant` class and uncomment the annotations on the two protected methods defined there.

The `@Access` annotations tells JPA these are for property access and the `@Column` indicates which column we are mapping. JPA doesn't actually know what data-member we are mapping that column to.

Rerun the `JpaRestaurantRepositoryTests` and `testNonParticipatingRestaurant` should now fail. Fix the test to expect a `NeverAvailable` policy and rerun it to see that our modification really works.



## Note

Mapping an object from a code like this is very useful and is often used with Enumerated types. It replaces the use of Hibernate's custom `UserType` mapping classes.

---

# Chapter 11. Spring MVC Essentials

## 11.1. Introduction

In this lab you will implement basic Spring MVC Controllers to invoke application functionality and display results to the user.

### What you will learn:

1. How to set up required Spring MVC infrastructure
2. How to expose Controllers as endpoints mapped to web application URLs

### Specific subjects you will gain experience with:

1. `DispatcherServlet`
2. `@Controller`
3. `InternalResourceViewResolver`

Estimated time to complete: 30 minutes

## 11.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> `Tasks` (*not Task List*)).

Occasionally, TODO'S defined within XML files may fail to appear in the `Tasks` view (i.e. gaps in the number sequence). To correct this, go to Preferences -> General -> Editors -> Structured Text Editor -> Task Tags pane. Check `Enable searching for Task Tags` and click `Clean` and `Redetect Tasks`. On the `Filters` tab, ensure XML content type is checked.

## 11.3. Detailed Instructions

The instructions for this lab are organized into two main sections. In the first section you will be briefed on the web application functionality you will implement in this lab, then you will review the pre-requisite



infrastructure needed to develop with Spring MVC. In the second section you will actually implement the required web application functionality.

### 11.3.1. Setting up the Spring MVC infrastructure

Spring MVC is a comprehensive web application development framework. In this section, you will review the goals of the web application you will be developing in this lab, then set up the initial infrastructure required to use Spring MVC.

#### 11.3.1.1. Assess the initial state of the web application

The web application you are developing should allow users to see a list of all accounts in the system, then view details about a particular account. This desired functionality is shown graphically below:



Figure 1: GET /accounts/accountList: View a listing of all accounts by name with links to view details



## Account Details

Account:	123456789	
Name:	Keith and Keri Donald	
<b>Beneficiaries:</b>		
<b>Name</b>	<b>Allocation Percentage</b>	<b>Savings</b>
Corgan	50%	\$0.00
Annabelle	50%	\$0.00

Figure 2: GET /accounts/accountDetails?entityId=0: Show details about account '0'

Currently, this desired functionality is half-implemented. In this first step you will assess the initial state of the web application.

(TODO-01) Begin by deploying the web application for this project as-is. Once deployed, navigate to the index page at <http://localhost:8080/mvc>. You should see the index page display. Now click the `View Account List` link. You should see a list of accounts display successfully. This 'accountList' functionality has been pre-implemented for you. We will review and change some of that later on, but it at least gets you started with the application.

Now try clicking on one of the account links. You will get a 404 indicating there is no handler for this request. This 'accountDetails' functionality has not yet been implemented. You'll implement this functionality in this lab.

### 11.3.1.2. Review the application configuration

Quickly assess the initial configuration of the "backend" of this web application. To do this, open `web.xml` in the `src/main/webapp/WEB-INF` directory. Notice that a `ContextLoaderListener` has already been defined for you. This listener is configured to bootstrap your application-layer from `RootConfig` class. Open this file to see the beans that make up this layer. You will see that it simply imports other configurations and establishes

transaction management, all with just a few annotations.

The `accountManager` is the key service that can load accounts from the database for display. The web layer, which will be hosted by the Spring MVC `DispatcherServlet`, will call into this service to handle user requests for account information.

With an understanding of the application-layer configuration, move on to the next step to review the web-layer configuration.

### 11.3.1.3. Review the Spring MVC `DispatcherServlet` configuration

The central infrastructure element of Spring MVC is the `DispatcherServlet`. This servlet's job is to dispatch HTTP requests into the web application to handlers you define. As a convenience, this lab has already deployed a `DispatcherServlet` for you with a basic boilerplate configuration. In this step, you will review this configuration and see how the existing functionality of the web application is implemented.

First, open `web.xml` and navigate to the definition of the `accounts` servlet. Notice it is a `DispatcherServlet` and that all `/accounts/*` requests are mapped to it. Also note how it is initialized with a configuration file. This file contains your web-layer beans.

Now open the `DispatcherServlet` configuration file and review it. First, notice how component scanning is used to detect all controllers within a specific package. This keeps us from having to define individual bean declarations for each controller, which can be great when we have dozens or hundreds of controllers.

Next, review the Java implementation of the `AccountController` to see how it works. Notice how the `@RequestMapping` annotation ties the `/accountList` URL to the `accountList()` method and how this method delegates to the `AccountManager` to load a list of `Accounts`. It then selects the `accountList.jsp` view to render the list. Finally it returns a `String` indicating to the `DispatcherServlet` what view to use to render the model.



## Note

Notice that the view name is specified as the full path relative to the Servlet's context root. The default `ViewResolver` simply forwards to the resource at that location.

Lets quickly summarize the big picture. Return to your web browser, and click on the "View Account List" link again. You should see the account list display again successfully. Clicking on that link issued a GET request to `http://localhost:8080/mvc/accounts/accountList` which set the following steps in motion:

1. The request was first received by the Servlet Engine, which routed it to the `DispatcherServlet`.
2. The `DispatcherServlet` then invoked the `accountList()` method on the `AccountController` based on the `@RequestMapping` annotation.

3. Next, the `AccountController` loaded the account list and selected the `"accountList.jsp"` view.
4. Finally, the `accountList.jsp` rendered the response which you see before you.

At this time, it might also be helpful to visualize the complete web application configuration across layers. To do this, graph the `web-context` config set in your Spring Explorer view. Notice how this config set merges both the `DispatcherServlet` and `Application` configuration files, and produces a graph that illustrates the relationship between your web-layer artifacts and your application-layer artifacts.

At this point you should have a good feel for how you could add the remaining `"accountDetails"` functionality to this application. You simply need to define a new method encapsulating this functionality, test it, and map it to the appropriate URL. You'll do that in the next section.

### 11.3.2. Implementing another Spring MVC handler method

In this section you will implement the handler method that will implement the functionality for the account details page. When you have completed this section, you will no longer get a 404 when you click on an account link from the list view. Instead, you will see the details of that account.

#### 11.3.2.1. Implement the `/accountDetails` request handler

(TODO-02) In the `AccountController`, add a method to handle requests for account details. The method should use the account identifier passed with the HTTP request to load the account, add it to the model, and then select a view.



#### Tip

In your web browser, try clicking on an account to see which parameter name is used to pass in the account identifier.



#### Tip

The JSP has already been implemented for you. Review it in the `/WEB-INF/views` directory.

When you're done with the implementation of the account details page try to run the web application again and make sure the functionality you implemented works. If it doesn't, try to chase where you might have gone wrong and possibly talk to your instructor.

#### 11.3.2.2. Testing the controller

We're almost done! There are two things we still have to do. First of all, we have to test the controller.

(TODO-03) Open `AccountControllerTests` and review how the `accountList()` method has been tested. As you can see, it just calls the handler method without having to do additional trickery and inspects if the model has been correctly filled. In this step, we will do the same for the `accountDetails()` method.

Implement a method called `testHandleDetailsRequest()` to test the controller and annotate with `@org.junit.Test`.



## Tip

The ability to test Spring MVC Controllers out-of-the-container is a great feature. Strive to create a test for each controller in your application. You'll find it proves more productive to test your controller logic using automated unit tests, than to rely solely on manual testing within your web browser.

When all tests pass move on to the next step.

### 11.3.2.3. Add a `ViewResolver`

(TODO-04) Up to this point, the view names have been established within each handler method using absolute paths. Each handler method is also aware of the specific type of view that will be rendered (JSPs in this case). It is recommended to decouple request handling from these response rendering details. In this step, you will add a `ViewResolver` to provide a level of indirection.

Navigate to the `MvcConfig` class and add a bean definition of type `InternalResourceViewResolver`. This will override the default `ViewResolver` and enable the use of logical view names within the Controller code. You should now specify two properties on the view resolver bean definition: `prefix` and `suffix`. Review the current view names to determine these values.



## Tip

The `DispatcherServlet` automatically recognizes any bean definitions of type `ViewResolver`. Therefore, you do not need to provide a bean name for your resolver.

Now refactor the existing controller and test so that only simple view names are used, such as `accountList`. Start by changing the expected values in the two test methods. Run those tests, and notice that they fail. After making those same changes in the `AccountController`, the tests should pass. At that point, redeploy the web application. If you are able to view the list and then the details view of a selected account, you are done with this lab.

---

# Chapter 12. Securing the Web Tier

## 12.1. Introduction

In this lab you will gain experience with Spring Security. You will enable security in the web-tier, and you will establish role-based access rules for different resources. Then you will specify some users along with their roles and manage the login and "access denied" behavior of the application. Finally you will see how to hide links and/or information from users based on their roles.

### **What you will learn:**

1. How to use Spring Security namespace
2. How to define role-based access rules for web resources
3. How to provide users and roles to the security infrastructure
4. How to control login and logout behavior
5. How to display information or links based on role

### **Specific subjects you will gain experience with:**

1. Spring Security namespace
2. The `<security/>` Tag Library
3. sha-256 encoding

Estimated time to complete: 45 minutes

## 12.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> `Tasks` (*not Task List*)).

Occasionally, TODO'S defined within XML files may fail to appear in the `Tasks` view (i.e. gaps in the number sequence). To correct this, go to `Preferences -> General -> Editors -> Structured Text Editor ->`

Task Tags pane. Check `Enable searching for Task Tags` and click `Clean and Redetect Tasks`. On the `Filters` tab, ensure XML content type is checked.

**Warning:** After changing the Spring Security configuration (`security-config.xml`) you will usually need to restart Tomcat to make the changes take effect.

**Note:** To log out at any time use this URL: [http://localhost:8080/security/j\\_spring\\_security\\_logout](http://localhost:8080/security/j_spring_security_logout)

## 12.3. Detailed Instructions

The instructions for this lab are organized into five sections. In the first section, you'll use Spring Security to protect part of the web application. In the second section, you will manage login and "access denied" scenarios. In the third section, you will handle unsuccessful attempts to log in. In the fourth section, you will configure some additional users and roles and experiment with different role-based access rules. In the final section, you will use the security tag library to display links and data based on role.

**Warning:** After changing the Spring Security configuration (`security-config.xml`) you will usually need to restart Tomcat to make the changes take effect.

**Note:** To log out at any time use this URL: [http://localhost:8080/security/j\\_spring\\_security\\_logout](http://localhost:8080/security/j_spring_security_logout)

### 12.3.1. Setting up Spring Security in the application

Currently, the Reward Network web application allows any user to not only view Account information, but also to edit Account information. Of course, in a typical application, certain roles would most likely be required for those actions. The first step in enforcing such role-based access is to intercept the requests corresponding to those actions. Spring Security utilizes standard *Servlet Filters* to make that possible.

(TODO-01) Begin by deploying the web application for this project and navigate to the index page at <http://localhost:8080/security>. You should see a link to 'View Account List'; click on this link and the list of accounts should appear. Be sure the application start successfully before moving on to the next step. You may wish to remove previous projects from your server to allow for quicker startup.

### 12.3.2. Define the Filter class

(TODO-02) Open `web.xml` (within the `src/main/webapp/WEB-INF` directory) and add the relevant `<filter/>` and `<filter-mapping/>` definitions.

### 12.3.3. Include Security Configuration in the Root Application Context

(TODO-03) Next, import the bean configuration file containing the security configuration into the `RootConfig` class. This will include those beans when bootstrapping the application context.

At this point, the filter should be fully configured and ready to intercept incoming requests. Save all work, restart the server and navigate to the index page at <http://localhost:8080/security>. You should see a link to 'View Account List'; click on this link. If your filter is configured correctly, then you *should* get a 404 response. This happens because the resource mapped to `accountList.htm` is secured and you have not configured a real login page yet. The XML currently defines `<security:form-login login-page="/TODO" .../>` and there is no such page as `TODO`.



Figure 1: Accessing Secured Resource

In the next step, you will explore the security constraints that make this happen, and you will configure the login page and the access denied page.

### 12.3.4. Configuring authentication

In the previous section you defined the filter such that it would delegate to security settings to be configured inside Spring configuration. In this section you'll use the security namespace to configure the login page and the error handling policy.

#### 12.3.4.1. Specify the Login Page

(TODO-04) Open `security-config.xml`. Notice that the actual security constraints are defined inside a tag called `security:http`. Specifically notice that the `ROLE_EDITOR` role is required to access the `accountList` page. We can therefore imagine what happened when we tried to access this page: the application was trying to redirect to a login page. However, you haven't defined a login page yet.

Open `login.jsp` under the `src/main/webapp` folder. Notice that the input fields are `j_username` and `j_password`. Also notice that the form action is `j_spring_security_check`



### Note

The usual location for jsp files is somewhere under the `WEB-INF` directory so that web clients



can't directly access them. However, for simplicity several files will be located directly under the `webapp` directory. In a more robust deployment environment these files would be placed in the `WEB-INF` directory and authorization rules would be defined to allow access to these resources by unauthenticated users.

Back inside `security-config.xml`, you can now configure the login page by modifying the `login-page` attribute of the `<security:form-login>` tag.

#### 12.3.4.2. Login as a Valid User

Save all work, restart the web application, and navigate to the index page at <http://localhost:8080/security>. This time when you click the 'View Account List' it should redirect you to the login form.

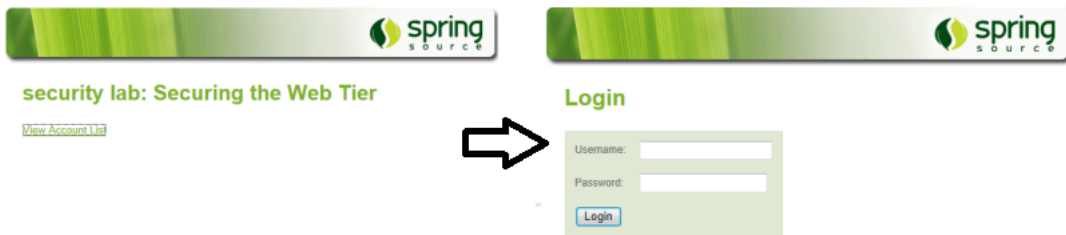


Figure 2: Implementing Login Page



### Note

Feel free to try logging in with a random username and password. If the values are invalid, then you should receive another 404 error message (the authentication failure url will be defined later).

To determine a valid username/password combination, you can explore the authentication configuration in `security-config.xml`. You will find that an in-memory `authentication-provider` is being used. Have a look in the properties file that it references, and there you will find a username along with its password and role.

Try logging in using the user called `vince`. Look carefully at the error message that occurs. You will see an error, since `vince` does not have the rights to access the `accountList` page yet. Before giving `vince` the right to access this page, you will set up a denied access page. This should be set using an attribute of the `security:http` tag. An access denied page has been created for you already. It can be reached on `/denied.jsp`.

Save all work, restart the web application. Revisit the index page at <http://localhost:8080/security>. Attempting to view the account list should now send you to the access denied page.

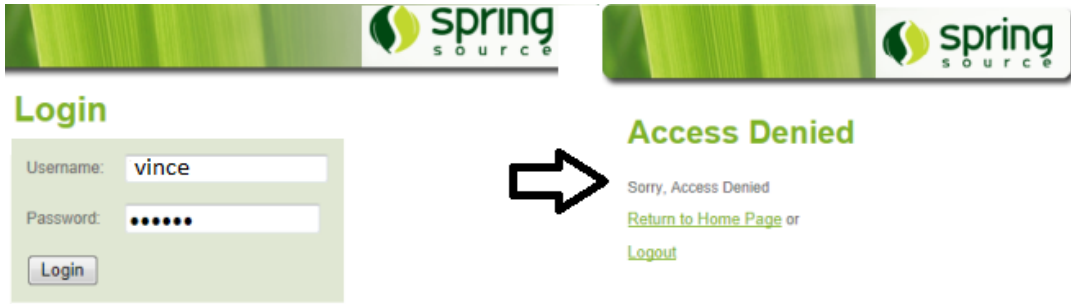


Figure 3: Implementing Customized Error Page

### 12.3.5. Handling unsuccessful attempts to log in

(TODO-05) When a user is not allowed to log in, errors should be handled gracefully and an error message should be shown on the login page. Try to log in using incorrect user/password. You should see a 403 error page since we have not set up properly the error handling policy yet.

Open `login.jsp`. Notice that there is a test to determine if a parameter named `login_error` is empty. This will be a parameter passed as an HTTP GET request. If a url such as `/login.jsp?login_error=value` is called, the message will be displayed.

Open `security-config.xml`. As you can see, the `security:form-login` tag holds an attribute called `authentication-failure-url`. You can replace its contents with `/login.jsp?login_error=true`. In that way, in case of authentication failure, the user will be redirected to the login page and a request parameter called `login_error` will be set to `true`.

Save all work, restart the web application and try logging in using incorrect user/password again. An error message should appear.



Figure 6: Handling Login Errors

## 12.3.6. Managing Users and Roles

(TODO 06) In the previous sections you worked on Spring Security general configuration. In this section, you will modify the access rules and define additional users.

### 12.3.6.1. Configure Role-Based Access

So far you have only been logging in as a user with the `ROLE_VIEWER` role, and you have been denied access to the account list. Perhaps the restriction is too severe. To edit an account should require the `ROLE_EDITOR` role, but accessing the `accountList` and `accountDetails` views should be available to a user with the `ROLE_VIEWER` role.

Find the `intercept-url` tags and modify the rules for `/accounts/account*` to enable access for viewers as well.

Save all work and restart the web application. Using the user `vince`, you should now be able to access the account list and the account details. On the Account details page, click on 'Edit Account'. This link should send you to the 'Access Denied' page as `vince` does not have the `EDITOR` privileges.

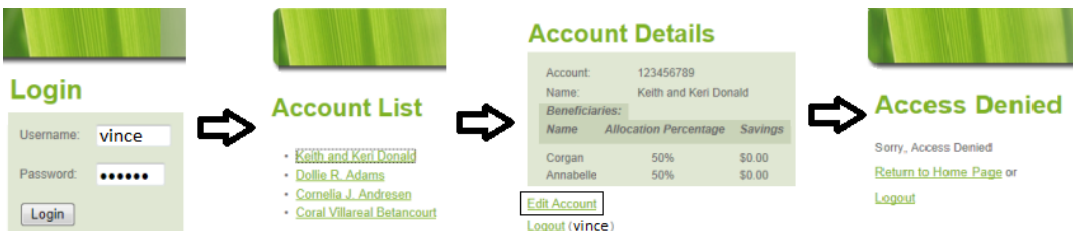


Figure 4: Configure Role-Based Access

### 12.3.6.2. Add a User

(TODO 07) Notice that the account list page provides a `logout` link. Open `accountList.jsp` within `WEB-INF/views` to see the corresponding URL. That value is automatically recognized by the `logout` mechanism.

At this point, logging out doesn't help much since you only have one user defined. However, by adding a user with the `EDITOR` role, then you should be able to login as that user and successfully edit the account.

Revisit the properties file where users are defined, and add a user called `edith` with the `ROLE_EDITOR` role.



## Note

Spring Security provides many out-of-the-box options for *where* and/or *how* the user details are stored. For development and testing, it is convenient to use the in-memory option. Since there is a layer of abstraction here, and since the authentication and authorization processes are completely decoupled, the strategy can be modified for other environments without impacting the rest of the behavior.

Save all work, restart the web application, log in with the user `edith` and try editing the account information. This time you should be able to access the `editAccount` page. Also, verify that a user that does not have the editor role is still redirected to the access denied page.

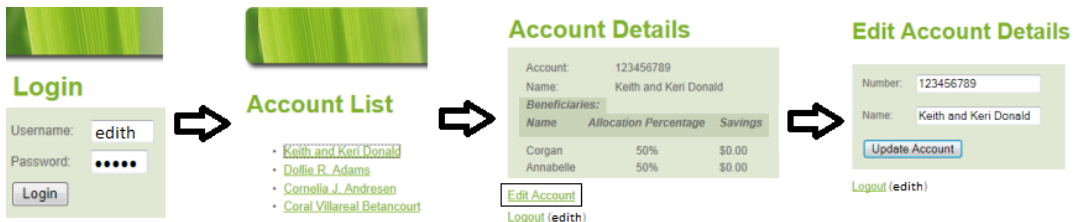


Figure 5: Configure Role-Based Access

### 12.3.6.3. Add a catch all

(TODO 08) Currently you secure URLs starting with `/accounts/edit` and `accounts/account`. To get a more robust configuration, you should also enforce that people must at least be logged in to show anything else starting with `/accounts/`.

Log out by clicking on the 'log out' link. Then try to access <http://localhost:8080/security/accounts/hidden>. As

you can see, this URL is currently not protected.

Inside `security-config.xml`, add another `intercept-url` element at the end of the list with the pattern `/accounts/**` which enforces that the user should be fully authenticated. Be sure to do this using the pre-defined Spring Security expressions. Save all work, restart the web application and check that you cannot access <http://localhost:8080/security/accounts/hidden> anymore.

## 12.3.7. Using the Security Tag Library

Spring Security includes a JSP tag library to support common view-related tasks while still promoting the best practice of scriptlet-free JSPs.

### 12.3.7.1. Hide a Link Based on Role

(TODO-09) A fairly common requirement for web-tier security is to only display certain information and/or links to users with a specified role. For example, you could hide the 'Edit Account' link unless a user would be able to access that page. This provides a much better user experience than constantly being redirected to the access denied page.

Open `accountDetails.jsp` and find the link for `editAccount.htm`. Surround that link within the body of an `<security:authorize>` tag. Then, see if you can determine what attribute of that tag to use in order to hide its contents.

Save your work and revisit the account details view (you should not need to restart your application). If you are currently logged in as an editor you should still see the link. On the other hand, if you are logged in as a viewer, you should not see the link. Try logging in as a user with and without the editor role and verify that you see the correct behavior.

### 12.3.7.2. Hide Information Based on Role

(TODO-10) Apply the same procedure to the table within the account details view that lists the beneficiary information. However, this time a viewer should be able to see the contents of the table while a non-viewer should only see the account number and name. It is quite common to encounter requirements for hiding detailed information from another user even if that user has more access privileges.

The interesting thing about this requirement is that an editor who is also a viewer will be able to view the beneficiary information, but an editor who is *not* a viewer will not be able to view the beneficiary information. After adding the necessary tag, verify that this is indeed the case.



## Note

Notice the other available attributes on the `<security:authorize/>` tag. Feel free to apply the

tag to other data and/or other JSPs. As you have seen, it's also trivial to define additional users and roles in order to have more options.

### 12.3.8. Bonus question: SHA-256 encoding

Even though your application's security has dramatically improved, you still have plain-text passwords inside `users.properties` file. This point will be improved using SHA-256 encoding.

(TODO-11) Open `security-config.xml` file and declare sha-256 encoding. You will use a tag called `password-encoder`. Set its `hash` attribute accordingly. Now, passwords need to be encoded. Open the `users.properties` file and change the plain-text passwords into sha-256-encoded ones. You will not need to setup any salt source.

Save all work, restart the web application and try logging in again. It should work in the same way as before. Your application is now using password encoding.

If you see the behavior as described, then you have completed this lab. Congratulations!



#### Tip

Normally there is no way to get back the password from a sha-256 hash, at least not with mathematics, but in the Internet you will find so called Rainbow Tables which are lookup tables for pre-generated hash/plaintext values. Sometimes you can even enter the hash value in google and get back the plaintext. By appending a salt to the user password before the hash is calculated this attack is more difficult, often infeasible. In real life we would recommend to append a salt to the user password.

---

# Chapter 13. Building RESTful applications with Spring MVC

## 13.1. Introduction

In this lab you'll use some of the features that were added in Spring 3.0 to support RESTful web services. Note that there's more than we can cover in this lab, please refer back to the presentation for a good overview.

### What you will learn:

1. Working with RESTful URLs that expose resources
2. Mapping request- and response-bodies using HTTP message converters
3. Writing a programmatic HTTP client to consume RESTful web services

### Specific subjects you will gain experience with:

1. Processing URI Templates using `@PathVariable`
2. Using `@RequestBody` and `@ResponseBody`
3. Using the `RestTemplate`

Estimated time to complete: 40 minutes

## 13.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> `Tasks` (*not Task List*)).

Occasionally, TODO'S defined within XML files may fail to appear in the `Tasks` view (i.e. gaps in the number sequence). To correct this, go to `Preferences -> General -> Editors -> Structured Text Editor -> Task Tags` pane. Check `Enable searching for Task Tags` and click `Clean` and `Redetect Tasks`. On the `Filters` tab, ensure XML content type is checked.

## 13.3. Detailed Instructions

The instructions for this lab are organized into sections. In the first section you'll add support for retrieving a JSON-representation of accounts and their beneficiaries and test that using the `RestTemplate`. In the second section you'll add support for making changes by adding an account and adding and removing a beneficiary. The optional bonus section will let you map an existing exception to a specific HTTP status code.

### 13.3.1. Exposing accounts and beneficiaries as RESTful resources

In this section you'll expose accounts and beneficiaries as RESTful resources using Spring's URI template support, HTTP Message Converters and the `RestTemplate`.

#### 13.3.1.1. Inspect the current application

(TODO-01) First open the `web.xml` deployment descriptor to see how the application is bootstrapped: the `RootConfig` class contains the necessary configuration for the root context, which contains a `accountManager` bean that provides transactional data access operations to manage `Account` instances. The `MvcConfig` class contains the configuration for Spring MVC, and since it uses component scanning of the `accounts.web` package it will define a bean for the `AccountController` class. The `<mvc:annotation-driven/>` element ensures that a number of default HTTP Message Converters will be defined and that we can use the `@RequestBody` and `@ResponseBody` annotations in our controller methods.

Under the `src/test/java` source folder you'll find an `AccountClientTests` JUnit test case: this is what you'll use to interact with the RESTful web services on the server.

Finally, deploy the application to your local server, start the server and verify that the application deployed successfully by accessing <http://localhost:8080/rest-ws> from a browser. When you see the welcome page, the application was started successfully.

#### 13.3.1.2. Expose the list of accounts

(TODO-02) Open the `AccountController`. Notice that it offers several methods to deal with various requests to access certain resources. Add the necessary annotations to the `accountSummary` method to make it respond to GET requests to `/accounts`.



### Tip

You need one annotation to map the method to the correct URL and HTTP Method, and another one to ensure that the result will be written to the HTTP response by an HTTP Message Converter (instead of an MVC View).



When you've done that, save all work and restart the server. Now try to access <http://localhost:8080/rest-ws/app/accounts> from that same browser. Depending on the browser used, you may see the response inline or you may see a popup asking you what to do with the response: save it to a local file and open that in a local text editor (Notepad is available on every Windows machine). You'll see that you've just received a response using a JSON representation (JavaScript Object Notation). How is that possible?

The reason is that the project includes the Jackson library on its classpath:

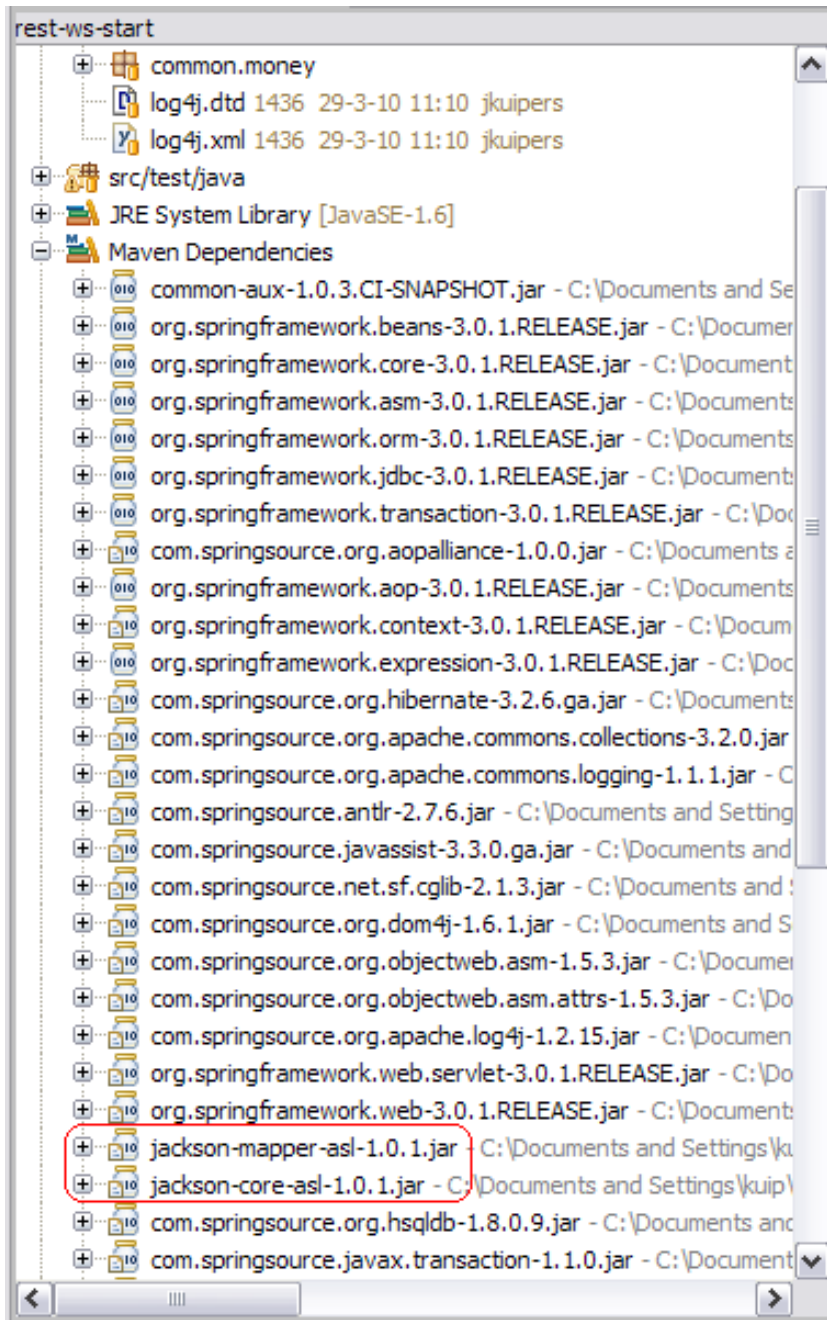


Figure 1: The Jackson library is on the classpath

If this is the case, an HTTP Message Converter that uses Jackson will be active by default when you specify `<mvc:annotation-driven/>`. The library mostly 'just works' with our classes without further configuration: if you're interested you can have a look at the `MonetaryAmount` and `Percentage` classes and search for the `Json` annotations to see the additional configuration.

### 13.3.1.3. Retrieve the list of accounts using a `RestTemplate`

(TODO 03) A client can process the shown JSON contents anyway it sees fit. In our case, we'll rely on the same HTTP Message Converter to deserialize the JSON contents back into `Account` objects. Open the `AccountClientTests` class under the `src/test/java` source folder in the `accounts.client` package. This class uses a plain `RestTemplate` to connect to the server. Use the supplied template to retrieve the list of accounts from the server, from the same URL that you used in your browser.



#### Tip

You can use the `BASE_URL` variable to come up with the full URL to use.



#### Note

We cannot assign to a `List<Account>` here, since Jackson won't be able to determine the generic type to deserialize to in that case: therefore we use an `Account[]` instead.

When you've completed this TODO, run the test and make sure that the `listAccounts` test succeeds. You'll make the other test methods pass in the following steps.

### 13.3.1.4. Expose a single account

(TODO 04) To expose a single account, we'll use the same `/accounts` URL followed by the `entityId` of the `Account`, e.g. `/accounts/1`. Switch back to the `AccountController` and complete the `accountDetails` method.



#### Tip

Since the `{accountId}` part of the URL is variable, use the `@PathVariable` annotation to extract its value from the URI template that you use to map the method to GET requests to the given URL.

If you want to test your code, just try to access <http://localhost:8080/rest-ws/app/accounts/0> to verify the result.

(TODO 05) When you're done with the controller, complete the `AccountClientTests` by retrieving the account

with id 0.



## Tip

The `RestTemplate` also supports URI templates, so use one and pass 0 as the value for the `urlVariables` varargs parameter.

Run the test and ensure that the `getAccount` test now succeeds as well.

### 13.3.1.5. Create a new account

So far we've only exposes resources by responding to GET methods: now you'll add support for creating a new account as a new resource.

(TODO 06) Implement the `createAccount` method is mapped to POSTs to `/accounts`. The body of the POST will contain a JSON representation of an `Account`, just like the representation that our client received in the previous step: make sure to annotate the `account` method parameter appropriately to let the request's body be deserialized! When the method completes successfully, the client should receive a `201 Created` instead of `200 OK`, so annotate the method to make that happen as well.

(TODO 07) RESTful clients that receive a `201 Created` response will expect a `Location` header in the response containing the URL of the newly created resource. Complete the TODO by setting that header on the response.



## Tip

To help you coming up with the full URL on which the new account can be accessed, we've provided you with a helper method called `getLocationForChildResource`. Since URLs of newly created resources are usually relative to the URL that was POSTed to, you only need to pass in the original request and the identifier of the new child resource that's used in the URL and the method will return the full URL, applying URL escaping if needed. This way you don't need to hard-code things like the server name and servlet mapping used in the URL in your controller code!

(TODO 08 - 09) When you're done, complete the test method by POSTing the given `Account` to the `/accounts` URL. The `RestTemplate` has two methods for this: use the one that returns the location of the newly created resource and assign that to a variable. Then complete TODO 09 by retrieving the new account on the given location. The returned `Account` will be equal to the one you POSTed, but will also have received an `entityId` when it was saved to the database.

Run the tests again and see if the `createAccount` test runs successfully. Regardless of whether this is the case or not, proceed with the next step!

### 13.3.1.6. Seeing what happens at the HTTP level

(TODO 10) If your test did not work, you may be wondering what caused an error. Because of all the help that you get from Spring, it's actually not that easy to see what's happening at the HTTP transport level in terms of requests and responses when you exercise the application. For debugging or monitoring HTTP traffic, Eclipse ships with a built-in tool that can be of great value: the TCP/IP Monitor. To open this tool, which is just an Eclipse View, press Ctrl+3 and type 'tcp' in the resulting popup window; then press Enter to open the TCP/IP Monitor View. Click the small arrow pointing downwards and choose "properties".

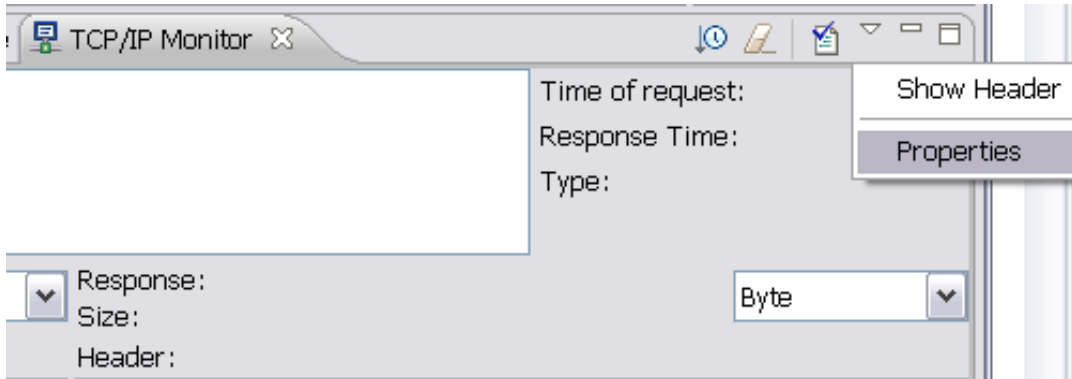


Figure 2: The "properties" menu entry of the TCP/IP Monitor view

Choose "Add..." to add a new monitor. As local monitoring port, enter 8081 since this port is probably unused. As host name, enter "localhost" and as port enter 8080 since this is the port that Tomcat is running on. Press OK and then press "Start" to start the newly defined monitor.



#### Tip

Don't forget to start the monitor after adding it!

Now switch to the `AccountClientTests` and change the `BASE_URL`'s port number to 8081 so all requests pass through the monitor.



#### Note

This assumes that you've used that variable to construct all your URLs: if that's not the case, then make sure to update the other places in your code that contain the port number as well!

Now run the tests again and switch back to the TCP/IP Monitor View (double-click on the tab's title to

maximize it if it's too small). You'll see your requests and corresponding responses. Click on the small menu arrow again and now choose 'Show Header': this will also show you the HTTP headers, including the Location header you specified for the response to the POST that created a new account.



## Note

Actually, there's one request missing: the request to retrieve the new account. This is because the monitor rewrites the request to use port 8080, which means the Location header will include that port number instead of the 8081 the original request was made to. We won't try to fix that in this lab, but it wouldn't be too hard to come up with some interceptor that changes the port number to make all requests pass through the filter.

If your `createAccount` test method didn't work yet, then use the monitor to debug it. Proceed to the next step when the test runs successfully.

### 13.3.1.7. Create and delete a beneficiary

(TODO 11) Complete the `addBeneficiary` method in the `AccountController`. This is similar to what you did in the previous step, but now you also have to use a URI template to parse the `accountId`. Make sure to return a `201 Created` status again! This time, the response's body will only contain the name of the beneficiary: an HTTP Message Converter that will convert this to a `String` is enabled by default, so simply annotate the method parameter again to obtain the name.

(TODO 12) Set the Location header to the URL of the new beneficiary.



## Note

As you can see in the `getBeneficiary` method, the name of the beneficiary is used to identify it in the URL.

(TODO 13) Complete the `removeBeneficiary` method. This time, return a `204 No Content` status.

(TODO 14 - 17) To test your work, switch to the `AccountClientTests` and complete the TODOs. When you're done, run the test and verify that this time all test methods run successfully. If this is the case, you've completed the lab!

### 13.3.1.8. BONUS (Optional): return a 409 Conflict when creating an account with an existing number

(TODO 18) The current test ensures that we always create a new account using a unique number. Let's change

that and see what happens. Edit the `createAccount` method in the test case to use a fixed account number, like `"123123123"`. Run the test: the first time it should succeed. Run the test again: this time it should fail. When you look at the exception in the JUnit View or at the response in the TCP/IP monitor, you'll see that the server returned a `500 Internal Server Error`. If you look in the Console View for the server, you'll see what caused this: a `DataIntegrityViolationConstraint`, ultimately caused by a `SQLException` indicating that the number is violating a unique constraint.

This isn't really a server error: this is caused by the client providing us with conflicting data when attempting to create a new account. To properly indicate that to the client, we should return a `409 Conflict` rather than the `500 Internal Server Error` that's returned by default for uncaught exceptions. To make it so, add a new exception handling method that returns the correct code in case of a `DataIntegrityViolationConstraint`.



## Tip

Have a look at the existing `handleNotFound` method for a way to do this.

When you're done, run the test again (do it twice as the database will re-initialize on redeploy) and check that you now receive the correct status code. Optionally you can even restore the test method and create a new test method that verifies the new behavior.

---

# Chapter 14. Simplifying Messaging with Spring JMS

## 14.1. Introduction

In this lab you will gain experience with Spring's JMS support. You will complete an implementation of a `DiningBatchProcessor` that sends dining event notifications to the reward network as messages. You will also configure a logger to receive the reward confirmations asynchronously.

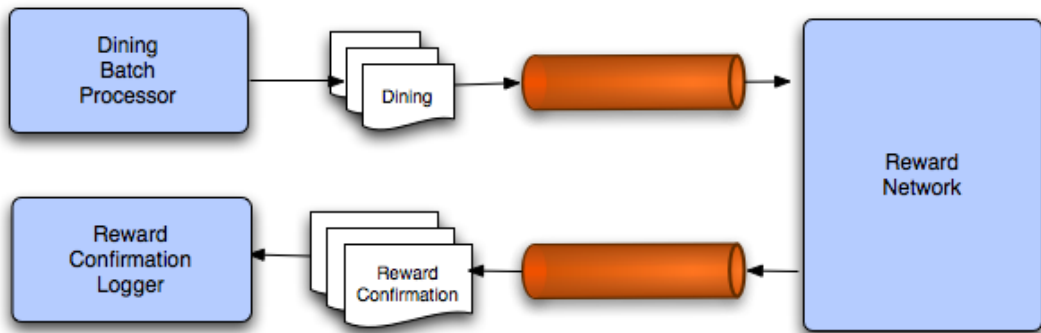


Figure 1: The batch processing of dining events with asynchronous messaging.

### What you will learn:

1. How to configure JMS resources with Spring
2. How to send messages with Spring's `JmsTemplate`
3. How to configure a Spring message listener container
4. How to delegate Message content to a plain Java object

### Specific subjects you will gain experience with:

1. `JmsTemplate`
2. The `jms:xxx` namespace



Estimated time to complete: 45 minutes

## 14.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)).

## 14.3. Detailed Instructions

The instructions for this lab are organized into five sections. In the first section, you will establish the messaging infrastructure. In the second section, you will learn how to send dining notifications as messages. In the third and fourth sections, you will define and configure listeners to enable message reception by *Message-Driven POJOs*. In the final section, you will complete a test case to verify that the batch of dining notifications is successfully producing the corresponding confirmation messages.

### 14.3.1. Providing the messaging infrastructure

In this section you will configure the necessary infrastructure to support the Reward Network in a messaging environment.

#### 14.3.1.1. Define the `ConnectionFactory`

(TODO-01) In JMS-based applications, the `Connection` is obtained from a `ConnectionFactory`. Spring's JMS support will handle the resources, but it does require a `ConnectionFactory` bean definition. In this step you will provide exactly that.

Open the `JmsInfrastructureConfig` class in the `config` package. Provide a bean definition there for an instance of `org.apache.activemq.ActiveMQConnectionFactory`. Also provide a value for the `brokerURL` property.



### Tip

For this simple lab, you will be using an embedded broker with persistence disabled. In ActiveMQ, the `brokerURL` should be `'vm://embedded?broker.persistent=false'`.

When you've done this, move on to the next step!

#### 14.3.1.2. Define the message queues

(TODO-02) Now you will need to create two queues, one for handling dining notifications and the other for handling the reward confirmations. Create two bean definitions of type `org.apache.activemq.command.ActiveMQQueue` and call them "diningQueue" and "confirmationQueue". Provide a unique name for each queue using constructor injection.

You are now ready to move on to the next section.

### 14.3.2. Sending Messages with `JmsTemplate`

In the previous section you configured a queue for dining notifications. In this section you will provide the necessary code to send dining notifications to that queue from a batch processor.

#### 14.3.2.1. Establish a dependency on `JmsTemplate`

(TODO-03) Navigate to the `JmsDiningBatchProcessor` within the `rewards/jms/client` package. This class will be responsible for sending the dining notifications via JMS. Provide a field for an instance of Spring's `JmsTemplate` so that you will be able to use its convenience method to send messages. Add a setter or constructor to allow you to set this dependency later via dependency injection.

#### 14.3.2.2. Implement the batch sending

(TODO-04) Now complete the implementation of the `processBatch(...)` method by calling the one-line convenience method provided by the `JmsTemplate` for each `Dining` in the collection.



### Note

Here you can rely on the template's default message conversion strategy. The `Dining` instance will be automatically converted into a JMS `ObjectMessage`.

#### 14.3.2.3. Define the template's bean definition

(TODO-05) Open the `ClientConfig` class within the `config` package. Define a bean definition for the `JmsTemplate`. Keep in mind that it will need a reference to the `ConnectionFactory` as well as its destination.

Once you have defined the bean, inject it into the `JmsDiningBatchProcessor` that is already defined in that same file. Then move on to the next section.

### 14.3.3. Configuring the `RewardNetwork` as a message-driven object

In the previous section you implemented the dining notification sending. In this section you will provide the

necessary configuration for receiving those messages and delegating their content to the `RewardNetwork`. You will do this using Spring's JMS namespace that was introduced in Spring 2.5.

#### 14.3.3.1. Define the listener container

(TODO-06) Open the `JmsRewardsConfig` class within the `config` package. In this file you will provide the necessary bean definitions to configure the existing `RewardNetworkImpl` as a Message-Driven POJO. No code modifications or new code will be required.

First define a `listener-container` bean definition. The listener-container element is defined in the JMS namespace and can be configured using a variety of attributes, such as the maximum amount of concurrent listener, the transaction manager reference and the connection factory reference. The default for the connection factory reference is `connectionFactory` and this is also the name of our connection factory bean, so you don't necessarily have to specify this.

Within the listener container, define a listener. The job of the listener is to listen to a queue for incoming messages and call a method on an object, passing the message payload. The queue we want to listen to is the dining queue defined earlier, use the `destination` attribute to specify the name of the queue, not the bean ID of the queue. (Recall that in JMS, the term `destination` refers to both where messages come from as well as where they go to.)

Next, the listener should use the `ref` and `method` attributes to call the method on the `rewardNetwork` bean to process the message. Since there is only one method (`rewardAccountFor`), you do not necessarily need to specify the method name. You should also note that this method takes a single argument of type `Dining`, which happens to correspond to the payload of the messages being placed on the dining queue. Finally, use `response-destination` to place the return value from the method, a `Confirmation` object, on the confirmation queue you setup earlier. Again, indicate the name of the queue, not the reference to the bean ID.

Once you have configured the listener-container, move on to the next step.

#### 14.3.3.2. Define the listener adapter

Now that you have the container in place, you can start adding listeners to it. Each listener will have a corresponding `listener` element defined **inside** the `listener-container` element.

Define a listener for the `RewardNetwork`. The reference of the listener should be set to `rewardNetwork`. The method also needs to be set (the method that will handle the reward request).

You also need to set the queue to which the listener is going to listen. Look up the name of the queue (remember: this is not the bean name) in `JmsInfrastructureConfig` and configure the `destination` attribute using this name. Since the `rewardAccountFor()` method returns an object, we also need to set the `defaultResponseQueueName` property. Review the diagram above, lookup the queue name for the destination queue in `JmsInfrastructureConfig` and set the `defaultResponseQueueName` attribute of the listener element.

Now that you have configured the `RewardNetworkImpl` as a message-driven object, you are ready to move on to the next section.

### 14.3.4. Receiving the asynchronous reply messages

In the previous section, you configured the reward network to receive messages and also to reply automatically to a queue with reward confirmations. Now you will define another Message-Driven POJO so that those confirmations will be received and logged.

#### 14.3.4.1. Define the listener container and adapter

(TODO-07) Open `ClientConfig` and define another listener container and corresponding listener adapter. This time, you should set the destination (source of messages) to the confirmation queue you setup earlier. The listener should delegate to the `logger` bean that is already provided. Have a look at that class to determine the method name. Also notice that it is a `void` method declaration so there is no need to provide a response destination this time.

### 14.3.5. Testing the message-based batch processor

At this point the messaging configuration should be fully established. It is now time to verify that configuration. Luckily a test case is already provided with all but two remaining tasks to complete.

#### 14.3.5.1. Send the batch of dining notifications

(TODO-08) Navigate to the `DiningBatchProcessorTests` in the `rewards/jms/client` package in the `src/test/java` folder. Notice that the class makes use of Spring's support for integration testing and that the `diningBatchProcessor` and `confirmationLogger` fields will be automatically injected using the `@Autowired` annotation..

In the `testBatch()` method, a number of `Dining` objects are being instantiated and added to a `List`. Here you simply need to invoke the method that you implemented previously in the `JmsDiningBatchProcessor` class.

(TODO-09) Finally, provide an assertion to verify that the entire batch was sent and that the `confirmationLogger` has received the same number of replies. If this assertion fails then carefully read any exception messages, and work for the green bar.



## Tip

If you are having trouble and not receiving any useful error messages, then first lower the log level for `org.springframework.jms` in the `log4j.xml` file. If that is still not helpful, then add breakpoints in some logical places (consider where you are sending and receiving messages)

and step through with Eclipse's debugger.

Once you have the green bar, you have completed this lab. Congratulations!

---

# Chapter 15. JMX Management of Performance Monitor

## 15.1. Introduction

In this lab you will use JMX to monitor a running application remotely. You will use the `RepositoryPerformanceMonitor` to collection performance metrics and expose them via JMX.

### What you will learn:

1. How to expose a Spring bean as a JMX MBean
2. How to control the management interface of the exposed JMX MBean
3. How to export pre-existing MBeans

### Specific subjects you will gain experience with:

1. `@ManagedResource`, `@ManagedAttribute`, `@ManagedOperation`
2. `@EnableMBeanExport`

Estimated time to complete: 30 mins

## 15.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)).

## 15.3. Detailed Instructions

### 15.3.1. Exposing the `MonitorFactory` via JMX

#### 15.3.1.1. Assess the initial state of the `JamonMonitorFactory`

Find and open the `JamonMonitorFactory` class in the `rewards.internal.monitor.jamon` package. Notice that this is an implementation of the `MonitorFactory` interface that uses the JAMon library to accomplish its performance monitoring.

When you are comfortable with the implementation of this class, move on to the next step where you export an instance of this bean via JMX

#### 15.3.1.2. Add JMX metadata to the implementation class

(TODO 01) Add Spring's JMX annotations `@ManagedResource`, `@ManagedAttribute` and `@ManagedOperation` to the class as well as methods you want to expose via JMX. Use `statistics:name=monitorFactory` as name for the bean exposed.

By placing the data collection and exposure of performance metrics in the `JamonMonitorFactory` class, we've ensured that the `RepositoryMonitorFactory` is completely decoupled from any reporting mechanism. The `MonitorFactory` interface is very generic, but allows each implementation strategy to expose any data it sees fit.

When you have finished exporting the `JamonMonitorFactory` class to JMX, move on the next step

#### 15.3.1.3. Activate annotation driven JMX in application configuration

(TODO 2) Find and open the `AspectsConfig` class in the `config` package. In this file activate annotation driven JMX by adding the appropriate annotation.

#### 15.3.1.4. Start the MBeanServer and deploy the web application

In this step, you will deploy the project as a web application as described Appendix C, *Using Web Tools Platform (WTP)*. However, before you can do that, you must tell the Java VM to start an `MBeanServer`. To do this, open the Window menu, go to "Preferences...", then select "Java > Installed JREs" on the left. Select "Edit..." for the JRE that you are using and add `-Dcom.sun.management.jmxremote` as a VM argument. This value instructs the JVM to start the internal `MBeanServer` and also allows connections to it via shared memory, so that when you run `jconsole` it will see the process and allow you to directly connect to it, instead of needing to use a socket connection, with a name/password required.

Now deploy the project as a web application. Once deployed, open <http://localhost:8080/jmx> in your browser. You should see the welcome page display containing a form that submits to the `RewardServlet`.

#### 15.3.1.5. View the monitor statistics using JConsole

From the command line of your system, or Windows Explorer, run the `$JDK_HOME/bin/jconsole` application. When this application starts up, choose the process that identifies your web application and open it.



## Tip

If you can not see the process you started, in `jconsole`, it is possible you do not have adequate security rights in your environment. In this case, you will have to connect to the process via a socket connection instead. In the VM arguments tab of your launch configuration, add the following arguments:

```
-Dcom.sun.management.jmxremote.port=8181  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false
```

Then restart the process, and connect via `jconsole` by using the 'Remote' tab, specifying a host of `localhost` and port of `8181`.

Once connected to the application, navigate to the `MBeans` tab and find the MBean you exported.

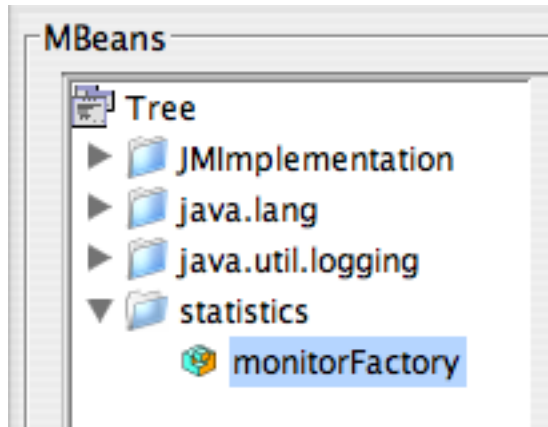


Figure 1: The `MonitorFactory` MBean

Once you have found the MBean, execute a few rewards operations in the browser and refresh the MBean attributes. You should see something similar to this



Attributes		Operations	Notifications	Info
Name	Value			
CallsCount	9			
LastAccessTime	Wed Apr 25 14:05:31 PDT 2007			
TotalCallTime	280			

Refresh

Figure 2: The `MonitorFactory` attributes

## Tip

Double clicking on any scalar value will create a graph over time

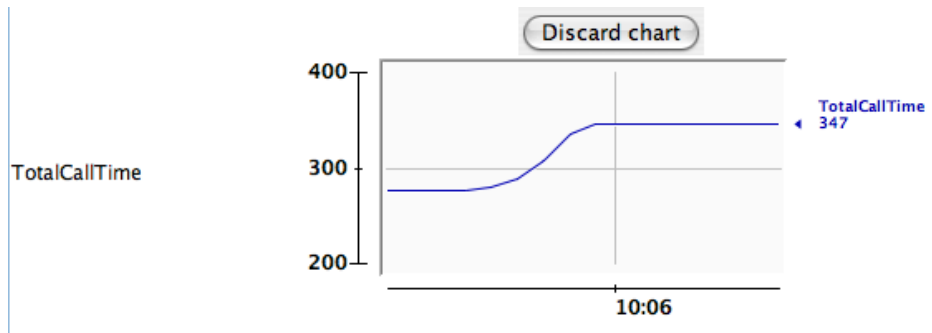


Figure 3: Scalar value graph

Explore the attributes and operations of the MBean and when you are finished move to the next section

### 15.3.2. Exporting pre-defined MBeans

By using Spring's `@EnableMBeanExport` element you not only have triggered annotation based JMX export. The element will also pick up classes that follow JMX naming conventions (a class implementing an interface `_${className}MBean`).

#### **15.3.2.1. View the Hibernate statistics bean**

In your JConsole you should now see a `org.hibernate.jmx` folder that includes the `StatisticsService`. Be sure to activate it by flipping the `StatisticsEnabled` flag. Now issue a few queries and refresh the statistics service. You should see the updates.

Once you have completed this step, you have completed the lab.

---

# Chapter 16. Spring Remoting

## 16.1. Introduction

In this lab you will gain experience with Spring's support for a variety of remoting protocols. You will expose the reward network on multiple endpoints and then test each of these from a standalone client running in another JVM. The lab will demonstrate Spring's consistent exporting and consuming strategies across different remoting protocols.

### What you will learn:

1. How to configure service exporters
2. How to configure client side proxies
3. How to deploy remote endpoints in a web application

### Specific subjects you will gain experience with:

1. RmiServiceExporter
2. RmiProxyFactoryBean
3. HttpInvokerServiceExporter
4. HttpInvokerProxyFactoryBean

Estimated time to complete: 30 minutes

## 16.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)).

Occasionally, TODO'S defined within XML files may fail to appear in the `Tasks` view (i.e. gaps in the number sequence). To correct this, go to Preferences -> General -> Editors -> Structured Text Editor -> Task Tags pane. Check `Enable searching for Task Tags` and click `Clean` and `Redetect Tasks`. On the `Filters` tab, ensure `XML content type` is checked.

## 16.3. Detailed Instructions

### 16.3.1. Remoting with RMI

In this section you will establish a service exporter for the `RewardNetwork` using Java's RMI protocol and a client-side proxy to call the service.

#### 16.3.1.1. Define the service exporter

(TODO-01) Spring provides exporters that allow you to decorate existing POJOs in order to expose them on remote endpoints. In this step, you will configure an RMI-based exporter to expose the existing `RewardNetworkImpl` bean.

Find and open the `rmi-server-config.xml` file in the `rewards.remoting` package in the `src/test/java` source folder. Create a bean definition in that file of type `RmiServiceExporter`. You will need to provide the following properties:

- `service` (the reference to the actual POJO to export)
- `serviceInterface` (the interface that the POJO implements)
- `serviceName` (the name used when binding to the `rmiRegistry` - such as 'rewardNetwork')



### Tip

You can set the `alwaysCreateRegistry` property to `true` in order to save time on startup (no need to search for an existing `rmiRegistry` for testing)

When you've done this, move on to the next step.

(TODO-02) Next, start a server containing the RMI exported `RewardNetwork`. Find and run the `RmiExporterBootstrap` class in the `rewards.remoting` package. Right-click on the class and choose "Run as -> Java Application" to start the application. Note that this application will run in the background and wait for client RMI requests to come in. We will do this next.

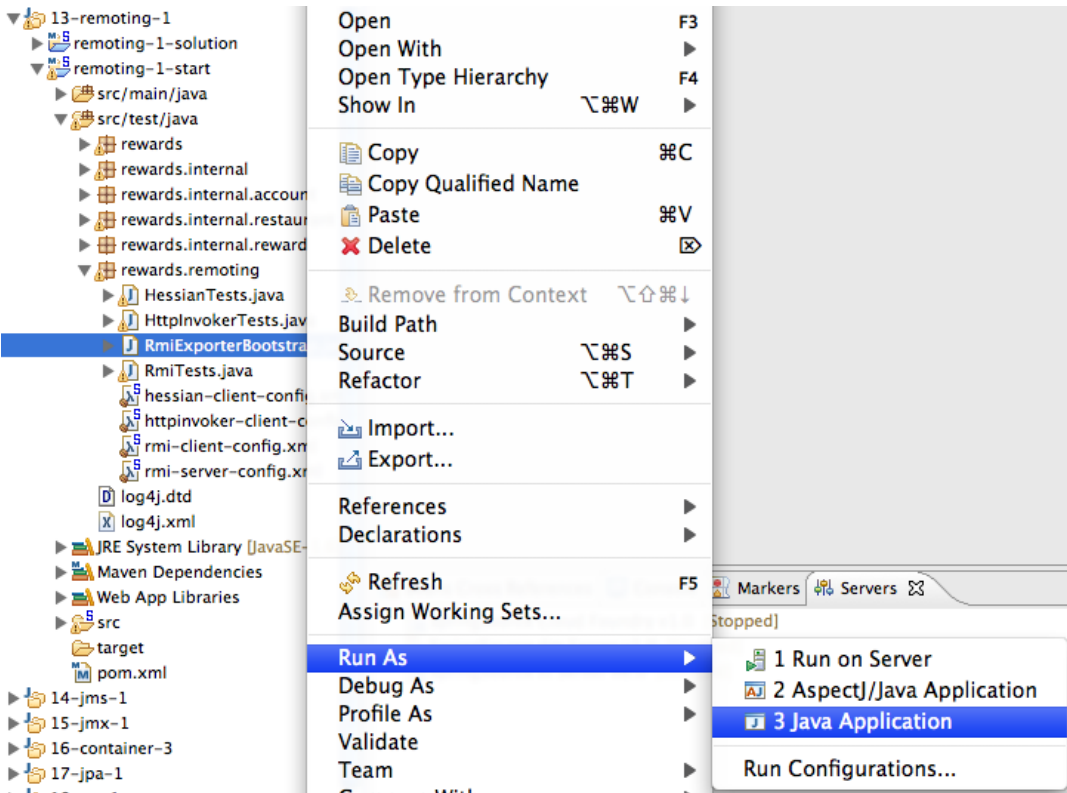


Figure 2: Run the RmiExporterBootstrap class

### 16.3.1.2. Define the client-side proxy

(TODO-03) Just as exposing the service is transparent, consumption is as well. On the client, Spring uses a proxy based mechanism to consume remoted services so that an application does not know that a dependency is remote. Spring provides a `RmiProxyFactoryBean` that generates this proxy.

Find and open the `rmi-client-config.xml` file in the `rewards.remoting` package. Create a bean named `rewardNetwork` of type `RmiProxyFactoryBean` and provide the `serviceInterface` and `serviceUrl` properties.



## Tip

The URL will be of the form: `rmi://host:port/serviceName`. Use 'localhost' for the host name and 1099 (the default RMI port) for the port number.

### 16.3.1.3. Run the tests

(TODO-04) Find and open the `RmiTests` class in the `rewards.remoting` package. Notice that most of the test has been written for you. First, create the application context by providing the name of the configuration file where the client-side context is configured. Use 'classpath:' as the prefix or start the file name with a leading '/'.

Next, notice that the test relies on `@Autowired` to inject a `RewardNetwork` implementation. By relying on polymorphism, we get a proxy injected that looks like `RewardNetwork` and directs the method call to a remote service.

(TODO-05) Next complete the `testRmiClient` method by calling the `rewardAccountFor(Dining)` method on the `RewardNetwork`. When complete, run this test. If you see the green bar you've completed this section. Move on to the next section.

## 16.3.2. Remoting with Spring's HttpInvoker

In the previous step, you tested an exporter/client-proxy pair with the RMI remoting protocol. Now you will establish a remoting scenario for the reward network based upon Spring's HTTP-based protocol.

### 16.3.2.1. Examine the web application deployment descriptor

Find and open the `web.xml` file in the `src/main/webapp` directory. Take note of the configuration of the `DispatcherServlet` (playing the role of 'Front Controller') and how it will be loading beans from the `/WEB-INF/remoting-config.xml` file.

### 16.3.2.2. Define the service exporter

(TODO-06) Find and open the `remoting-config.xml` file in the `src/main/webapp/WEB-INF` directory. In this file, create a bean definition for a `HttpInvokerServiceExporter`. Its `id` should be `/httpInvoker/rewardNetwork`. Inject values for the `serviceInterface` and `service` properties.

Once completed, go to the next step.

### 16.3.2.3. Start the web application

Now start the web application for this project. Once started, the welcome page (just a static index page at the context root) should be accessible as <http://localhost:8080/remoting>

### 16.3.2.4. Define the client-side proxy

(TODO-07) Find and open the `httpinvoker-client-config.xml` file in the `rewards.remoting` package in the

`src/test/java` source folder. In this file, define a bean definition of type `HttpInvokerProxyFactoryBean` named `rewardNetwork`. Set the `serviceInterface` and `serviceUrl` properties.



### Tip

The `serviceUrl` property is a concatenation of the web application url and the URL endpoint the service is exported to. Therefore, you should use `http://localhost:8080/remoting/rewards/httpInvoker/rewardNetwork` as the value.

Continue to the next step.

#### 16.3.2.5. Run the tests

(TODO-08) Find and open the `HttpInvokerTests` class in the `rewards.remoting` package. Notice that most of the test has been written for you. First, create the application context by providing the name of the configuration file where the client-side context is configured.

(TODO-09) Next complete the `testHttpInvokerClient` method by calling the `rewardAccountFor(Dining)` method on the `RewardNetwork`. Run this test, it should pass.

Finally, compare this test code to the `RmiTests` class you completed earlier. Other than the configuration, the client code is identical. This is part of the advantage of using Spring for remoting, all details regarding transport have been abstracted away and handled by the framework. Congratulations, you have now completed this lab.

---

# Chapter 17. Exposing SOAP Endpoints using Spring WS

## 17.1. Introduction

In this lab you will gain experience using Spring WS to expose the rewards application at a SOAP endpoint. You'll create an XSD defining the document to be exchanged across SOAP and then use Spring WS to create endpoint. Then you will use Spring WS to call that SOAP service from a client application.

### What you will learn:

1. How to use SOAP with a contract-first approach
2. How to use Spring WS to expose a SOAP endpoint
3. How to use Spring WS to consume a SOAP endpoint

### Specific subjects you will gain experience with:

1. XML Schema Definition (XSD)
2. The `WebServiceTemplate` template class

Estimated time to complete: 45 minutes

## 17.2. Instructions

The instructions for this lab are organized into three sections. In the first section, you'll define the contract that clients will use to communicate with you via SOAP. In the second section you'll export a SOAP endpoint for access. In the third section you'll consume that SOAP service using Spring WS.

### 17.2.1. Defining the message contract

When designing SOAP services the important thing to keep in mind is that the SOAP services are meant to be used by disparate platforms. To effectively accomplish this task, it is important that a contract for use of the service is designed in a way that is accessible to all platforms. The typical way to do this is by creating an



XML Schema Definition (XSD) of the messages that will be passed between the client and the server. In the following step you will define the message contract for the rewards application you created earlier.

#### 17.2.1.1. Create a sample message

(TODO 01) In the `ws` project, open the `sample-request.xml` file from the `src/main/webapp/WEB-INF/schemas` directory. This is currently a bare-bone sample message which only contains the root element and the desired namespace. Complete the sample message by adding attributes for `amount`, `creditCardNumber` and `merchantNumber`. Fill in some useful values in these attributes, like `100.00` for the amount, and so on.

#### 17.2.1.2. Infer the contract

(TODO 02) You now need to infer a contract out of your sample message, in our case an XML Schema (XSD). If you are already experienced with XSDs you could of course also skip the sample message part, and write your schema yourself. But it often saves some time if you start with the sample message and use tools to create a corresponding XSD.

You will use Trang in this lab, which is a Open Source schema converter. You already have a working Run Configuration in Eclipse. Just right-click on the file `ws-1 Trang.launch` in your project root and select "Run As/1 ws-1 Trang". Trang will create a XSD named `trang-schema.xsd` in `src/main/webapp/WEB-INF/schemas`.



### Tip

You need to refresh the project (select the project and press F5) before you see this file.

Open the file and inspect it. Trang should have generated a definition for the element `rewardAccountForDiningRequest` of type `complexType` with the 3 attributes in it from the previous step. Trang has also generated the types for the attributes. The `amount` attribute should be of type `xs:decimal` and the other two of type `xs:string`.



### Note

You may find the generated types to be different than the types described above. If this is the case, manually edit the types so they match the expected types.

When you've finished defining the `rewardAccountForDiningRequest` element place the cursor between the `xs:complexType` elements and select the 'Design' tab from the lower left of the editor window. If you have properly created the XSD, your `(rewardAccountForDiningRequestType)` will look like Figure 1.



(rewardAccountForDiningRequestType)	
⑧ amount	decimal
⑧ creditCardNumber	string
⑧ merchantNumber	string

Figure 1: (rewardAccountForDiningRequestType) structure

(TODO 03) We also need a response message, but this has already been created for you. Open the `reward-network.xsd` file from the `src/main/webapp/WEB-INF/schemas` directory. You'll see the definition of `rewardAccountForDiningResponse`. Copy your generated definition of `rewardAccountForDiningRequest` also into this file. Now you have completed your contract definition, move on to the next step.

## 17.2.2. Generate the classes with JAXB2

(TODO 04) In this lab we use JAXB2 to convert between Objects and XML. So we need to generate the classes out of your previously created XML Schema with `xjc`, the JAXB2 compiler. You will find an Ant buildfile for this in the root of the project with the name `create-classes.xml`. Right click on it and select "Run As/Ant Build". After refreshing the project (select the project and press F5) you will see the generated classes in the package `rewards.ws.types`.

Open `RewardAccountForDiningRequest` and see how the properties and types align with your schema definition.

## 17.2.3. Exporting the `RewardNetwork` as a SOAP endpoint

### 17.2.3.1. Add the `MessageDispatcherServlet`

(TODO 05) Much like Spring MVC, Spring WS uses a single servlet endpoint for the handling of all SOAP calls. Open the `web.xml` file in the `src/main/webapp/WEB-INF` directory. Add a new servlet named `rewards` with a servlet class of `org.springframework.ws.transport.http.MessageDispatcherServlet`.

Next define an initialization parameter for the servlet called `contextConfigLocation` that has a value that points to the servlet configuration file defined in the same directory.

### 17.2.3.2. Create the SOAP endpoint

(TODO 06) Now that the Spring WS infrastructure has been set up, you must create an endpoint to service the `RewardNetwork` requests. You will use the annotation style mapping in this lab.

Such an endpoint has been started for you. Open `RewardNetworkEndpoint` from the `reward.ws` package. Notice that the class is already annotated with `@Endpoint` and is autowired with a `RewardNetwork` service. The only missing piece is the method which processes the request. Create a new method: you can choose any name you like, something like `reward` would make sense. Give it a parameter of type `RewardAccountForDiningRequest` and use `RewardAccountForDiningResponse` as the return type. These are your JAXB2 generated classes: they can be automatically converted for you by Spring WS using JAXB2, but you'll have to annotate the parameter with `@RequestPayload` and the method with `@ResponsePayload` to indicate that this is necessary!

Now you have to implement the logic inside of the method. As the generated classes are not your domain classes you must convert them to the classes which are used in the service. Create a new `Dining` object with `Dining.createDining(String amount, String creditCardNumber, String merchantNumber)`. You will get the needed values out of `RewardAccountForDiningRequest`. Then call the method `rewardAccountFor` on the `rewardNetwork`. Finally create a `RewardAccountForDiningResponse` object and return it.

Complete your endpoint now by mapping the method to the correct request by placing an annotation on the method that uses the payload root's element name.

### 17.2.3.3. Complete the Spring WS configuration

(TODO 07) Open the `ws-config.xml` file from the `src/main/webapp/WEB-INF` directory. This file contains the configuration for Spring Web Services. Notice how component scanning is already enabled: this will ensure that your endpoint class is defined as a Spring bean automatically. You just have to use the new `ws:namespace` to enable the annotation-driven programming model, which will enable support for all the annotations you've applied in your endpoint class. You don't have to explicitly configure an OXM marshaller for JAXB2, Spring-WS 2.0 enables it automatically when you've added the annotation-driven model. Once you've completed this move on to the next step.

### 17.2.3.4. Start the web application

(TODO 08) Now that the SOAP endpoint has been wired properly you must start the web application to export it. Start the web application for this project as described Appendix C, *Using Web Tools Platform (WTP)*. Once started, the welcome page (just a static index page at the context root) should be accessible as <http://localhost:8080/ws>

## 17.2.4. Consuming services from a SOAP endpoint

At this point you've successfully exported a service to a SOAP endpoint without changing the original class. If you are acting as a provider of services to other clients this would be all that you need to do. But there are many cases where you need to consume SOAP services as well. When doing this, it is important to hide the fact that SOAP is being used from the client.

### 17.2.4.1. Test the web service

(TODO 09) Open and run the `SoapRewardNetworkTests` test class in the `rewards.ws.client` package of the `src/test/java` source folder. If you see a green bar, your web service works properly. Notice that the test method `testWebServiceWithXml()` uses plain XML (in this case DOM) and not the generated classes. As we started by defining the contract, JAXB2 is just an implementation detail and therefore the client doesn't have to use it.

### 17.2.4.2. Using the TCP/IP monitor to see the SOAP messages

(TODO 10) Whether your test ran OK or not, you've probably noticed that there's not much to see when you run it: the actual content of the SOAP request and response is not available. When writing web services or web services clients, it's nice to see what XML is actually sent from the client to the server and vice versa. Several tools exist to help you with this. One of these tools is built-in with Eclipse's Web Tools Plugin and is called the *TCP/IP Monitor*. It is a view that you can add to your perspective.

Type `Ctrl-3` and enter `TCP` plus `Enter` to add the TCP/IP Monitor view to your perspective. Click the small arrow pointing downwards and choose "properties".

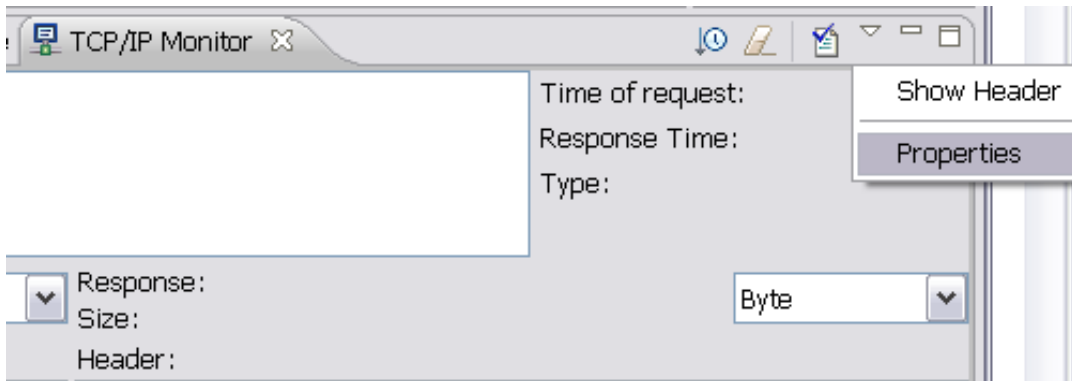


Figure 2: The "properties" menu entry of the TCP/IP Monitor view

Choose "Add..." to add a new monitor. As local monitoring port, enter 8081 since this port is probably unused. As host name, enter "localhost" and as port enter 8080 since this is the port that Tomcat is running on. Press

OK and then press "Start" to start the newly defined monitor.

Now open `client-config.xml` class and change the port number in the request URL from 8080 to 8081. This ensures that the request will go through our monitor, which will log and forward it to the server. The response will follow the same route back from the server to the client. Run the test again. Now switch to the Monitor view: you should see one request and response passing by. If you change the pulldowns from "Byte" to to "XML", the view will render the messages in a more readable way.

This is an excellent tool to help you to debug your web services: if there was an error when running your test, try to fix it now using the monitor as a tool to see what the actual request and response are holding.

#### **17.2.4.3. Using WebServiceTemplate with JAXB2**

(TODO 11) There is also an empty method called `testWebServiceWithJAXB` in `SoapRewardNetworkTests`. This method should do the same as `testWebServiceWithXml()`, but by using JAXB2 and not DOM. Implement this method now and use your generated JAXB2 classes. The `marshalSendAndReceive()` from the `WebServiceTemplate` should be the right one for this. Pass in `RewardAccountForDiningRequest` and you will get back a `RewardAccountForDiningResponse`. Use the input data and the assertions from `testWebServiceWithXml()`. If you see a green bar, you've completed this lab. Congratulations.

---

# Chapter 18. ORM simplification using Spring

## 18.1. Introduction

In this lab you will implement the repositories of the rewards application with Hibernate. You'll configure Hibernate to map database rows to objects, use native Hibernate APIs to query objects, and write tests to verify mapping behavior.

### What you will learn:

1. How to write Hibernate mapping information to map relational structures to domain objects
2. How to use Hibernate APIs to query objects
3. How to configure Hibernate in a Spring environment
4. How to test Hibernate-based repositories

### Specific subjects you will gain experience with:

1. Hibernate and JPA mapping Annotations
2. `SessionFactory` and `Session`
3. `AnnotationSessionFactoryBean`

Estimated time to complete: 45 minutes

## 18.2. Quick Instructions

If you feel you have a good understanding of the material, you can work with the TODOs listed in the `Tasks` view in Eclipse/STS. To display them, click on Window -> Show view -> Tasks. Alternatively, the next section contains more detailed step-by-step instructions. Each task in STS is also described in more detail by a corresponding section in the step-by-step instructions.

## 18.3. Detailed Instructions

The instructions for this lab are organized into three sections. The first two sections focus on using Hibernate within a *domain module* of the application. The first addresses the `Account` module, and the second addresses the `Restaurant` module. In each of these sections, you'll map that module's domain classes using Hibernate, implement a Hibernate-based repository if needed, and unit test your repository to verify Hibernate mapping behavior. In the third and final section, you'll integrate Hibernate into the application configuration and run a top-down system test to verify application behavior.

### 18.3.1. Using Hibernate in the Account module

#### 18.3.1.1. Create the `Account` mapping using Annotations

Recall the `Account` entity represents a member account in the reward network that can make contributions to its beneficiaries. In this step, you'll finish the Hibernate mappings that map the `Account` object graph to the database.

##### Tasks

1. Inside the package `rewards.internal.account`, open the `Account` class. This file needs more annotations to define how it is mapped to the database.
2. Notice the mapping has already been started for you. Specifically, the `Account` class has already been mapped to the `T_ACCOUNT` table with the `entityId` property mapped to the `ID` primary key column.

Complete `TODO 1` by mapping the remaining `Account` properties. This includes the `number`, `name`, `beneficiaries` and `creditCards` properties. Use the [\[reward dining database schema\]](#) to help you.



#### Tip

JPA knows how to map primitive types, Strings and BigDecimals. But it still needs to know what columns they correspond to.



#### Tip

Since an `Account` can have many beneficiaries, its `beneficiaries` property is a collection. Map this property as a one-to-many relationship. The foreign key column in the beneficiary table is `ACCOUNT_ID`. Same goes for the `creditCards` property.

3. When you have finished mapping the `Account` entity, complete the mapping of its `Beneficiary` associate. Recall that an `Account` distributes contributions to its beneficiaries based on an allocation percentage.

Complete the `Beneficiary` mapping by opening the `Beneficiary` class and adding mappings for the `name`, `allocationPercentage` and `savings` properties (TODO 2).

4. The `CreditCard` entity is very simple and has been mapped for you. Take a quick look at the class to see how it has been mapped.

When you have completed mapping the `Account`, `Beneficiary` classes, move on to the next step!

#### 18.3.1.2. Review `HibernateAccountRepository`

You just defined the metadata Hibernate needs to map rows in the account tables to an account object graph. Now you will check the data access logic to query `Account` objects.

##### Tasks

1. Open `HibernateAccountRepository`. The `findByCreditCard(String)` method has already been implemented. You should review this method before moving to the next step. You will need to write a similar query in a short while.

#### 18.3.1.3. Test `HibernateAccountRepository`

It is now time to proof-test your Hibernate configuration.

##### Tasks

1. In the `src/test/java` source folder, run the `rewards.internal.account.HibernateAccountRepositoryTests` class. When you get the green bar, your repository works indicating your account object-to-relational mappings are correct. Move on to the next section!
2. Review the methods in the `AccountRepository` interface. It is different. If you aren't sure compare it to a previous lab.

What has changed?

Specifically, the `updateBeneficiaries(Account)` method has been removed because it is simply no longer needed with a ORM capable of transparent persistence. Changes made to the `Beneficiaries` of an account will automatically be persisted to the database when the transaction is committed. Explicit updates of persistent domain objects are no longer necessary, as long as those changes are made within the scope of a `Session`. This is the power of an ORM over managing database data manually.



## 18.3.2. Using Hibernate in the Restaurant module

### 18.3.2.1. Create the `Restaurant` mapping

Recall the `Restaurant` entity represents a merchant in the reward network that calculates how much benefit to reward to an account for dining. In this step, you'll create the Hibernate mapping file that maps the `Restaurant` object graph to the database.

#### Tasks

1. In the package `rewards.internal.restaurant`, open the `Restaurant` class. This is the file that will define the `Restaurant` object-to-relational mapping rules using annotations.

Finish mapping the `Restaurant` object (TODO 3). If you are not sure what to do, refer back to the `Account` class. The mappings are similar



#### Tip

Use the reward dining database schema to help you.



#### Note

The `benefitAvailabilityPolicy` is an enumeration. However, JPA can translate column values to and from an enumeration, provided the value in the database is a string. Thus the enumerated value `ALWAYS_AVAILABLE` is mapped to the string `'ALWAYS_AVAILABLE'`. An enum can then be mapped just like any other.

In practice, enumerated values are not usually stored as strings. It is possible to copy an enumerated data-member to a encoded value in the database (typically a number or a code) but that is outside the scope of this section.

When you have completed the `Restaurant` mapping, move on to the next step!

### 18.3.2.2. Implement `HibernateRestaurantRepository`

You just defined the metadata Hibernate needs to map rows in the `T_RESTAURANT` table to a `Restaurant` object graph. Now you will implement the data access logic to query `Restaurant` objects.

#### Tasks

1. Open `HibernateRestaurantRepository`.
2. Complete TODO 4 by implementing the `findByMerchantNumber(String)` method.



## Tip

Use the `createQuery(String)` method to find the Restaurant.

### 18.3.2.3. Test `HibernateRestaurantRepository`

#### Tasks

1. In the `src/test/java` source folder, run the `HibernateRestaurantRepositoryTests` class. When you get the green bar your repository implementation works. Move on to the next section!

## 18.3.3. Integrating Hibernate into the Rewards Application

Now that you have tested your Hibernate based repositories, you'll add them to the overall application configuration. In this section you'll update the application configuration as well as the system test configuration. Then, you'll run your system test to verify the application works!

### 18.3.3.1. Define the Hibernate configuration for the application

#### Tasks

1. In the `rewards.internal` package, open `application-config.xml`. In this file, define beans for the `HibernateAccountRepository` and the `HibernateRestaurantRepository` (TODO 5). Remember that each of the repositories needs a `SessionFactory` injected. The session factory will be defined as part of your test infrastructure.
2. Next, in the `src/test/java` source folder, open `rewards/system-test-config.xml`. You will define there 2 beans of type `SessionFactory` and `HibernateTransactionManager` (TODO 6).

Firstly, define a factory to create the `SessionFactory` you referenced earlier. The factory bean's class is `AnnotationSessionFactoryBean`. Set the `dataSource` and `annotatedClasses` properties appropriately.

3. You can set additional Hibernate configuration properties by setting the `hibernateProperties` property. For example, you could pass in `hibernate.show_sql=true` to output the SQL statements that Hibernate is passing to the database and `hibernate.format_sql=true` to format the SQL statements.

4. Finally, define a `transactionManager` bean so the Reward Network can drive transactions using Hibernate APIs. Use the `HibernateTransactionManager` implementation. Set its `sessionFactory` property appropriately.
5. Now go to the Spring Explorer view in Eclipse and show the graph of the `hibernate -> system-test-config.xml`. If you configured your application context properly the graph should look something like Figure 18.1:

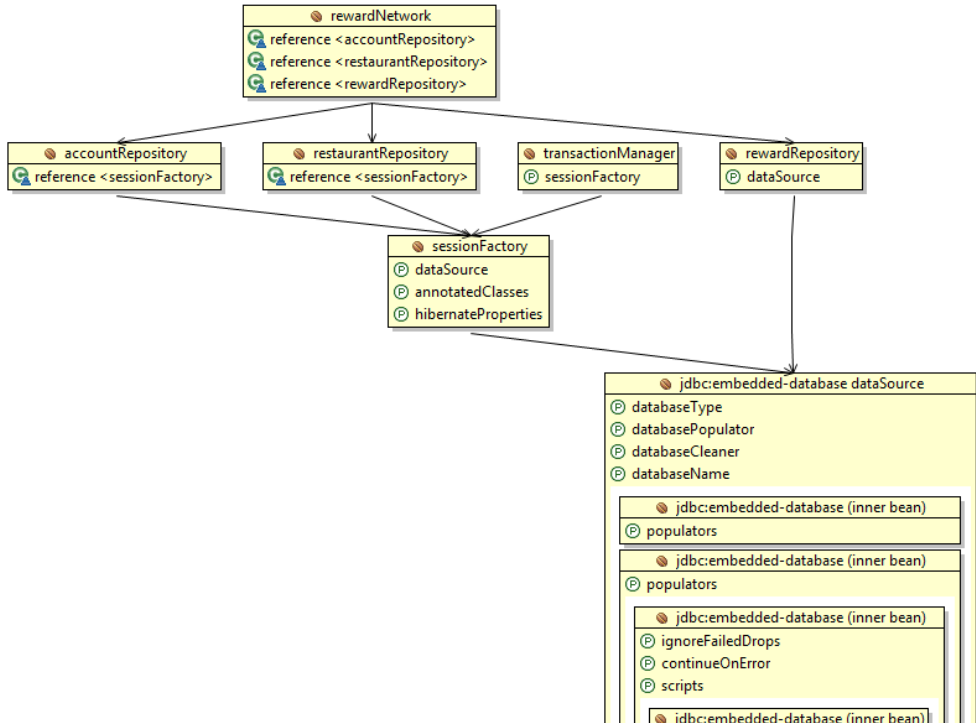


Figure 18.1. Spring Bean Configuration



## Tip

It sometimes happen that the graph does not refresh properly. If that is the case, you just need to follow those steps:

- a. In the Spring explorer view, right click on the `hibernate` project and select `Properties`

- b. Go into the Beans Support section, uncheck Enable support for import elements in configuration files and click on OK
- c. Go back to the Beans Support section and check Enable support for import elements in configuration files again

If your graph looks correct, you've completed this step. Move on to the next step!

#### **18.3.3.2. Run the application system test**

Interfaces define a contract for behavior and abstract away implementation details. Plugging in Hibernate-based implementations of the repository interfaces should not change the overall application behavior. So our integration tests should still work.

To verify this, find and run the `RewardNetworkTests` class. If you get a green bar, the application is now running successfully with Hibernate for object persistence!

Congratulations, you have completed the lab!

---

# Appendix A. Spring XML Configuration Tips

## A.1. Bare-bones Bean Definitions

```
<bean id="rewardNetwork" class="rewards.internal.RewardNetworkImpl">
</bean>

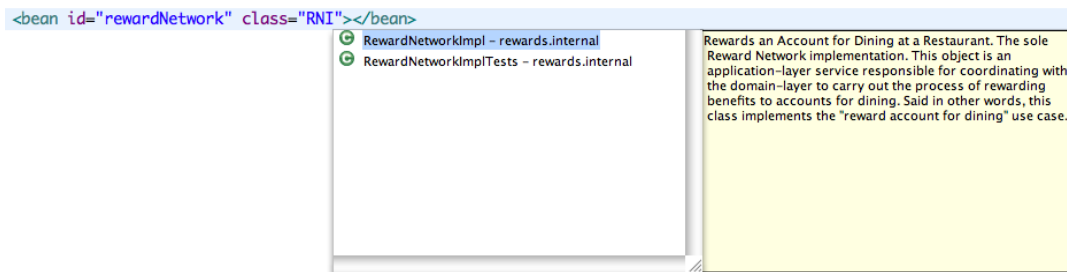
<bean id="accountRepository" class="rewards.internal.account.JdbcAccountRepository">
</bean>

<bean id="restaurantRepository" class="rewards.internal.restaurant.JdbcRestaurantRepository">
</bean>

<bean id="rewardRepository" class="rewards.internal.reward.JdbcRewardRepository">
</bean>
```

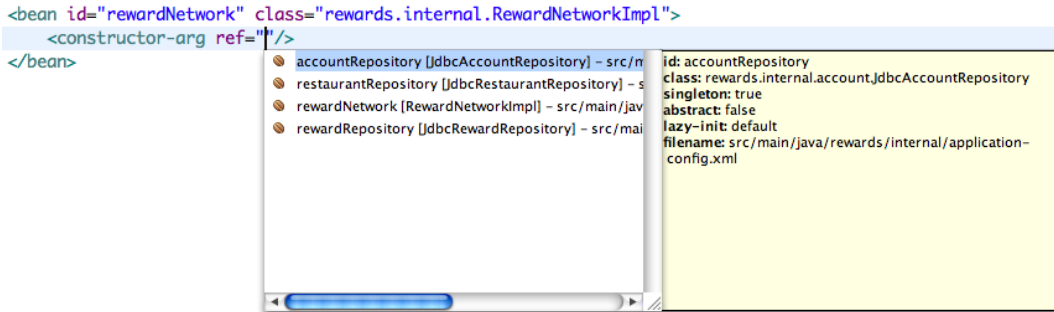
Bare-bones bean definitions

## A.2. Bean Class Auto-Completion



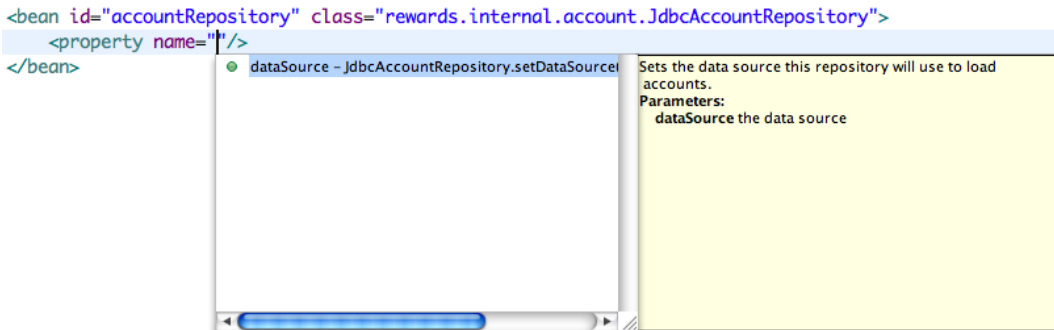
Bean class auto-completion

## A.3. Constructor Arguments Auto-Completion



Constructor argument auto-completion

## A.4. Bean Properties Auto-Completion



Bean property name completion

---

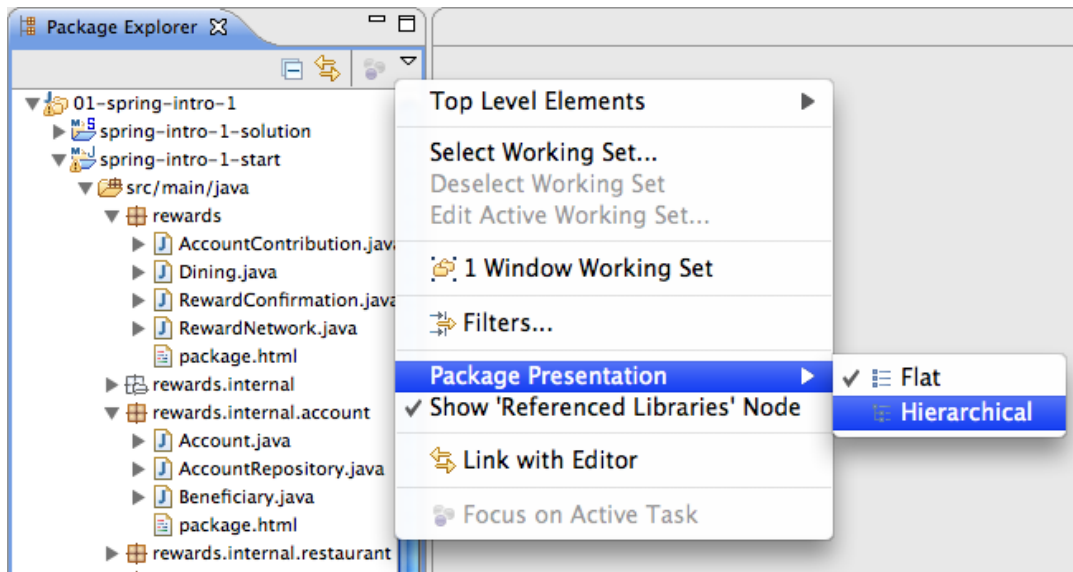
# Appendix B. Eclipse Tips

## B.1. Introduction

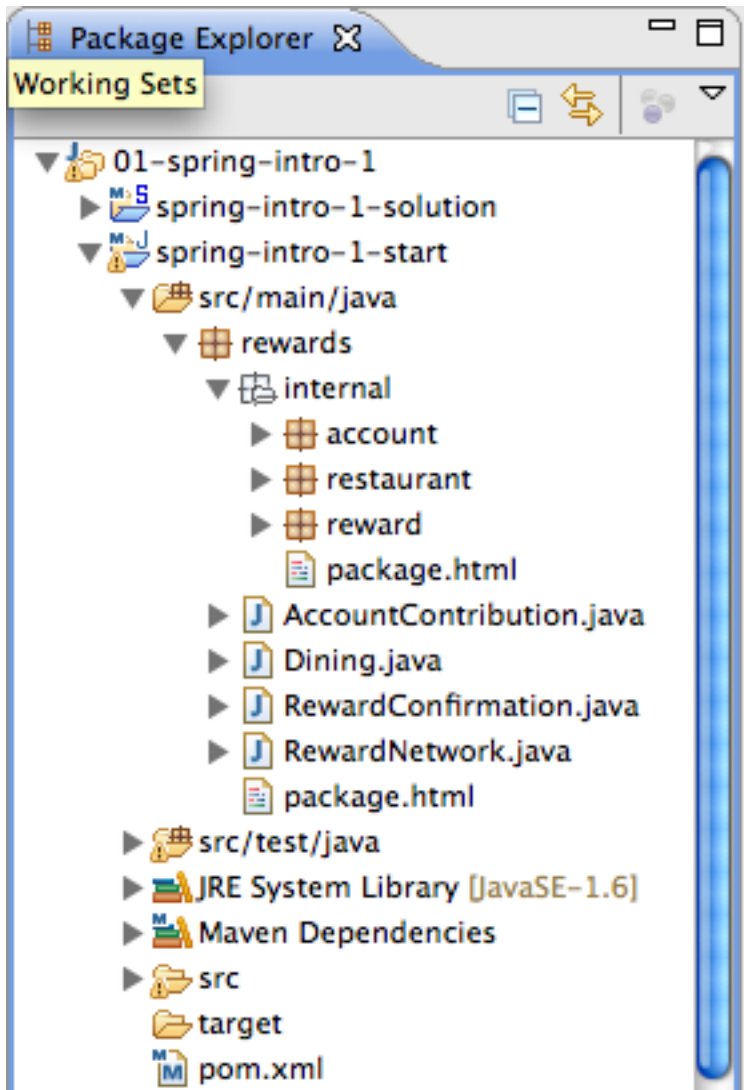
This section will give you some useful hints for using Eclipse.

## B.2. Package Explorer View

Eclipse's Package Explorer view offers two ways of displaying packages. Flat view, used by default, lists each package at the same level, even if it is a subpackage of another. Hierarchical view, however, will display subpackages nested within one another, making it easy to hide an entire package hierarchy. You can switch between hierarchical and flat views by selecting the menu inside the package view (represented as a triangle), selecting either Flat or Hierarchical from the Package Presentation submenu.



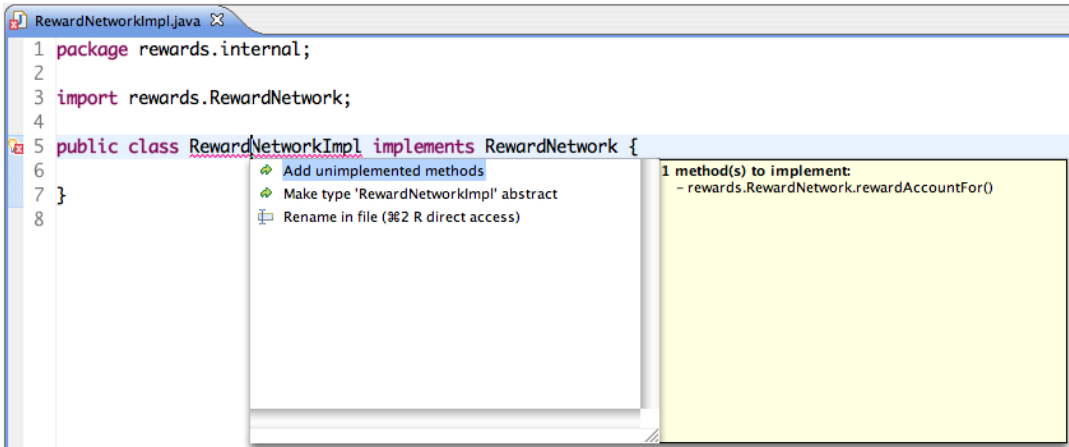
Switch between hierarchical and flat views by selecting the menu inside the package view (represented as a triangle), selecting either Flat or Hierarchical from the Package Presentation submenu



The hierarchical view shows nested packages in a tree view

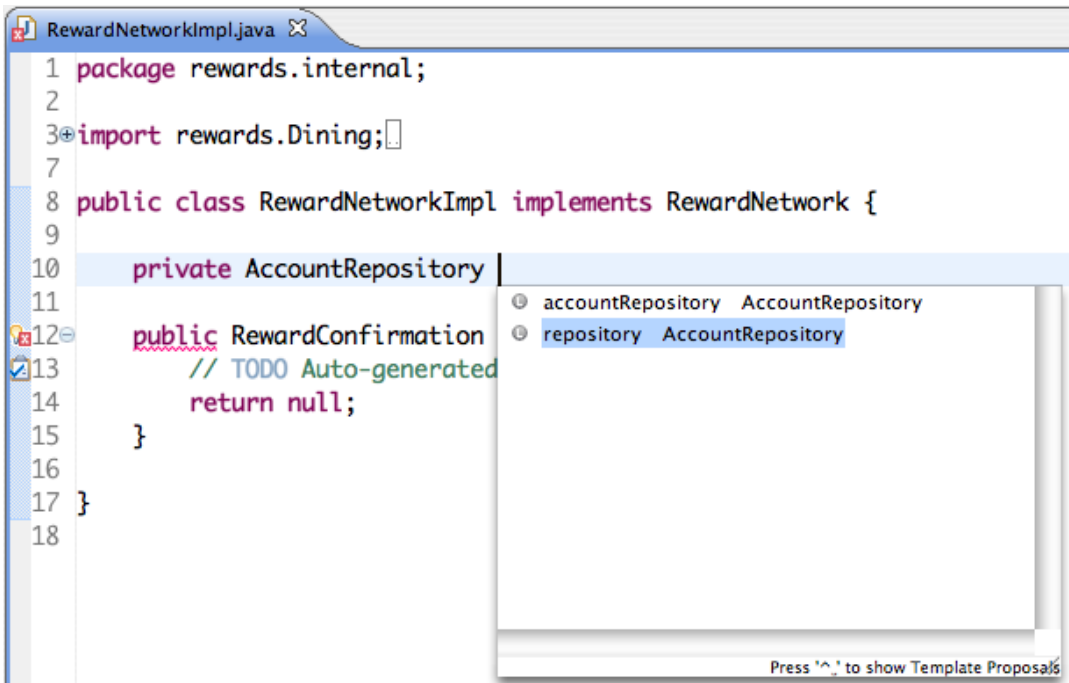
## B.3. Add Unimplemented Methods





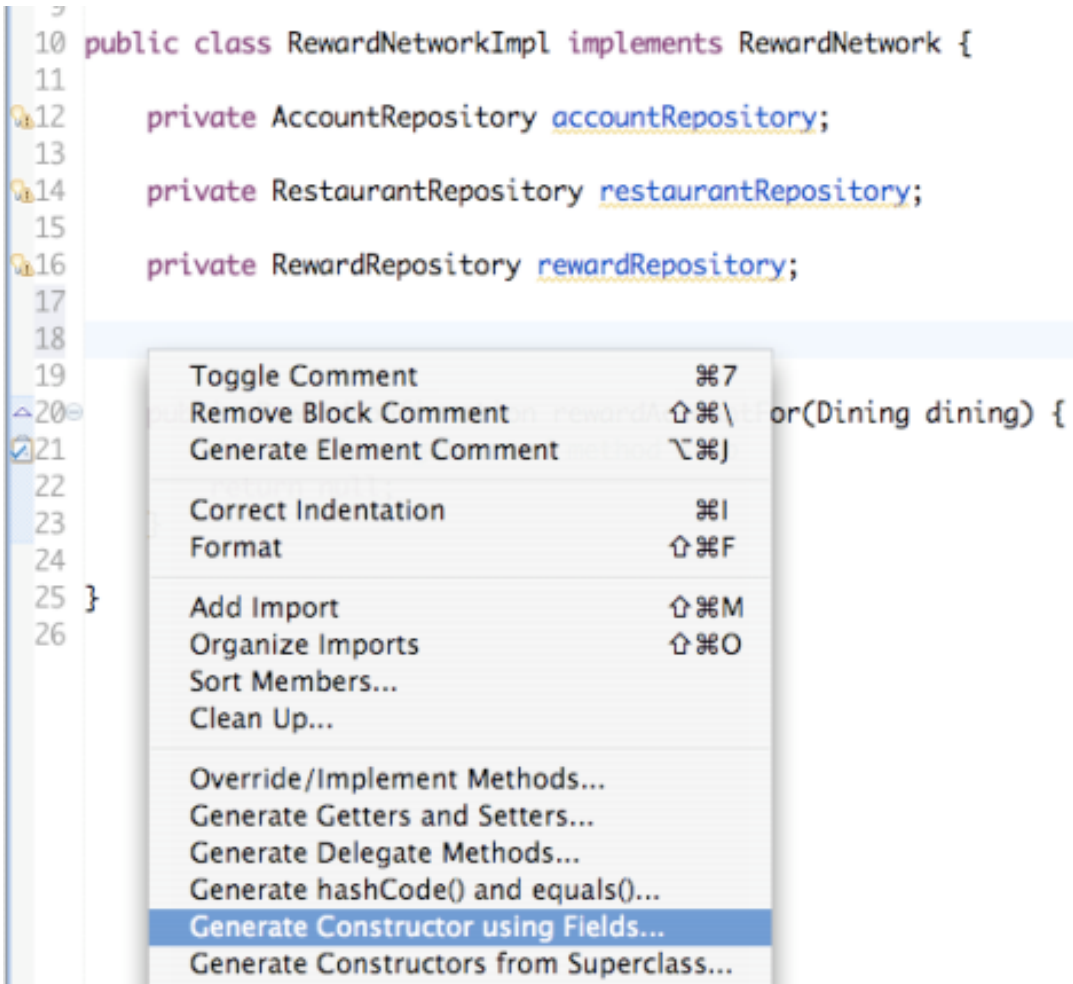
"Add unimplemented methods" quick fix

## B.4. Field Auto-Completion



Field name auto-completion

## B.5. Generating Constructors From Fields

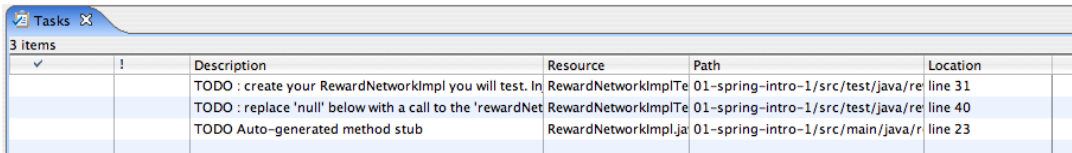


"Generate Constructor using Fields" using the Source Menu (ALT + SHIFT + S)

## B.6. Field Naming Conventions

A field's name should describe the role it provides callers, and often corresponds to the field's type. It should not describe implementation details. For this reason, a bean's name often corresponds to its service interface. For example, the class `JdbcAccountRepository` implements the `AccountRepository` interface. This interface is what callers work with. By convention, then, the bean name should be `accountRepository`.

## B.7. Tasks View

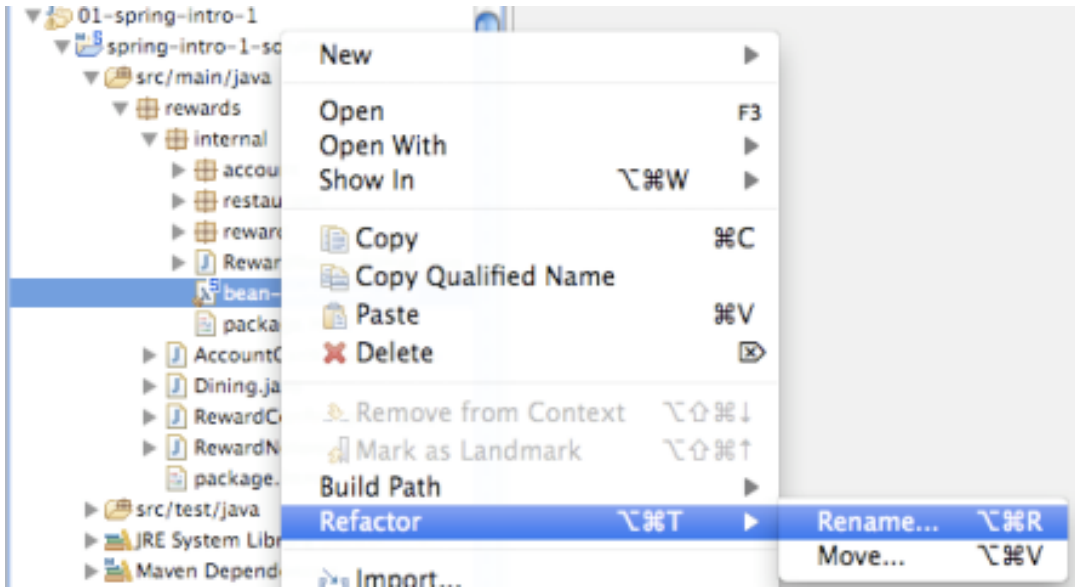


Tasks					
3 items					
✓	!	Description	Resource	Path	Location
		TODO : create your RewardNetworkImpl you will test. In	RewardNetworkImplTe	01-spring-intro-1/src/test/java/re	line 31
		TODO : replace 'null' below with a call to the 'rewardNet	RewardNetworkImplTe	01-spring-intro-1/src/test/java/re	line 40
		TODO Auto-generated method stub	RewardNetworkImplJa	01-spring-intro-1/src/main/java/r	line 23

The tasks view in the bottom right page area

You can configure the Tasks View to only show the tasks relevant to the current project. In order to do this, open the dropdown menu in the upper-right corner of the tasks view (indicated by the little triangle) and select 'Configure Content...'. Now select the TODO configuration and from the Scopes, select 'On any element in same project'. Now if you have multiple project opened, with different TODOs, you will only see those relevant to the current project.

## B.8. Rename a File



Renaming a Spring configuration file using the Refactor command

---

# Appendix C. Using Web Tools Platform (WTP)

## C.1. Introduction

This section of the lab documentation describes the general configuration and use of the [Web Tools Platform](#) plugin for Eclipse using Tomcat 6.0 for testing web application labs and samples.

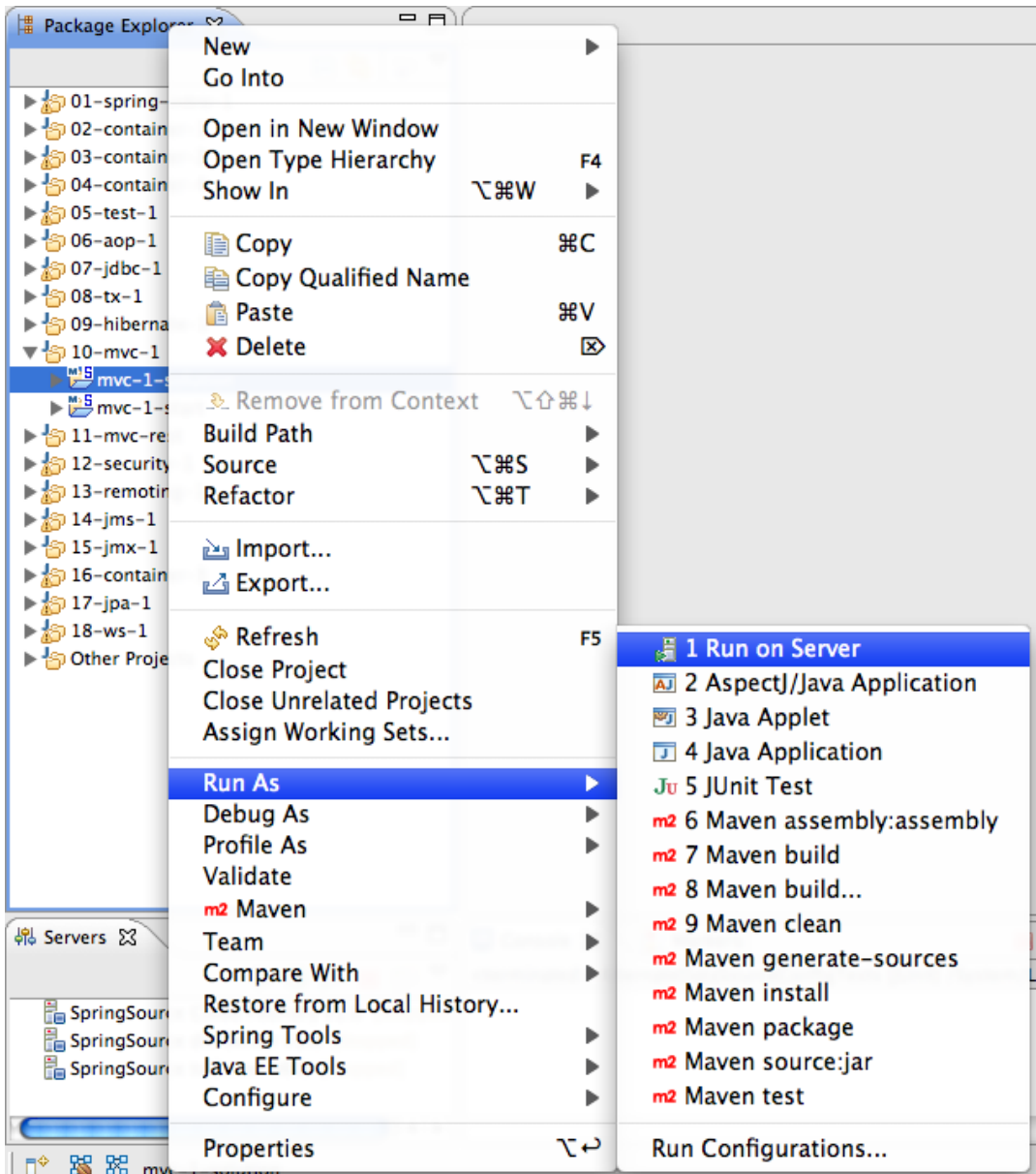
## C.2. Verify and/or Install the Tomcat Server

The *Servers* view provided by the WTP plugin needs to be open, so that you can see the status of the Tomcat server. Verify that you can see the Servers view. The tab for this view should be beside other views such as *Problems* and *Console*. If the view is not open, open it now via the 'Windows / Show View / Other ... / Server / Servers' menu sequence.

Your workspace may already contain a pre-created entry for a Tomcat 6.0 server, visible in the *Servers* view as 'Tomcat v6.0 Server at localhost'. If it does, proceed to the next step. Otherwise, you will need to install a new server runtime. Do this by clicking on the Spring icon in the toolbar below the main menus. This will launch the Spring Tool Suite Dashboard, on which there is a *Configuration* tab with a link to 'Create Server Instance'. Click on this link and verify that a server runtime is created in the *Servers* view. Please ask your instructor for assistance if you get stuck.

## C.3. Starting & Deploying to the Server

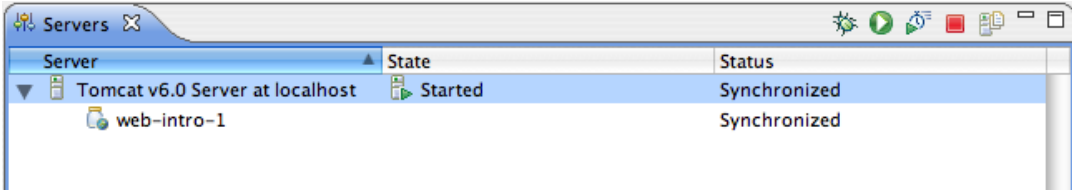
The easiest way to deploy and run an application using WTP is to right-click on the project and select 'Run As' then 'Run On Server'.



Run On Server

The console view should show status and log information as Tomcat starts up, including any exceptions due to project misconfiguration.

After everything starts up, it should show as a deployed project under the server in the *Servers* tab.



### Running on Server

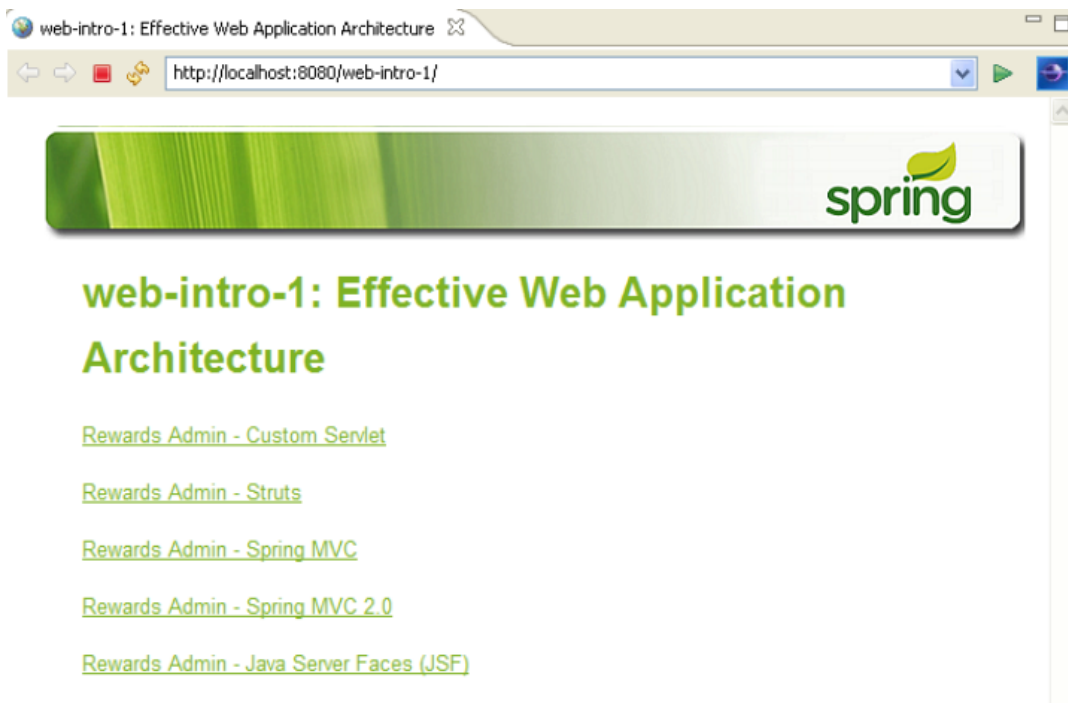
Similarly, the server can be shut down (stopped) by pressing the red box in the toolbar of the Servers tab.



### Tip

When you run the server as described above, you are running it against the project in-place (with no separate deployment step). Changes to JSP pages will not require a restart. However, changes to Spring Application Context definition files will require stopping and restarting the server for them to be picked up, since the application context is only loaded once at web app startup.

WTP will launch a browser window opened to the root of your application, making it easy to start testing the functionality.



Start Browser



## Tip

It is generally recommended that you only run one project at a time on a server. This will ensure that as you start or restart the server, you only see log messages in the console from the project you are actively working in. To remove projects that you are no longer working with from the server, right click on them under the server in the *Servers* view and select 'Remove'.