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:

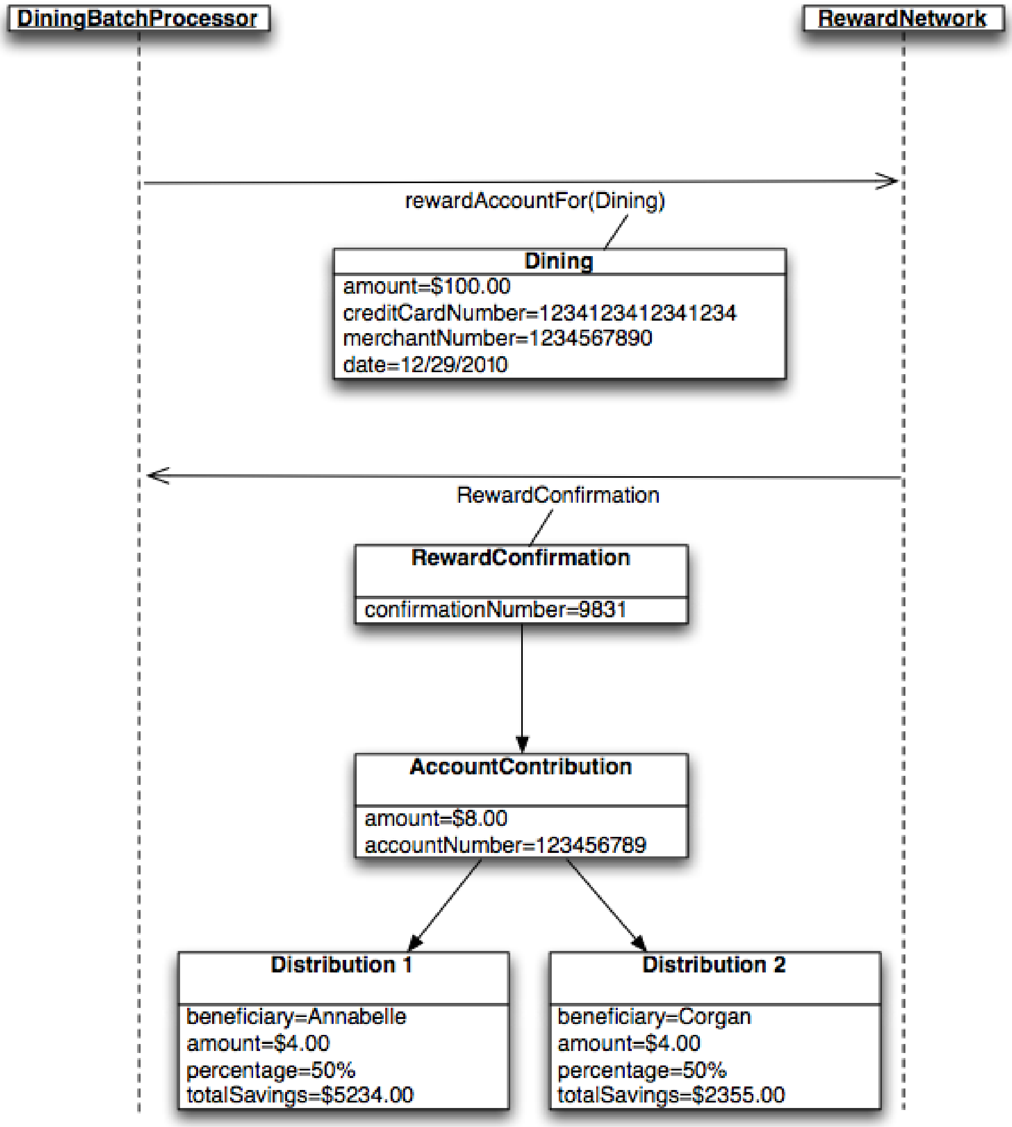


Figure 3: A client calling the `RewardNetwork` to reward an account for dining.

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:

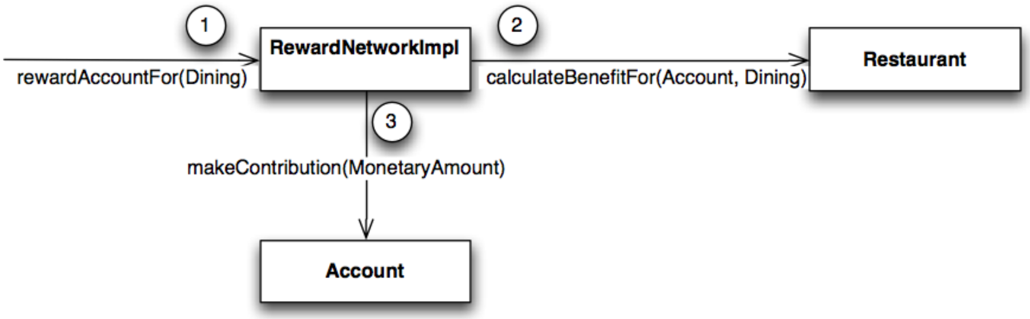

1
rewardAccountFor(Dining)
RewardNetworkImpl
2
calculateBenefitFor(Account, Dining)
Restaurant
3
makeContribution(MonetaryAmount)
Account

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:

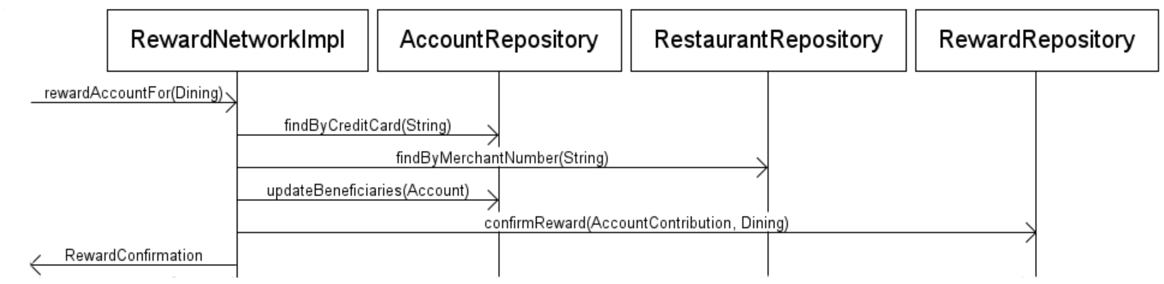


Figure 5: The complete `RewardNetworkImpl` `rewardAccountForDining(Dining)` sequence

4. Reward Dining Database Schema

The Reward Dining applications share a database with this schema:

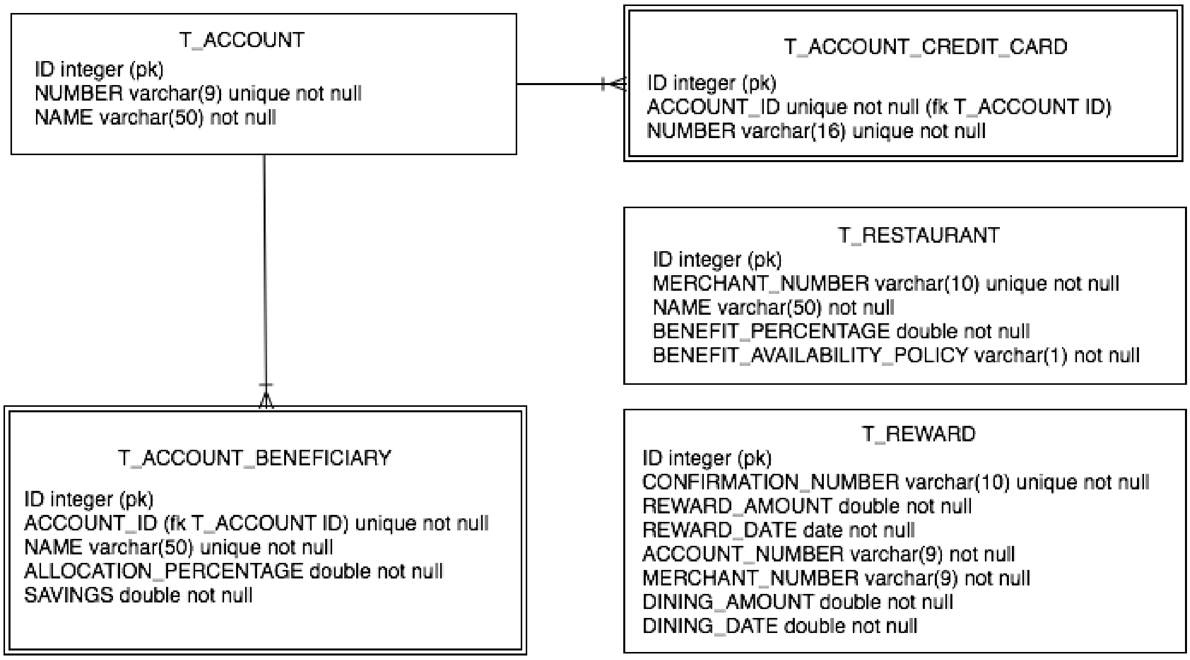

T_ACCOUNT
ID integer (pk)
NUMBER varchar(9) unique not null
NAME varchar(50) not null
T_ACCOUNT_CREDIT_CARD
ID integer (pk)
ACCOUNT_ID unique not null (fk T_ACCOUNT ID)
NUMBER varchar(16) unique not null
T_RESTAURANT
ID integer (pk)
MERCHANT_NUMBER varchar(10) unique not null
NAME varchar(50) not null
BENEFIT_PERCENTAGE double not null
BENEFIT_AVAILABILITY_POLICY varchar(1) not null
T_ACCOUNT_BENEFICIARY
ID integer (pk)
ACCOUNT_ID (fk T_ACCOUNT ID) unique not null
NAME varchar(50) unique not null
ALLOCATION_PERCENTAGE double not null
SAVINGS double not null
T_REWARD
ID integer (pk)
CONFIRMATION_NUMBER varchar(10) unique not null
REWARD_AMOUNT double not null
REWARD_DATE date not null
ACCOUNT_NUMBER varchar(9) not null
MERCHANT_NUMBER varchar(9) not null
DINING_AMOUNT double not null
DINING_DATE double not null

Figure 6: The Reward Dining database schema

Part I. Labs

Chapter 1. spring-intro-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 familarized 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!*

What you will learn:

1. Basic features of the SpringSource 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 SpringSource Tool Suite

The SpringSource 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 SpringSource Tool Suite by using the shortcut link on your desktop.

Figure 1: STS Desktop Icon

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

springsourcetoolsuite™
© Copyright SpringSource and others. 2007, 2010. All right reserved. Java and all Java-related trademarks and logos are trademarks of Sun Microsystems, Inc. in the U.S. other countries, or both. Eclipse is a trademark of the Eclipse Foundation, Inc.
springsource®
A division of vmware®

Figure 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

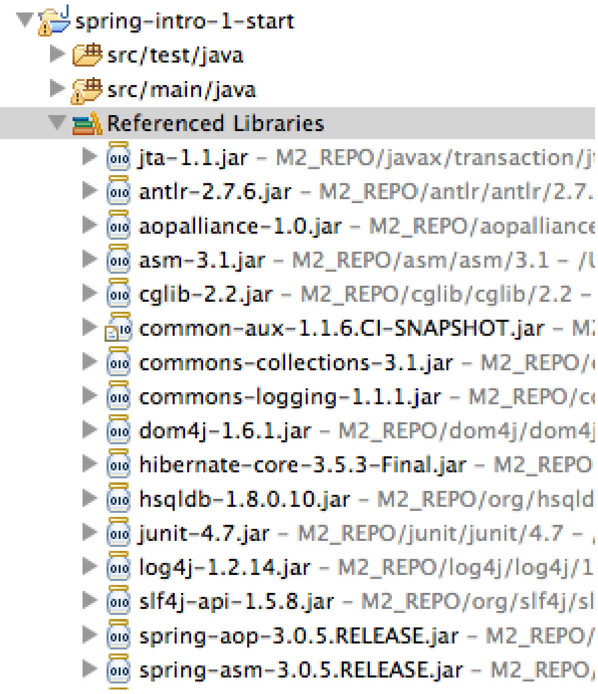
If you've just opened STS, it may be still starting up. Wait several moments until the progress indicator on the bottom right finishes. When complete, you should have no red error markers within the *Package Explorer* or *Problems* views

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-1* Working Set. Within you'll find two projects: *spring-intro-1-start* and *spring-intro-1-solution*. This pair of *-start* and *-stolution* projects is a common pattern throughout the labs in this course.

Open the *spring-intro-1-start* project and expand its *Referenced Libraries* node. Here you'll see a number of dependencies similar to the screenshot below:



```
▼ spring-intro-1-start
    ▶ src/test/java
    ▶ src/main/java
    ▼ Referenced Libraries
        ▶ jta-1.1.jar - M2_REPO/javax/transaction/j
        ▶ antlr-2.7.6.jar - M2_REPO/antlr/antlr/2.7.
        ▶ aopalliance-1.0.jar - M2_REPO/aopalliance
        ▶ asm-3.1.jar - M2_REPO/asm/asm/3.1 - /U
        ▶ cglib-2.2.jar - M2_REPO/cglib/cglib/2.2 -
        ▶ common-aux-1.1.6.CI-SNAPSHOT.jar - M2
        ▶ commons-collections-3.1.jar - M2_REPO/
        ▶ commons-logging-1.1.1.jar - M2_REPO/c
        ▶ dom4j-1.6.1.jar - M2_REPO/dom4j/dom4j
        ▶ hibernate-core-3.5.3-Final.jar - M2_REPO
        ▶ hsqldb-1.8.0.10.jar - M2_REPO/org/hsqld
        ▶ junit-4.7.jar - M2_REPO/junit/junit/4.7 -
        ▶ log4j-1.2.14.jar - M2_REPO/log4j/log4j/1
        ▶ slf4j-api-1.5.8.jar - M2_REPO/org/slf4j/sl
        ▶ spring-aop-3.0.5.RELEASE.jar - M2_REPO/
        ▶ spring-asm-3.0.5.RELEASE.jar - M2_REPO
```

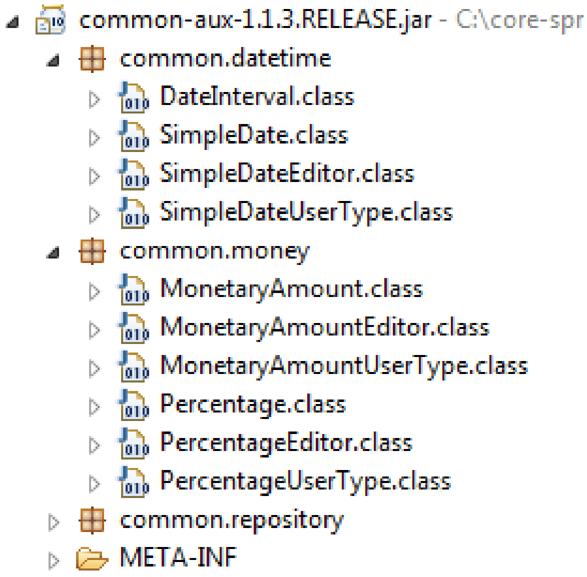
Figure 3: Referenced Libraries

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.

Notice near the top there is a jar named *common-aux*.



```
common-aux-1.1.3.RELEASE.jar - C:\core-spr
  common.datetime
    DateInterval.class
    SimpleDate.class
    SimpleDateEditor.class
    SimpleDateUserType.class
  common.money
    MonetaryAmount.class
    MonetaryAmountEditor.class
    MonetaryAmountUserType.class
    Percentage.class
    PercentageEditor.class
    PercentageUserType.class
  common.repository
  META-INF
```

Figure 4: CommonAux components

This dependency 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.

1.2.2. 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:

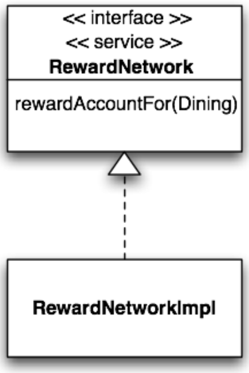
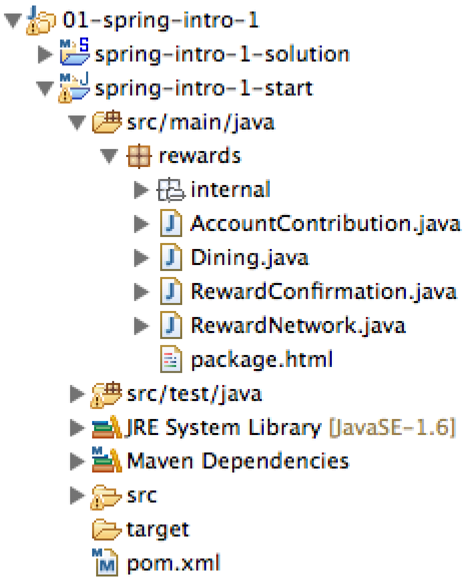


Figure 5: `RewardNetworkImpl` implements the `RewardNetwork` interface

Take a look at your `spring-intro-1-start` 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:



```
▼ 01-spring-intro-1
  ▶ spring-intro-1-solution
  ▼ spring-intro-1-start
    ▼ src/main/java
      ▼ rewards
        ▶ internal
        ▶ AccountContribution.java
        ▶ Dining.java
        ▶ RewardConfirmation.java
        ▶ RewardNetwork.java
          package.html
    ▶ src/test/java
    ▶ JRE System Library [JavaSE-1.6]
    ▶ Maven Dependencies
    ▶ src
      target
      pom.xml
```

Figure 6: 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:

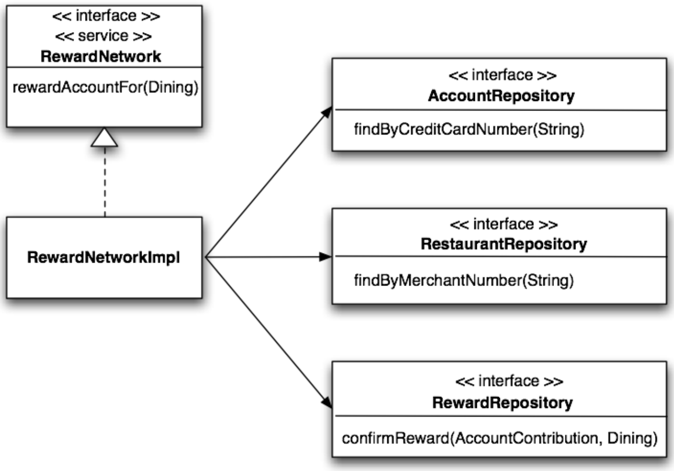


Figure 7: RewardNetworkImpl class diagram

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 much at the moment.

Now code the following steps to complete the method implementation:

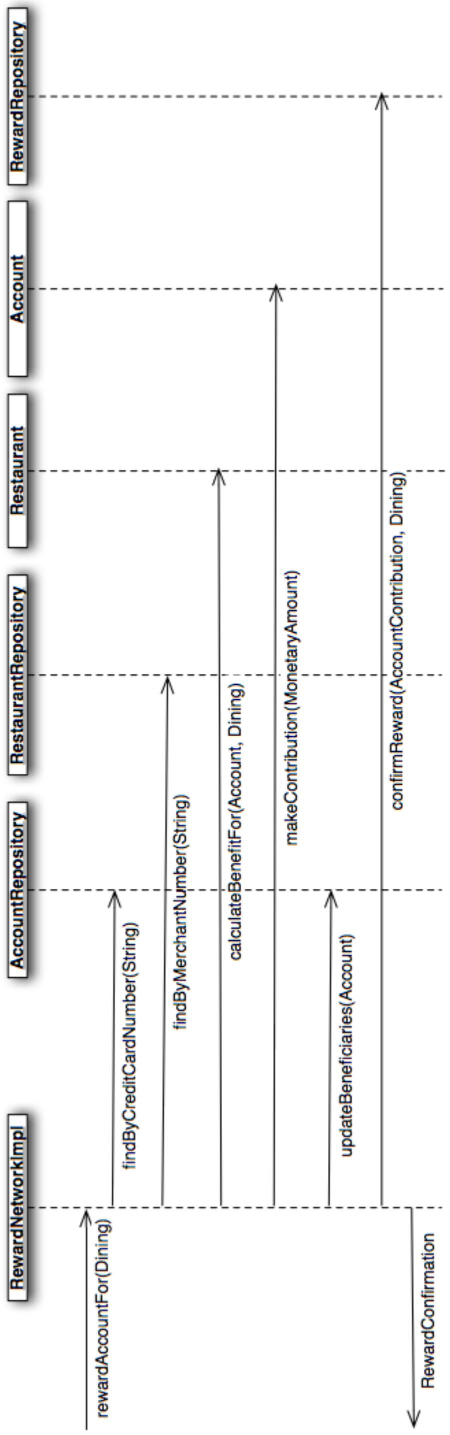

RewardNetworkImpl
AccountRepository
RestaurantRepository
Restaurant
Account
RewardRepository
rewardAccountFor(Dining)
findByCreditCardNumber(String)
findByMerchantNumber(String)
calculateBenefitFor(Account, Dining)
makeContribution(MonetaryAmount)
updateBeneficiaries(Account)
confirmReward(AccountContribution, Dining)
RewardConfirmation

Figure 08: The RewardNetworkImpl rewardAccountFor(Dining) sequence

Tip

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

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.

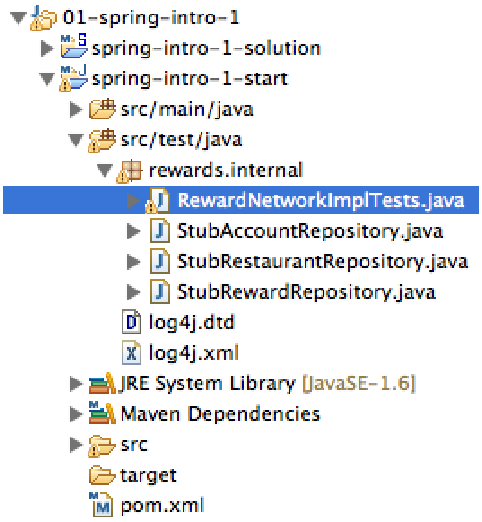


Figure 09: The rewards test tree

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. Module container-1: Using Spring to Configure an Application

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 to use SpringSource Tool Suite to visualize your application's configuration
6. How Spring combined with modern development tools facilitates a test-driven development process

Specific subjects you will gain experience with:

1. Constructor dependency injection
2. Setter dependency injection
3. Spring XML configuration syntax
4. Spring 3.0 embedded database support

5. SpringSource Tool Suite

6. Factory Beans

Estimated time to complete: 45 minutes

2.2. 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!

2.2.1. 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:

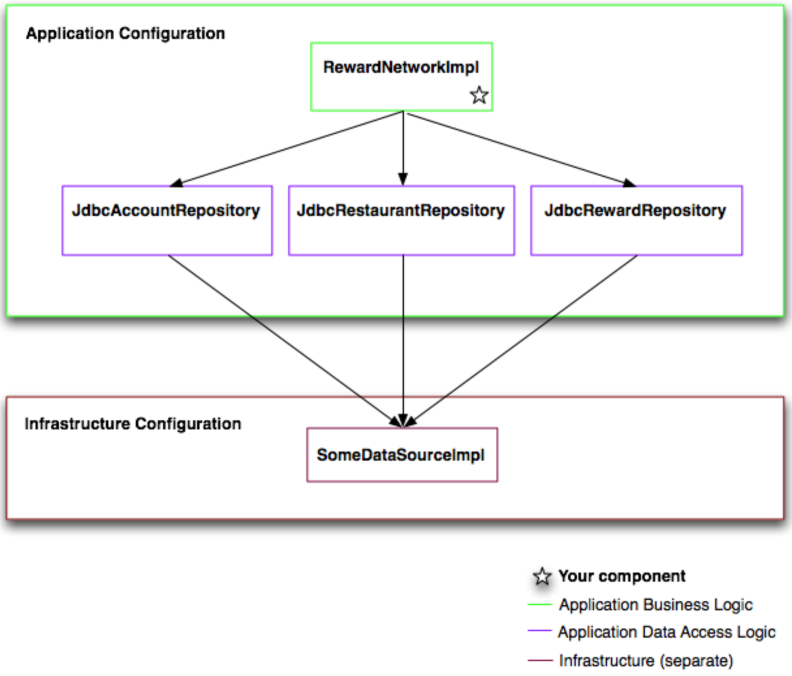

Application Configuration
RewardNetworkImpl
JdbcAccountRepository
JdbcRestaurantRepository
JdbcRewardRepository
Infrastructure Configuration
SomeDataSourceImpl
☆ Your component
Application Business Logic
Application Data Access Logic
Infrastructure (separate)

Figure 1: The rewards application configuration diagram

Figure 1 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.

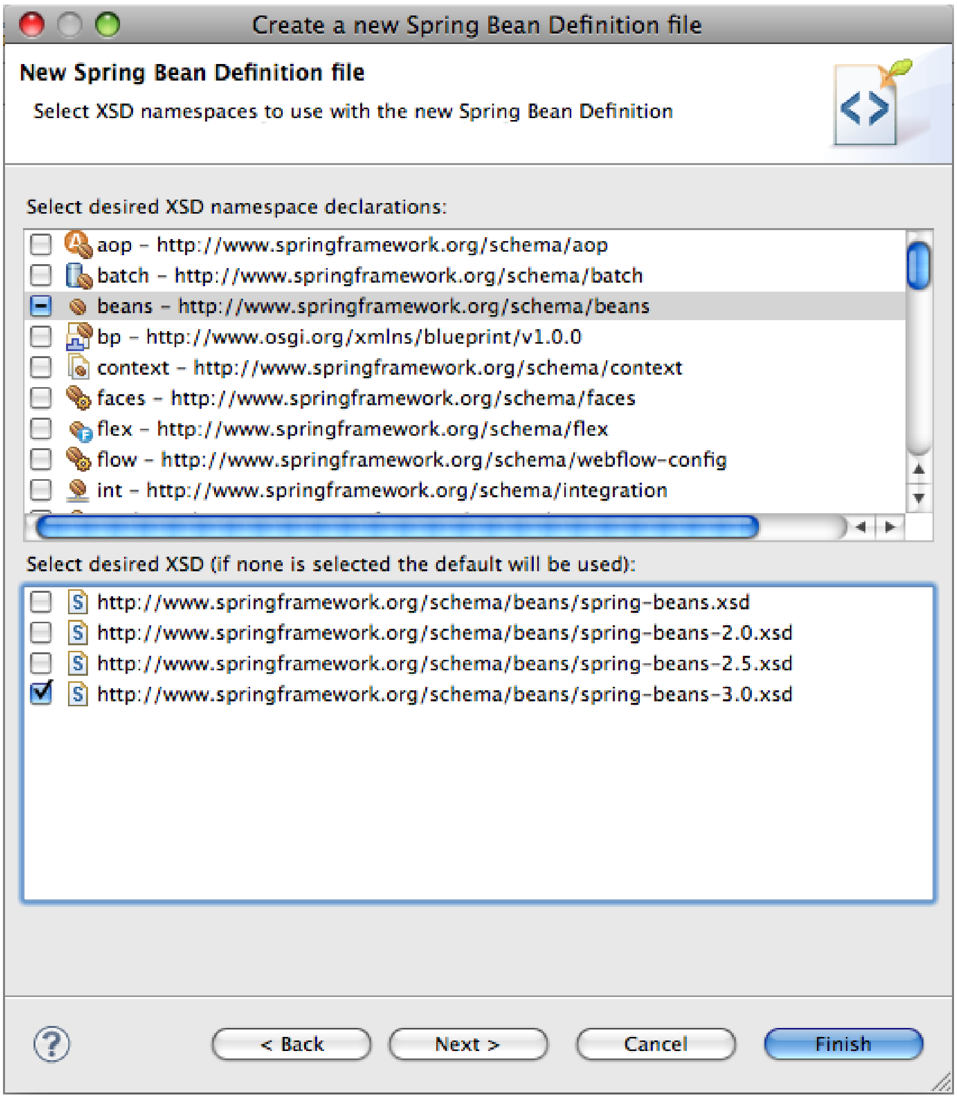
In your project, you'll find your familiar `RewardNetworkImpl` in the `rewards.internal` package. You'll find each JDBC-based repository implementation it needs located within the domain packages inside the `rewards.internal` package. Each repository uses the JDBC API to execute SQL statements against 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.

2.2.1.1. Create the application configuration file

Spring configuration information is typically externalized from Java code, partitioned across one or more XML files. In this step you'll create a single XML file that tells Spring how to configure your application components. You'll then validate your configuration visually using SpringSource Tool Suite's graphical visualizer.

Under the `src/main/java` folder, right-click the `rewards.internal` package and select *New -> Spring Bean Configuration File*. Use `application-config.xml` as the file name to indicate this is where your application configuration will reside. Leave the 'Add Spring project nature if required' checkbox checked and click *Next*.

In the next screen, note that by clicking on the 'beans' namespace, you can select XSD files for different version of Spring. The default is 3.0 so you can either choose that explicitly or leave it unchecked. After that, click *Finish*.



New Spring Bean Definition file

Select XSD namespaces to use with the new Spring Bean Definition

Select desired XSD namespace declarations:

- □ aop - http://www.springframework.org/schema/aop
- □ batch - http://www.springframework.org/schema/batch
- ⊟ beans - http://www.springframework.org/schema/beans
- □ bp - http://www.osgi.org/xmlns/blueprint/v1.0.0
- □ context - http://www.springframework.org/schema/context
- □ faces - http://www.springframework.org/schema/faces
- □ flex - http://www.springframework.org/schema/flex
- □ flow - http://www.springframework.org/schema/webflow-config
- □ int - http://www.springframework.org/schema/integration

Select desired XSD (if none is selected the default will be used):

- □ http://www.springframework.org/schema/beans/spring-beans.xsd
- □ http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
- □ http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
- ☑ http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

Figure 2: XSD selection

Now, in application-config.xml create the configuration illustrated in the 'application-config.xml' box below:

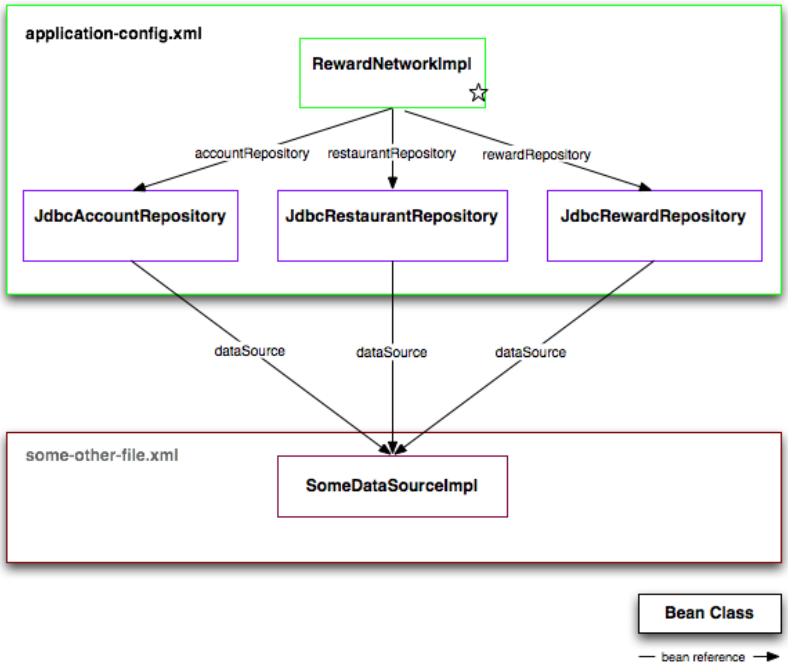


Figure 3: Application configuration

In Figure 3, the colored rectangular boxes represent bean definitions, and the labeled arrows between them represent bean references. The entire configuration is shown split out across two files, application-config.xml and some-other-file.xml. This is because it is generally a best-practice to separate application and infrastructure configuration, as infrastructure typically varies across environments. For example, in a test environment you might use a simple, in-memory data source, but in production you'll use a shared connection pool talking to a database server. By putting your infrastructure in another file, you can change it without effecting your application configuration.

Don't worry about the infrastructure configuration for now. Simply define four beans in your application-config.xml file to reflect the configuration in the 'application-config.xml' box of Figure 3. It's okay to assume the dataSource bean referenced by each repository is defined in another file: you'll see how to combine and visualize beans partitioned across multiple files in a later step. For consistency with the rest of the lab, give your `RewardNetworkImpl` bean an id of `rewardNetwork`.

Define each bean, then define the references between beans

As you start working in `application-config.xml`, consider creating a bare-bone bean definition for each of your four application components first, then going back and adding bean references to match Figure 3 above.

Follow bean naming conventions

As you define each bean, follow bean naming conventions. The arrows in Figure 3 representing bean references follow the recommended naming convention.

A bean's name should describe the *service* it provides callers. 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 XML auto-completion

As you define each bean, have Eclipse auto-suggest XML tags for you. Press `Ctrl+Space` inside a tag and Eclipse will suggest what's legal based on the XSD you are using. In-line documentation of each tag will also be displayed.

Use bean class name auto-completion

As you define each bean, have SpringSource Tool Suite auto-complete the bean's class name for you. When you add a `bean` tag, for the `class` attribute type the capital letters of the class name (i.e. 'JAR' for `JdbcAccountRepository`) or the first couple of letters of the name (i.e. 'JdbcA' for

`JdbcAccountRepository`) and press `Ctrl+Space` and you'll see a list of suggested class names.

Use constructor-arg auto-completion

For beans with constructor arguments, have SpringSource Tool Suite auto-complete constructor-args for you. When you add a `constructor-arg` tag press `Ctrl+Space` in the `ref` attribute and you'll see suggestions of legal values.

Use property auto-completion

For beans with settable properties, have SpringSource Tool Suite auto-complete properties for you. When you add a `property` tag press `Ctrl+Space` in the `name` and `ref` attributes and you'll see suggestions of legal values.

Once you have the four beans defined and referenced as shown in Figure 3, move on to the next step!

2.2.1.2. Visualize the application configuration file

So you defined your four beans in `application-config.xml`. How can you verify you defined them correctly? One way is to use SpringSource Tool Suite to visualize your configuration. In this next step you'll use SpringSource Tool Suite to graph the beans in your `application-config.xml`.

To enable SpringSource Tool Suite graphing, you must tell Eclipse your project is a SpringSource Tool Suite project. In your case that was done automatically when you added the Spring Bean Configuration File, otherwise you can add the Spring project nature through the *Spring Tools* context menu of your project.

You'll see your project's icon now has a "S" annotation indicating it's a SpringSource Tool Suite project:

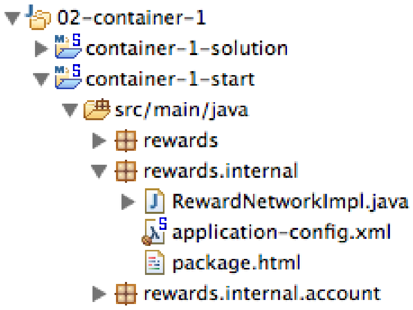


Figure 4: `container-1-start` is a Spring project

SpringSource Tool Suite also needs to know about your bean definition files. Again, this was done automatically when you used the wizard to add the new configuration file. To do this manually for files that were not created with the wizard, you can right-click your `container-1-start` project, select *Properties*, and navigate to the `Spring -> Beans Support` tab.

With the file added, now tell SpringSource Tool Suite to visualize it. In the *Spring Explorer* view, expand the `container-1-start` project, right-click on `application-config.xml` and select *Open Graph*.

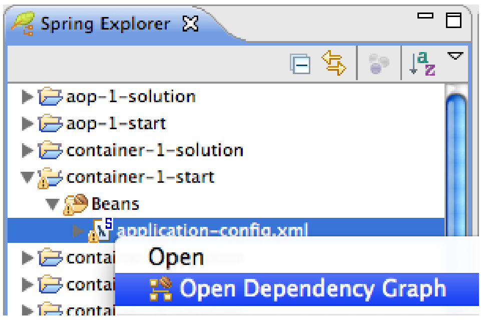


Figure 5: Open Dependency Graph

The resulting graph should look like:

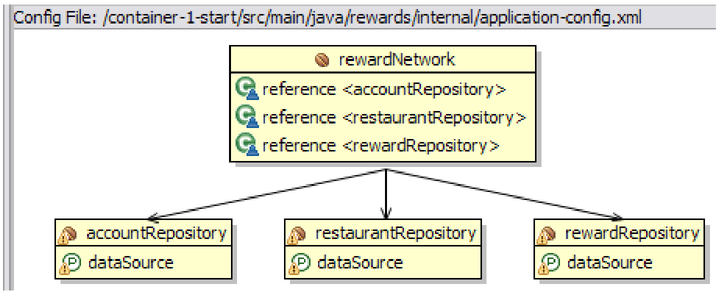

Config File: /container-1-start/src/main/java/rewards/internal/application-config.xml

Figure 6: `application-config.xml` visualized with SpringSource Tool Suite

When you see the equivalent of Figure 6 in your graph of application-config.xml, move on to the next step! If you see something different, head back to your configuration to find the issue. When you make a change to your configuration, refresh your graph.

2.2.1.3. Create the infrastructure configuration needed to test the application

In the previous step you created and visualized bean definitions for your application components. In this step you'll create the infrastructure configuration necessary to test your application. You'll then visualize the entire test configuration.

What's left to be able to test your application? Recall 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 Figure 6 as the 'dataSource' references are dangling). In this step you'll define the DataSource in a separate configuration file in your test tree.

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

You'll see a TODO asking you to complete a coding task. This is where you will define the DataSource your application uses to acquire database connections in a test environment.

What DataSource implementation should you use? You want something simple that lets you quickly test your application inside Eclipse. You also need to ensure your database is created and populated with test data *before* your application is initialized. Otherwise your tests would fail as there would be no data to test against.

Spring 3.0 ships with decent support for creating DataSources based on in-memory databases such as H2, HSQLDB and Derby. So the class you need to configure is EmbeddedDatabaseFactoryBean residing in org.springframework.jdbc.datasource.embedded package. Make sure you give the bean the id dataSource.

As we need to populate the DataSource with test data before we can run actual tests against them, we also need to configure a ResourceDatabasePopulator from the org.springframework.jdbc.datasource.init package. You should configure its scripts property to receive schema.sql as well as test-data.sql. The list element is probably really helpful here.

Both of these files can be loaded as scripts and both 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. Remember too that the root of the classpath is the parent directory of the /rewards directory, so you will need to configure the paths to the scripts with this in mind.

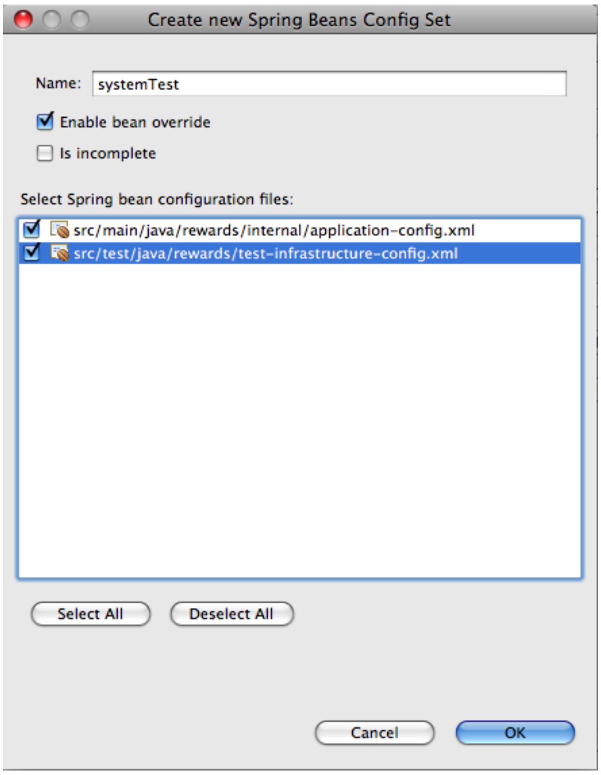
If you have configured the populator you can then wire it to the databasePopulator property of your EmbeddedDatabaseFactoryBean. Once you have the dataSource and populator bean defined in test-infrastructure-config.xml with the correct scripts, move on to the next step!

2.2.1.4. Visualize the complete test configuration

So far you've defined the configuration for your application and the infrastructure necessary to system test your application. In this step you'll use SpringSource Tool Suite to visualize your complete system test configuration across the two files.

Right-click on your container-1-start project and select *Properties*. Select the Spring -> Beans Support tab and *Add...* your test-infrastructure-config.xml to the Config Files list.

Now group the two files together into a logical 'systemTest' Config Set. To do this, select the Config Sets tab and select *New...* In the dialog, enter the Config Set name, select the config files to include, and select *OK*:



Create new Spring Beans Config Set

Name: systemTest

☑ Enable bean override

□ Is incomplete

Select Spring bean configuration files:

☑ src/main/java/rewards/internal/application-config.xml

☑ src/test/java/rewards/test-infrastructure-config.xml

Figure 10: New Config Set

Now graph your new 'systemTest' Config Set. In your Spring Explorer view, right-click on the set and select *Open Graph*. Your graph should look like:

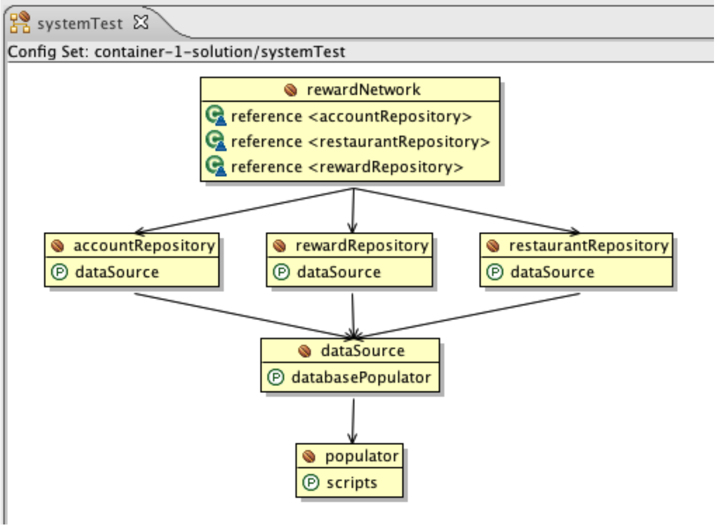


Figure 11: The rewards application - system test configuration

Notice how the `dataSource` bean is now part of the graph, as the graph is now visualizing the combined configuration across the `application-config.xml` and `test-infrastructure-config.xml` files.

Note

It's possible that you still see warning or even error overlays in the icons shown in the diagram. This is caused by a bug in the Eclipse icon cache and does not mean that you did something wrong! As long as there are no warnings or errors in your config files you're fine.

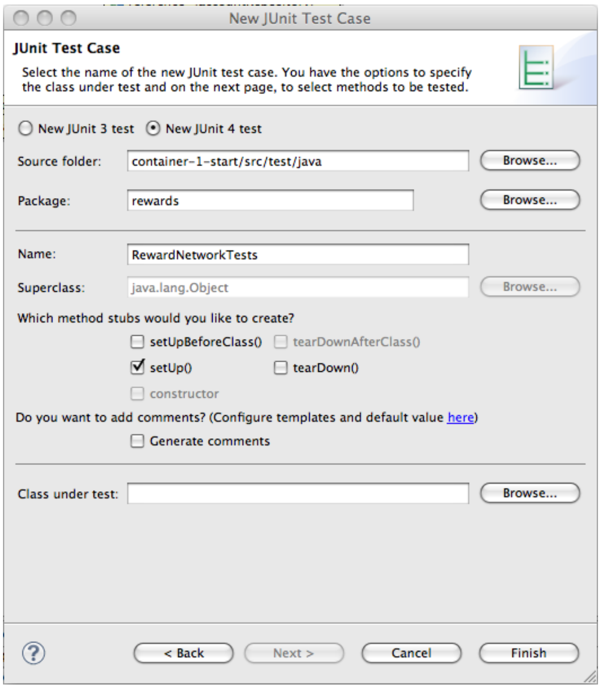
When you see the equivalent of Figure 11 in your graph of `application-config.xml`, move on to the next step!

2.2.2. 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.

2.2.2.1. 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):



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

○ New JUnit 3 test ● New JUnit 4 test

Source folder: container-1-start/src/test/java

Package: rewards

Name: RewardNetworkTests

Superclass: java.lang.Object

Which method stubs would you like to create?

□ setUpBeforeClass() □ tearDownAfterClass()

☑ setUp() □ tearDown()

□ constructor

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Class under test:

Figure 12: Creating the RewardNetworkTests TestCase using the JUnit Test Case wizard

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

2.2.2.2. 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 is done for you when you checked the `setUp()` checkbox in wizard.)

Within `setUp()`, create a new `ClassPathXmlApplicationContext`, providing it the class paths to your `application-config.xml` and `test-infrastructure-config.xml` files. 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!

2.2.2.3. 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 `RewardNetworkTest.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. Module container-2: Understanding the Bean Lifecycle

3.1. Introduction

In this lab you will gain experience with the lifecycle features of Spring's bean container in the context of the rewards application.

What you will learn:

1. How to implement your own bean lifecycle behaviors

2. How to modify Spring bean definitions at runtime, and when that is useful

3. How to apply custom configuration behaviors to objects created by Spring

Specific subjects you will gain experience with:

1. @PostConstruct / @PreDestroy / <context:annotation-config>

2. BeanFactoryPostProcessor / PropertyPlaceholderConfigurer / <context:property-placeholder>

3. BeanPostProcessor / RequiredAnnotationBeanPostProcessor

Estimated time to complete: 45 minutes

3.2. Instructions

3.2.1. Implementing your own bean lifecycle behaviors

Since Spring manages the lifecycle of your application, it is well-placed to issue your callbacks at various points within that lifecycle.

In this section, you'll implement your own lifecycle behaviors for a component (bean) of your application.

In the reward dining domain, restaurant data is read often but rarely changes. This makes that data a good candidate for caching. In this section you will enhance `JdbcRestaurantRepository` to cache `Restaurant` objects to improve performance. You'll leverage Spring to control cache initialization and destruction, and you'll run tests to verify your behavior.

Here are the design requirements you should follow:

1. When your `JdbcRestaurantRepository` is initialized it should eagerly populate its cache by loading all restaurants from its `DataSource`.

2. Each time your repository is used it should query from its cache.

3. When your repository is destroyed it should clear its cache to release memory.

The desired 'initialization' sequence is shown graphically below:

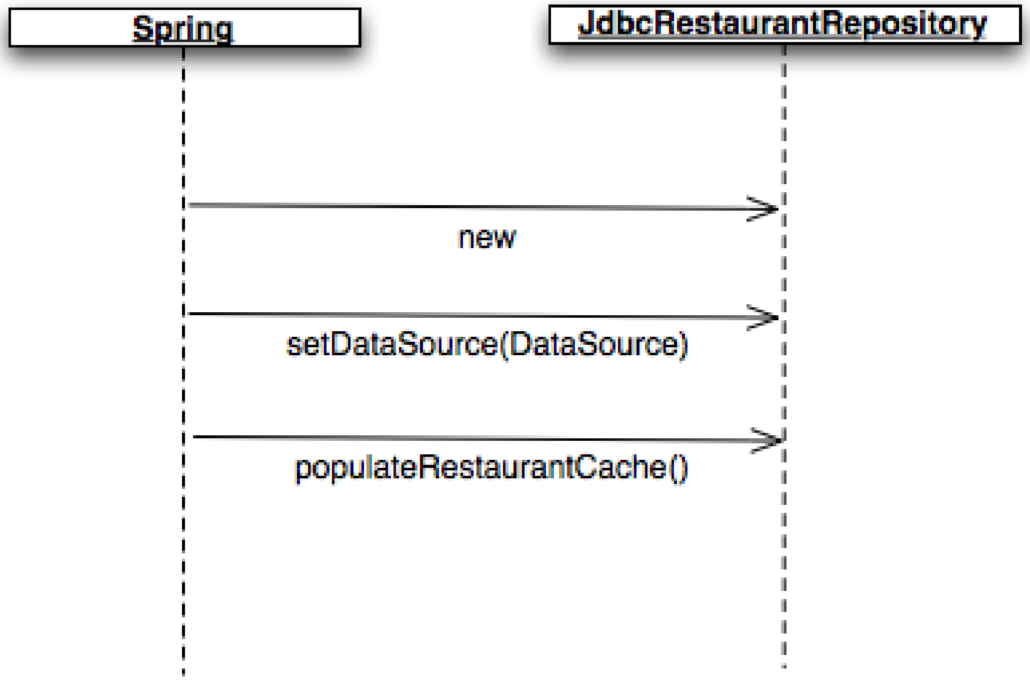


Figure 1: A `JdbcRestaurantRepository` being initialized

The desired 'use' sequence is shown below:

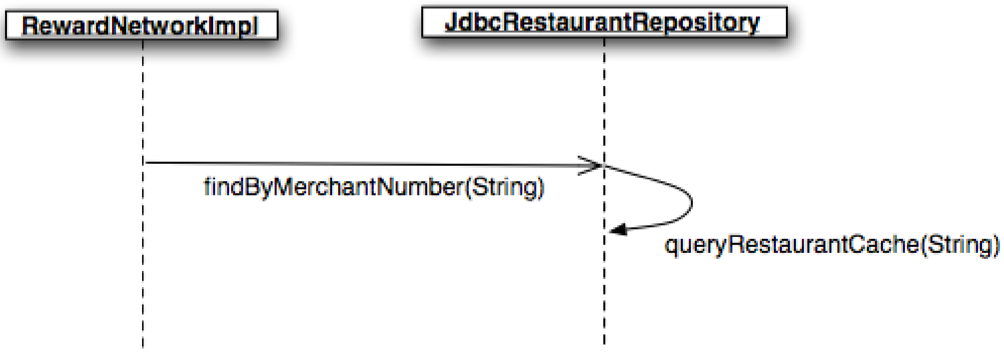


Figure 2: A `RewardNetworkImpl` calling a `JdbcRestaurantRepository` to load Restaurants

The desired 'destruction' sequence is shown below:

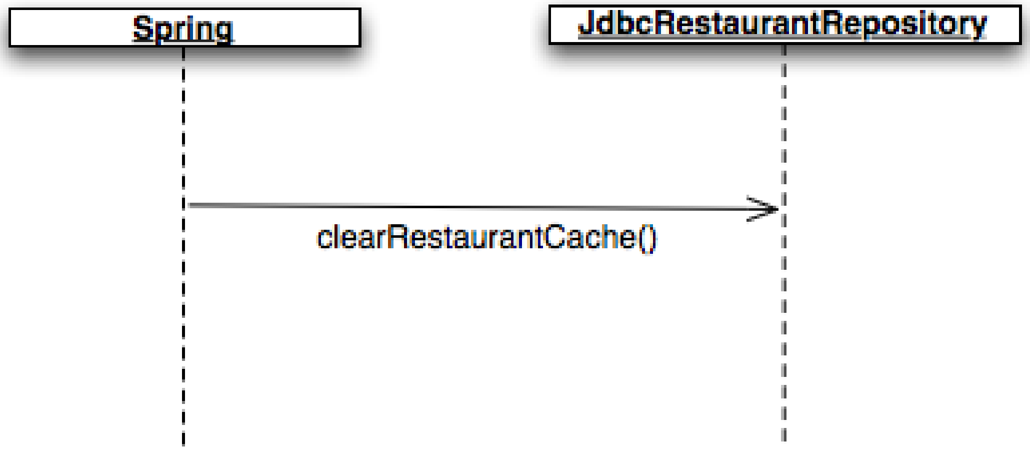


Figure 3: A `JdbcRestaurantRepository` being destroyed

With the design outlined, you are ready to implement.

3.2.1.1. Enhance `JdbcRestaurantRepository`

Take a look in your `container-2-start` project in the Eclipse IDE. Navigate from the `src/main/java` source folder into the `rewards.internal.restaurant` package. Within that package you'll find `JdbcRestaurantRepository`. Open it.

You'll see two `TODOs` indicating tasks to be completed. To complete these, enhance your `JdbcRestaurantRepository` so it will:

1. Receive an initialization callback when deployed as a Spring bean. Upon initialization, the `populateRestaurantCache` method should be called to populate the cache.

2. Query its cache by calling `queryRestaurantCache(String)` when asked to find a `Restaurant`.

3. Receive a destruction callback when deployed as a Spring bean. Upon destruction, the `clearRestaurantCache` method should be called to clear the cache.

When you've completed your enhancements, move on to the next step!

3.2.1.2. Unit test `JdbcRestaurantRepository`

Now it is time to verify your enhancements work. In this step, you will unit test your `JdbcRestaurantRepository` in isolation. The unit test won't use Spring, but will simulate the Spring bean lifecycle to initialize and destroy your repository.

In the `src/test/java` source folder, navigate to the `rewards.internal.restaurant` package. There you will find the unit test class `JdbcRestaurantRepositoryTests`. Before you go any further, run the test as-is to see what happens. You'll see the red bar indicating 3 failures, along with an explanation of why each test failed.

Now open the test class. First, you'll focus on properly testing initialization behavior. In `setUp` you'll see a `JdbcRestaurantRepository` being initialized like it would if deployed as a Spring bean. The default constructor is being called and the `DataSource` is being set (injected). There is also a `TODO` asking you to issue the initialization callback as Spring would do. Do this and re-run your test. You should still see red, but the failures should be different.

If you implemented your repository initialization and use logic correctly, you'll see that 2 of the 3 tests cases pass now, leaving 1 failure. The first tests a successful find, and the second tests a find with a bogus merchant number. Together they verify your cache was initialized and queried successfully. If you see more than 1 failure, head back to your `JdbcRestaurantRepository` implementation to debug. Run the test again when you think you have fixed the problem.

So 2 of your 3 tests are passing, but 1 is still failing: the test that verifies the cache is cleared when your repository is destroyed. In `tearDown` you'll see a `TODO` asking you to issue the destruction callback as Spring would do. Do this. Then, inspect the `restaurantCacheClearedAfterDestroy` test and see how it forces an eager tearDown, then verifies the cache is cleared. Re-run your test. If you implemented your repository destroy logic correctly, you should see all green.

When you see the green bar you have proof your `JdbcRestaurantRepository` works in isolation. Move on to the next step!

3.2.1.3. Re-run the `RewardNetwork` system test

You have verified your `JdbcRestaurantRepository` works in isolation. However, you haven't verified that everything else continues to work when your repository is integrated and used in your application. Specifically:

1. You don't know yet if Spring will properly initialize and destroy your repository. So far you've only simulated the bean lifecycle, you actually haven't tested it with Spring.

2. You don't know yet if your repository causes something else in the application to fail.

In this step, you will re-run your existing `RewardNetworkTests` to verify your application, configured by Spring, still works with your repository enhancements.

Find `RewardNetworkTests` within the root `rewards` package in `src/test/java`. Open it and run it as is. You should see a red bar indicating a test failure. The test fails because Spring has not yet been configured to respect the initialization and destruction annotations added in Step 1. Open `application-config.xml` in the `src/main/java` in the `rewards.internal` and add the configuration element that enables processing these annotations.

Tip

The `context` namespace declaration has already been added to the XML for you. If you need help remembering the syntax for the element that enables annotation processing, try typing `<context:` and use auto-completion (`ctrl-space`) to bring up a menu of available elements.

Tip

It's a best practice to put the directive enabling the processing of these annotations in a configuration file not specific to a deployment environment. Regardless of the environment, we always want annotation processing to happen. This is why we are putting the directive in the `application-config.xml` file and not in the `test-infrastructure-config.xml` file.

Once you've updated the configuration XML accordingly, re-run the test - you should have the green bar.

3.2.2. Close the application context properly

Your test seems to run fine, let us now have a closer look. Open `JdbcRestaurantRepository` and add a breakpoint to the `@PreDestroy` annotated method. Re-run `RewardNetworkTests` in debug mode. As you can see, this method is not called. In the Spring lifecycle, when is any method annotated with `@PreDestroy` supposed to be called? What is missing in your test?

Make the appropriate changes inside `RewardNetworkTests.tearDown()`. You can run the test in debug mode one more time. The `@PreDestroy` annotated method should now be called.

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!

3.2.3. Using bean factory post processors to modify Spring bean definitions

Spring has several extension points that allow you to add custom behavior at well-defined points within the bean container lifecycle. In this section you will learn about one of these extension points called the *bean factory post processor*. You'll then gain experience with a concrete implementation provided by the Spring Framework.

A `BeanFactoryPostProcessor` is a special bean that can modify the definitions of other beans before they are used to create objects. How does this work? Well, Spring issues each `BeanFactoryPostProcessor` a callback before any other bean is created. Each can then change the definition of any other bean in the factory. This process is shown graphically in Figure 5:

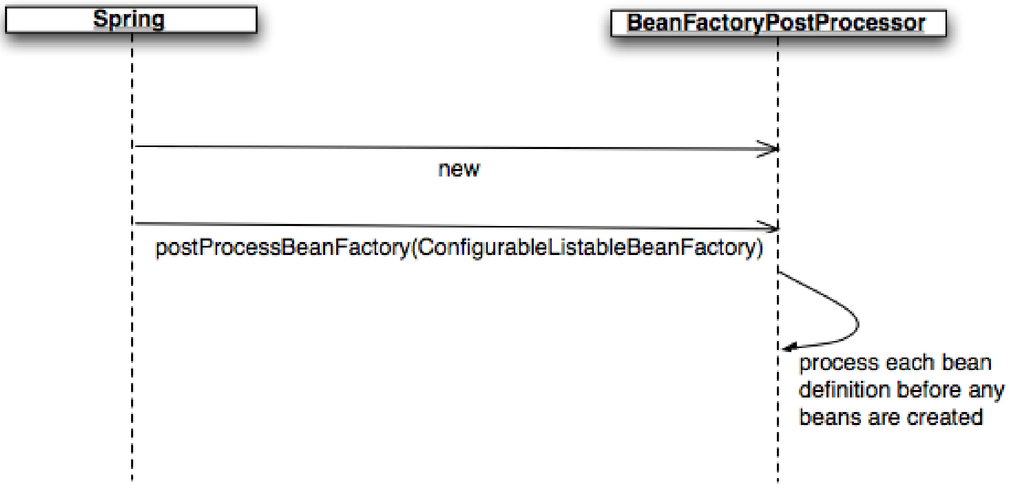

Spring
BeanFactoryPostProcessor
new
postProcessBeanFactory(ConfigurableListableBeanFactory)
process each bean definition before any beans are created

Figure 5: The bean factory post processor

Bean factory post processors are useful when you want to apply changes or transformations to groups of bean definitions at runtime. A good example would be to replace bean property value placeholders with values from `.properties` files to allow administrators to easily tweak configuration parameters. In this section, you'll do exactly that with the `<context:property-placeholder>` element.

In the following steps you'll gain experience with using the `<context:property-placeholder>`, element. This element uses `PropertyPlaceholderConfigurer`, a concrete `BeanFactoryPostProcessor` implementation. Specifically, you will move the configuration of your `TestDataSourceFactory` 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.

3.2.3.1. 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.testdb` 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`. Once you've verified this, move on to the next step!

3.2.3.2. 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 `dataSource` factory bean 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...

3.2.3.3. 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.

When you have the green bar, congratulations, you have completed this section and are ready for the next one!

3.2.4. Using bean post processors to apply behaviors to objects created by Spring

Another extension point provided by Spring is the *bean post processor*. In this section you will learn when it is useful, and will gain experience with a concrete implementation provided by the Spring Framework.

A `BeanPostProcessor` is a special bean that can modify other beans *after they are created* by Spring. This is different than a BeanFactoryPostProcessor, which can modify *bean definitions* before any objects are created.

How do they work? Spring issues each BeanPostProcessor two callbacks for every bean it creates as part of the bean initialization lifecycle. Each bean post processor can then manipulate each bean instance as it sees fit. A bean post processor can even return a different object back to the factory to use in place of the object created by Spring. Figures 6 shows this process graphically:

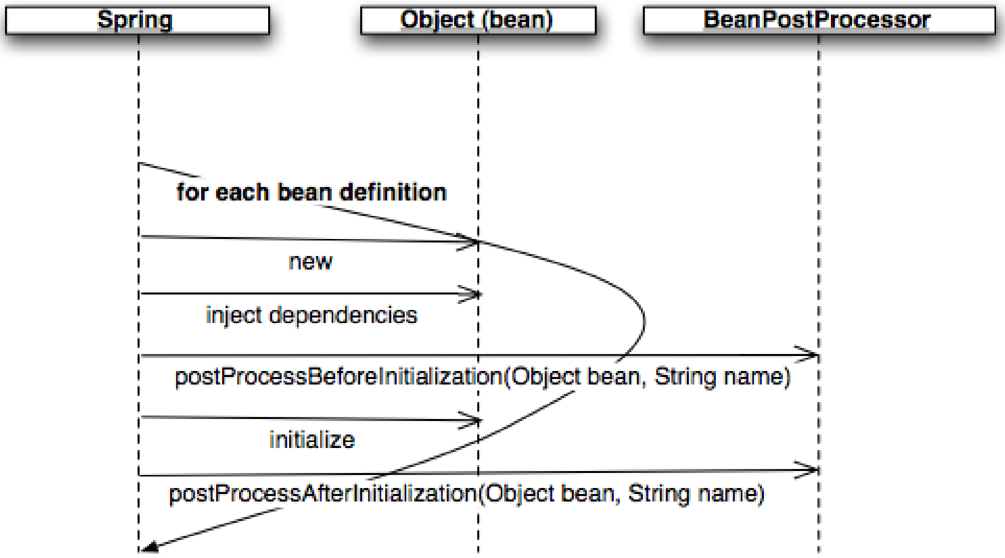

Spring
Object (bean)
BeanPostProcessor
for each bean definition
new
inject dependencies
postProcessBeforeInitialization(Object bean, String name)
initialize
postProcessAfterInitialization(Object bean, String name)

Figure 6: The bean post processor within the bean initialization lifecycle

Bean post processors are useful when you need to apply custom configuration behaviors to groups of bean instances. Often, post processors key off annotation metadata present on a bean's class to determine what behaviors to apply. A good example would be to perform a dependency check on a bean properties whose setter methods are marked @Required to enforce they have been set. In this section, you will do exactly that with the RequiredAnnotationBeanPostProcessor, a concrete BeanPostProcessor implementation.

In the rewards application there are several required properties. Recall that each JDBC-based repository needs a DataSource to do its work. Currently, this data source is injected using setter dependency injection. But what if you forget to call the setter? No error would be detected until the repository was used. In the following steps you'll use a `RequiredAnnotationBeanPostProcessor` to verify *at configuration time* all required properties are set properly.

3.2.4.1. Remove the `dataSource` property from the `JdbcAccountRepository`

In this step you will break your application by removing the instruction to set the `DataSource` for each repository.

Before you do anything, first run your `RewardNetworkTests` to verify your application still works. Now break it temporarily by opening `application-config.xml` and commenting out the `property` tag that sets the `dataSource` for each repository. Re-run your `RewardNetworkTests` and get the red bar as you would expect. But note how the error was not caught until the application was used and the description of what went wrong isn't informative. You can do better!

3.2.4.2. Add the `@Required` annotation

In this step you'll add a @Required annotation to each required `dataSource` property.

Open each JDBC-based repository class and add a `@Required` annotation to the `dataSource` setter method. Re-run your `RewardNetworkTests`. You'll get the red bar, but notice how the error is now much more descriptive and was caught earlier. Much better!

But what about configuring a `RequiredAnnotationBeanPostProcessor`?

Notice that we did not explicitly define a `RequiredAnnotationBeanPostProcessor` bean, yet post-processing of `@Required` annotations is clearly working? This is because we already defined `<context:annotation-config>` (in the `application-config.xml` file), and remember that it enables *multiple* bean post processors, including RequiredAnnotationBeanPostProcessor! Consult the reference guide's section on annotation-config [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/xsd-config.html#xsd-config-body-schemas-context-ac] for more details.

3.2.4.3. Restore your application configuration

Nobody likes the red-bar, so put back your property tag to set the dataSource and re-run your test. You should get the green bar, but now if it breaks again you'll catch the bug earlier with a better error message thanks to the RequiredAnnotationBeanPostProcessor.

Now re-run your RewardNetworkTests. If you get all green, congratulations! You've completed this lab!

Chapter 4. Module container-4: Configuration with annotations

4.1. 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 and Spring's @ContextConfiguration annotation. Finally, if you have time, you will play with Spring's JavaConfig support to achieve the same results.

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

Specific subjects you will gain experience with:

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

Estimated time to complete: 30 minutes

4.2. 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 will use Spring's annotations to configure dependency injection of the components of the rewards application. In the third section, you'll configure Spring's component scanning feature to create beans from the application classes automatically. The final optional section will explore achieving the same goals using Spring's JavaConfig support. We will use a JUnit 4 system test throughout to verify that we haven't broken anything. Have fun!

4.2.1. 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 learning how to define that shortly. In this lab we will rewrite some of the application code to make use of annotations for configuration.

4.2.1.1. First verify that everything works

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 review the configuration. If you haven't already done so, open the Spring Explorer by going to 'Window', 'Show View' and by selecting the Spring Explorer (sometimes it's not available yet, then you have to select it from the 'Other' option). Open the Spring dependency graph from the Spring Explorer tab that just appeared. Expand 'Beans' and then right-click the `system-test` config set and choose *Open Graph*. This should give you a nice overview of the dependencies between the different application and infrastructure components.

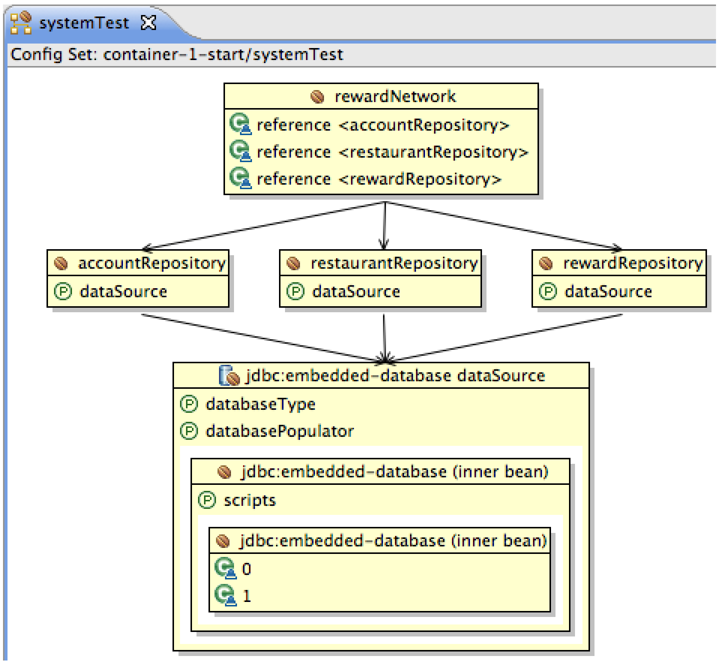

systemTest
Config Set: container-1-start/systemTest
rewardNetwork
reference <accountRepository>
reference <restaurantRepository>
reference <rewardRepository>
accountRepository
dataSource
restaurantRepository
dataSource
rewardRepository
dataSource
jdbc:embedded-database dataSource
databaseType
databasePopulator
jdbc:embedded-database (inner bean)
scripts
jdbc:embedded-database (inner bean)
0
1

Figure 1: The dependency graph

Now open the application configuration (use CTRL+SHIFT+R to quickly navigate to it) called `application-config.xml` and review the XML that wires up all the dependencies. As you can see, we're using constructor arguments. We're including the context namespace and are enabling the processing of annotations like @PostConstruct and @PreDestroy using the `<context:annotation-config/>` directive.

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

4.2.2. Dependency injection using Spring's @Autowired annotations

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 available in the `application-config.xml` file using Spring's @Autowired annotations. In `application-config.xml`, remove the constructor-arguments and property definitions for the all beans. In other words, the only things that'll be left are the bean identifiers and the implementation types.

Tip

Note that removing the constructor-arguments on the `RewardNetworkImpl` bean should have caused a validation error in the XML. This is because we're now asking Spring to invoke the default constructor in `RewardNetworkImpl` and if you look at the Java code, it doesn't have one. SpringSource Tool Suite assists you greatly in this regard to ensure that the coupling between the XML configuration and your Java code are correct.

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 hints using @Autowired. Previously, we shortly talked about the `<context:annotation-config/>` directives in `application-config.xml`. This element also processes the @Autowired annotations, so we don't need to further modify the configuration files.

Open the `RewardNetworkImpl` class and annotate its constructor with the `@Autowired` annotation. This tells Spring that it should try to automatically find the beans that matches the types of the constructor arguments to instantiate the class.

Tip

Note that this should have fixed the validation error in `application-config.xml`. SpringSource Tool Suite understands the @Autowired annotation and uses it in its validation processing.

Remember that you can quickly navigate from a bean definition to the Java code by pointing the cursor at its class name and pressing F3.

Now open up the `JdbcRewardRepository` and annotate the `setDataSource()` with that same @Autowired annotation. This will tell Spring to inject the setter with a instance of a bean matching the `DataSource` type. You can replace the @Required annotation, because @Autowired implies @Required.

Open up the `JdbcRestaurantRepository` and the `JdbcAccountRepository` and do the same thing as you did for the reward repository.

Now that you've finished adding instructions to automatically wire the dependencies, try to run the integration test again. You should see that it succeeds.

Tip

Remember that if you're using configuration annotations such as @Autowired, you have to enable detection of these in your Spring XML configuration somewhere, otherwise they will be ignored.

4.2.3. Defining Spring beans using component-scanning

As you might have noticed, we have so far only configured dependency injection using annotations. The instructions to Spring to actually create the application components is still defined in XML. In this section, we will look at how to use Spring annotations to create Spring beans automatically from your classes.

In your `application-config.xml` file, remove the `<context:annotation-config/>` element and add replace it with: `<context:component-scan base-package="rewards"/>`. This configuration directive 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 which you enabled with `<context:annotation-config/>`. First re-run the test to ensure that it still passes.

Now remove all of the bean definitions from the configuration file, leaving just the `<context:component-scan .../>` element. The test will now fail. Spring is now scanning for classes which make up the components of your application, but you haven't yet annotated those classes, which is why the test is failing. So this is the next step. Open the Java types for those bean definitions that you've just removed (`JdbcRestaurantRepository`, `JdbcAccountRepository`, `RewardNetworkImpl` and `JdbcRewardRepository`). Annotate all repositories with `@Repository` and the service (the `RewardNetworkImpl`) with `@Service`. This will allow the component scanning feature to find these components.

Re-run the test and see that it passes.

4.2.4. Using Spring's JavaConfig to configure the application

In this optional section, we will look at how you could use Java code to create your Spring beans and the dependencies between them. We will also see how you can combine this with @Autowired and component-scanning.

The first thing to do is to remove all instances of the @Autowired, @Service and @Repository annotations from all of your application classes. Instead, we're going to have all of the bean definitions and dependency injection defined in a single Java class. The main advantage of this is that all of the configuration is centralized in one place and you benefit from the stronger type checking of the Java compiler. It also allows you greater scope for flexibility and customization as you are effectively going to be writing a factory method for each bean.

Verify that the test is broken. Now, create a new Java Class in the `rewards.internal` package and call it something like

`ApplicationConfig`. Annotate the class with @Configuration. This tells Spring that the class represents Spring configuration and also ensures that it will be picked up by the component scanning which is already configured.

Next we need to pass in a DataSource that we can inject into our repository beans. Define a private field of type `DataSource` and annotate it with @Autowired. This ensures that the ApplicationContext will pass in the DataSource that we defined in our `test-infrastructure-config.xml` file.

Next we need to create the bean definitions. A bean definition is simply a method annotated @Bean which returns the instance of a bean. The convention is that the method name will become the bean id in the ApplicationContext. You'll therefore need to create four methods annotated with @Bean: One for the rewardNetwork bean and three others for the repositories. Start by creating method stubs for all four beans. Note that you should use the interface types as the method return types and that the methods don't have to be public but cannot be private.

Once you have the method stubs, you can start wiring things together. The rewardNetwork() method should simply create and return an instance of RewardNetworkImpl using `new` and invoke the other factory methods when calling the constructor.

Finally, the repositories are coded in a similar way. You just need to create an instance of the repository implementation, inject the dataSource and then return the object. Note that it would have required less coding if we'd chosen to use constructor injection for the dataSources!

Just for fun, re-run the test.

Chapter 5. Module aop-1: Introducing Aspect Oriented Programming

5.1. 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
2. How to weave an aspect into your application
3. How to visualize where an aspect will be applied

Specific subjects you will gain experience with:

1. @AspectJ AOP syntax
2. Spring IDE AOP visualization

Estimated time to complete: 45 minutes

5.2. Instructions

5.2.1. Creating an Aspect

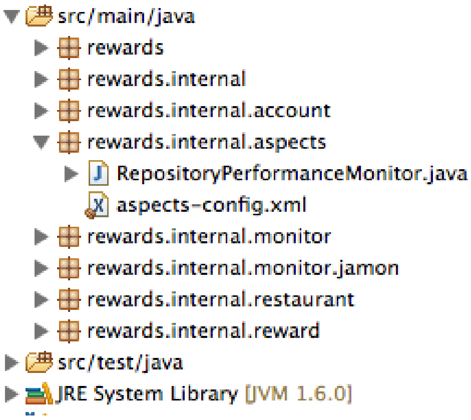
Up until now you have used Spring to configure and test your main-line application logic. Real-world enterprise applications also demand supporting services that cut across your main-line logic. An example would be security: there may be many places in your application where you need to perform a security check. 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.

5.2.1.1. Create an aspect

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

You're in luck: the definition of the aspect has already been started for you. Find it in the `rewards.internal.aspects` package:



- src/main/java
  - rewards
  - rewards.internal
  - rewards.internal.account
  - rewards.internal.aspects
    - RepositoryPerformanceMonitor.java
    - aspects-config.xml
  - rewards.internal.monitor
  - rewards.internal.monitor.jamon
  - rewards.internal.restaurant
  - rewards.internal.reward
- src/test/java
- JRE System Library [JVM 1.6.0]

Figure 1: The RepositoryPerformanceMonitor

Open this file and you'll see several TODOs for you to complete. First, complete `TODO 1` by annotating the `RepositoryPerformanceMonitor` 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.

Next, scroll down to the definition of the `monitor(ProceedingJoinPoint)` method. This is the method you will weave into your application to monitor its performance.

You aren't interested in monitoring *every* method of your application, though, only a subset. Specifically, you're only interested in monitoring your repositories, the objects responsible for data access in the application. This will give you a gauge of your data access performance.

Here is what should happen: before any repository is invoked, your `monitor` method should be called. It should then start a monitor, proceed with the repository invocation, then stop the monitor after the invocation returns and log a report. This is clearly "around" advice, as it will execute logic before *and* after your repositories. The desired "around" behavior is shown graphically below:

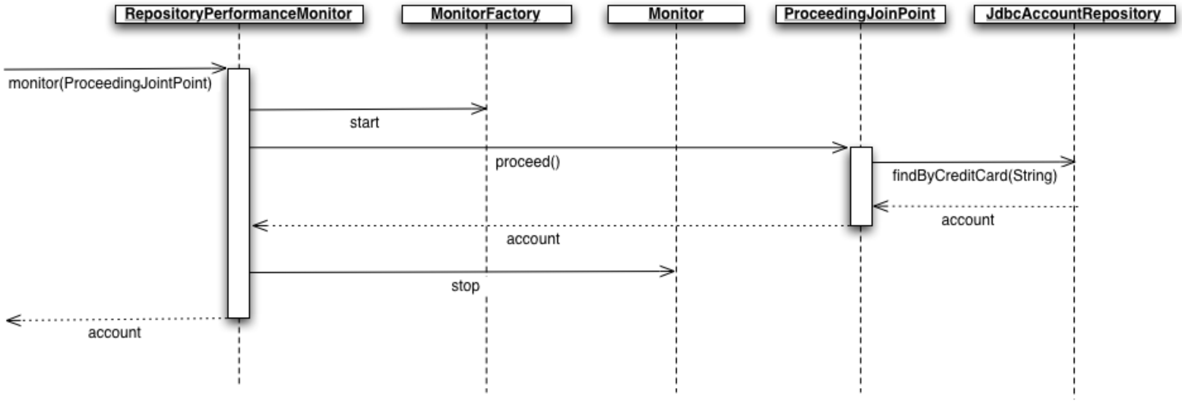


Figure 2: A Stop Watch around a repository method

To make this happen, complete `TODO 2` by annotating the `monitor(ProceedingJointPoint)` method with the `@Around` annotation. Its expression should match any method on the `AccountRepository`, `RestaurantRepository`, or `RewardRepository` interfaces.

Tip

Refer back to the AspectJ pointcut expression language reference in the training material to help you.

Now in `monitor(ProceedingJoinPoint)` 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 `TODO 3` 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!

Once you've added the proceed call, run the `RepositoryPerformanceMonitorTest` class in the test tree. If you get the green bar, your monitoring behavior has been implemented correctly. Now all you have to do is make sure this behavior gets woven into the right places.

You now have fully defined the aspect expressing where performance monitoring behavior should apply. Move on to the next step!

5.2.1.2. Configure Spring to weave the aspect into the application

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

Find the `aspects-config.xml` file inside the `rewards.internal.aspects` package. In this file, first define a bean of class `RepositoryPerformanceMonitor` (TODO 4). This will deploy your aspect as a Spring bean. Note that the RepositoryPerformanceMonitor takes a monitor factory as a constructor argument. A factory has already been defined for you.

Next, add the `<aop:aspectj-autoproxy>` tag to this file (TODO 5). 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:

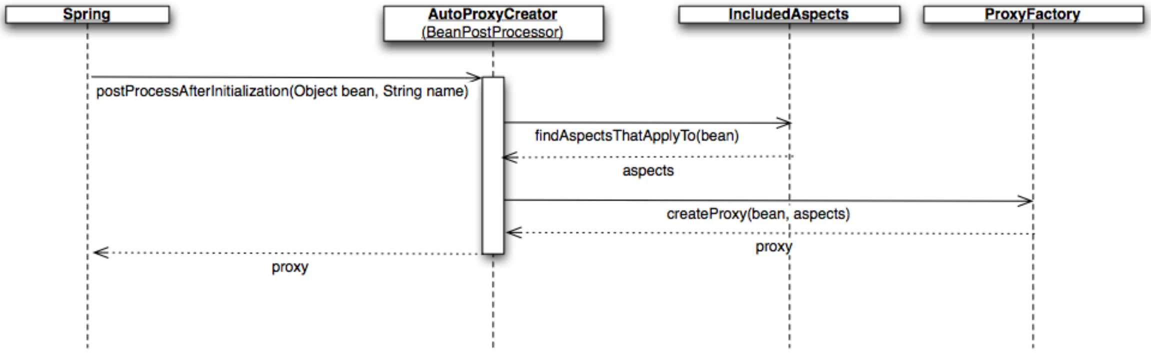

Spring
AutoProxyCreator
(BeanPostProcessor)
IncludedAspects
ProxyFactory
postProcessAfterInitialization(Object bean, String name)
findAspectsThatApplyTo(bean)
aspects
createProxy(bean, aspects)
proxy
proxy

Figure 3: Spring's auto proxy creator weaving an aspect into the application using the proxy pattern

Figure 4 shows the internal structure of a created proxy and what happens when it is invoked:

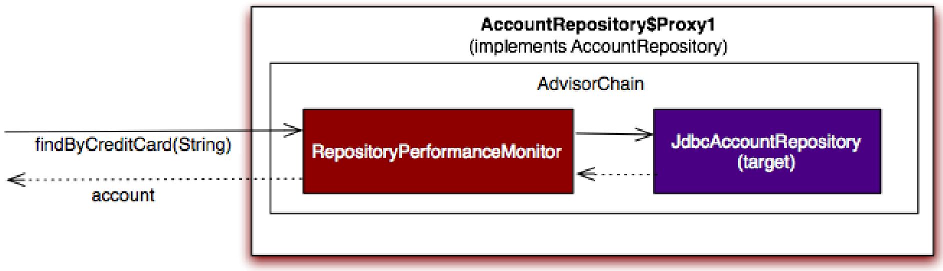


Figure 4: A proxy that applies performance monitoring to a `JdbcAccountRepository`

Note

Note it is not required, but it is generally recommended that you explicitly denote what beans are aspects in the XML configuration. To do this add an `<aop:include/>` tag inside of the `<aop:aspectj-autoproxy>` tag. The name attribute should reference the id of the bean that is an aspect.

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

5.2.1.3. Monitor application data access performance

To see this aspect in action, plug it into the application's system test configuration. To do that, simply add an import for `aspects-config.xml` in the `system-test-config.xml` file (TODO 6).

After the configuration file has been added, run `RewardNetworkTests` in Eclipse and watch the console. Note that this is not the same test you ran before. The `RepositoryPerformanceMonitorTest` is a unit test and only tests the aspect. The `RewardNetworkTests` is the integration test that actually applies the aspect. You should see output that looks like this:

```
AccountRepository.findByCreditCard: Last=1.0, Calls=1.0...
RestaurantRepository.findByMerchantNumber: Last=1.0...
AccountRepository.updateBeneficiaries: Last=1.0...
RewardRepository.confirmReward: Last=8.0, Calls=1.0...
```

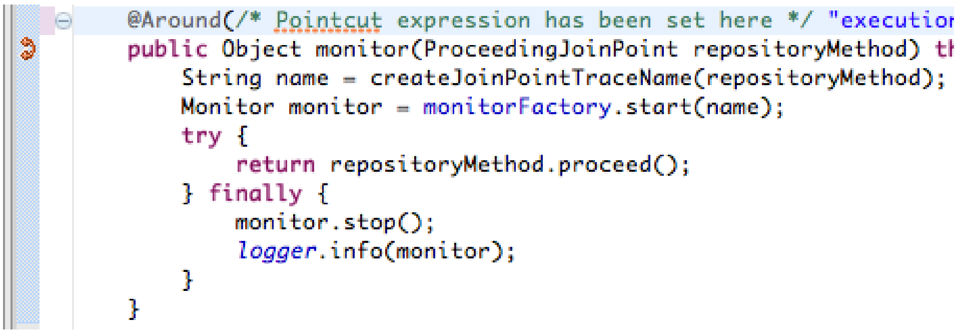
When you see repository performance monitoring output, your aspect is being applied! Move on to the next step!

5.2.2. Visualizing Aspect Weaving

AOP is a powerful way of adding cross-cutting behavior to an application, but it can be difficult to visualize exactly where an aspect will be applied. In the following steps you will explore different ways to visualize aspects using Spring IDE. The SpringSource Tool Suite will have to know about your aspects config file, but as you added an `<import/>` statement in the previous step that's taken care of automatically.

5.2.2.1. Visualize bean cross references

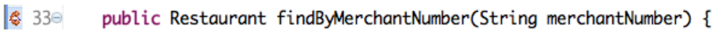
With your aspect configuration added, you can now open the classes for both your aspect(s) and bean(s) and see visual indicators of cross-cutting behavior. To verify, first open your `RepositoryPerformanceMonitor` class and scroll-down to the `monitor(ProceedingJoinPoint)` method:



```
@Around(/* Pointcut expression has been set here */ "executio
public Object monitor(ProceedingJoinPoint repositoryMethod) th
    String name = createJoinPointTraceName(repositoryMethod);
    Monitor monitor = monitorFactory.start(name);
    try {
        return repositoryMethod.proceed();
    } finally {
        monitor.stop();
        logger.info(monitor);
    }
}
```

Figure 5: The arrow gutter icon indicates that the `monitor` method is an around advice that advises one or more beans

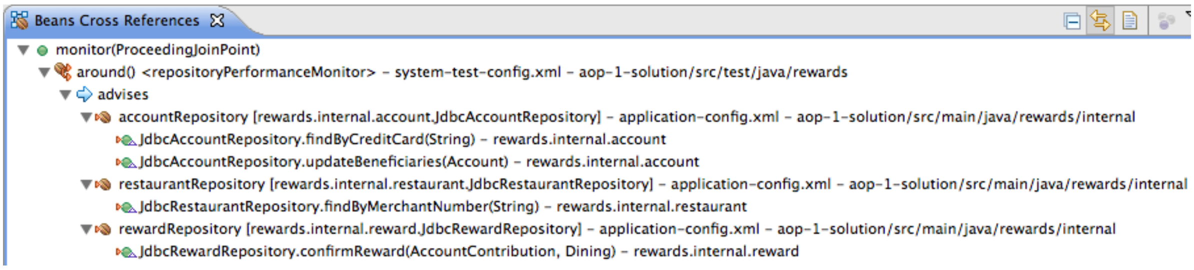
As another example, open `JdbcRestaurantRepository` and scroll to the `findByMerchantNumber(String)` method:



```
33  public Restaurant findByMerchantNumber(String merchantNumber) {
```

Figure 6: The arrow gutter icon indicates the `findByMerchantNumber` method is advised

These particular displays are somewhat limited because they only indicate a method is advice or is being advised. For more information on what is being advised by an aspect you turn on the Beans Cross References view. To do this, from the menu bar select *Window* -> *Show View* -> *Other...* Then, expand the Spring node and select Beans Cross References. You'll see the Beans Cross References view appear in the bottom page area:



```
▼ monitor(ProceedingJoinPoint)
  ▼ around() <repositoryPerformanceMonitor> - system-test-config.xml - aop-1-solution/src/test/java/rewards
    ▼ advises
      ▼ accountRepository [rewards.internal.account.JdbcAccountRepository] - application-config.xml - aop-1-solution/src/main/java/rewards/internal
          JdbcAccountRepository.findByCreditCard(String) - rewards.internal.account
          JdbcAccountRepository.updateBeneficiaries(Account) - rewards.internal.account
      ▼ restaurantRepository [rewards.internal.restaurant.JdbcRestaurantRepository] - application-config.xml - aop-1-solution/src/main/java/rewards/internal
          JdbcRestaurantRepository.findByMerchantNumber(String) - rewards.internal.restaurant
      ▼ rewardRepository [rewards.internal.reward.JdbcRewardRepository] - application-config.xml - aop-1-solution/src/main/java/rewards/internal
          JdbcRewardRepository.confirmReward(AccountContribution, Dining) - rewards.internal.reward
```

Figure 7: The `monitor` around advice advises several beans

Tip

If the cross references aren't being displayed, check that the view is "live" by clicking the icon with the two gold arrows pointing in opposite directions (it shows as a depressed button when it is active).

Explore the Beans Cross References view to see how the `monitor` around advice advises your beans, then move to the next step.

5.2.2.2. Visualize cross-cutting behavior application-wide with the AJDT Visualizer

The Beans Cross References view is a useful way of visualizing how a single aspect is applied to multiple classes. However, it doesn't provide a high-level visualization of all the cross-cutting behavior in your system. To provide this, Spring IDE integrates with the AJDT Aspect Visualization perspective.

Open the Aspect Visualization perspective by selecting *Window -> Open Perspective -> Other... -> Aspect Visualization*. When the perspective is open click on the *Menu* pull down in the *Visualize* view and choose *Preferences*.

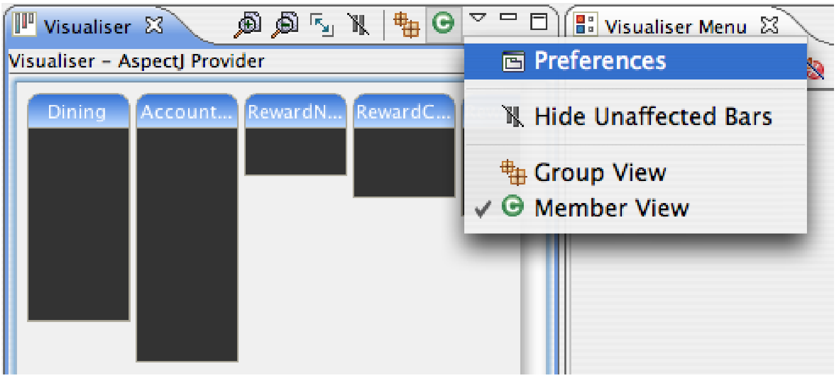
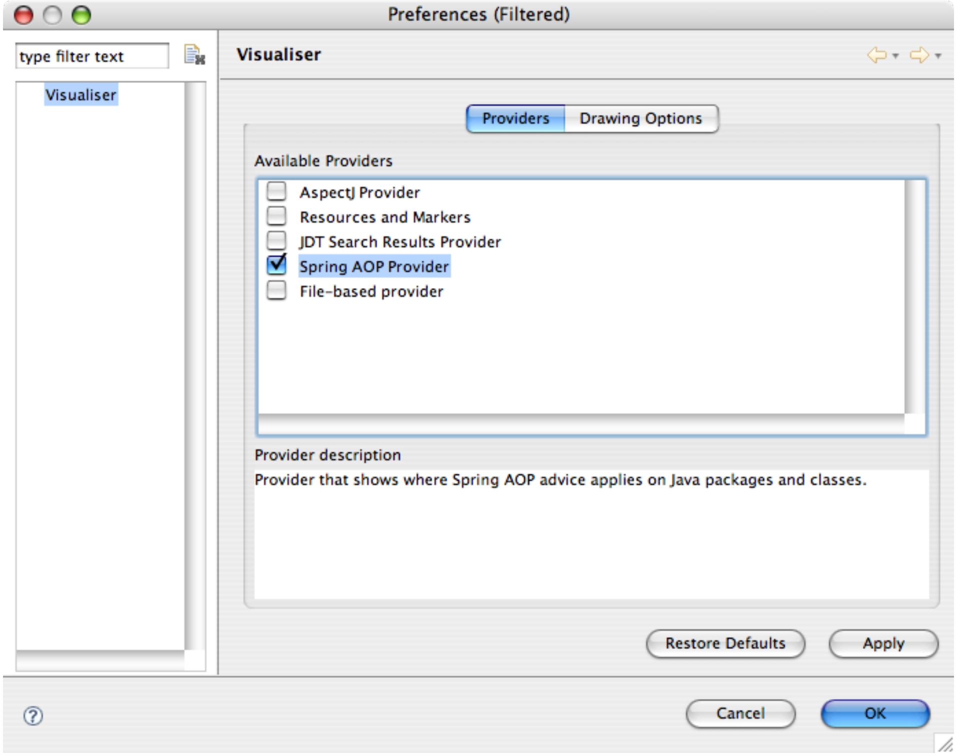


Figure 8: Open the visualizer preferences with the *Menu* pull down

Next choose the Spring AOP provider and press OK. This will tell AJDT to use Spring AOP when drawing its aspect graphics.



Available Providers

□ AspectJ Provider
□ Resources and Markers
□ JDT Search Results Provider
☑ Spring AOP Provider
□ File-based provider

Provider description

Provider that shows where Spring AOP advice applies on Java packages and classes.

Figure 9: The Spring AOP Provider

In the left most panel, select the `aop-1-start` project. You will see all of the classes in the project displayed with colored bands where an aspect is applied. In this example, you should see blue bands on your repositories indicating they are advised by the `RepositoryPerformanceMonitor`. This is shown below:

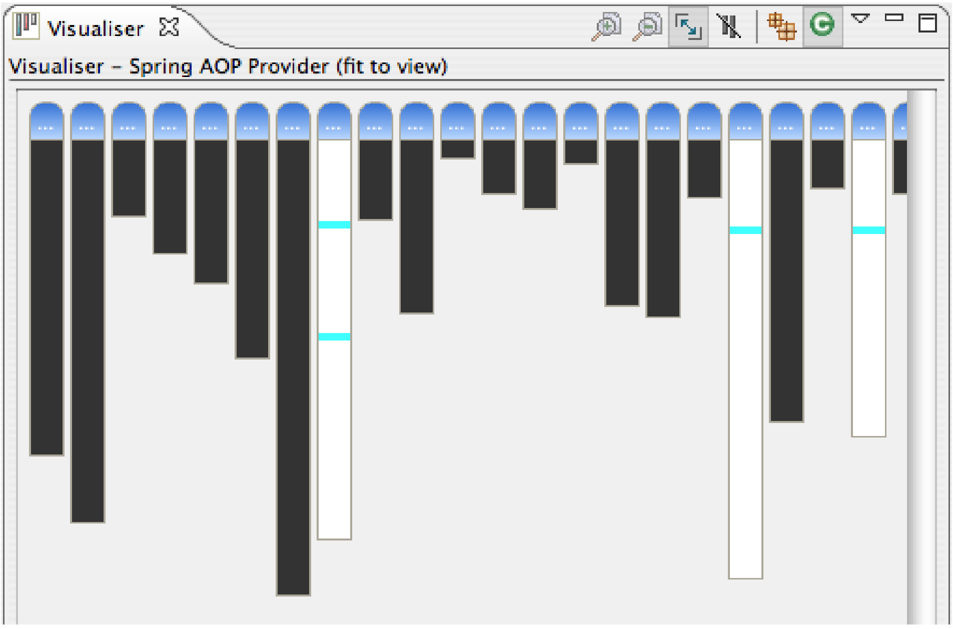


Figure 10: The AJDT visualizer showing aspects application-wide

Explore the AJDT Visualizer. When you're finished, you've completed the lab! You've just created a piece of cross-cutting logic called an aspect and applied it to your application. In the second half of the lab, you used visualization tools provided by SpringSource Tool Suite to understand where your aspect is applied.

Chapter 6. jdbc-1: JDBC Simplification using the JdbcTemplate

6.1. 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

6.2. Instructions

6.2.1. 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.

6.2.1.1. Use `JdbcTemplate` in a test to verify insertion

Before making any changes to `JdbcRewardRepository`, let's first make sure the existing fuctionality works by testing it and taking advantage of the `JdbcTemplate` class. 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 1).

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` (TODO 2). 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.

6.2.1.2. Refactor `JdbcRewardRepository` to use `JdbcTemplate`

We are now going to refactor an existing Repository class so it can use the `JdbcTemplate` (TODO 3). 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.

6.2.2. Using a `RowMapper` to create complex objects

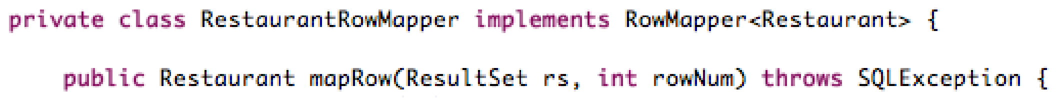
6.2.2.1. Use a `RowMapper` to create `Restaurant` objects

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 (TODO 4).

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.

6.2.3. Using a `ResultSetExtractor` to traverse a `ResultSet`

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

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 (TODO 5).

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 7. Module tx-1: Transaction Management with Spring

7.1. 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: 45 minutes

7.2. 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.

7.2.1. 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.

7.2.1.1. Add `@Transactional` annotation

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 (TODO 1). Adding the annotation will identify this method as a place to apply transactional semantics at runtime.

Before we are going to tell Spring to start looking for the `@Transactional` annotations, we first need to configure the transaction management infrastructure. Navigate to the `system-test-config.xml` file and complete TODO 2 by wiring up a `DataSourceTransactionManager`. Remember to set the `dataSource` property on this bean definition.

Finally, find and open the `application-config.xml` file in the same package. In this file 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 bean definition for `<tx:annotation-driven/>` (TODO 3). Make sure to populate the `transaction-manager` attribute on that bean definition with a reference to a bean named 'transactionManager'.

Note

If no value is specified for the `transaction-manager` attribute, the `<tx:annotation-driven/>` tag will look for and autowire a bean named `transactionManager`.

Go to the Spring Explorer view in Eclipse and show the graph of the `tx-1-start -> Beans -> system-test-config.xml`. If you configured your application context properly the graph should look like Figure 1.

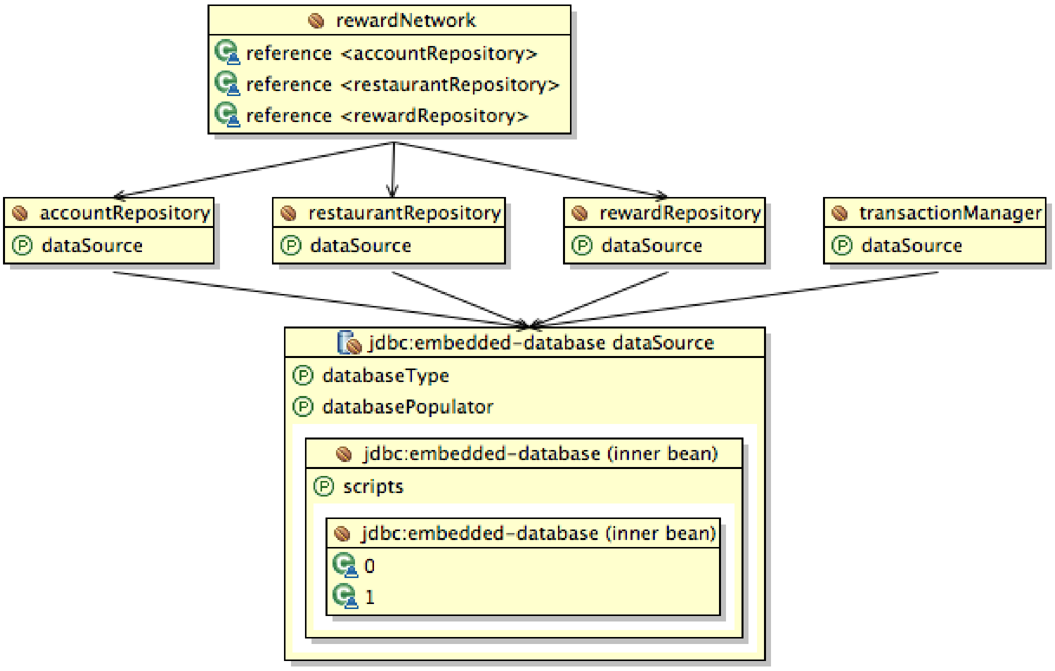

rewardNetwork
reference <accountRepository>
reference <restaurantRepository>
reference <rewardRepository>
accountRepository
dataSource
restaurantRepository
dataSource
rewardRepository
dataSource
transactionManager
dataSource
jdbc:embedded-database dataSource
databaseType
databasePopulator
jdbc:embedded-database (inner bean)
scripts
jdbc:embedded-database (inner bean)
0
1

Figure 1: The configuration of the context

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

7.2.1.2. 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.

7.2.2. 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`.

7.2.2.1. Modify Propagation Behavior

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.

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` (TODO 4). 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.

7.2.3. 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.

7.2.3.1. 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.

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` (TODO 5). 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 8. Module hibernate-1: ORM simplification using Spring

8.1. 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

8.2. 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.

8.2.1. Using Hibernate in the Account module

8.2.1.1. 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!

8.2.1.2. 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.

8.2.1.3. 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.

8.2.2. Using Hibernate in the Restaurant module

8.2.2.1. 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!

8.2.2.2. 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`.

8.2.2.3. 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!

8.2.3. 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!

8.2.3.1. 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-1 -> system-test-config.xml`. If you configured your application context properly the graph should look something like Figure 4:

Figure 4: The configuration of the context.

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-1` 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!

8.2.3.2. 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!

Chapter 9. Module mvc-1: Spring MVC Essentials

9.1. 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

9.2. Instructions

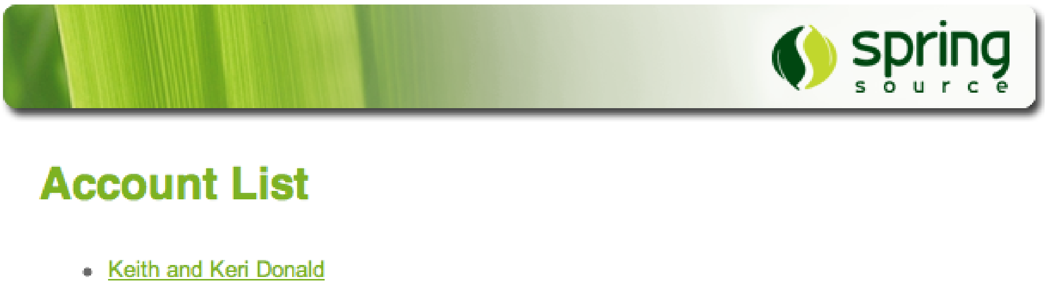
The instructions for this lab are organized into two main sections. In the first section you will be briefed on the web application functionality you will implement in this lab, then you will review the pre-requisite infrastructure needed to develop with Spring MVC. In the second section you will actually implement the required web application functionality.

9.2.1. 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.

9.2.1.1. Assess the initial state of the web application

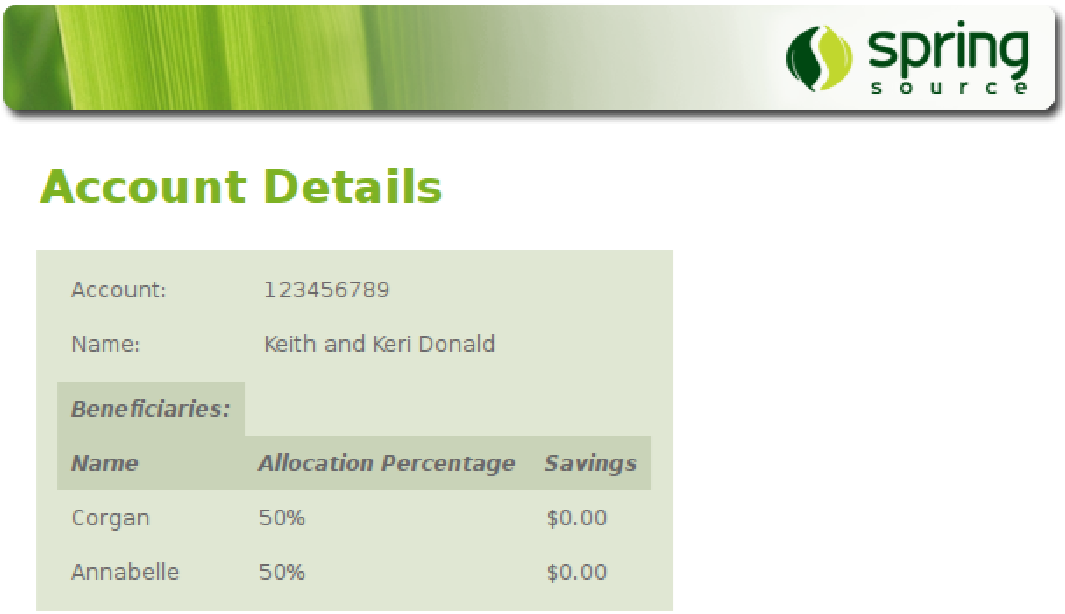
The web application you are developing should allow users to see a list of all accounts in the system, then view details about a particular account. This desired functionality is shown graphically below:



Account List

- Keith and Keri Donald

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	
Beneficiaries:		
Name	*Allocation Percentage*	*Savings*
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.

Begin by deploying the web application for this project as-is. Once deployed, navigate to the index page at http://localhost:8080/

mvc-1-start. 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.

9.2.1.2. 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 `app-config.xml`. Open this file to see the beans that make up this layer. You may also visualize this file using STS.

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.

9.2.1.3. 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 the `AccountController` bean is defined and how the `AccountManager` dependency is injected as a constructor argument.

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-1-start/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.

9.2.2. 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.

9.2.2.1. Implement the /accountDetails request handler

In the `AccountController`, add a method to handle requests for account details (TODO 1). 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.

9.2.2.2. 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.

Open up `AccountControllerTests` in the test tree 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` (TODO 2).

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.

9.2.2.3. Add a ViewResolver

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 `mvc-config.xml` file and add a bean definition of type `InternalResourceViewResolver` (TODO 3). 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 code 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, move on to the next step.

9.2.2.4. Simplify the configuration by using @Component and @Autowired

Web controllers are perfect candidates to be autodetected by Spring. Often, they're in one package (in this case the `accounts.web`) package and have a Spring dependency in the form of the `@Controller` annotation. In this step, we will simplify the configuration and have Spring automatically pick up the `@Controller` classes.

First, navigate to the `mvc-config.xml` file and add the `context` namespace to the XML header (TODO 4). This enables you to start using the component-scanning features.

Tip

The `namespaces` tab of the Spring Bean configuraiton editor allows you to enable/disable additional custom namespaces that Spring provides for you.

Next, **remove** the controller bean definition and turn on component scanning by putting in the line `<context:component-scan base-package="accounts.web"/>` (TODO 5). This enables component-scanning. Any `@Controller` will now automatically be picked up.

Navigate to the AccountController. As you can see, it has a constructor that takes an argument. Since we do not have an explicit bean definition anymore, Spring will not know how to inject the `AccountManager` argument. Make sure to provide Spring a hint that it can use to autowire the constructor (TODO 6).

Try to re-run your web application and see if everything still works. If so, you've successfully used the `@Controller` and `@RequestMapping` functionality to map URLs to handler methods, used an InternalResourceViewResolver to be able to return logical view names from methods instead of hard-coded paths and the component-scanning features to not have to wire up each and every controller anymore. This means you are done with this lab.

Chapter 10. Module security-1: Securing the Web Tier

10.1. 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. md5 encoding

Estimated time to complete: 45 minutes

10.2. 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 configure some additional users and roles and experiment with different role-based access rules. In the fourth section, you will handle unsuccessful attempts to log in. In the final section, you will use the security tag library to display links and data based on role.

10.2.1. 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.

10.2.1.1. Define the Filter class

Open `web.xml` (within the `src/main/webapp/WEB-INF` directory) and add the relevant <filter/> and <filter-mapping/>definitions (TODO 01).

10.2.1.2. Include Security Configuration in the Root Application Context

Next, import the bean configuration file containing the security configuration into the `app-config.xml` (TODO 02). 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. Deploy the web application for this project and navigate to the index page at http://localhost:8080/security-1-start. 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 the login page yet. In the next step, you will explore the security constraints that make this happen, and you will configure the proper login page.

10.2.2. 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.

10.2.2.1. Specify the Login Page

Open `security-config.xml`. Notice that the actual security constraints are defined inside a tag called `security:http`. Specifically notice that the *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.

You can now configure the login page inside `security-config.xml` by modifying the `login-page` attribute of the `<security:form-login>` tag (TODO 03).

10.2.2.2. Login as a Valid User

Redeploy the web application, and navigate to the index page at http://localhost:8080/security-1-start. This time when you click the 'View Account List' it should redirect you to the login form.

Note

Feel free to try logging in with a random username and password. If the values are invalid, then you should receive an 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 `keith`. Look carefully at the error message that occurs. You will see a 403 error, since Keith does not have the rights to access the accountList page yet. Before giving Keith 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` (TODO 04).

Tip

Use Eclipse's autocomplete to help determine the appropriate attribute.

Redeploy the web application. Revisit the index page at http://localhost:8080/security-1-start. Attempting to view the account list should now send you to the access denied page.

10.2.3. Managing Users and Roles

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

10.2.3.1. Configure Role-Based Access

So far you have only been logging in as a user with the 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 EDITOR role, but accessing the accountList and accountDetails views should be available to a user with the VIEWER role.

Find the `intercept-url` tags and modify the rules for `/accounts/account*` to enable access for viewers as well (TODO 05).

Redeploy the web application. Using the user `keith`, 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 `keith` does not have the EDITOR privileges.

10.2.3.2. Add a User

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 `admin` with the EDITOR role (TODO 06). While you are at it, add a third user called `keri` that has both VIEWER and EDITOR roles. In the next section, you will want to experiment with all three user/role combinations.

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.

Redeploy the web application, log in with the user `admin` 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.

10.2.3.3. Add a catch all

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/`.

To do this, add another `intercept-url` element at the end of the list with the pattern `/accounts/**` which enforces that the user is fully logged in (TODO 07). Be sure to do this using the pre-defined Spring Security expressions.

10.2.3.4. MD5 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 MD5 encoding.

Normally there is no way to get back the password from a MD5 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.

Open security-config.xml file and declare MD5 encoding (TODO 08). You will use a tag called `password-encoder`. Set its `hash` attribute accordingly. Also place a `salt-source` tag inside of the `password-encoder` and fill the `system-wide` attribute with `MySalt`, to use a static salt. Now, passwords need to be encoded. Open the users.properties file and change the plain-text passwords into MD5-encoded ones (TODO 09).

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

10.2.4. Handling unsuccessful attempts to log in

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 404 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` (TODO 10). 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`.

Redeploy the web application and try logging in using incorrect user/password again. An error message should appear.

10.2.5. 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.

10.2.5.1. Hide a Link Based on Role

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 (TODO 11). Then, see if you can determine what attribute of that tag to use in order to hide its contents.

Revisit the account details view. 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.

10.2.5.2. Hide Information Based on Role

As a final step, apply the same procedure to the table within the account details view that lists the beneficiary information (TODO 12). 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.

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

Chapter 11. Module remoting-1: Spring Remoting

11.1. 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

11.2. Instructions

11.2.1. 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.

11.2.1.1. Define the service exporter

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` (TODO 1). 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.

11.2.1.2. Define the client-side proxy

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 (TODO 2).

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.

11.2.1.3. Run the tests

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 (TODO 3). 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.

Next complete the testRmiClient method by calling the rewardAccountFor(Dining) method on the RewardNetwork (TODO 4).

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.

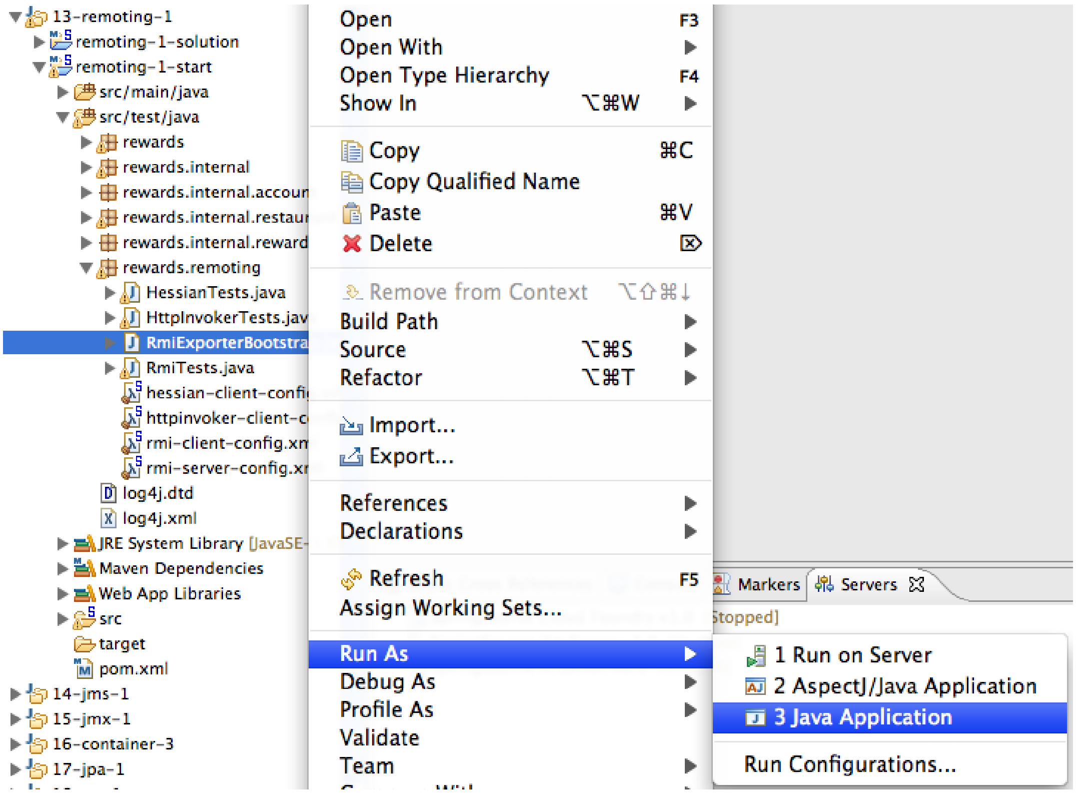


Figure 2: Run the `RmiExporterBootstrap` class

Finally, run the `RmiTests` class. If you see the green bar you've completed this section. Move on to the next section.

11.2.2. 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.

11.2.2.1. 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.

11.2.2.2. Define the service exporter

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` with a name of `/httpInvoker/rewardNetwork` (TODO 5). Inject values for the `serviceInterface` and `service` properties.

> **Tip**
>
> By default, the `DispatcherServlet` will use a `BeanNameUrlHandlerMapping` to resolve URLs to beans. Therefore, you will need to use the `name` attribute instead of the `id` attribute when naming the `HttpInvokerServiceExporter` bean.

Once completed, go to the next step.

11.2.2.3. 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-1-start

11.2.2.4. Define the client-side proxy

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` (TODO 6). 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-1-start/rewards/httpInvoker/rewardNetwork` as the value.

Continue to the next step.

11.2.2.5. Run the tests

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 7).

Next complete the `testHttpInvokerClient` method by calling the `rewardAccountFor(Dining)` method on the `RewardNetwork` (TODO 8).

Finally, run the `HttpInvokerTests` class. If you see the green bar, you have now completed this lab. Congratulations!

Chapter 12. Module jms-1: Simplifying Messaging with Spring JMS

12.1. 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.

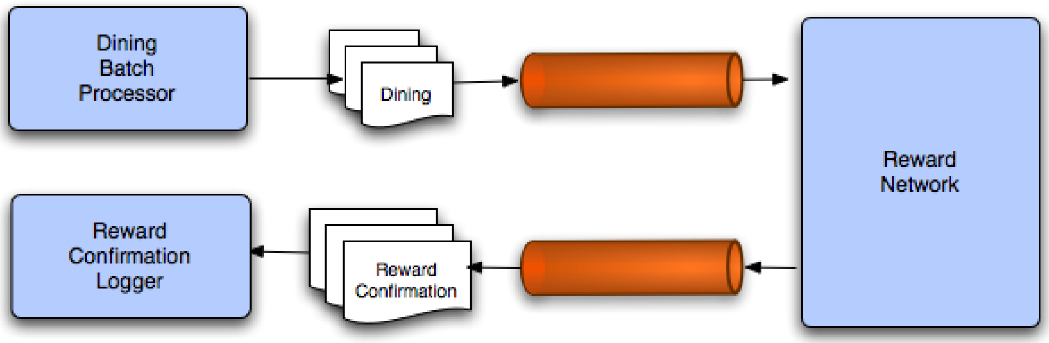


Figure 1: The batch processing of dining events with asynchronous messaging.

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

12.2. 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.

12.2.1. Providing the messaging infrastructure

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

12.2.1.1. Define the `ConnectionFactory`

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 `jms-infrastructure-config.xml` file in the `rewards/jms` package. Provide a bean definition there (TODO 01) 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!

12.2.1.2. Define the message queues

Now you will need to create two queues (TODO 02): 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.

12.2.2. 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.

12.2.2.1. Establish a dependency on `JmsTemplate`

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` (TODO 03) so that you will be able to use its convenience method to send messages.

12.2.2.2. Implement the batch sending

Now complete the implementation of the `processBatch(..)` method (TODO 04) 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`.

12.2.2.3. Define the template's bean definition

Open the `client-config.xml` within the `rewards/jms/client` package. Define a bean definition for the `JmsTemplate` (TODO 05). 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.

12.2.3. 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.

12.2.3.1. Define the listener container

Open the `jms-rewards-config.xml` file within the `rewards/jms` 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 (TODO 06). 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.

Tip

At this point, you might want to open the graph for the `systemTest` Spring IDE config set in order to visualize beans defined in the following files:

- `rewards/internal/application-config.xml`
- `rewards/jms/client/client-config.xml`
- `rewards/jms/jms-infrastructure-config.xml`
- `rewards/jms/jms-rewards-config.xml`
- `rewards/system-test-config.xml`

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

12.2.3.2. 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 `jms-infrastructure-config.xml` and configure the `destination` attribute using this name. Since the `rewardAccountFor()` method returns an object, we also need to specify the `response-destination` property. Review the diagram above, lookup the queue name for the destination queue in `jms-infrastructure-config.xml` and set the `response-destination` 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.

12.2.4. 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.

12.2.4.1. Define the listener container and adapter

Open `client-config.xml` and define another `jms:listener-container` and corresponding `listener` (TODO 07). This time, you should delegate to the `confirmationLogger` 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.

12.2.5. 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.

12.2.5.1. Send the batch of dining notifications

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 08).

Finally, provide an assertion (TODO 09) 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 13. Module container-3: Simplifying Application Configuration

13.1. Introduction

In this lab you will gain experience with features of Spring's bean container commonly used to simplify application configuration.

What you will learn:

1. Techniques for reducing the amount of Spring configuration code
2. How to import XML namespaces
3. How to use the <util:/> namespace to simplify common configuration tasks

Specific subjects you will gain experience with:

1. Bean Definition Inheritance
2. Importing Configuration Files
3. XML Schema (XSD) Namespaces
4. Lazy Initialization
5. Factory method
6. Bean aliasing

Estimated time to complete: 30 minutes

13.2. Instructions

13.2.1. 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*.

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 inherit 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.

13.2.1.1. 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 find `application-config.xml`. Open it. 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. Once you've done this, move on to the next step!

13.2.1.2. Update each repository bean definition

In this step you'll update each repository bean to extend from your `abstractJdbcRepository` to inherent its configuration.

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

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

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

13.2.1.3. Visualize the updated application configuration

In this step you'll see how the changes you made effect the graph of your application configuration.

In the Spring Beans view for the `container-3-start` project, graph the `systemTest` config set. It should look like this:

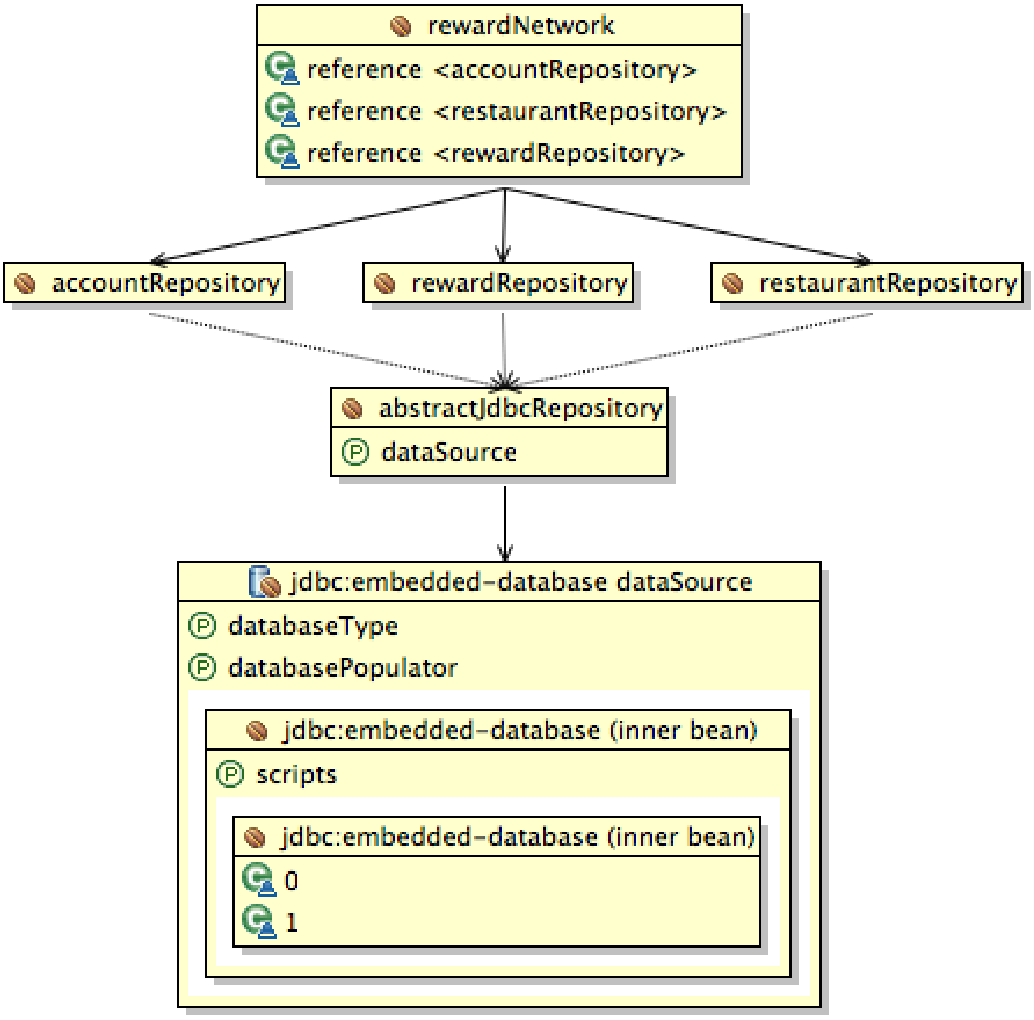

rewardNetwork
reference <accountRepository>
reference <restaurantRepository>
reference <rewardRepository>
accountRepository
rewardRepository
restaurantRepository
abstractJdbcRepository
dataSource
jdbc:embedded-database dataSource
databaseType
databasePopulator
jdbc:embedded-database (inner bean)
scripts
jdbc:embedded-database (inner bean)
0
1

Figure 1: The updated rewards application 'systemTest' config set

When you see the equivalent of Figure 1, move on to the next step!

13.2.1.4. 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. Good job!

13.2.2. 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.

13.2.2.1. 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.

To get started first rename the `test-infrastructure-config.xml` file to `system-test-configuration.xml` indicating that this file will fully define the configuration needed to run the system test. Update your test to include only this file.

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. Move on to the next section.

13.2.3. Working with the <util:/> namespace to simplify common configuration tasks

So far you have defined your entire application configuration using a simple XML-based bean definition language. With this language, you have already achieved a lot. You've configured Spring to:

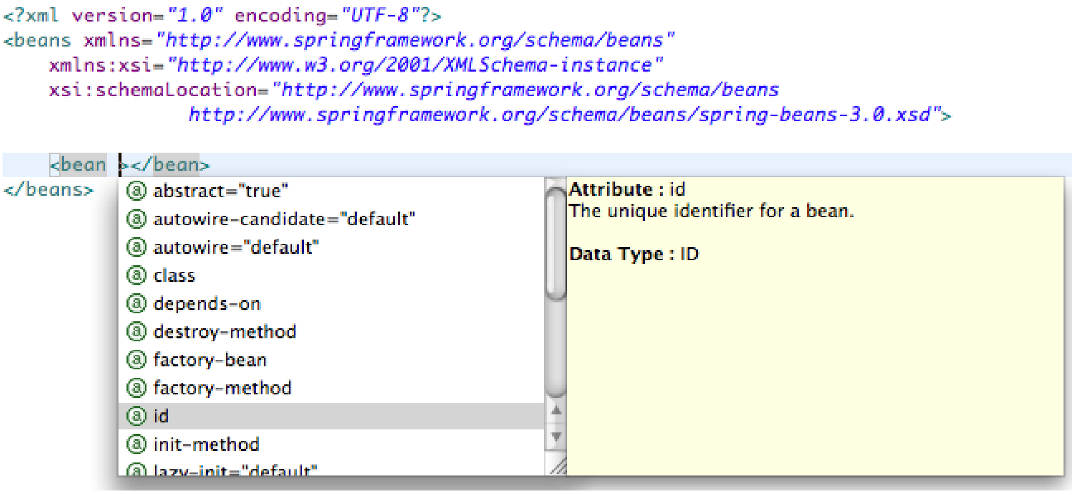
1. Create and configure your application components

2. Delegate to custom factories to create application components (FactoryBean)

3. Perform bean lifecycle callbacks (InitializingBean, DisposableBean)

4. Invoke special beans that add in custom configuration behaviors (BeanFactoryPostProcessor, BeanPostProcessor)

To do this, you imported Spring's generic bean definition language for each configuration file you defined:



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean ></bean>
</beans>
```

Attribute : id
The unique identifier for a bean.

Data Type : ID

Figure 2: The core bean definition language imported from `spring-beans-3.0.xsd`

Now, Spring also provides several higher-level *domain-specific languages* that build on its generic bean definition language to further simplify common configuration tasks. Each language is defined by its own XML schema with its own namespace you may import. In this section, you'll learn how to import one of these languages, `<util/>`, then use it to perform useful configuration tasks.

13.2.3.1. Import the <util/> XML namespace

The `<util/>` namespace contains tags that offer useful configuration utilities. In this step, you'll import the namespace and browse the new tags available to you.

Open `alternate-datasource-config.xml` in the `rewards.testdb.config` package within `src/test/java`. This is the Spring configuration file you'll work primarily in during this section. Here over the next few steps you'll complete several bean definitions to gain experience with the `<util/>` namespace.

In `alternate-datasource-config.xml` you'll see several TODOs, starting with TODO 1 that asks you to import the `<util/>` namespace. You'll complete TODO 1 in this step.

Import the `<util/>` namespace: In your editor, switch to the 'Namespaces' tab and select the 'util' namespace. Then switch back to the 'Source' tab. Notice that the editor has added a reference to the 'util' namespace URI and XML schema and associated the namespace URI with a `util:` prefix.

When you have completed your import, test it. Hit `Ctrl+Space` in your file body and look for new tags with the `util:` prefix. If you see them, move on to the next step!

13.2.3.2. Run `AlternateDataSourceConfigTests`

In the following steps you will use the new `<util/>` tags to configure four beans in different ways. You'll go through each bean one-by-one until you've completed them all. As you go, you'll run tests to verify you configured each bean correctly.

First things first, run `AlternateDataSourceConfigTests`. Verify all 4 test cases fail, indicating all 4 beans need completing. On to the first bean!

13.2.3.3. Complete `dataSource-createdFromList`

In this step, you'll complete the configuration of the bean named `dataSource-createdFromList`.

Complete `dataSource-createdFromList` (TODO 1 - 2) by firstly setting the `databaseName` property on Spring's `EmbeddedDatabaseFactoryBean` class. This class is part of the Spring 3.0 embedded database support. For this particular bean, the `databaseName` can just be a literal string, such as "rewards".

You now need to pass the initialization scripts into this bean as a list. The list will be passed into the `scripts` property of an inner bean of type `ResourceDatabasePopulator`. The list should contain two values which are the resource locations of the `schema.sql` and `test-data.sql` files (in that order). Try using the `<util:list>` tag to configure the list.

When you're done, run `AlternateDataSourceConfigTests` to verify the test for this bean is now passing. You should only get 3 failures now. When it is passing, move on to the next step!

13.2.3.4. Complete `dataSource-createdFromConstants`

In this step, you'll complete the configuration of the bean named `dataSource-createdFromConstants`.

Complete `dataSource-createdFromConstants` (TODOs 3 - 4) by injecting constant values for the properties using `<util:constant/>` tags. You will see the available constants by opening the `rewards.testdb.config.Constants` class. Note that that this bean definition still uses the `EmbeddedDatabaseFactoryBean`, so the configuration simply involves replacing the literal values for `databaseName` and the `scripts` in the previous step with constants from the Java class.

Tip

Note how this bean, as well as the other beans, are marked lazy (lazy-init="true"). This keeps them from being instantiated eagerly when the `applicationContext` is created on test `setUp`. Instead, they are instantiated only when referenced by another object, such as the test methods that call `applicationContext.getBean(String)`. This lazy-init behavior is useful for this lab, as it allows you to test each bean's construction process independently.

All done? Run `AlternateDataSourceConfigTests` to verify the test for this bean passes. You should only get 2 failures now. When the test passes, move on to the next step!

13.2.3.5. Complete `dataSource-createdFromMap`

In this step, you'll complete the configuration of the bean named `dataSource-createdFromMap`.

Complete `dataSource-createdFromMap` (TODO 5) by passing a `java.util.Map` argument to the `createDataSourceFromMap` static factory method on a special Factory class which has been created for you called `TestDataSourceFactory`. You will need to use the `factory-method` attribute to ensure that the correct method is called.

The map should contain three entries for the properties `testDatabaseName`, `schemaLocation`, and `testDataLocation`.

Tip

Inspect the `AlternateDataSourceFactory.createDataSourceFromMap` method to see its signature.

Tip

Use the `constructor-arg`tag to declare a single factory method argument.

Tip

Nest the `<util:map>` tag within the `constructor-arg` tag to define a map as the argument value.

Note how the `constructor-arg` tag is overloaded to define factory method arguments as well as true constructor arguments. Spring will not actually call a constructor here, but instead will call a static factory method *on a class* to construct the bean. When the factory method is invoked the return value gets assigned to the bean's name automatically. This is shown graphically below:

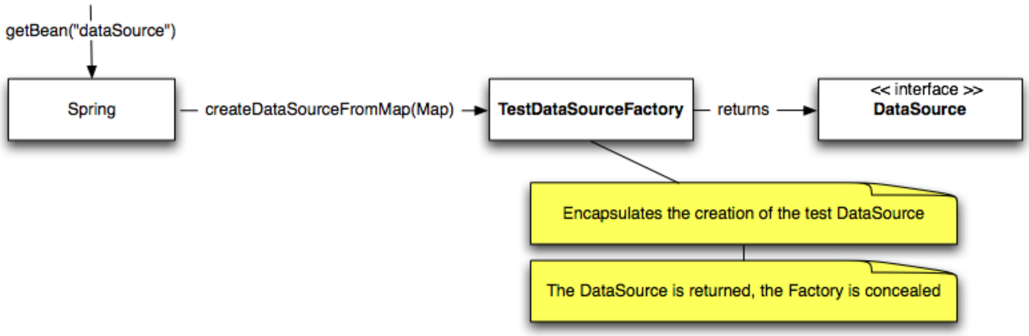

getBean("dataSource")
Spring
createDataSourceFromMap(Map)
TestDataSourceFactory
returns
<< interface >>
DataSource
Encapsulates the creation of the test DataSource
The DataSource is returned, the Factory is concealed

Figure 3: Calling the `createDataSourceFromMap` method to get the `dataSource` bean.

All done? Run `AlternateDataSourceConfigTests` to verify the test for this bean passes. You should only get 1 failure now. When the test passes, move on to the next step!

13.2.3.6. Complete `dataSource-createdFromProperties`

In this step, you'll complete the configuration of the bean named `dataSource-createdFromProperties`.

Complete `dataSource-createdFromProperties` (TODO 6) by passing a `java.util.Properties` argument to the `createDataSourceFromProperties` static factory method on the `TestDataSourceFactory`. Load the properties from the `testdb.properties` file in the `rewards.testdb` package. Use the `<util:properties/>` tag to do this.

Once you've done this, run `AlternateDataSourceConfigTests` to verify the test for this bean is passing. You should get the green bar!

13.2.3.7. Plug the alternate data sources into your application

In this last step, you'll plug each of your alternate data sources into your application one at a time, then re-run your `RewardNetworkTests` to verify your application still works with each.

In `system-test-config.xml`, first link in your alternate data source beans by importing `testdb/config/alternate-datasource-config.xml`. Then, rename the bean currently named `dataSource` to `dataSource-createdFromPlaceholders`. Lastly, define a `dataSource` alias that references the first bean you want to test. Doing this will link that bean into the application.

Tip

You can import beans defined in other files using the `import` tag.

Tip

You can define a bean alias by using the `alias` tag.

A graph showing a `dataSource` aliasing that plugs in the `dataSource-createdFromConstants` bean is shown below:

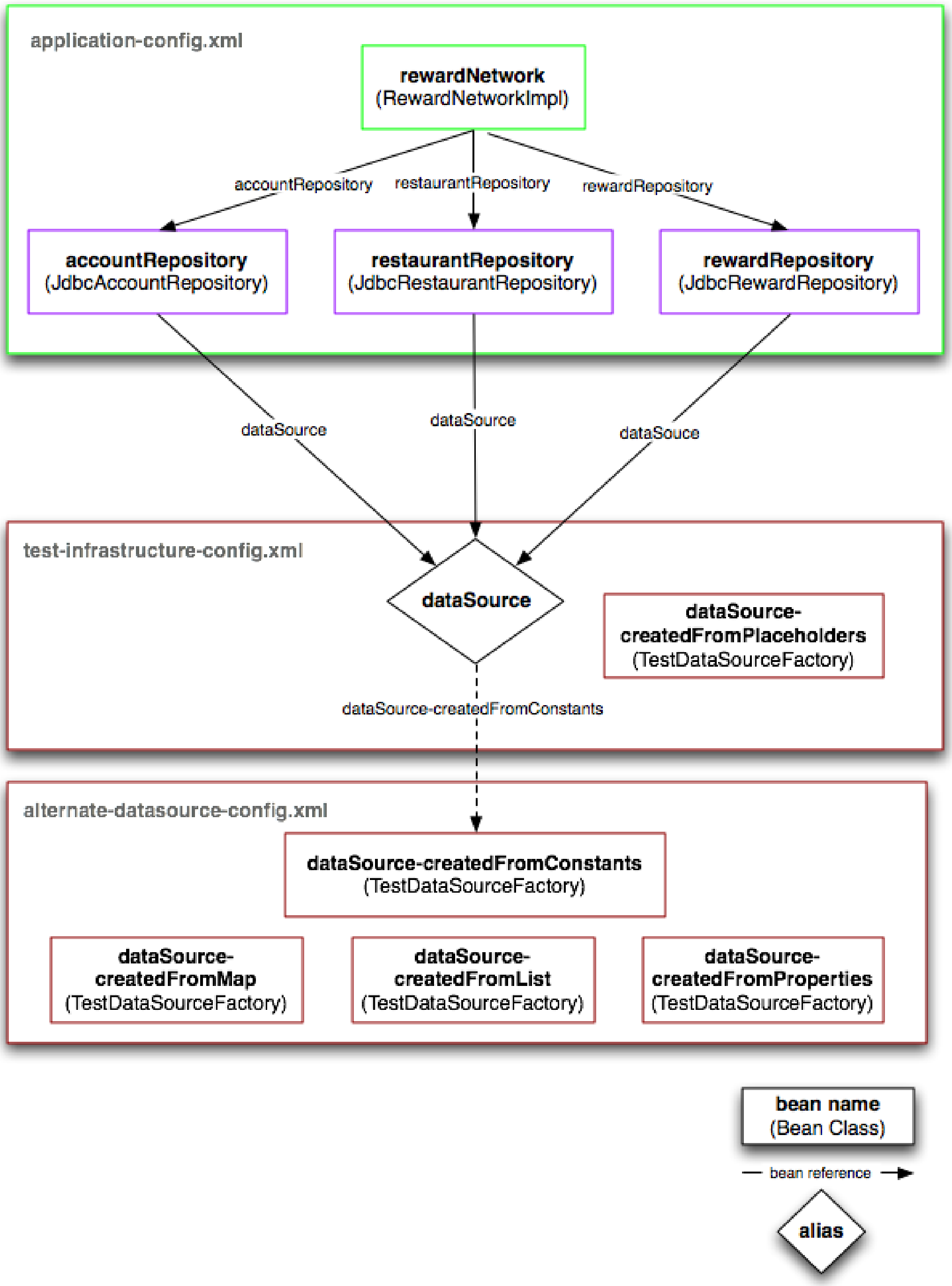

application-config.xml
rewardNetwork (RewardNetworkImpl)
accountRepository
restaurantRepository
rewardRepository
accountRepository (JdbcAccountRepository)
restaurantRepository (JdbcRestaurantRepository)
rewardRepository (JdbcRewardRepository)
dataSource
dataSource
dataSouce
test-infrastructure-config.xml
dataSource
dataSource-createdFromPlaceholders (TestDataSourceFactory)
dataSource-createdFromConstants
alternate-datasource-config.xml
dataSource-createdFromConstants (TestDataSourceFactory)
dataSource-createdFromMap (TestDataSourceFactory)
dataSource-createdFromList (TestDataSourceFactory)
dataSource-createdFromProperties (TestDataSourceFactory)
bean name (Bean Class)
bean reference
alias

Figure 4: `dataSource` aliases `dataSource-createdFromConstants`, plugging it into the application. The aliased bean can be switched easily.

Once you have the `dataSource` alias pointed at one of your alternate data source implementations, re-run `RewardNetworkTests` and verify you get the green bar indicating your application still works. Repeat the test with the remaining data sources by updating where the alias points, and verify you get the green bar each time.

If you got all green, congratulations! You've completed this lab!

Chapter 14. Module test-1: Introducing Unit and System Testing

14.1. Introduction

In this lab you will enhance the rewards application, then unit test your enhancement with JUnit. Once you verify your enhancement works in isolation, you'll run system tests to verify it integrates properly into your application. You'll also see how Spring's system test support library can be used to simplify and improve the performance of your system tests.

What you will learn:

1. How to write JUnit test scenarios

2. How to create stubs

3. The recommended way of system testing an application configured by Spring

Specific subjects you will gain experience with:

1. JUnit

2. Spring's TestContext framework

Estimated time to complete: 30 minutes

14.2. Instructions

14.2.1. Meeting a new Restaurant requirement

Recall that a `Restaurant` is responsible for calculating how much benefit to reward an account for dining:

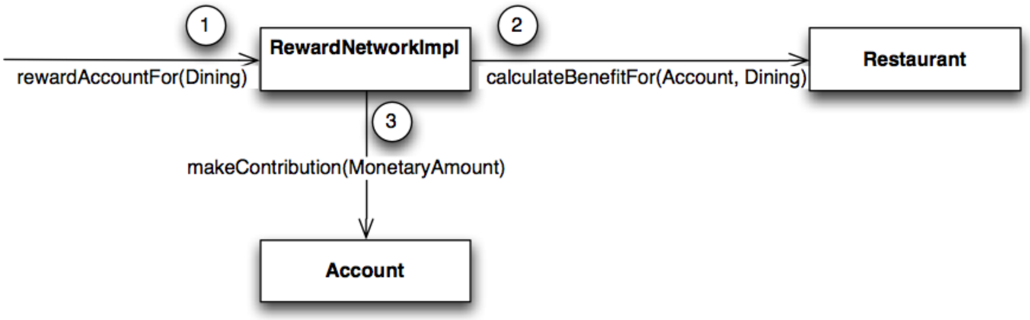

1
rewardAccountFor(Dining)
RewardNetworkImpl
2
calculateBenefitFor(Account, Dining)
Restaurant
3
makeContribution(MonetaryAmount)
Account

Figure 1: The Restaurant is asked (2) to calculate how much money to reward an Account for Dining

This is because benefit calculation rules are specific to each restaurant. For example, some restaurants may reward more than others. By putting this logic in the Restaurant object, you encapsulate restaurant-specific business rules and can change them without impacting the rest of the application.

See for yourself. Open your `RewardNetworkImpl` in the `rewards.internal` package and navigate to where `restaurant.calculateBenefitFor(Account, Dining)` is called. Then open the method declaration (F3) and review the current implementation:

```
public MonetaryAmount calculateBenefitFor(Account account,
   Dining dining) {
    return dining.getAmount().multiplyBy(benefitPercentage);
}
```

This implementation simply multiplies the total dining amount by a percentage to calculate the benefit amount. Restaurants can then have different benefit percentages to affect how much they are willing to reward. For example, Apple Bee's might offer a 8% benefit, while Bennigan's only offers 4%.

The current implementation isn't quite enough, though. You see, restaurants have strict policies about when benefit is available and when it is not. Some of these policies can get complex. For example, AppleBee's only rewards benefit for dining on week days when the total amount already rewarded to an account has not exceeded the monthly maximum. In this section you will enhance your `Restaurant` implementation to support this benefit availability policy requirement, then unit test your enhancement using `JUnit`.

14.2.1.1. Add a `Restaurant BenefitAvailabilityPolicy`

One way to isolate the complexity and variability in a business policy is to introduce a *strategy* interface that encapsulates it. In this step you'll introduce a restaurant `BenefitAvailabilityPolicy` strategy that encapsulates the availability calculation on a restaurant-by-restaurant basis. You'll then delegate to the policy inside your `calculateBenefitFor(Account, Dining)` method.

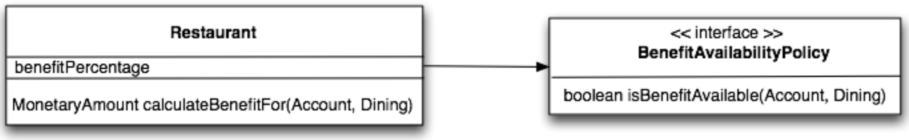

Restaurant
benefitPercentage
MonetaryAmount calculateBenefitFor(Account, Dining)
<< interface >>
BenefitAvailabilityPolicy
boolean isBenefitAvailable(Account, Dining)

Figure 2: A Restaurant with a Benefit Availability Policy

Time to get started. Turn to your `test-1-start` project in Eclipse. You'll see the `BenefitAvailabilityPolicy` interface already exists in the `restaurant` package. Quickly review it, then open your `Restaurant` class where you'll see two TODOs to complete. Complete the first by adding a private field for the benefit availability policy and generating a getter and setter so it can be configured. Complete the second by incorporating the policy into your benefit calculation logic as follows:

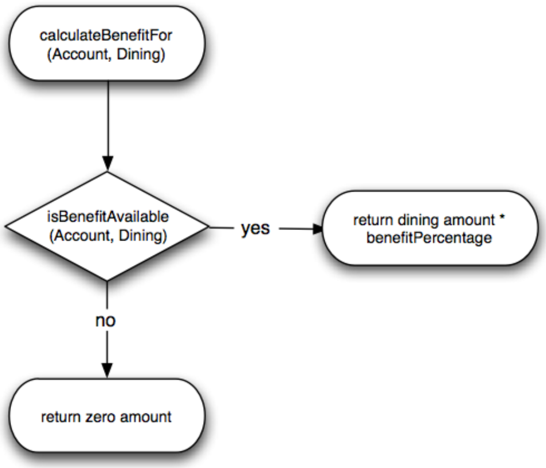


Figure 3: Enhanced `calculateBenefitFor(Account, Dining)` logic

Tip

The `MonetaryAmount` class represents money in the rewards application. Review this class to see how to use it.

When you have completed your enhancement, move on to the next step!

14.2.1.2. Unit test your `Restaurant` logic

You just enhanced your `Restaurant` class to apply a benefit availability policy, but how do you know your enhancement works? You don't, not without a test that proves it. In this step you will unit test your `Restaurant` class to verify your enhancement is correct.

You're in luck - this lab already has a unit test started for you. Open `RestaurantTests` in the `rewards.internal.restaurant` package within the `src/test/java` source folder to review it. You'll see it is already a JUnit 4 Test and is already setting up some test fixtures for you. You'll also see several TODOs for you to complete.

First run the test as-is. You'll see it succeed, as both its test methods are empty: the test logic has yet to be written.

Before diving into writing your test methods, complete the first TODO to setup your test fixtures. In a set up method (annotated with

@Before), create a new `Restaurant` object and assign it to a private field. Pass it your own merchant number and name. Set your own benefit percentage. This is the unit you will use in your test methods.

Tip

The `Percentage` class represents percentages in the rewards application. Review this class to see how to use it to set the benefit percentage.

Once you've setup the Restaurant fixture, it is time to write your some tests. First, ask yourself what scenarios are there to test for? That will determine how many test methods you need. See the following visualizations of the logic flow:

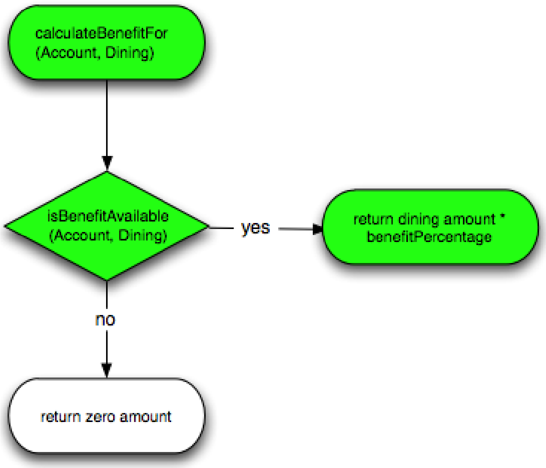

calculateBenefitFor (Account, Dining)
isBenefitAvailable (Account, Dining)
yes
return dining amount * benefitPercentage
no
return zero amount

Figure 4: Scenario 'benefit available'

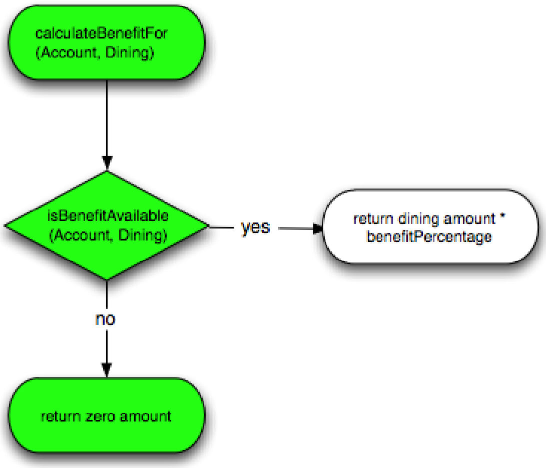

calculateBenefitFor (Account, Dining)
isBenefitAvailable (Account, Dining)
yes
return dining amount * benefitPercentage
no
return zero amount

Figure 5: Scenario 2 'benefit not available'

From Figures 4 and 5 you can see there are two distinct scenarios to test for: 'benefit available' and 'benefit not available'.

In `RestaurantTests`, complete the second TODO by implementing a test method that tests the 'benefit available' scenario shown in Figure 4.

Tip

To exercise the 'benefit available' scenario, configure a dummy `BenefitAvailabilityPolicy` that simply returns `true`. The `StubBenefitAvailabilityPolicy` static inner class has been provided for you to use as a convenience.

Tip

Verify the returned benefit amount with JUnit's `assertEquals(Object expected, Object actual)` method.

Once your test method is written, re-run your test case. When you have the green bar, congratulations! When benefit is available your benefit calculation logic works as expected.

Now complete the third TODO by implementing a test method to test the remaining 'benefit not available' scenario shown in Figure 5. When you have the green bar for both test scenarios, you have complete code coverage. Move on to the next step!

14.2.2. Integrating your Restaurant enhancement into the application

So far you have verified your enhanced `Restaurant` implementation works in isolation. However, you have not verified if the rewards application applies your new enhancement correctly. In the following steps, you will run system tests to ensure your work is successfully integrated into the rewards application.

Where in the application is the `Restaurant` unit touched? That will help determine what other tests need to be run. Recall the

`RewardNetworkImpl` uses a Restaurant to calculate the benefit amount for a reward transaction:

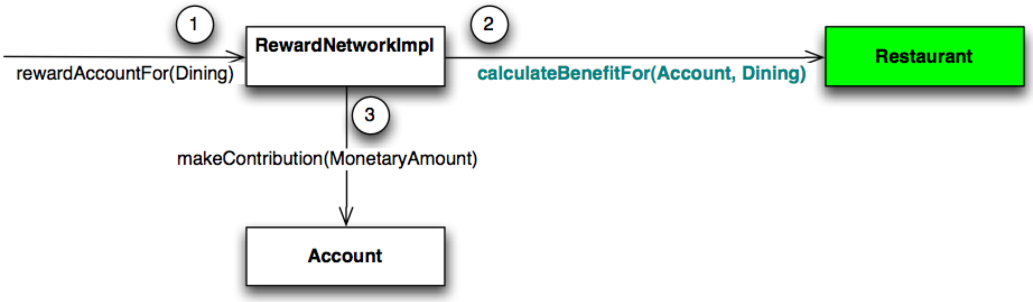

1
rewardAccountFor(Dining)
RewardNetworkImpl
2
calculateBenefitFor(Account, Dining)
Restaurant
3
makeContribution(MonetaryAmount)
Account

Figure 7: A Restaurant calculating the benefit amount to reward

So that's one place. But how does this Restaurant object get *created* before it is used? Who is in charge of that? What are the lifecycle semantics?

All restaurant data is stored centrally in the database, where each restaurant entity is tracked by an unique merchant number and may be updated by other applications. Therefore, for each reward transaction a `Restaurant` needs to be restored from its database representation, asked to do the benefit calculation, then go out of scope. The reward network delegates to a data access object (DAO) called a "repository" to do this. The repository encapsulates the data source implementation and the complexity of the object restoration logic. This is highlighted below:

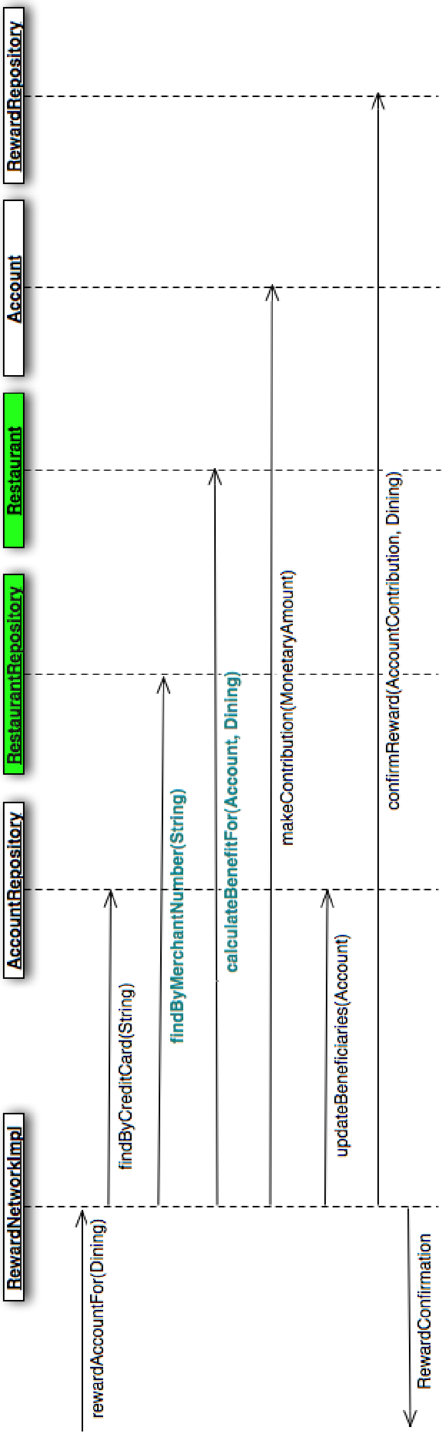

RewardNetworkImpl
AccountRepository
RestaurantRepository
Restaurant
Account
RewardRepository
rewardAccountFor(Dining)
findByCreditCard(String)
findByMerchantNumber(String)
calculateBenefitFor(Account, Dining)
makeContribution(MonetaryAmount)
updateBeneficiaries(Account)
confirmReward(AccountContribution, Dining)
RewardConfirmation

Figure 8: The RestaurantRepository restoring a Restaurant tracked by its merchant number

So clearly there are two areas in your application where the Restaurant unit is touched. A `Restaurant` is:

1. Restored by the `RestaurantRepository`.

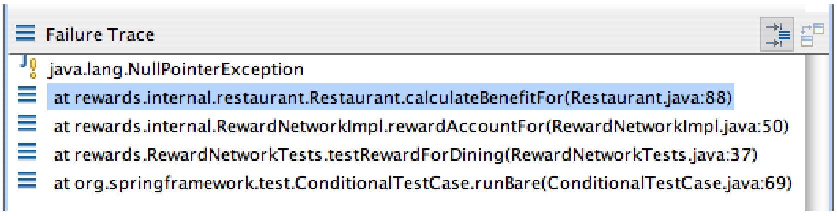
2. Used by the `RewardNetworkImpl`.

In the following steps you'll see if any change is needed in these areas to support your enhancement.

14.2.2.1. Run `RewardNetworkTests`

A good first test would be to verify if the rewards application still works as a whole since your enhancement. Run `RewardNetworkTests` in the root `rewards` package in the test tree. What do you see? You should see the red bar indicating test failure. Something went wrong.

Now, figure out what went wrong. Inspect the failure trace by double clicking on the top of the stack (as shown below). This will take you to the line where the exception was thrown.



```
java.lang.NullPointerException
at rewards.internal.restaurant.Restaurant.calculateBenefitFor(Restaurant.java:88)
at rewards.internal.RewardNetworkImpl.rewardAccountFor(RewardNetworkImpl.java:50)
at rewards.RewardNetworkTests.testRewardForDining(RewardNetworkTests.java:37)
at org.springframework.test.ConditionalTestCase.runBare(ConditionalTestCase.java:69)
```

Figure 9: RewardNetworkTests failure trace

So what happened? Clearly a `NullPointerException` was thrown because the Restaurant's `benefitAvailabilityPolicy` reference was null. What component of the application is responsible for setting this reference? The `RestaurantRepository` is responsible for this, as part of restoring a Restaurant object from its persistent form. Clearly the repository implementation didn't do the restoration correctly. There must be a bug.

Time to investigate. First, determine the `RestaurantRepository` implementation in use in your application. To do this, in the Spring Beans view, graph your `systemTest` config set. Then double click on the `restaurantRepository` bean. Note the class of the bean definition. In the next step you will fix the bug in this class to properly restore Restaurant benefit availability policies.

14.2.2.2. Fix `JdbcRestaurantRepository`

In this step you will fix the `RestaurantRepository` implementation in use, `JdbcRestaurantRepository`, to properly map the benefit availability policy from the restaurant's database representation.

Open `JdbcRestaurantRepository` in the `restaurant` package. First, confirm that it is not mapping a restaurant `BenefitAvailabilityPolicy` from the result set. You can confirm this by navigating to where the `mapRestaurant(ResultSet)` method is called and opening its declaration. Within the method body, you'll see other properties being mapped such as the name and benefit percentage, but a `TODO` asking you to map the benefit availability policy.

The restaurant mappings rules that should be followed are shown below:

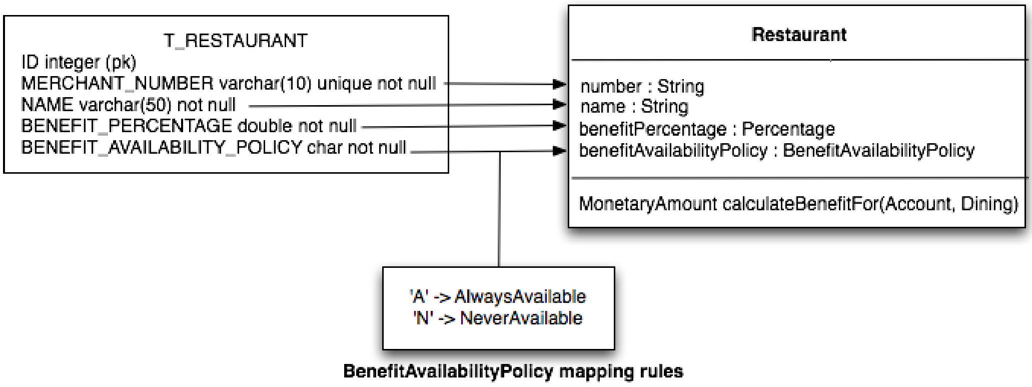

T_RESTAURANT
ID integer (pk)
MERCHANT_NUMBER varchar(10) unique not null
NAME varchar(50) not null
BENEFIT_PERCENTAGE double not null
BENEFIT_AVAILABILITY_POLICY char not null
Restaurant
number : String
name : String
benefitPercentage : Percentage
benefitAvailabilityPolicy : BenefitAvailabilityPolicy
MonetaryAmount calculateBenefitFor(Account, Dining)
'A' -> AlwaysAvailable
'N' -> NeverAvailable
BenefitAvailabilityPolicy mapping rules

Figure 9: The Restaurant relational-to-object mapping rules

Complete the `TODO` by mapping the value of the `BENEFIT_AVAILABILITY_POLICY` column to the corresponding `BenefitAvailabilityPolicy` instance; then, setting the instance as the value of the restaurant's `benefitAvailabilityPolicy` property. Use the `mapBenefitAvailabilityPolicy(ResultSet)` method to help you do this (hint: `NeverAvailable` and `AlwaysAvailable` are the class names of the corresponding policies). Doing this will complete restoration of the `Restaurant` so it can do its job.

When you have applied this fix, move on to the next step!

14.2.2.3. Re-run `RewardNetworkTests`

With the fix applied you are now ready to re-test your application. Re-run `RewardNetworkTests`. Verify you now get the green bar. The test exercises a 100.00 dining at restaurant '1234567890' with a 8% benefit percentage and availability policy of 'A' - Always Available, so if your test is passing a reward of 8.00 is being confirmed as expected!

Tip

Review the `test-data.sql` file in the `rewards.testdb` package to review the data you are testing against.

When you have the green bar, you've completed this section! You've successfully integrated your restaurant enhancement into the application with basic support for 'A' Always Available and 'N' Never Available benefit availability policies driven by relational data.

14.2.3. Improving the performance of your system test

Spring provides system test support classes as part of its `spring-test` library. In this section, you'll refactor your existing `RewardNetworkTests` class to take advantage of this support. You'll see how this support simplifies your test setup code, as well as improves test performance.

14.2.3.1. Refactor to use Spring's TestContext framework

One of the central components in the TestContext framework is the SpringJUnitClassRunner. In this step, you'll tell JUnit to run your test with it and then refactor your test as necessary to work with it.

Return to your `RewardNetworkTests` class and add an `@RunWith` annotation to it passing in `SpringJUnit4ClassRunner.class` as the runner. Be sure to use `Ctrl+Space` to get code completion on this long class name (for example, by typing `SpJ` in the RunWith annotations value and then pressing `Ctrl+Space`)

Once you have told JUnit to use Spring's TestRunner to run this test you can start telling Spring what to do to setup your test fixture. You will do this using annotation based dependency injection and specific test meta data.

To complete the refactoring you need to make two changes. First, annotate the test with `@ContextConfiguration`.

Set the `locations` property of the annotation to the same `String[]` you defined in your `setUp` method. Then, delete the original `setUp` method as it is no longer needed. Spring's test runner will automatically create (and cache) an `ApplicationContext` for you.

Now run your test. It should fail with the red bar because your `rewardNetwork` field is now null. Look over your test code: the field is clearly not being set now.

One more change left to make. It's an easy one: simply 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! The autowiring is based on the type, but can be fine tuned using qualifiers.

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.

When you have the green bar, congratulations, you've completed this lab! You've successfully integrated an enhancement to the rewards application, and at the same time simplified your system test by leveraging Spring's test support. In addition, the performance of your system test has improved as the `ApplicationContext` is now created once per test case run (and cached) instead of once per test method.

Tip

@ContextConfiguration assumes a default XML file name of `<Classname>-context.xml` if a location is not specified. Try renaming the `system-test-config.xml` file to `RewardNetworkTests-context.xml` and using @ContextConfiguration without specifying the location. This is an example of convention over configuration.

Tip

Also notice that this is the first time we've used the `<import>` element to define the dependencies between the configuration files. This means that we no longer need a config set defined as SpringSource Tool Suite can parse the `<import>` element in the same way as the ApplicationContext. Test this out by navigating to the Spring Explorer and opening a dependency graph on the XML file.

Chapter 15. Module jpa-1: JPA Simplification using Spring

15.1. 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

15.2. 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 1 below), and you can use this fact to simplify the JPA mapping of the `Account` entity.

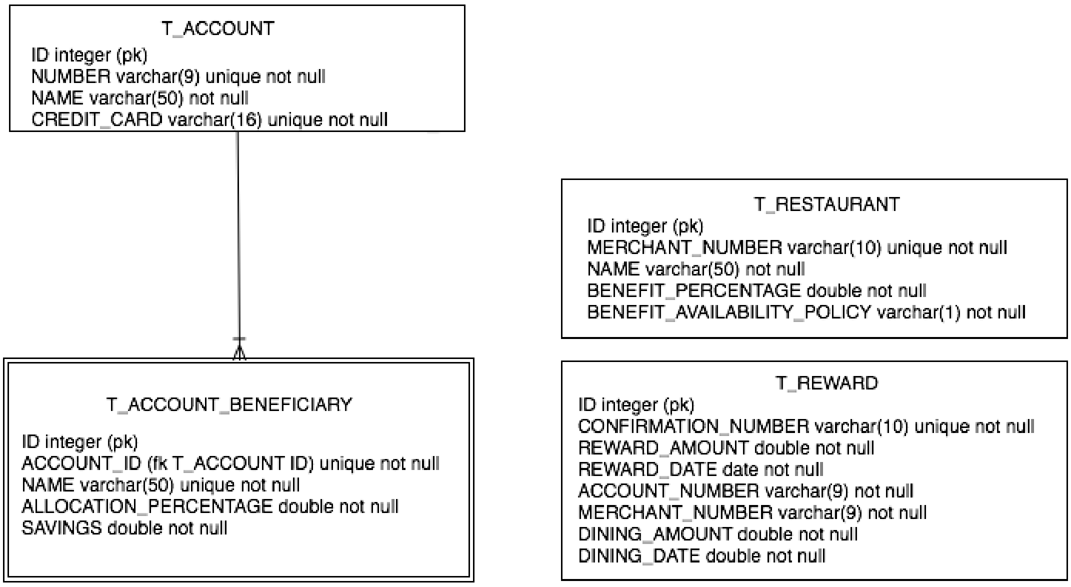

T_ACCOUNT
ID integer (pk)
NUMBER varchar(9) unique not null
NAME varchar(50) not null
CREDIT_CARD varchar(16) unique not null
T_RESTAURANT
ID integer (pk)
MERCHANT_NUMBER varchar(10) unique not null
NAME varchar(50) not null
BENEFIT_PERCENTAGE double not null
BENEFIT_AVAILABILITY_POLICY varchar(1) not null
T_ACCOUNT_BENEFICIARY
ID integer (pk)
ACCOUNT_ID (fk T_ACCOUNT ID) unique not null
NAME varchar(50) unique not null
ALLOCATION_PERCENTAGE double not null
SAVINGS double not null
T_REWARD
ID integer (pk)
CONFIRMATION_NUMBER varchar(10) unique not null
REWARD_AMOUNT double not null
REWARD_DATE date not null
ACCOUNT_NUMBER varchar(9) not null
MERCHANT_NUMBER varchar(9) not null
DINING_AMOUNT double not null
DINING_DATE double not null

Figure 1: The database schema for this lab, showing the credit card number as part of the account table.

15.2.1. Using JPA in the Account module

15.2.1.1. 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. Add an `Entity` annotation on this class to specifiy 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` (TODO 1).

3. Every entity needs a primary key so that it can be used by the JPA implementation. All the tables in this lab have auto-generated numeric keys. We have already added a long integer `entityId` field to the classes you will be using.

4. Go ahead and annotate the `entityId` as the Id for this class (TODO 2).

5. We need to tell JPA to let the databaase automatically generate each primary key for us. Add a `GeneratedValue` annotation to the `entityId`.

6. By default JPA uses the field name as the column name to map a field into a databased table. Because the column name doesn't match with the field name, we need to override it desired field name.

7. Complete the mapping for the remaining Account fields - the `number`, `name`, `beneficiaries`, and `creditCardNumber` properties (TODO 3). 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. Annotate the `Beneficiary` class as a JPA entity and specify its table. (TODO 4).

2. Add the mappings for the `entityId` and `name` fields - refer back to what you have already done for `Account`.

3. Finally map the `savings`, and `allocationPercentage` fields (TODO 5).

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!

15.2.1.2. Implement `JpaAccountRepository`

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 by completing TODO 6.

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!

15.2.1.3. 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

You may have already noticed that the `AccountRepository` interface has changed from previous labs. Specifically, the `updateBeneficiaries(Account)` method has been removed. This method was removed because it is simply 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.

15.2.2. Using JPA in the Restaurant module

15.2.2.1. 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. Open the `Restaurant` in the `rewards.internal.restaurant` package. We will configure all object-to-relational mapping rules using annotations inside this class (TODO 7).

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 reward dining database schema 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!

15.2.2.2. Implement `JpaRestaurantRepository`

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 (TODO 8).

Tip

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

15.2.2.3. 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!

15.2.3. 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!

15.2.3.1. Define the JPA configuration for the application

We need to setup our Spring configuration.

1. Open `application-config.xml` in the `rewards.internal` package. In this file, define beans for the `JpaAccountRepository` and the `JpaRestaurantRepository` (TODO 7 & 8).

2. In contrast to the previous JDBC based implementations, these classes don't need any external dependencies. Spring will automatically recognize the `PersistenceContext` annotation inside the `JpaAccountRepository` and `JpaRestaurantRepository` - check the methods `setPersistenceContext(EntityManager)` in each class. However you need to tell Spring to use this annotation.

   Modify `application-config.xml` to enable the `PersistenceContext` annotation. Use the `context` namespace.

3. Now to setup the Entity Manager Factory. There are three steps.

   a. Open `system-test-config.xml` 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`.

   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 an inner bean for 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.

4. Finally, define a `transactionManager` bean so the Reward Network can drive transactions using JPA APIs. Use the `JpaTransactionManager` implementation. Set its `entityManagerFactory` property appropriately.

5. Now go to the Spring Explorer view in Eclipse and show the graph of the `jpa-1-start -> systemTest`. If you configured your application context properly the graph should look something like Figure 4:

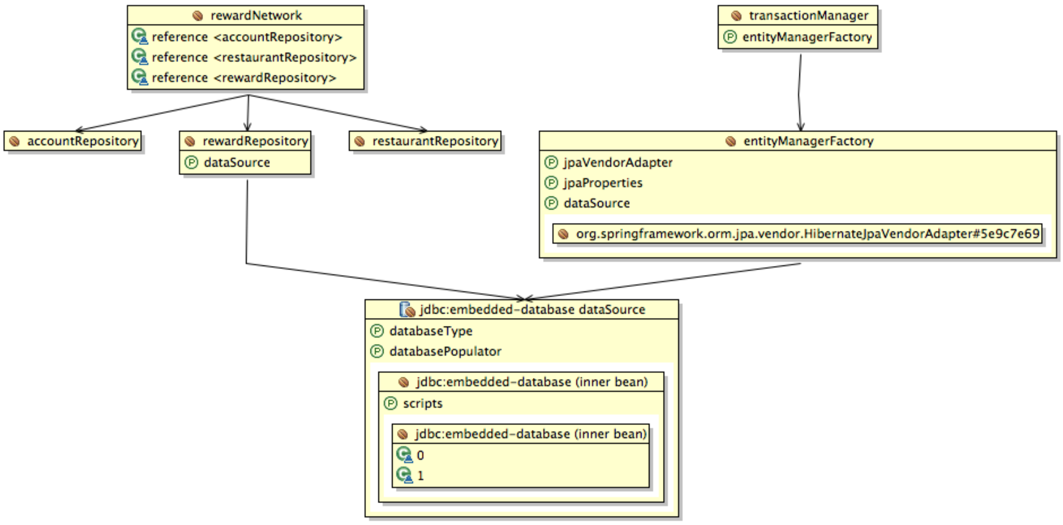

rewardNetwork
reference <accountRepository>
reference <restaurantRepository>
reference <rewardRepository>
transactionManager
entityManagerFactory
accountRepository
rewardRepository
dataSource
restaurantRepository
entityManagerFactory
jpaVendorAdapter
jpaProperties
dataSource
org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter#5e9c7e69
jdbc:embedded-database dataSource
databaseType
databasePopulator
jdbc:embedded-database (inner bean)
scripts
jdbc:embedded-database (inner bean)
0
1

Figure 4: The configuration of the context.

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

15.2.3.2. 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!

15.2.4. BONUS CREDIT

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

15.2.4.1. Type-safe Query

Redo the query you wrote in `JpaAccountRepository` to use the Criteria Query API instead.

Note

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).

15.2.4.2. Mapping Benefit Availability Policy

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, 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, if it is "N" to NeverAvailable, 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 (TODO 9).

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 it works.

Chapter 16. Module jmx-1: JMX Management of Performance Monitor

16.1. 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. `context:mbean-server`

3. `context:mbean-export`

Estimated time to complete: 30 mins

16.2. Instructions

16.2.1. Exposing the `MonitorFactory` via JMX

16.2.1.1. 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 it's 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

16.2.1.2. Add JMX metadata to the implementation class

Add Spring's JMX annotations `@ManagedResource`, `@ManagedAttribute` and `@ManagedOperation` to the class as well as methods you want to expose via JMX (TODO 1). 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

16.2.1.3. Activate annotation driven JMX in application configuration

Find and open the `aspects-config.xml` file in the `rewards.internal.aspects` package. In this file activate annotation driven JMX by using `mbeans-export` and `mbean-server` from the `context` namespace. (TODO 2).

16.2.1.4. 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-1-start in your browser. You should see the welcome page display containing a form that submits to the RewardServlet.

16.2.1.5. 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.

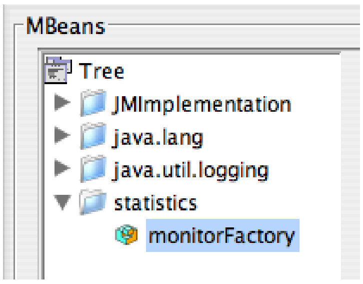
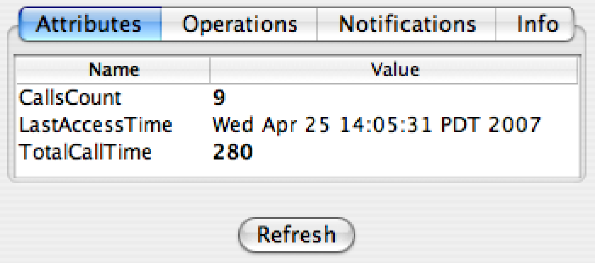


Figure 1: The `MonitorFactory` MBean

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



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

Figure 2: The `MonitorFactory` attributes

Tip

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

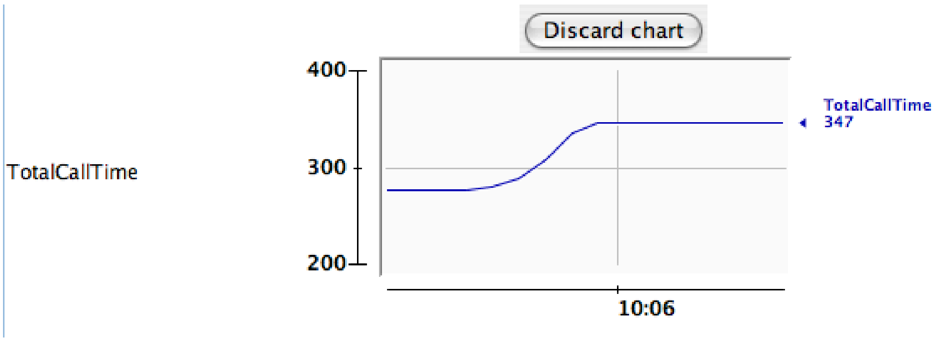


Figure 3: Scalar value graph

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

16.2.2. Exporting pre-defined MBeans

By using Spring's `context:mbean-export` 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`).

16.2.2.1. 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 17. ws-1: 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

In the `ws-1-start` 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

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

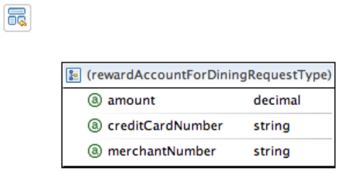
You need to refresh the project (select the project and press F5) before you see this file.

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)	
ⓐ amount	decimal
ⓐ creditCardNumber	string
ⓐ merchantNumber	string

Figure 1: `(rewardAccountForDiningRequestType)` structure

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 (TODO 1). Now you have completed your contract definition, move on to the next step.

17.2.2. Exporting the `RewardNetwork` as a SOAP endpoint

17.2.2.1. Add the `MessageDispatcherServlet`

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` (TODO 2).

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.

17.2.2.2. Generate the classes with JAXB2

In this lab we use JAXB2 to convert between Objects and XML. So the first step is 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.

17.2.2.3. Create the SOAP endpoint

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 (*TODO 3*). 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.

17.2.2.4. Complete the Spring WS configuration

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 (*TODO 4*). 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.

17.2.2.5. Start the web application

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-1-start

17.2.3. 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.

17.2.3.1. Test the web service

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.

17.2.3.2. Using the TCP/IP monitor to see the SOAP messages

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".

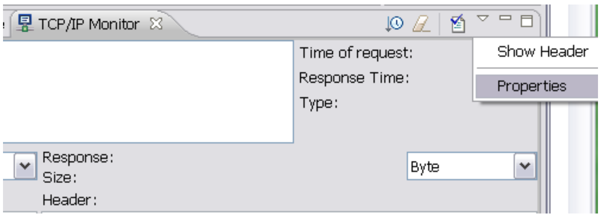


Figure 2: The "properties" menu entry of the TCP/IP Monitor view

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.

17.2.3.3. Using WebServiceTemplate with JAXB2

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 (TODO 5). 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. Module rest-ws: Building RESTful applications with Spring MVC

18.1. 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

18.2. 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.

18.2.1. 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`.

18.2.1.1. Inspect the current application

First open the `web.xml` deployment descriptor to see how the application is bootstrapped: the `app-config.xml` file contains the necessary configuration for the root context, which contains a `accountManager` bean that provides transactional data access operations to manage `Account` instances. The `mvc-config.xml` 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.

18.2.1.2. Expose the list of accounts

Open the `AccountController`. Notice that it offers several methods to deal with various requests to access certain resources. Complete `TODO 01` by adding the necessary annotations to the `accountSummary` method to make it respond to GET requests to `/accounts`.

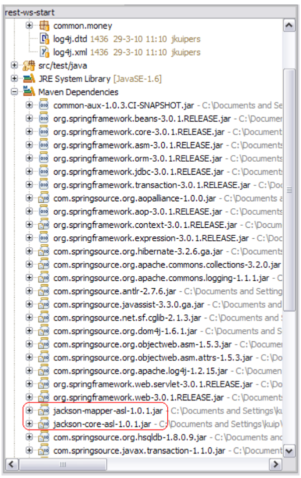
Tip

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

Now try to access http://localhost:8080/rest-ws-start/app/accounts from that same browser. You should 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:



```
rest-ws-start
    common.money
    log4j.dtd 1436  29-3-10 11:10  jkuipers
    log4j.xml 1436  29-3-10 11:10  jkuipers
  src/test/java
  JRE System Library [JavaSE-1.6]
  Maven Dependencies
    common-aux-1.0.3.CI-SNAPSHOT.jar - C:\Documents and Se
    org.springframework.beans-3.0.1.RELEASE.jar - C:\Documer
    org.springframework.core-3.0.1.RELEASE.jar - C:\Document
    org.springframework.asm-3.0.1.RELEASE.jar - C:\Documents
    org.springframework.orm-3.0.1.RELEASE.jar - C:\Documents
    org.springframework.jdbc-3.0.1.RELEASE.jar - C:\Documents
    org.springframework.transaction-3.0.1.RELEASE.jar - C:\Doc
    com.springsource.org.aopalliance-1.0.0.jar - C:\Documents a
    org.springframework.aop-3.0.1.RELEASE.jar - C:\Documents
    org.springframework.context-3.0.1.RELEASE.jar - C:\Docum
    org.springframework.expression-3.0.1.RELEASE.jar - C:\Doc
    com.springsource.org.hibernate-3.2.6.ga.jar - C:\Documents
    com.springsource.org.apache.commons.collections-3.2.0.jar
    com.springsource.org.apache.commons.logging-1.1.1.jar - C
    com.springsource.antlr-2.7.6.jar - C:\Documents and Setting
    com.springsource.javassist-3.3.0.ga.jar - C:\Documents and
    com.springsource.net.sf.cglib-2.1.3.jar - C:\Documents and S
    com.springsource.org.dom4j-1.6.1.jar - C:\Documents and S
    com.springsource.org.objectweb.asm-1.5.3.jar - C:\Documen
    com.springsource.org.objectweb.asm.attrs-1.5.3.jar - C:\Do
    com.springsource.org.apache.log4j-1.2.15.jar - C:\Documen
    org.springframework.web.servlet-3.0.1.RELEASE.jar - C:\Do
    org.springframework.web-3.0.1.RELEASE.jar - C:\Documents
    jackson-mapper-asl-1.0.1.jar - C:\Documents and Settings\ku
    jackson-core-asl-1.0.1.jar - C:\Documents and Settings\kuip\
    com.springsource.org.hsqldb-1.8.0.9.jar - C:\Documents and
    com.springsource.javax.transaction-1.1.0.jar - C:\Document
```

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.

18.2.1.3. Retrieve the list of accounts using a `RestTemplate`

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 (`TODO 02`).

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.

18.2.1.4. Expose a single account

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 `TODO 03` by completing 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-start/app/accounts/0 to verify the result.

When you're done with the controller, complete `TODO 04` in the `AccountClientTests` by retrieving the account with id 0.

Tip

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

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

18.2.1.5. 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.

Implement `TODO 05` by making sure 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.

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 `TODO 06` 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!

When you're done, complete `TODO 07` 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 08` 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!

18.2.1.6. Seeing what happens at the HTTP level

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".

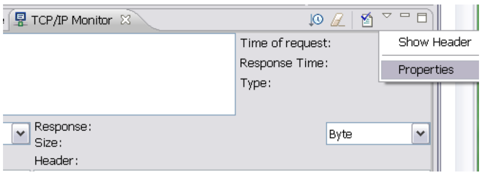


Figure 2: The "properties" menu entry of the TCP/IP Monitor view

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 speficied 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.

18.2.1.7. Create and delete a beneficiary

Complete `TODO 09` by completing 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.

Also finish `TODO 10` by setting 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.

Proceed by completing `TODO 11` and complete the `removeBeneficiary` method. This time, return a `204 No Content` status.

To test your work, switch to the `AccountClientTests` and complete `TODO`s 12 to 15. 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!

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

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, complete `TODO 16` by adding a new exception handling method that returns the correct code in case of a `DataIntegrityViolationConstraint`.

Tip

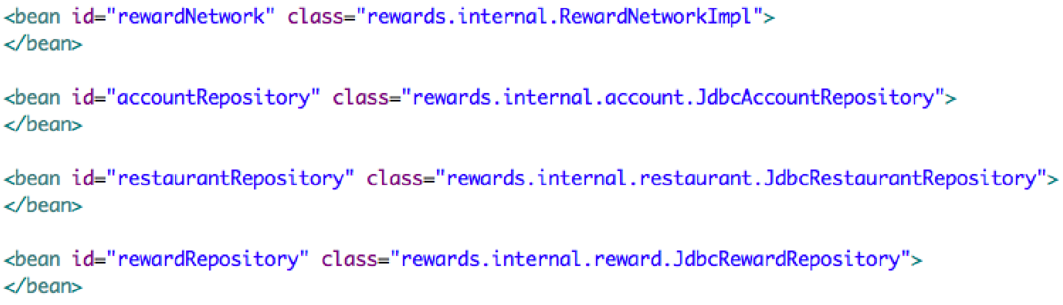
Have a look at the existing `handleNotFound` method for a way to do this.

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.

Part II. Appendices

Appendix A. Spring XML Configuration Tips

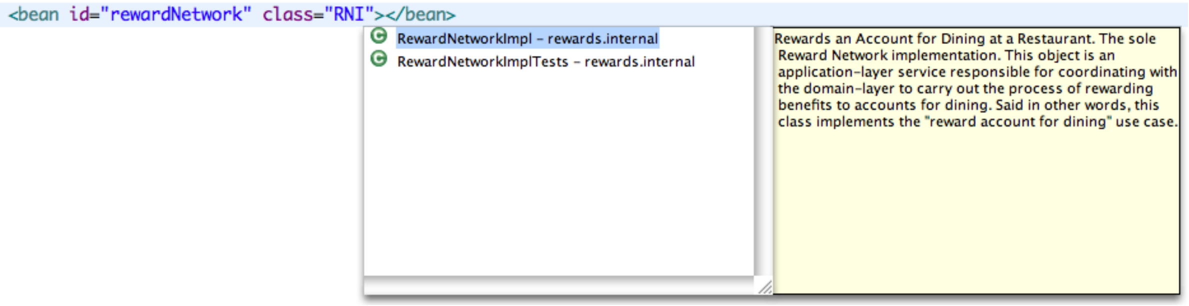
A.1. Bare-bones Bean Definitions



```
<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



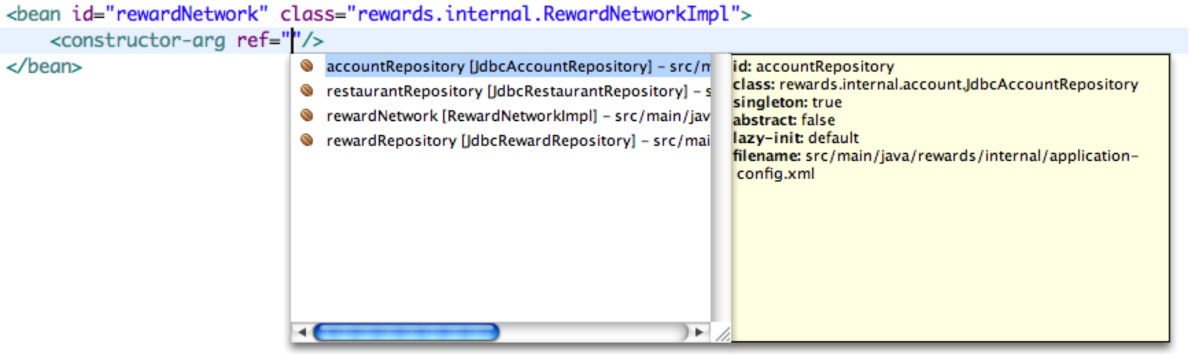
```
<bean id="rewardNetwork" class="RNI"></bean>
```

RewardNetworkImpl - rewards.internal
RewardNetworkImplTests - rewards.internal

Rewards an Account for Dining at a Restaurant. The sole Reward Network implementation. This object is an application-layer service responsible for coordinating with the domain-layer to carry out the process of rewarding benefits to accounts for dining. Said in other words, this class implements the "reward account for dining" use case.

Bean class auto-completion

A.3. Constructor Arguments Auto-Completion



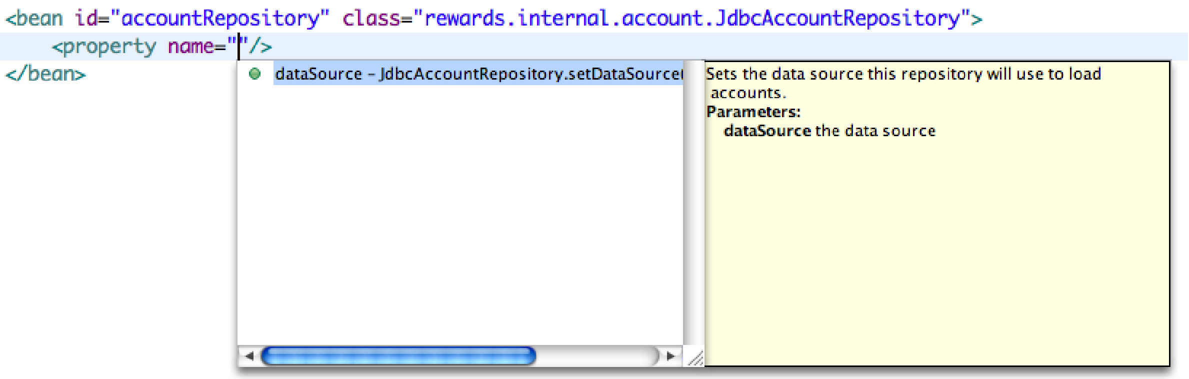
```
<bean id="rewardNetwork" class="rewards.internal.RewardNetworkImpl">
    <constructor-arg ref=""/>
</bean>
```

accountRepository [JdbcAccountRepository] - src/n
restaurantRepository [JdbcRestaurantRepository] - s
rewardNetwork [RewardNetworkImpl] - src/main/jav
rewardRepository [JdbcRewardRepository] - src/mai

id: accountRepository
class: rewards.internal.account.JdbcAccountRepository
singleton: true
abstract: false
lazy-init: default
filename: src/main/java/rewards/internal/application-config.xml

Constructor argument auto-completion

A.4. Bean Properties Auto-Completion



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

Sets the data source this repository will use to load accounts.
Parameters:
dataSource the data source

Bean property name completion

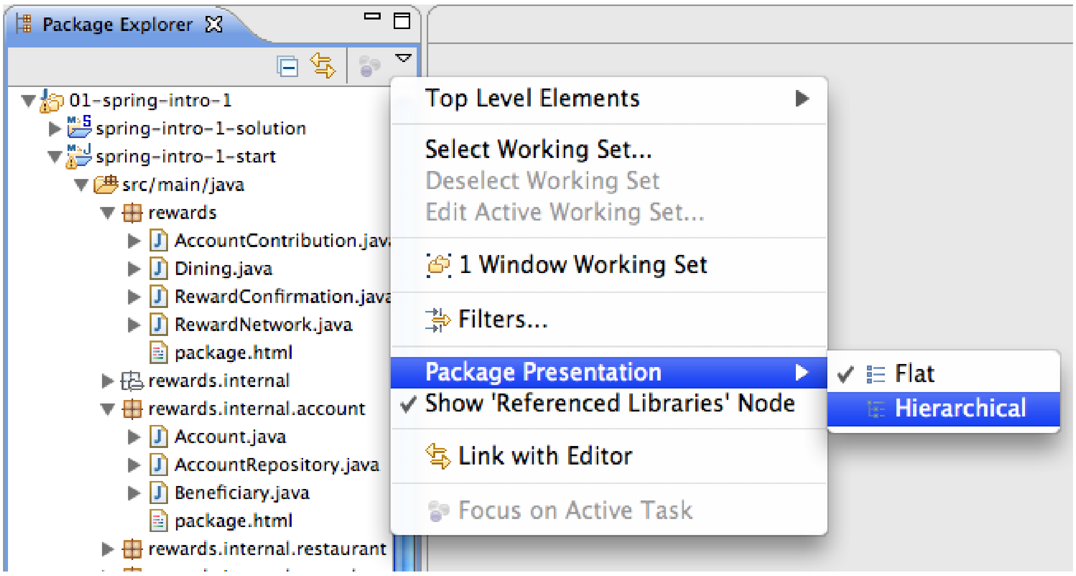
Appendix B. Eclipse Tips

B.1. Introduction

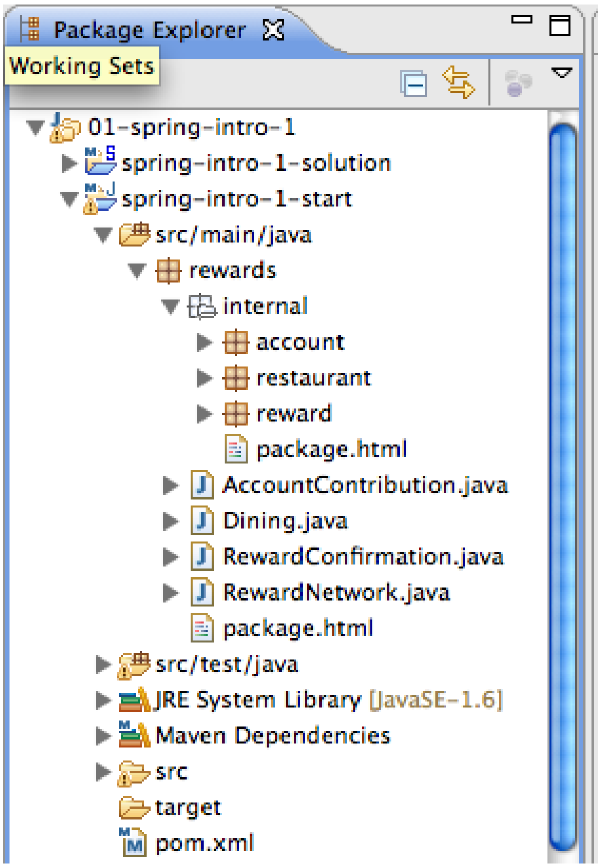
This section will give you some useful hints for using Eclipse.

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.



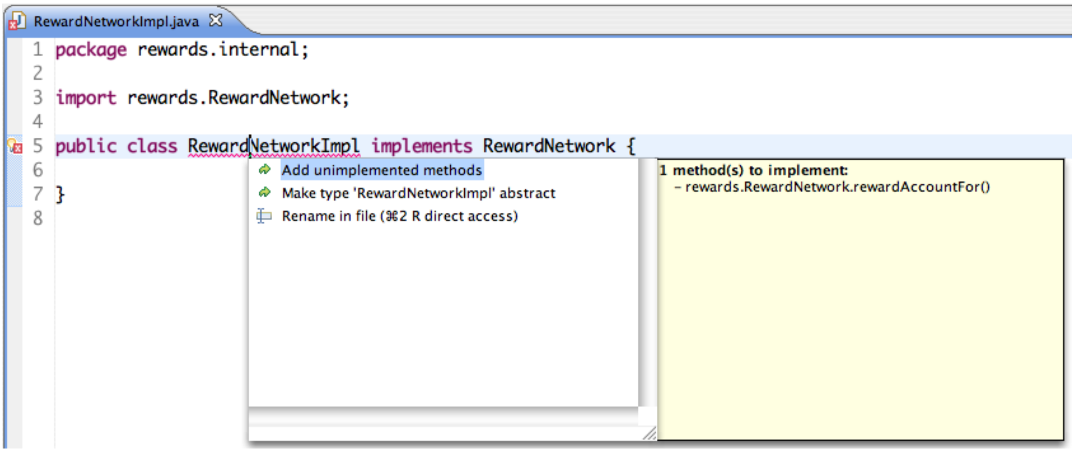
Switch between hierarchical and flat views by selecting the menu inside the package view (represented as a triangle), selecting either Flat or Hierarchical from the Package Presentation submenu



```
01-spring-intro-1
  spring-intro-1-solution
  spring-intro-1-start
    src/main/java
      rewards
        internal
          account
          restaurant
          reward
          package.html
        AccountContribution.java
        Dining.java
        RewardConfirmation.java
        RewardNetwork.java
        package.html
    src/test/java
    JRE System Library [JavaSE-1.6]
    Maven Dependencies
    src
    target
    pom.xml
```

The hierarchical view shows nested packages in a tree view

B.3. Add Unimplemented Methods



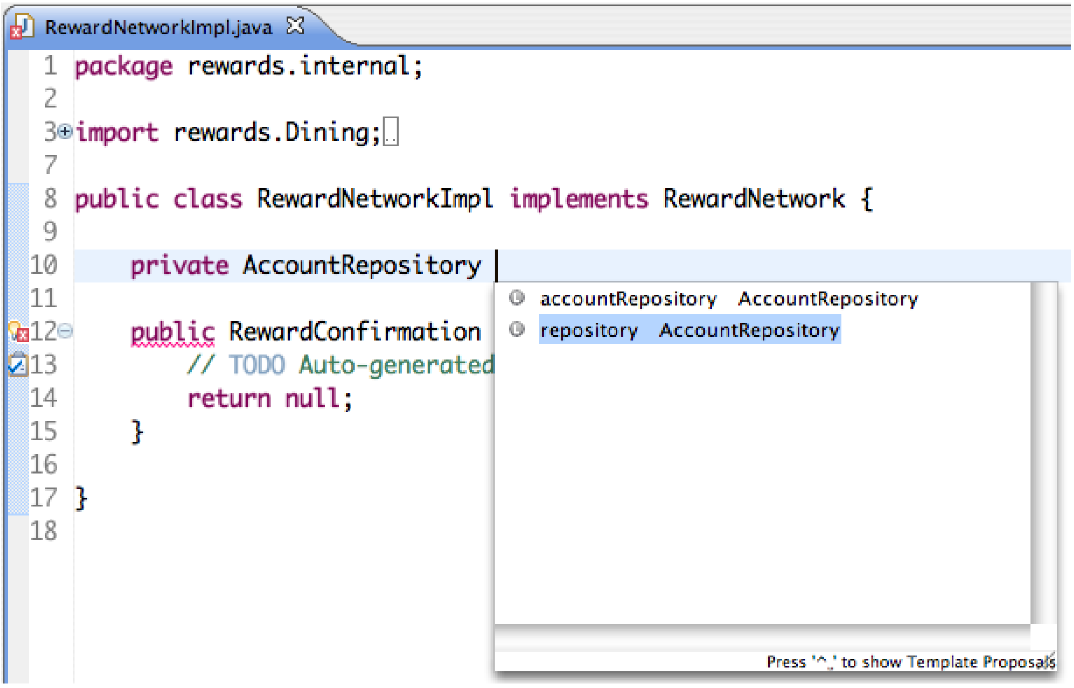
```
package rewards.internal;

import rewards.RewardNetwork;

public class RewardNetworkImpl implements RewardNetwork {

}
```

Add unimplemented methods
Make type 'RewardNetworkImpl' abstract
Rename in file (⌘2 R direct access)

1 method(s) to implement:
- rewards.RewardNetwork.rewardAccountFor()

"Add unimplemented methods" quick fix

B.4. Field Auto-Completion

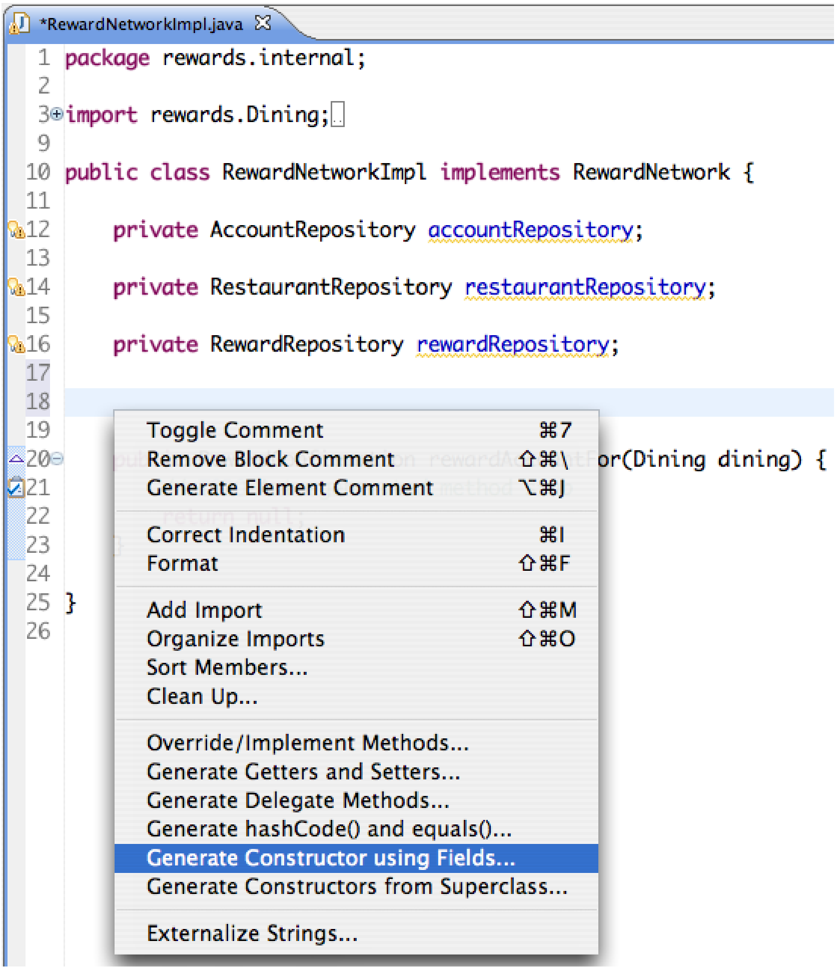


```
package rewards.internal;

import rewards.Dining;

public class RewardNetworkImpl implements RewardNetwork {

    private AccountRepository 

    public RewardConfirmation
        // TODO Auto-generated
        return null;
    }

}

```

```
accountRepository   AccountRepository
repository   AccountRepository
```

Field name auto-completion

B.5. Generating Constructors From Fields


*RewardNetworkImpl.java
Toggle Comment ⌘7
Remove Block Comment ⇧⌘\
Generate Element Comment ⌥⌘J
Correct Indentation ⌘I
Format ⇧⌘F
Add Import ⇧⌘M
Organize Imports ⇧⌘O
Sort Members...
Clean Up...
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate hashCode() and equals()...
Generate Constructor using Fields...
Generate Constructors from Superclass...
Externalize Strings...

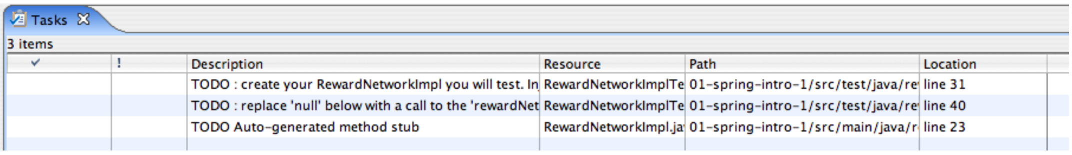
```
package rewards.internal;

import rewards.Dining;

public class RewardNetworkImpl implements RewardNetwork {

    private AccountRepository accountRepository;

    private RestaurantRepository restaurantRepository;

    private RewardRepository rewardRepository;



                                              or(Dining dining) {




}
```

"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 ☒

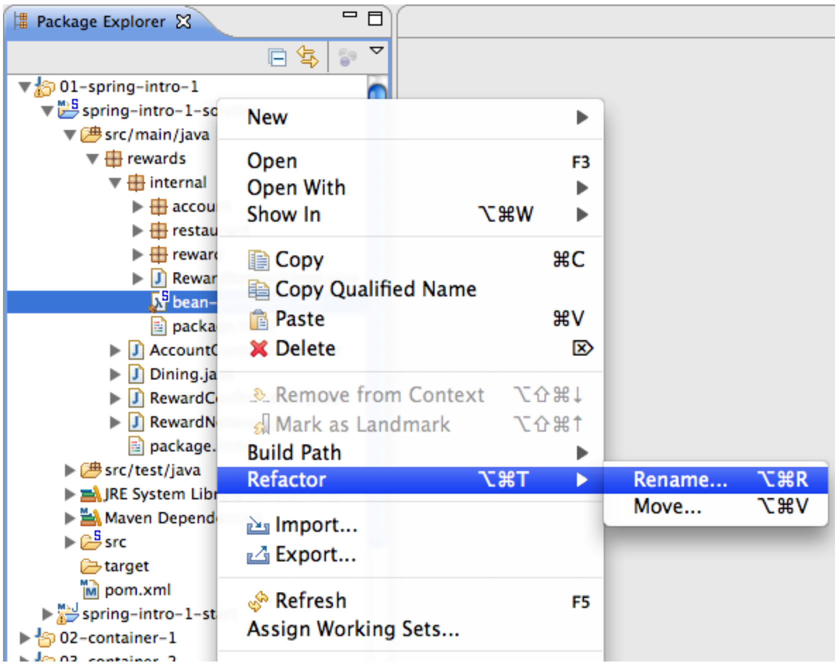
3 items

✓	!	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	RewardNetworkImpl.ja	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 [http://www.eclipse.org/webtools/] plugin for Eclipse using Tomcat 6.0 for testing web application labs and samples.

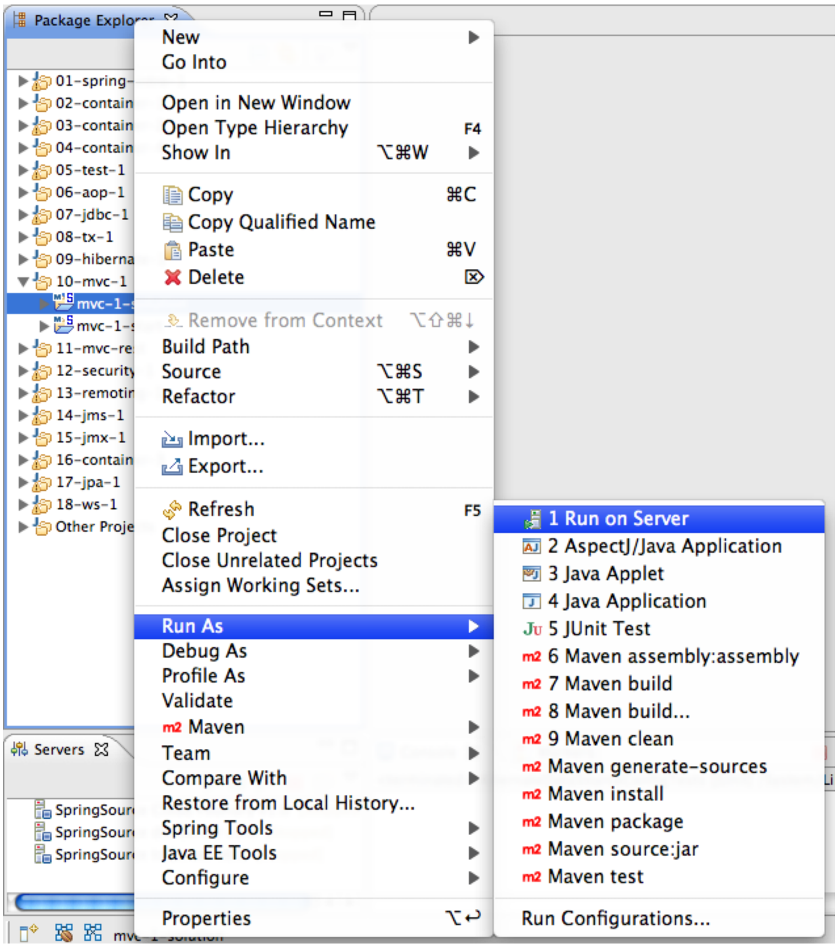
C.2. Verify and/or Install the Tomcat Server

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 SpringSource icon in the toolbar at below the main menus. This will launch the SpringSource ToolSuite 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

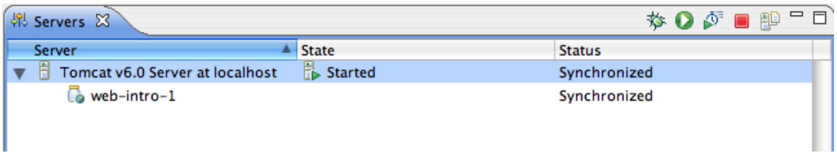
The easiest way to deploy and run an application using WTP is to right-click on the project and select '*Run As*' then '*Run On Server*'.



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.



Server	State	Status
Tomcat v6.0 Server at localhost	Started	Synchronized
web-intro-1		Synchronized

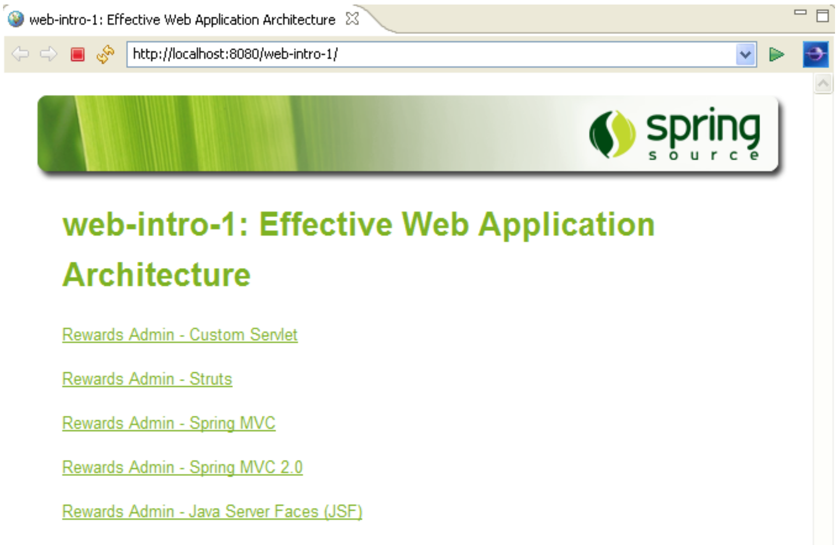
Running on Server

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.


web-intro-1: Effective Web Application Architecture
http://localhost:8080/web-intro-1/
spring source

web-intro-1: Effective Web Application Architecture

Rewards Admin - Custom Servlet

Rewards Admin - Struts

Rewards Admin - Spring MVC

Rewards Admin - Spring MVC 2.0

Rewards Admin - Java Server Faces (JSF)

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*'.