Core Spring

Lab Instructions

Building Enterprise Applications using Spring

Version 4.3.b

spring Pivotal

Copyright Notice

This Page Intentionally Left Blank

This Page Intentionally Left Blank

Table of Contents

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.

Papa Keith dines at a restaurant in the reward network

Figure 1. Keith and friend

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.

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

Figure 2. Annabelle's college fund

3. Reward Dining Domain Applications

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

3.1. The Rewards Application

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

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

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


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

3.1.1. Public Application Interface

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

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

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

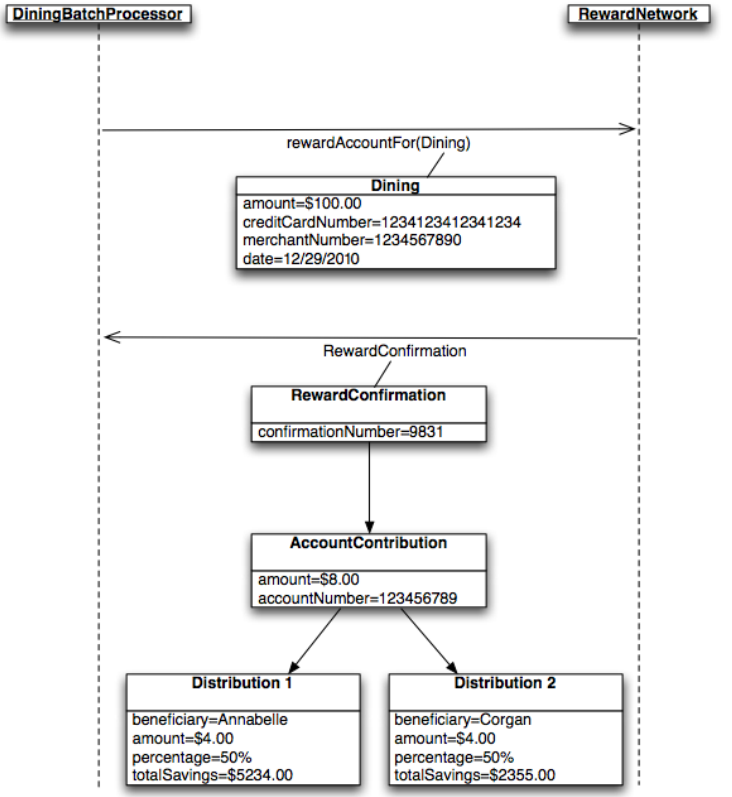


Figure 3. A client calling the `RewardNetwork` to reward an account for dining

In this example, the account with credit card 1234123412341234 is rewarded for a $100.00 dining at restaurant 1234567890 that took place on 12/29/2010. The confirmed reward 9831 takes the form of an $8.00 account contribution distributed evenly among beneficiaries Annabelle and her brother Corgan.

3.1.2. Internal Application implementation

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

This flow is shown below:

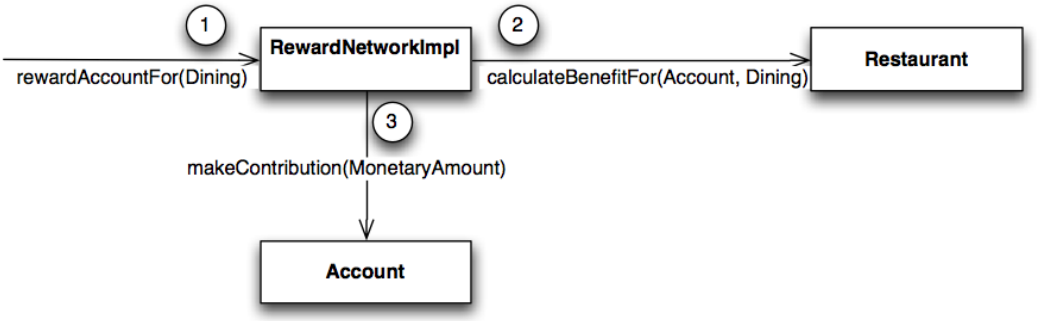


Figure 4. Objects working together to carry out the `rewardAccountFor(Dining)` use case

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

3.1.3. Supporting RewardNetworkImpl Services

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

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

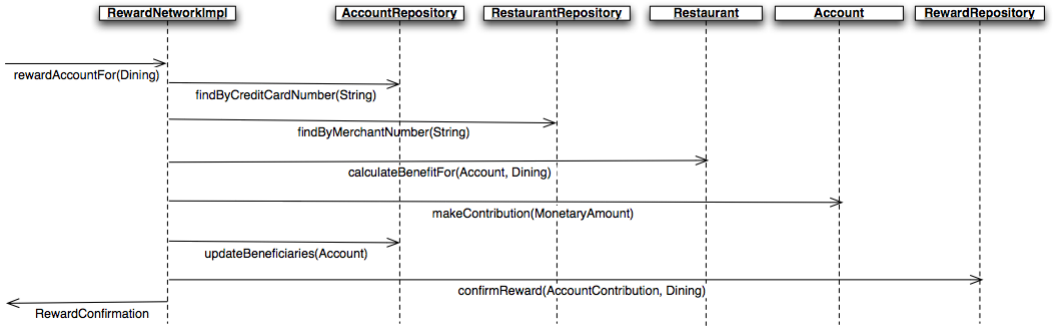


Figure 5. The complete `RewardNetworkImpl rewardAccountForDining(Dining)` sequence

4. Reward Dining Database Schema

The Reward Dining applications share a database with this schema:

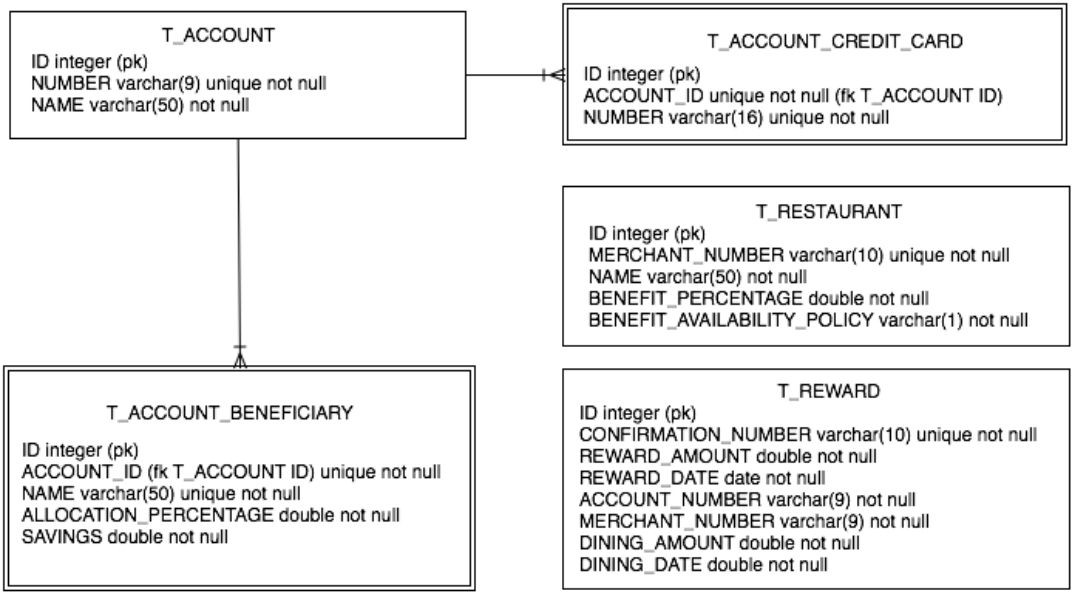


Figure 6. The Reward Dining database schema

Chapter 1. spring-intro: 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!*

Note

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

What you will learn:

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

Estimated time to complete: 30 minutes

1.2. Instructions

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

1.2.1. Getting Started with the Spring Tool Suite

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

1.2.1.1. Launch the Tool Suite

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

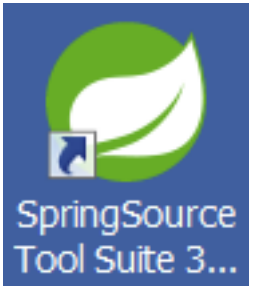


Figure 1.1. STS Desktop Icon

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

Figure 1.2. STS Splash Image

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

1.2.1.2. Understanding the Eclipse/STS project structure

Tip

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

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

1.2.1.3. Browse Working Sets and projects

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

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

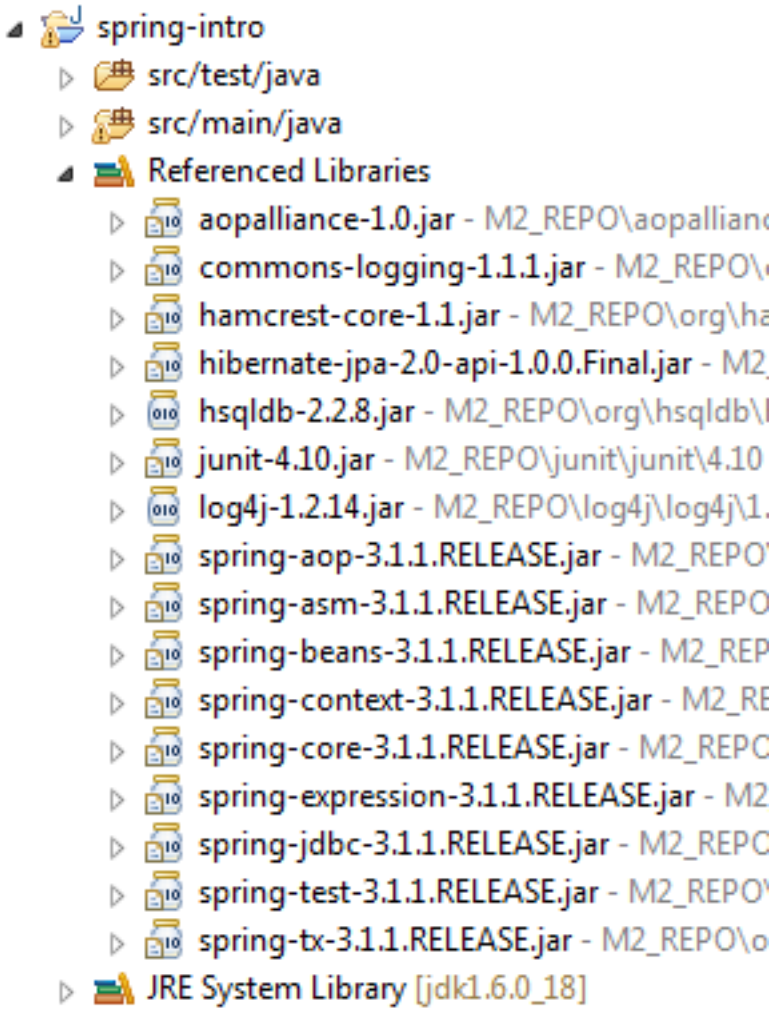


Figure 1.3. Referenced Libraries

Tip

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

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

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

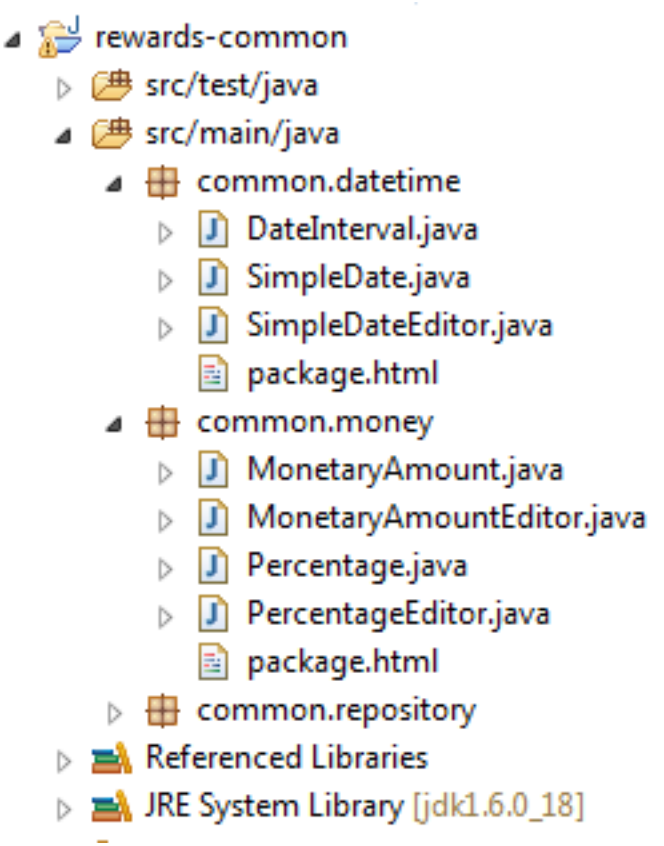


Figure 1.4. *rewards*-common common components

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

1.2.1.4. Configure the TODOs in STS

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

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

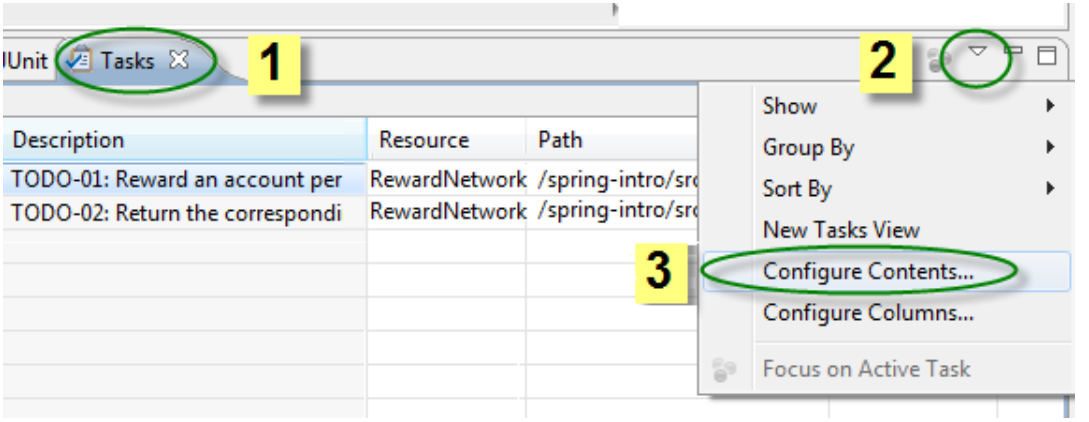


Figure 1.5. Configure TODOs

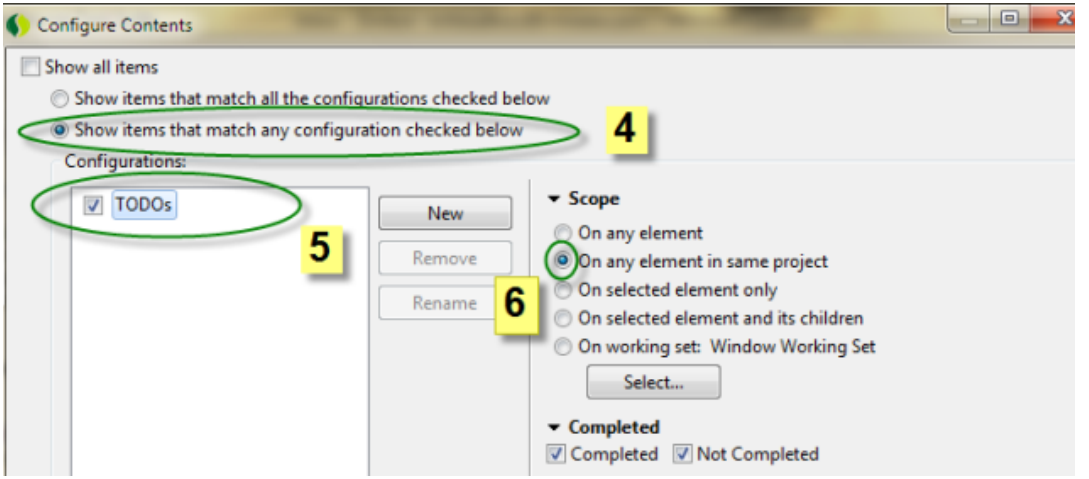


Figure 1.6. Configure TODOs

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

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

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

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

1.2.2.1. Review the `RewardNetwork` implementation class

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

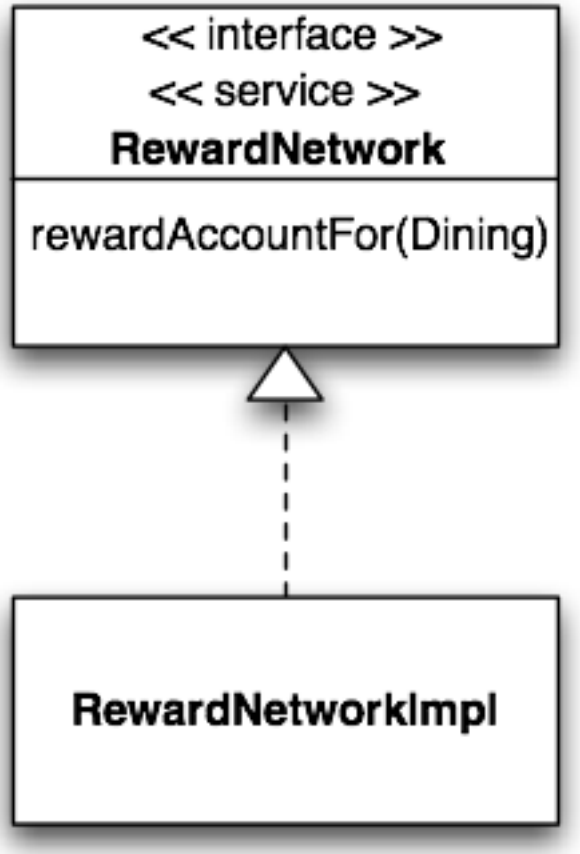


Figure 1.7. `RewardNetworkImpl` implements the `RewardNetwork` interface

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

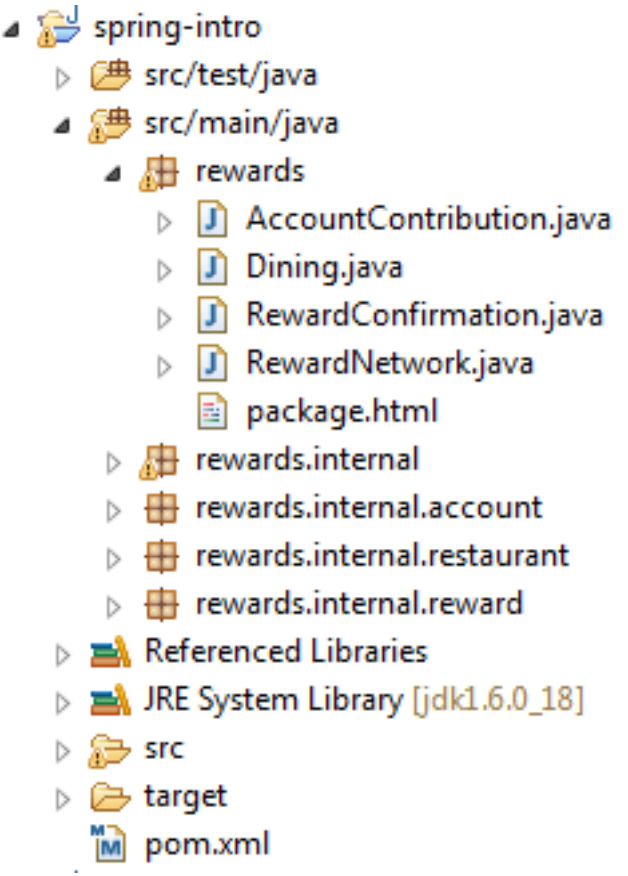


Figure 1.8. The `rewards` package

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

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

1.2.2.2. Review the `RewardNetworkImpl` configuration logic

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

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

This relationship is shown graphically below:

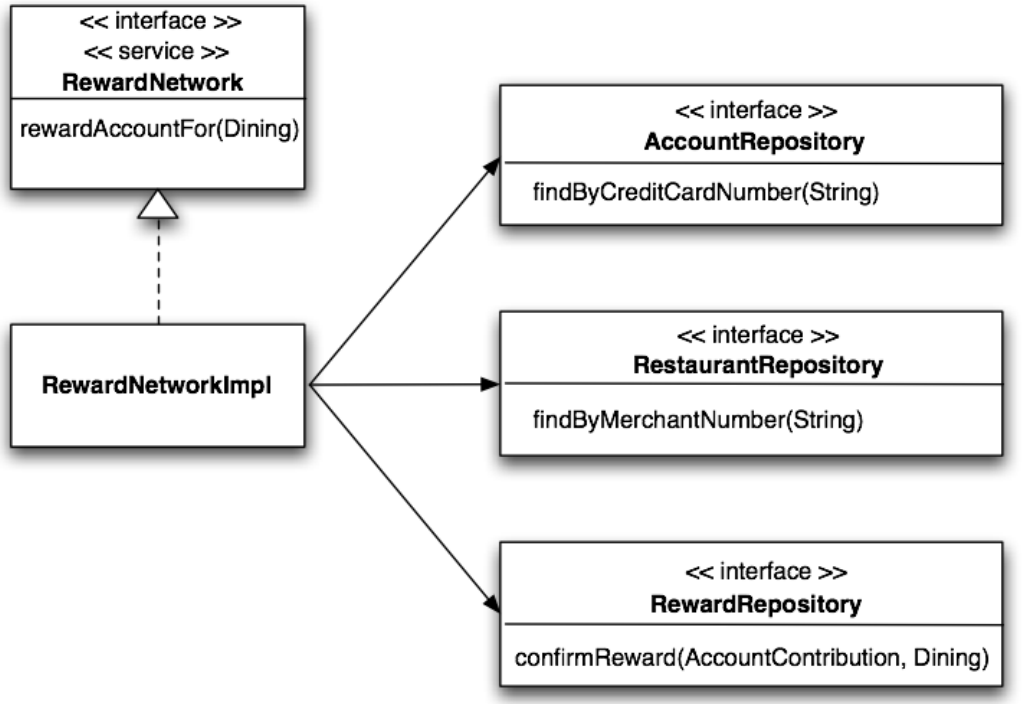


Figure 1.9. `RewardNetworkImpl` class diagram

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

1.2.2.3. Implement the `RewardNetworkImpl` application logic

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

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

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

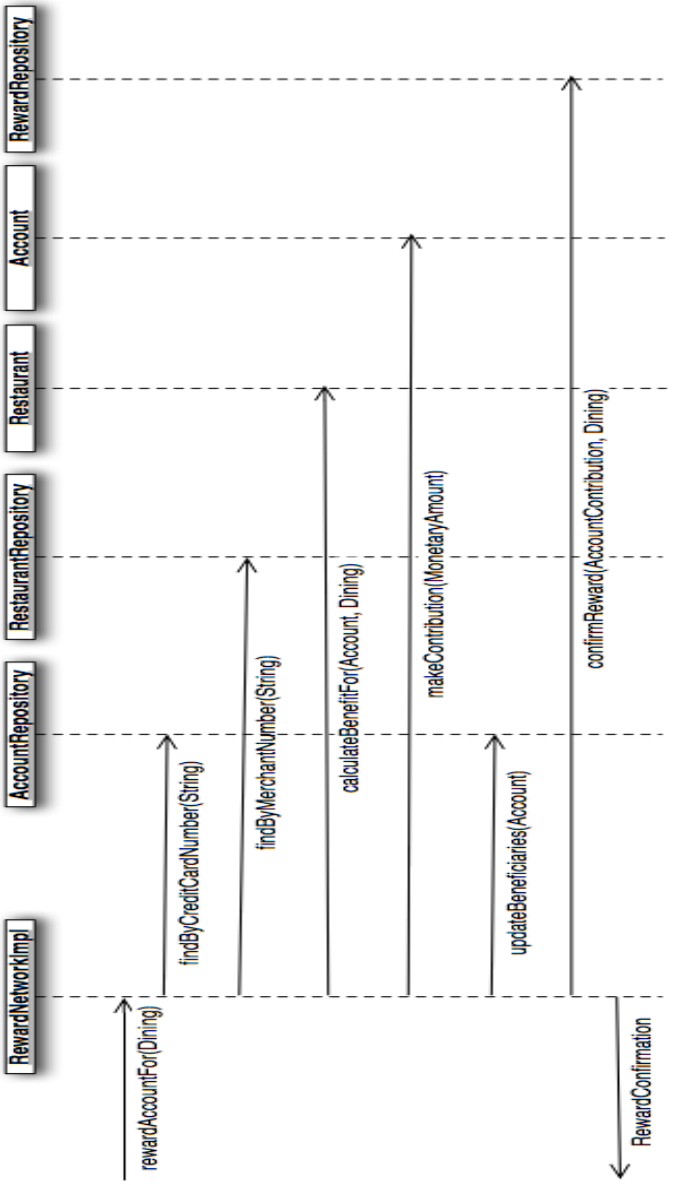


Figure 1.10. The `RewardNetworkImplrewardAccountFor(Dining)` sequence

Tip

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

Tip

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

Tip

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

1.2.2.4. Unit test the `RewardNetworkImpl` application logic

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

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

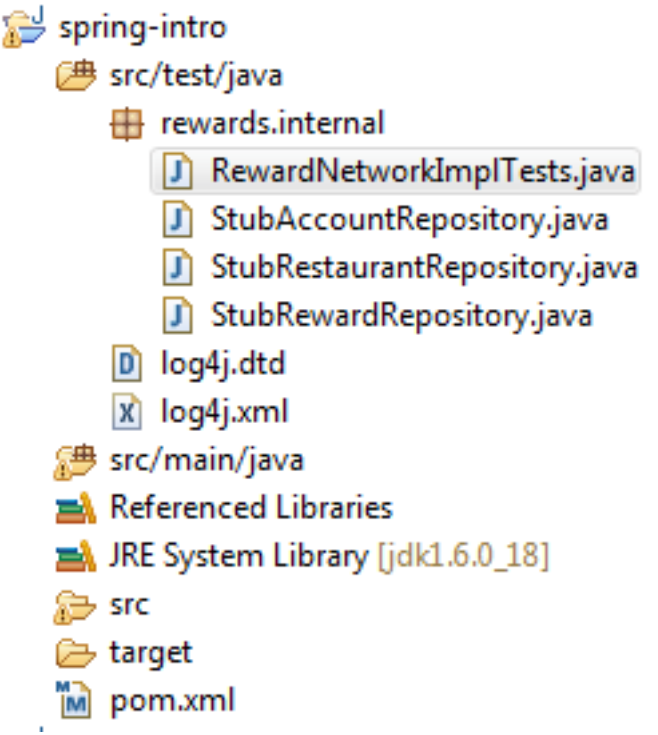


Figure 1.11. The `rewards` test tree

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

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

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

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

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

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

Chapter 2. javaconfig-dependency-injection: Dependency Injection with Java Configuration

2.1. Introduction

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

What you will learn:

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

Specific subjects you will gain experience with:

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

Estimated time to complete: 45 minutes

2.2. Quick Instructions

If you feel you have a good understanding of the material, follow the steps listed here. However, if you would like more detailed guidance, the next section contains more detailed step-by-step instructions.

If you aren't sure, try the quick instructions first and refer to the detailed instructions if you need more help. Each quick-instruction has a link to the corresponding detailed instructions.

Create the Application Configuration

1. **Creating application configuration file** (details)
   Go to the `config` package. Select new / class, name the file `RewardsConfig` and click Finish. Note that the

class does not need to extend any other classes or implement any interfaces.

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

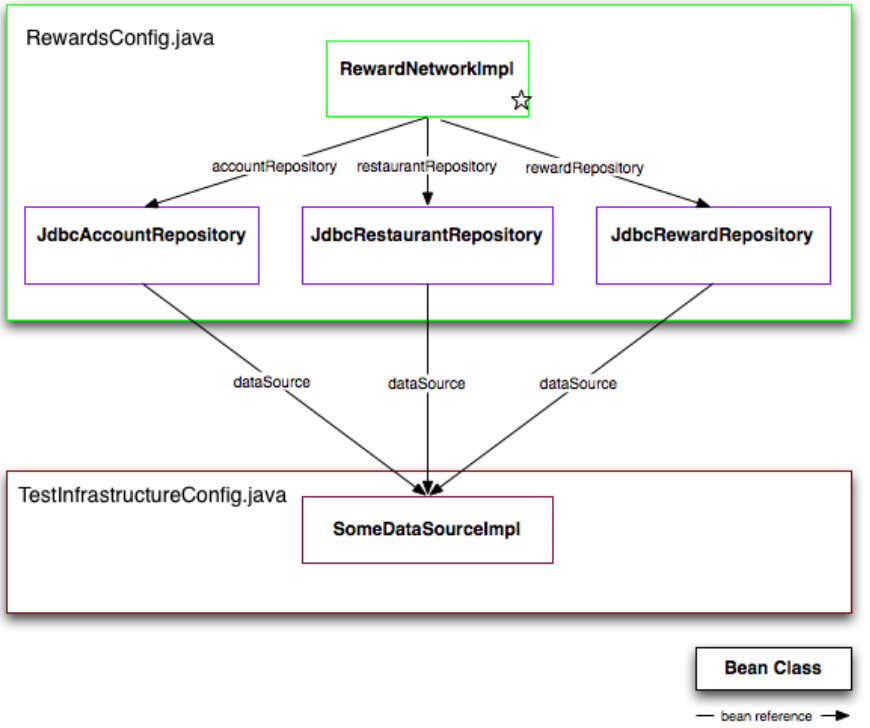


Figure 2.1. Application configuration

2.

Infrastructure configuration (details)

Next review the infrastructure configuration necessary to test your application. We need a datasource for your application to use to acquire database connections in a test environment. Open

`TestInfrastructureConfig.java` file and verify the datasource and database connection code.
This `TestInfrastructureConfig.java` will also serve as the master configuration class for our upcoming test. To have it serve in this role, add an `@Import` to the class to reference your new `RewardsConfig.class`.

System testing the application with Spring and JUnit

1. **Create the system test class** (details)
   Create a new JUnit test called `RewardNetworkTests` in the `rewards` package inside the `src/test/java` source folder.

2. **Implement test setup logic** (details)
   Add the test setup logic required to run your system test. In this method, you will need to create an application context using the master configuration class from the last step (`TestInfrastructureConfig.java`). In addition, you need to obtain a reference to the the *rewardNetwork* bean from the application context. Assign this as a private field that you can reference in your test methods.

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

4. **Run the test**
   With the test setup logic implemented, you're ready to test your application. Run your new unit test. It will invoke `RewardNetwork.rewardAccountFor(Dining)` method to verify all pieces of your application work together to carry out a successful reward operation. You should get a green bar in the JUnit view.
   Congratulations the lab is finished.

2.3. Detailed Instructions

2.3.1. Creating the application configuration

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

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

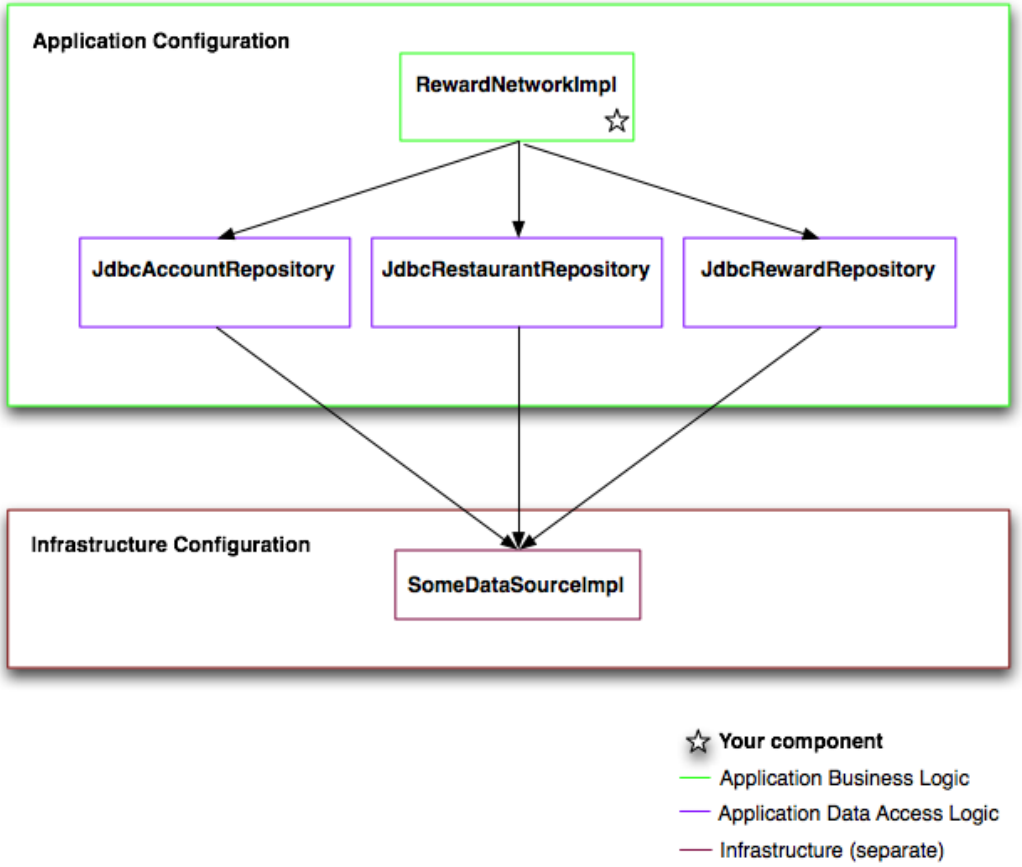


Figure 2.2. The rewards application configuration diagram

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

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

2.3.1.1. Create the application configuration class

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

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

Next we will add code to the `RewardsConfig` class to create the result illustrated in the '`RewardsConfig`' box below:

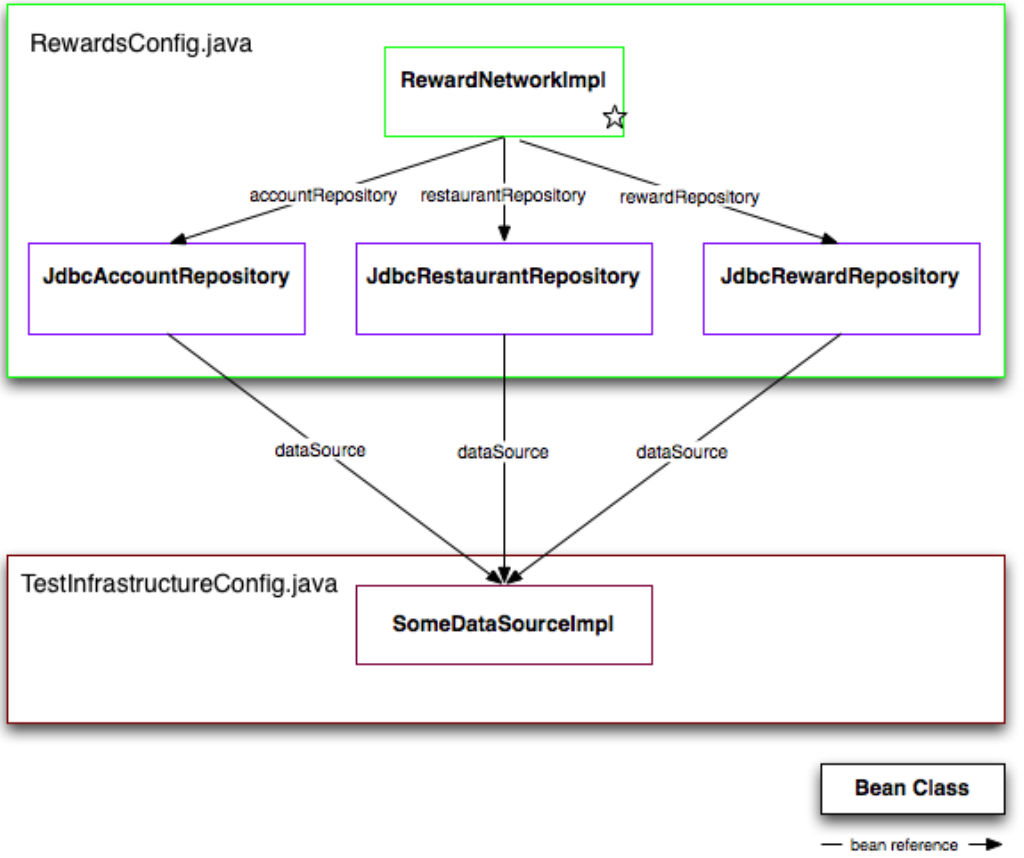


Figure 2.3. Application configuration

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

Tip

When typing in any classname in STS, use `CTRL+Space` before completing the full name, this will cause the IDE to prompt you for possible completions AND correctly add the relevant import statement. To add import statements after the fact, you can use `CTRL+SHIFT+O` (upper-case letter O) to *Organize Imports* (use `COMMAND+SHIFT+O` on MacOS).

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

Looking back at the illustration, you can see that each of the three repositories has a dependency on a `DataSource` that will be defined elsewhere. This means in each repository method we must make a call to the repository's `setDataSource()`, passing in a reference to the `dataSource`. But where will we get the `DataSource` from when it is defined in another file (in our case in `TestInfrastructureConfig.java`)?

Spring is able to populate fields on our configuration class with references to beans defined elsewhere. To do this, create a field (instance variable / member variable) on the class of type `DataSource` named `dataSource` and mark it with the `@Autowired` annotation. When Spring sees this, it will automatically populate this field with a reference to the DataSource defined in `TestInfrastructureConfig.java`, assuming both configuration files are specified at startup.

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

Follow bean naming conventions

As you define each bean, follow bean naming conventions. The arrows in the diagram representing bean references follow the recommended naming convention.
For best practices, a bean's name should describe the *service* it provides. It should not describe implementation details. For this reason, a bean's name often corresponds to its *interface*. For example, the class `JdbcAccountRepository` implements the `AccountRepository` interface. This interface is what callers work with. By convention, then, the bean name should be `accountRepository`.

Use Eclipse auto-completion

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

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

2.3.1.2. The infrastructure configuration needed to test the application

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

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

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

You will see that a dataSource is already configured for you. You don't need to make any changes to this bean, but you do need to understand what we have defined here for you. This `TestInfrastructureConfig.java` will also serve as the master configuration class for our upcoming test. To have it serve in this role, add an `@Import` to the class to reference the `RewardsConfig.class`.

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

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

2.3.2. System testing the application with Spring and JUnit

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

2.3.2.1. Create the system test class

Start by creating a new JUnit Test Case called `RewardNetworkTests` in the `rewards` package inside the `src/test/java` source folder. Use the *New -> Other -> Java -> JUnit Test Case* wizard to help you. Check the box next to "setUp" to automatically create a setup method, you will need this on the next step. Also note that you might need to change the version of JUnit that will be used to 4:

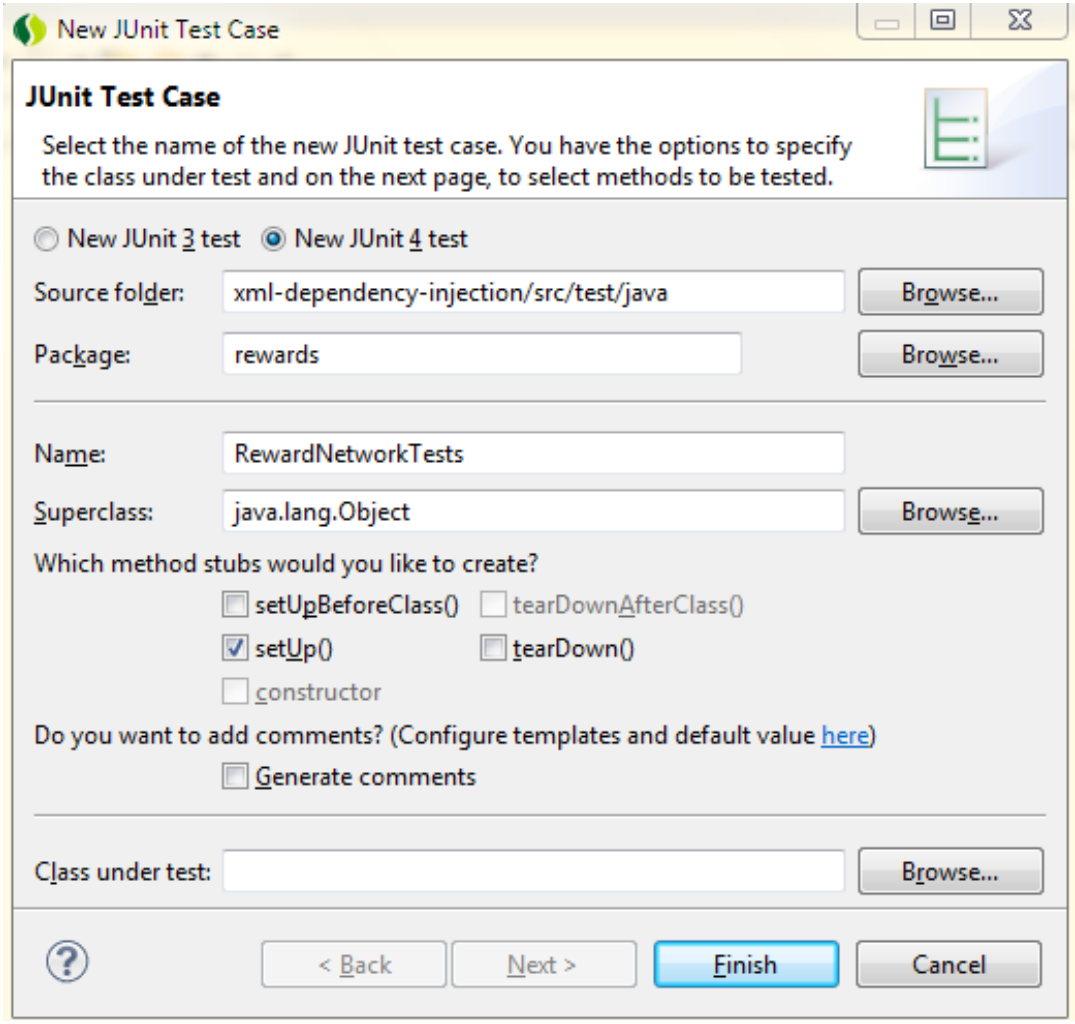


Figure 2.4. Creating the RewardNetworkTests TestCase using the JUnit Test Case wizard

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

2.3.2.2. Implement the test setup logic

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

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

Within `setUp()`, call `SpringApplication.run`, providing it the `TestInfrastructureConfig.class` that you want to load. Doing this will bootstrap your application by having Spring create, configure, and assemble all beans defined in the two configuration files (since one imports the other).

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. The most useful information is usually at the *bottom* of the stack trace, so you my need to scroll down to see it.

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

2.3.2.3. Implement the test logic

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

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

Tip

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

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

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. annotations: Configuration with Annotations

3.1. Introduction

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

What you will learn:

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

Specific subjects you will gain experience with:

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

Estimated time to complete: 25 minutes

3.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)). Use the view's small down arrow to select a `Configure Contents...` menu, you'll find the instructions are easy to follow if you configure TODOs to display `On any element in the same project`.

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

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

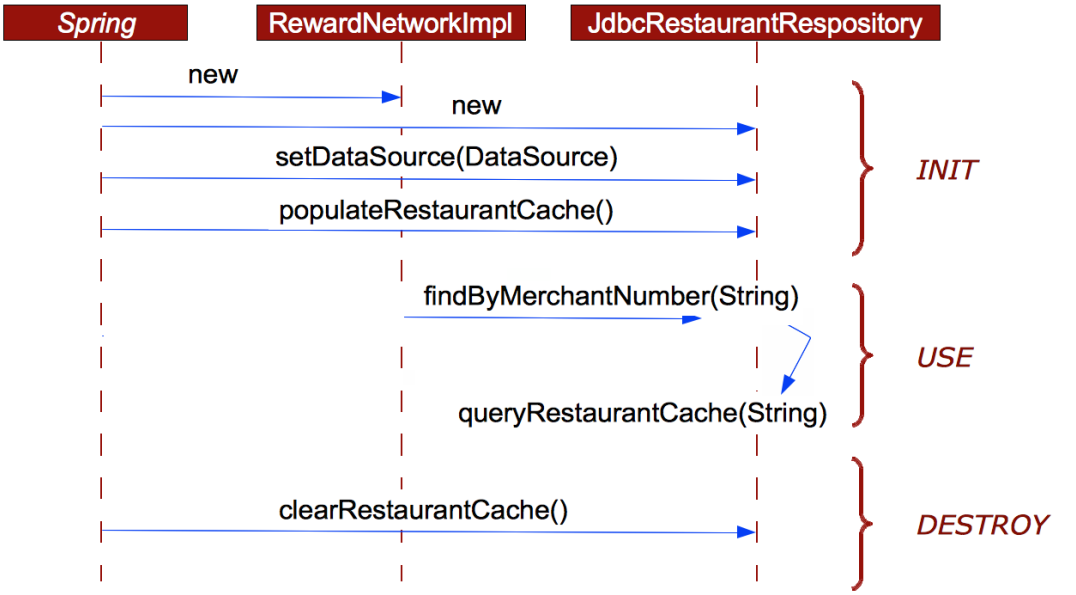


Figure 3.1. Lifecycle of the `JdbcRestaurantRepository`

3.3. Detailed Instructions

3.3.1. Reviewing the application

In this lab, we are using a version of the `rewards` application that is already fully functional. Essentially, it is the completed version of the last exercise. It has repository implementations that are backed by JDBC and which connect to an in-memory embedded HSQLDB database. We will then rewrite some of the application code to make use of annotations.

3.3.1.1. First verify that everything works

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

As...' followed by 'JUnit Test'. The test should run successfully.

Now open the application configuration called `RewardsConfig.java` (use `CTRL+SHIFT+T` [Windows/Linux] or `COMMAND+SHIFT+T` [Mac] to popup the "Open Type" search dialog and quickly navigate to it). Review the @Beans that wires up all the dependencies. As you can see, we're using constructor arguments.

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

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

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

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

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

(TODO 04) Now open `JdbcRewardRepository` and annotate it with a stereotype annotation. Since it is a repository class, the @Repository annotation is the obvious choice here. Mark the `setDataSource()` method with that same `@Autowired` annotation. This will tell Spring to inject the setter with a instance of a bean matching the `DataSource` type. You could use field-level injection instead if you prefer, the application will work the same either way.

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

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

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

called component scanning which looks for all classes annotated with annotations such as `@Component`, `@Repository` or `@Service` and creates Spring beans from those classes. It also enables detection of the dependency injection annotations. The "rewards" argument is the base package that we want Spring to look from, this will keep Spring from unnecessarily scanning all org.* and com.* packages on the classpath.

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

3.3.3. Working with Init and Destroy callbacks

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

The cache works as follows:

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

For convenience, the full sequence is shown again below.

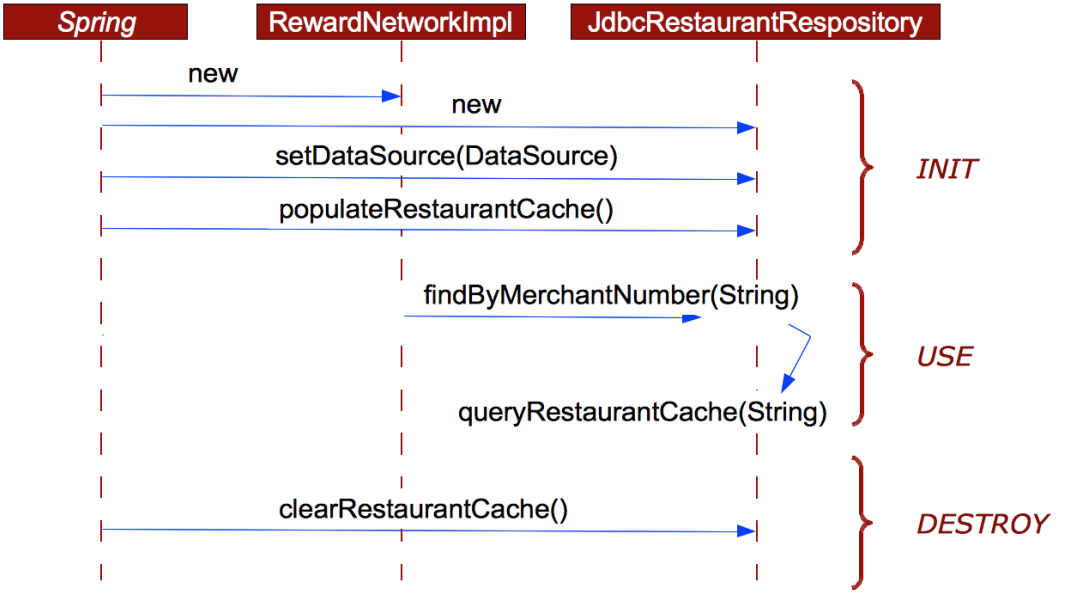


Figure 3.2. The `JdbcRestaurantRepository` life-cycle

3.3.3.1. Initialization

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

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

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

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

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

3.3.3.2. Destroy

(TODO-10) Your test seems to run fine, let us now have a closer look. Open `JdbcRestaurantRepository` and add a breakpoint inside `clearRestaurantCache`. Re-run `RewardNetworkTests` in debug mode. As you can see, the method `clearRestaurantCache` is not called, which means that your cache is never cleared. Add an annotation to mark this method to be called on shutdown.

Save your work and run `RewardNetworkTests` in debug mode one more time. You should then stop into the breakpoint.

Tip

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

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

Chapter 4. xml-di: XML Dependency Injection

4.1. Introduction

What you will learn:

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

Estimated time to complete: 30 minutes

4.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)). Use the view's small down arrow to select a `Configure Contents...` menu, you'll find the instructions are easy to follow if you configure TODOs to display `On any element in the same project`.

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

4.3. Detailed Instructions

4.3.1. First verify that everything works

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

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

4.3.2. Convert to XML configuration / Component Scanning

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

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

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

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

(TODO 04) At this point, we have an XML configuration file equivalent of our `RewardsConfig.java` class, and we also have a `test-infrastructure-config.xml` that is already prepared for us. Our test class is still coded to load the `TestInfrastructureConfig.java` Java configuration class. We need to change this configuration class to import our XML files.

Open `TestInfrastructureConfig.java`. Remove/comment out the existing `@Import` annotation and the entire `dataSource()` bean method. Instead add an `@ImportResource` annotation, and pass it an array of Strings indicating the XML configuration files to load. The first is the file you just modified, `rewards-config.xml` located in the `src/main/resources/config` folder. The second is an existing XML file already prepared for you, `test-infrastructure-config.xml` located in the `src/test/resources/rewards` folder. Note that both of these are classpath resources, so `@ImportResource` should find these as long as we indicate the correct folder locations.

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

4.3.3. Switch to XML-based Configuration

At this point, we are using Annotation-based configuration (via component scanning) to define the application

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3.4. Bonus - Remove Component Scanning

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

Chapter 5. test: Integration Testing with Profiles

5.1. Introduction

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

What you will learn:

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

Specific subjects you will gain experience with:

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

Estimated time to complete: 30 minutes

5.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)). Use the view's small down arrow to select a `Configure Contents...` menu, you'll find the instructions are easy to follow if you configure TODOs to display `On any element in the same project`.

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

5.3. Detailed Instructions

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

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

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

Tip

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

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

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

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

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

Note

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

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

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

5.3.2. Configure Repository Implementations using Profiles

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

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

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

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

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

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

5.3.3. Switching between Development and Production Profiles

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

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

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

Does `RewardNetworkTests` still run OK? Why not?

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

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

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

5.3.4. Optional Step - Further Refactoring

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

This is an example of convention over configuration. Does the test still run?

Note

When you copy the `TestInfrastructureConfig` class into `RewardNetworkTests`, remember to make it `static` - refer to example in notes if unsure.

Chapter 6. aop: Introducing Aspect Oriented Programming

6.1. Introduction

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

What you will learn:

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

Estimated time to complete: 35 minutes

Important Note 1: The JUnit tests you will run in this lab *already work*. Just getting a green test does not indicate success. *You must also get logging messages in the console*.

Important Note 2: Students often find this one of the hardest labs. If you get totally stuck *please ask a colleague or your instructor* - don't waste the whole lab trying to fix your first pointcut expression.

6.2. Quick Instructions

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

6.3. Detailed Instructions

6.3.1. Creating and Testing a simple Aspect (`@Before` advice)

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

6.3.1.1. Check Console Output

The tests we use in this lab already run successfully. Adding an aspect to a class does not necessarily change what that class does, often it just extends its functionality. In this lab we will add extra logging, so not only must tests pass, there should be the expected logging on the console.

Capturing output to the console means diverting `System.out` and `System.err` and saving all output to an internal buffer. Fortunately Spring Boot provides a convenient class to do this, `OutputCapture`, which can be used independently of the rest of Spring Boot. All the tests already have output-capture enabled, but currently output testing is turned off (until you write some aspects, there is no logging to check).

(TODO-01) Open `TestConstants` and change the CHECK_CONSOLE_OUTPUT boolean to `true`.

6.3.1.2. Create an Aspect

REQURIEMENT #1: Create a simple logging aspect for repository find methods.

In this section, you will first define the logging behavior, then the rules about where the behavior should be applied. You'll use the annotated `@Aspect` definition style.

(TODO-02) The definition of the aspect has already been started for you. Find it in the `rewards.internal.aspects` package.

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

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

(TODO-03) We aren't interested in logging *every* method of your application, though, only a subset. At this stage, you're only interested in logging the `find*` methods in your *repositories*, the objects responsible for data access in the application.

Try to define a pointcut expression that matches all the `find*` methods, such as `findByCreditCard()`, in the `AccountRepository`, `RestaurantRepository`, or `RewardRepository` interfaces.

Tip

If you get stuck - refer to the pointcut examples in the slides. Or try writing a pointcut expression that just matches `find*` methods, similar to the setter method example in the slides.

You can make it more specific later.

You will need a `@Before` advice on the `implLogging()` method which has already been implemented for you. It takes a `JoinPoint` object as a parameter, and logs information about the target objects invoked during the application execution.

6.3.1.3. Configure Spring to weave the aspect into the application

(TODO-04) Now that your aspect has been defined, you will create the Spring configuration needed to weave it into your application.

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

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

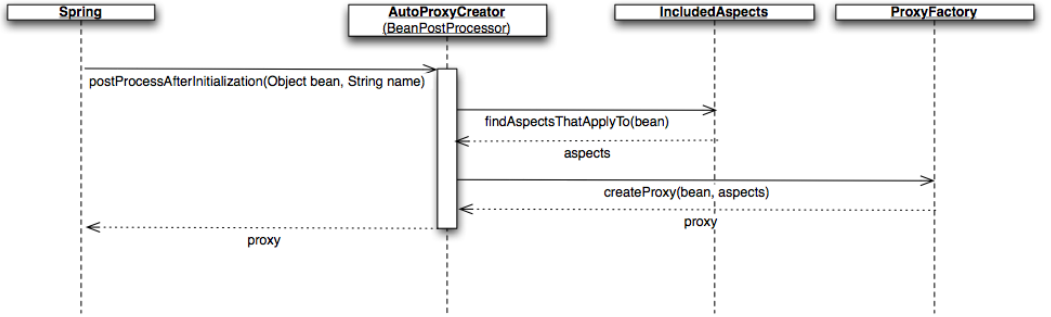


Figure 6.1. Spring's auto proxy creator weaving an aspect into the application using the proxy pattern

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

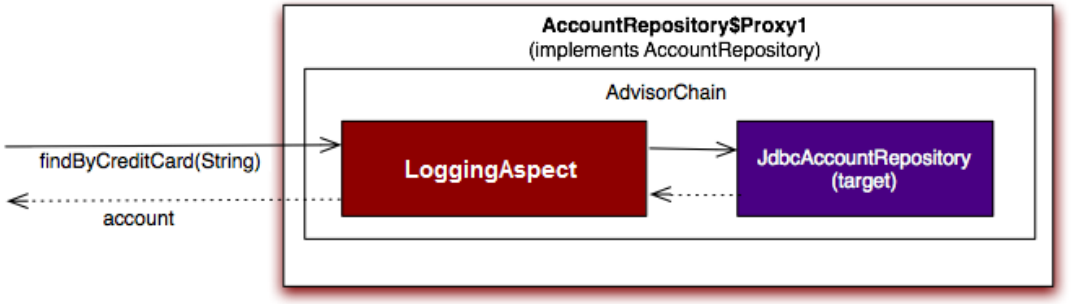


Figure 6.2. A proxy that applies logging behaviour to a `JdbcAccountRepository`

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

6.3.1.4. Test the Aspect Implementation

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

After the configuration file has been added, (TODO-06) run `LoggingAspectTest` in STS and watch the console. You should see:

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

Tip

If you don't see any console output your pointcut expression is most probably wrong. Refer back to the notes for TODO-03 and see if you can fix it. *Please ask a colleague or your instructor* if you really can't work out the problem. Don't waste the entire lab trying to fix your first pointcut expression.

When you see the logging output, your aspect is being applied.

If you wrote a very general pointcut expression earlier, as suggested by these notes (just `find*` methods), try to make it more specific to match `find*` methods on `*Repository` classes.

Once everything is working, move on to the next step!

6.3.2. Performance Monitor Aspect

REQURIEMENT #2: Implement a `Around` Advice which logs the time spent in each of your *update* repository methods.

- Return to the `LoggingAspect` class, and examine the `monitor(ProceedingJoinPoint)` method. Most of the method has been implemented for you. You will complete it now.
- (TODO-07) Specify `@Around` advice for the `monitor` method. Define a pointcut expression that matches all the `update*` methods (such as `JdbcAccountRepository.updateBeneficiaries(...)` on the `AccountRepository`, `RestaurantRepository`, or `RewardRepository` interfaces.
  Again there is a HINT in the TODO Text if you are stuck.
- (TODO-08) Now in `monitor(ProceedingJoinPoint)` method, notice the Monitor start and stop logic has already been written for you. What has not been written is the logic to proceed with the target method invocation after the watch is started. Complete this step by adding the `proceed` call.

Tip

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

- (TODO-09) Once you've added the proceed call, modify the `expectedMatches` in `RewardNetworkTests` class because now there should be 4 lines of logging output not 2. If the test passes and you can see relevant logging information in the console, your monitoring behavior has been implemented correctly.
  Again, *please ask a colleague or your instructor* if you can't get the test to pass.

6.3.3. OPTIONAL: Exception Handling Aspect

Create an exception handling aspect as follows:

- (TODO-10) Modify the `DBExceptionHandlingAspect` class by annotating the method `implExceptionHandling(Exception e)` to be used in the event of an exception. Which type of advice will you need?
- Add the advice annotation to this method and define a pointcut expression that matches all the methods in any of the three repositories (regardless of the method names).
- (TODO-11) Although this class is presently marked as an @Aspect, it isn't defined as a @Component, and therefore it is not picked up when component scanning. Change this by simply adding a @Component

anotation to the top of the class.

(TODO-12) After the configuration has been added, run `DBExceptionHandlingAspectTests` in Eclipse and watch the console. If you can see relevant logging information in the console, your exception handling behavior has been implemented correctly.

Congratulations, you've completed the lab!

Chapter 7. jdbc: JDBC Simplification using the JdbcTemplate

7.1. 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

7.2. Quick Instructions

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

7.3. Detailed Instructions

7.3.1. 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.

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

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

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

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

7.3.1.2. Refactor `JdbcRewardRepository` to use `JdbcTemplate`

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

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

Tip

The `Object...` means a variable argument list allowing you to append an arbitrary number of arguments to a method invocation, including no arguments at all.

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.

7.3.2. Using a `RowMapper` to create complex objects

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

(TODO-04) In many cases, you'll want to return complex objects from calls to the database. To do this you'll

need to tell the `JdbcTemplate` how to map a single `ResultSet` row to an object. In this step, you'll refactor `JdbcRestaurantRepository` using a `RowMapper` to create `Restaurant` objects.

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

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

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

```
private class RestaurantRowMapper implements RowMapper<Restaurant> {

    public Restaurant mapRow(ResultSet rs, int rowNum) throws SQLException {
```

Figure 7.1. `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.

Note

If you prefer to use a lambda instead please do so. Have it delegate to the existing `mapRestaurant(ResultSet)` method.

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

7.3.3. OPTIONAL STEP: Refactoring the `JdbcAccountRepository`

In this repository there are two different methods that need to be refactored: `updateBeneficiaries(Account)` and `findByCreditCard(String)`.

Only do this section if you have enough time left. You will need 10-15 mins.

7.3.3.1. Refactoring a SQL UPDATE

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

(TODO-05) 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`.

(TODO-06) 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`.

When you are done, rerun the `JdbcAccountRepositoryTests` tests. When you have the green bar, you are good.

7.3.3.2. EXTRA CREDIT: Use a `ResultSetExtractor` to traverse a `ResultSet` for creating `Account` objects

This is an optional step if you have at least 10 minutes of the lab remaining.

(TODO-07) 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 `findByCreditCard(String)` using a `ResultSetExtractor` to create `Account` objects.

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 existing `mapAccount(ResultSet)` method.

Note

If you prefer to use a lambda instead please do so. Have it delegate to the existing `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` tests once again. 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 8. tx: Transaction Management with Spring

8.1. Introduction

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

What you will learn:

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

Specific subjects you will gain experience with:

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

Estimated time to complete: 25 minutes

8.2. Quick Instructions

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

8.3. Detailed Instructions

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.

8.3.1. Demarcating Transactional Boundaries in the Application

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

8.3.1.1. Add `@Transactional` annotation

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

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

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

8.3.1.2. Verify transactional behavior

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

```
DEBUG: o.s.j.d.DataSourceTransactionManager - Creating new transaction with name [rewards.internal.RewardNetworkImpl.rew
    ardAccountFor]: ...
DEBUG: o.s.j.d.DataSourceTransactionManager - Acquired Connection [org.hsqldb.jdbc.JDBCConnection@176b75f7] for JDBC
    transaction
DEBUG: o.s.j.d.DataSourceTransactionManager - Switching JDBC Connection [org.hsqldb.jdbc.JDBCConnection@176b75f7] to
    manual commit
DEBUG: o.s.j.d.DataSourceTransactionManager - Initiating transaction commit
DEBUG: o.s.j.d.DataSourceTransactionManager - Committing JDBC transaction on Connection [org.hsqldb.jdbc.JDBCConnection@
    176b75f7]
DEBUG: o.s.j.d.DataSourceTransactionManager - Releasing JDBC Connection [org.hsqldb.jdbc.JDBCConnection@176b75f7] after
    transaction
```

Note

If you look in `setup()` of `RewardNetworkTests` you will see that we have enabled DEBUG logging for the `DataSourceTransactionManager`.

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.

8.3.2. Configuring Spring's Declarative Transaction Management

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

8.3.2.1. Modify Propagation Behavior

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

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

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

8.3.3. Developing Transactional Tests

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

8.3.3.1. Use `@Transactional` to isolate test cases

First (TODO-06) back out your propagation changes from the previous section (change the propagation back to `Propagation.REQUIRED` instead of `Propagation.REQUIRES_NEW`). This is the appropriate propagation setting for this method.

(TODO-07) Find and open `RewardNetworkSideEffectTests` from the `rewards` package in the `src/test/java`

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

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

Congratulations, you're done with the lab!

Chapter 9. jpa-spring-data: JPA Simplification using Spring Data

9.1. Introduction

In this lab you will get an introduction into JPA so that you can utilize Spring Data's automatic repositories.

What you will learn:

1. How to configure domain objects with annotations to map these to relational structures
2. How to use Spring Data JPA to dramatically reduce the amount of persistence code
3. How to test JPA-based repositories

Specific subjects you will gain experience with:

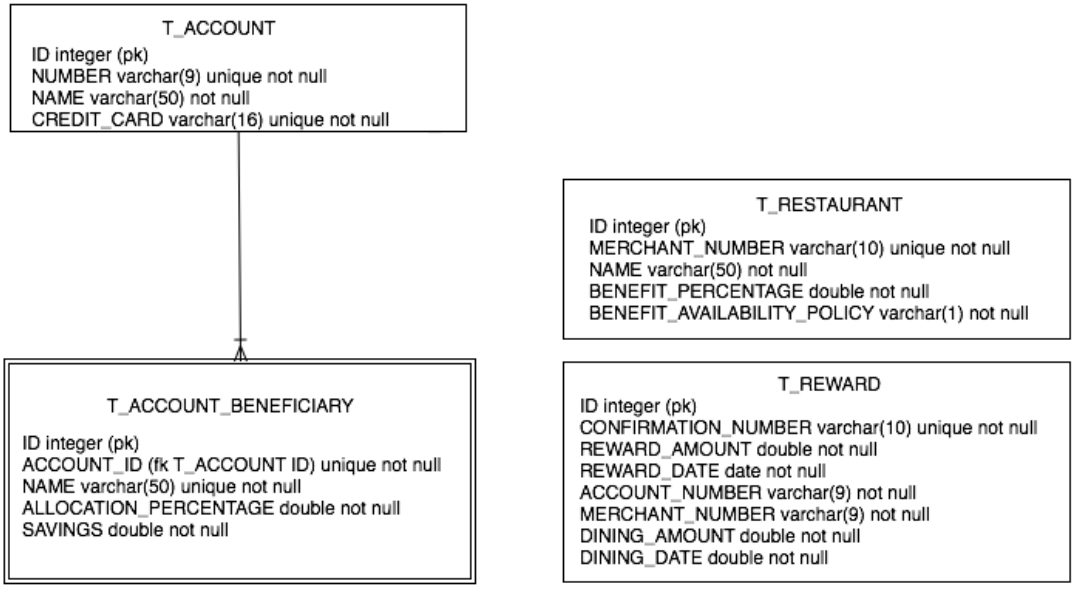
1. JPA Annotations
2. Spring Data JPA

Estimated time to complete: 50 minutes

9.2. Quick Instructions

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

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



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

Figure 9.1. Rewards Database Schema

9.3. Detailed Instructions

9.3.1. Review the `Account` and `Beneficiary` 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 review the JPA mapping annotations that map the Account object graph to the database.

1. (TODO-01) Open `Account` class in the `rewards.internal.account` package. As you can see this class is already annotated with several JPA annotations. Let's walk through these annotations one at a time to understand what each one is doing. Later in this lab, you will need to annotate a class on your own, so this is a good chance to look at a good object-to-relational mapping example.

2. Note the `@Entity` annotation on this class. This specifies 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`). Note the `@Table` annotation is used to override the default - in this case our database table is `T_ACCOUNT`.

3. Every entity needs a primary key so it can be managed by JPA. Every table in this lab uses auto-generated numeric keys. A long integer `entityId` field has been added to each classes to be mapped. As you can see, it is annotated with `@Id`, this means that the database will treat the matching column as the table's primary key.

4. Although you don't see it here, whenever we are creating new database rows we need to tell JPA how the primary key values are to be generated. For this the `@GeneratedValue` annotation is used to describe the strategy. In our case, we will never insert new values into the database so this annotation is not needed, so to keep things simple we've omitted it.

5. By default JPA uses the field name as the column name to map a field into a database table column. In our case, some of the field names don't match the column names, so we are overriding the mapping by using `@Column`.

6. Since an `Account` can have many beneficiaries, its `beneficiaries` property is a collection. In the database, this relationship is modeled by a one-to-many relationship with a BENEFICIARY table. To describe this relationship to JPA, the `@OneToMany` annotation is used. The foreign key column in the beneficiary table is `ACCOUNT_ID`, and we tell JPA about this foreign key using `@JoinColumn`.

Of course, there is a lot to JPA that we are glossing over - such as all of the options that we can express on the various annotations. JPA is a deep subject and we are only covering the basics here. Once you feel you have the basic idea down, move on to the next step.

(TODO-02) Open the `Beneficiary` class and observe its JPA annotations.

1. The `@Entity`, `@Table`, `@Id`, and `@Column` annotations are used exactly the same way as in the previous class.

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

3. 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 `@AttributeOverride`
   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. We must map the field name `value` to the column for `savings` and `allocationPercentage` respectively.

9.3.2. Configure the `Restaurant` mapping

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

1. (TODO-03) Open the `Restaurant` in the `rewards.internal.restaurant` package. We will configure all object-to-relational mapping rules using annotations inside this class.

2. Like the `Account` module, we need to mark this class as an entity, define its table and define its `entityId` field as an auto-generated primary key. Don't forget to use a `Column` annotation to specify the target column in the database for this field.

3. Complete the mapping for the remaining Restaurant fields: `number`, `name` and `benefitPercentage`. Like the `Beneficiary` mapping, the percentage is a custom type and needs mapping differently. Use the schema in Figure 1 to help you.

Tip

You will need to use the `@AttributeOverride` annotation again.
There is no need to map the `benefitAvailabilityPolicy` - it has been done for you.

When you have completed the Restaurant mapping, we have all of the domain classes annotated. In the next steps, we will implement some code to query the database using JPA.

9.3.3. Implement `AccountRepository` using Spring Data JPA

Now that we have our ORM mapping data defined, we can write some code that uses a JPA EntityManager to query the persistent objects in various ways. However, most of the typical queries are fairly predictable, and we can use Spring Data JPA to automatically implement most of our repository code for us.

(TODO-04) Open `AccountRepository`. As you can see, this is an ordinary Java interface (POJI). Alter this interface to extend the Spring Data JPA `Repository` interface. This interface is known to the Spring Data JPA framework, and it can use it to automatically implement a number of useful methods for us.

`Repository` is a typed interface, so we need to describe the data types that the implementation class is intended to work on. The first type is the Entity class that this repository is intended to operate on: `Account`. The second is the type of the ID column that we setup on the Account. If you re-open the Account class you will see that the data type of the `@Id` field is a basic java.lang.`Long`, so simply indicate `Long` as the second type.

Next, we need to define a method signature to be used whenever we wish to look up an Account using a credit card number. The Spring Data JPA framework follows a simple convention that we can employ to have it automatically implement this logic: the method syntax is `findBy<DataMember><Op>`, where "Op" can be Gt, Lt,

Ne, Between, Like, etc. For the equals case, "op" can be omitted, so the resulting method name would be `findByCreditCardNumber`. The method should take a String parameter and return an `Account`.

That's all the code we need to write! Spring Data JPA will automatically implement all the methods defined in the Repository interface, plus the method we defined using a naming convention. Because we have written 0 lines of executable Java code, we can bypass the traditional unit test and go straight to integration / system testing, which we will do in a later step. Of course there are many more options here, and we can even combine Spring Data JPA automatic repositories with our own.

9.3.4. Implement `RestaurantRepository` using Spring Data JPA

(TODO-05) Open `RestaurantRepository`. Alter this interface to extend the Spring Data JPA `Repository` interface. `Repository` is a typed interface, so we need to describe the data types that the implementation class is intended to work on (`Restaurant`) and the type of the ID column (`Long`).

Next, define a method signature to be used whenever we wish to look up an Restaurant using a merchant number. Following the method naming convention, and observing that `Restaurant` stores the merchant number in a data member called `number`, the resulting method name would be `findByNumber`. The method should take a String parameter and return a `Restaurant`.

9.3.5. Enable Spring Data JPA Automatic Repositories

The interfaces we have just embellished do nothing by themselves of course. In a traditional application we would have to write a Java class that implements them, then write the JPA code for the queries. With Spring Data JPA the implementations will be made automatically at runtime, we just need to "activate" the framework to tell it to do this.

(TODO-06) Open a configuration class such as `RewardsConfig` and add the `@EnableJpaRepositories` annotation to the class. Within this annotation, you should specify the base package under which all of the `Repository` methods can be found.

With this annotation in place, Spring Data JPA will automatically implement the required repository logic when the ApplicationContext is first loaded. Next, we need to define some beans for working with JPA, specifically the EntityManageFactoryBean and a JpaTransactionManager

1. (TODO-07) Open `SystemTestConfig.java`. In this file, setup the Entity Manager Factory. There are three steps.

    a. Define a bean method to create the `EntityManagerFactory`. The factory bean's class is `LocalContainerEntityManagerFactoryBean`. You can name the bean whatever you like, though `entityManagerFactory` is probably the most descriptive.

b. Set the `dataSource` and `jpaVendorAdapter` properties appropriately. The `jpaVendorAdapter` tells Spring which JPA implementation will be used to create an `EntityManagerFactory` instance. Use the class `HibernateJpaVendorAdapter` to define the `jpaVendorAdapter` property.

c. You can set additional JPA implementation specific configuration properties by setting the `jpaProperties` property. During development it is very useful to have Hibernate output the SQL that it is passing to the database. The two properties to pass in are `hibernate.show_sql=true` to output the SQL and `hibernate.format_sql=true` to make it readable. Note that in some cases a property may be available in the general `jpaProperties` AND in the specific `HibernateJpaVendorAdapter`.

2. (TODO-08) Finally, define a `transactionManager` bean so the Reward Network can drive transactions using JPA APIs. Use the `JpaTransactionManager` implementation. Note that this class requires an `entityManagerFactory`, so we will need to obtain one from the previously defined bean. Fortunately, Spring makes this very easy; simply define the bean method with a parameter of type `EntityManagerFactory`. When Spring calls this method at startup time, it will find a reference by calling the `getObject()` method on the `LocalContainerEntityManagerFactoryBean` defined earlier. By letting Spring handle this (instead of calling the method directly), you are allowing it to take care of any lifecycle requirements.

9.3.6. Adjust Application Code and Test

(TODO-09) Open the `RewardNetworkImpl` class. Notice that the `rewardAccountFor(Dining dining)` method has several lines of code commented out - these lines could not compile until our interfaces were completed. Now that they are, uncomment these lines, and remove the line at the end of the method which returns null. The code should compile at this point, if it does not take a good look at the method names and ensure that your earlier repository work is correct.

Next, we should be able to run a test and verify all your work. Open the `RewardNetworkTests` class, remove the `@Ignore` annotation on the `testRewardForDining` method, and run the test. It should pass.

If you have a successfully running test, congratulations! You have annotated domain objects with JPA annotations, setup automatic repositories, defined beans for the EntityManagerFactory, and enabled Spring Data JPA correctly.

Chapter 10. mvc: Spring MVC Essentials

10.1. 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

10.2. Setting up a Tomcat Server

There should be a Servers folder in STS on the left hand side and it should be open. It should contain a Tomcat server ready to run. If there is no such folder, or it is empty, you will need to setup a new server.

If there is no Tomcat server defined, please tell your instructor now. Your instructor can take the class through creating one for the lab. Or see Appendix for details.

10.3. Quick Instructions

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

10.4. Detailed Instructions

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

10.4.1. Setting up the Spring MVC infrastructure

Spring MVC is a comprehensive web application development framework. Let's see how we have used it to implement the current system.

10.4.1.1. The Requirement

The web application we 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 below:

Figure 10.1. GET /accounts/accountList

You should see a listing of all accounts by name with links to view details. Once implemented, clicking on a link will take you to the account details page as shown below (however this is not yet implemented).

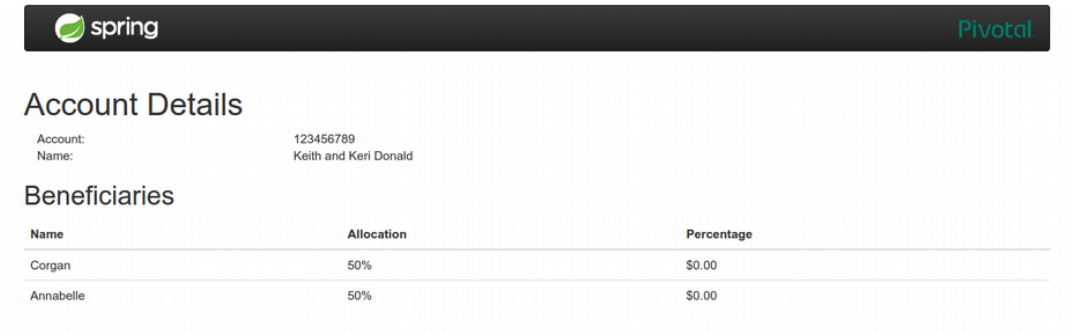


Figure 10.2. Show Details for Account '0'

10.4.1.2. Running the application

In this first step you will assess the initial state of the web application.

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

Note

If your application deploys successfully there should be a lot of logging output in the console window and the images on the page should appear correctly. If you have problems, try the following before asking for help. Stop the server. Right-click on the server in the `Servers` panel and select `Clean ...`. Now right-click again on the server and select `Clean Work Directory ...`. Try running the server again.

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

10.4.1.3. Review the application configuration

Quickly assess the initial configuration of the "backend" of this web application. To do this, open `WebInitializer` in the `config` package. Notice that the `root configuration` has already been defined to bootstrap your application-layer from `RootConfig` class. Open this file to see the beans that make up this layer. You will see that it simply imports other configurations and establishes transaction management, all with just a few annotations.

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

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

10.4.1.4. 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 made to 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.

Return to `WebInitializer` and notice `MvcConfig` is listed as the servlet config class. Also notice that a `/accounts/*` path is mapped to it.

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

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. This list is added to the model for display to the user. Finally it requests the `accountList.jsp` view to render the list by returning its location as a `String` telling the DispatcherServlet to use this view 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. We fix that next.

10.4.1.5. Reviewing the whole system

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

1. The request was first received by the Servlet Engine, which routed it to the DispatcherServlet.
2. The DispatcherServlet then invoked the `accountList()` method on the `AccountController` based on the `@RequestMapping` annotation.
3. Next, the AccountController loaded the account list and selected the "accountList.jsp" view.
4. Finally, the accountList.jsp rendered the response which you see before you.

At this point you should have a good feel for how everything works so far. You should also have an idea of how to 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 soon.

But first let's get rid of those absolute paths to views.

10.4.2. Add a View Resolver

The view name in our handler method uses an absolute path. this means the method is aware of the specific type of view that will be rendered (JSPs in this case). Spring recommends decoupling request handling from response rendering details. In this step, you will add a `ViewResolver` to provide a level of indirection.

(TODO-02) Navigate to the `MvcConfig` class and add a bean definition of type `InternalResourceViewResolver`. This will override the default `ViewResolver` and enable the use of logical view names within the Controller code. You should now specify two properties on the view resolver bean definition: `prefix` and `suffix`. Review the current view names to determine these values.

Tip

The `DispatcherServlet` automatically recognizes any bean definitions of type `ViewResolver`. Therefore, you can use any method name for your @Bean method (recall that the method name defines the bean name).

Now refactor the existing controller (TODO-03) and test (TODO-04) so that only a simple view name such as `accountList` is used. The `AccountControllerTests` should now pass.

Now (TODO-05) restart the web application. If you are still able to view the list of accounts, your changes are correct.

10.4.3. Implementing another Spring MVC handler method

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

10.4.3.1. Implement the account details request handler

(TODO-06) In the `AccountController`, add a method to handle requests for account details. The method should use the account identifier passed with the HTTP request to load the account, add it to the model, and then select a view. What name (key) will you use for the account attribute when you add it to the model?

Tip

In your web browser, try clicking on an account and look at the URL to see which parameter name is used to pass in the account identifier. Ignore the 404 error.

The JSP has already been implemented for you. Review it in the `WEB-INF/views` directory to see what attribute name it is expecting.

10.4.3.2. Testing the controller

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

(TODO-07) Open `AccountControllerTests` and review how the `accountList()` method has been tested. As you can see, it just calls the handler method (without having to worry about doing any web setup) 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 - in a similar way to `testHandleListRequest()`. There should be one attribute in the model. What is its name? What type is it? Get the attribute and confirm it contains the right data. Don't forget to annotate the new method with `@Test`.

Note

The ability to test Spring MVC Controllers out-of-the-container is a useful 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, carry on.

10.4.4. Running the application

Finally (TODO-08) try to run the web application again and make sure the functionality you implemented works. If it doesn't, try to find where you might have gone wrong and possibly talk to your instructor.

10.4.5. EXTRA CREDIT: Mock MVC Testing

If there is time left, you can try this optional section.

Spring's Mock MVC testing framework allows a JUnit test to drive Spring MVC as if it was running in a container - enabling more powerful testing that the simple `AccountControllerTests` you have been using.

Open the class `MockMvcTests` and follow the TODO steps. Most of the code has been written for you. The important part is to see if you understand how the tests work - we did not cover Mock MVC testing in the course notes.

For more information, refer to the *online documentation*.

Chapter 11. spring-boot: Creating a Web Application using Spring Boot

11.1. Introduction

In this lab you will gain experience with building a Spring Project using Spring Boot.

What you will learn:

1. How to develop a Spring MVC web application using Spring Boot
2. How to configure a Spring Boot project

Estimated time to complete: 30 minutes

URL for this project http://localhost:8080/spring-boot or http://localhost:8080/accountList, depending on specific instructions used.

11.2. General notes about this lab

This lab is different than all the other labs you have done so far. In this lab, you start with just a minimal Spring Boot application and build on it. There are no TODO items in the codes. Instead there is just one set of instructions - you are reading them!

Fortunately STS and Spring Boot will take care of most of the work for you. And to help you, there is a "cheat sheet" at the end of this section.

Note

STS has a built-in wizard to create new Spring Boot projects and *we already ran it for you* to create the `spring-boot` project you are using to do this lab.

11.3. Lab instructions

11.3.1. Inspect the Spring Boot project

The `spring-boot-start` project we provide looks like this:

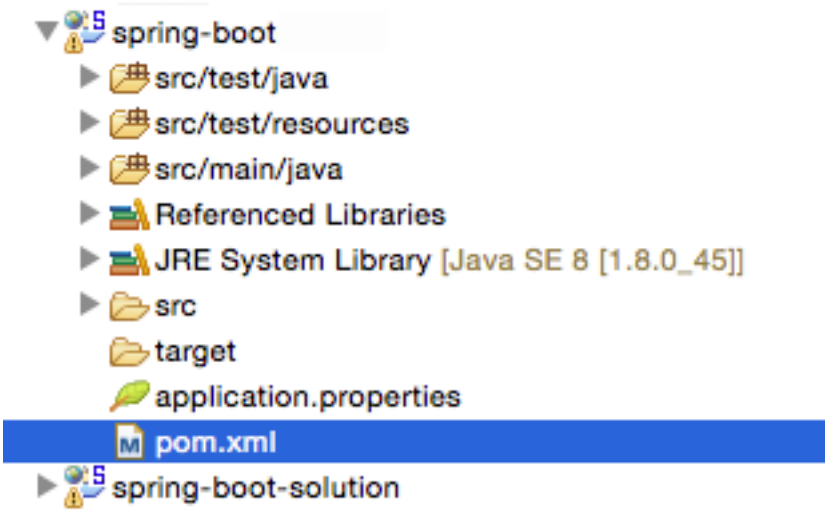


Figure 11.1. Minimal Spring Boot project

Let's have a look at what a simple Spring Boot project looks like:

1. The standard Java / Maven project folder structure.

2. A Maven build file. Boot works perfectly well with Gradle but this lab environment is established for Maven.
   Open the Maven build file (`pom.xml`) and click on the pom.xml tab on the bottom of the editor.
   Unlike a typical Spring Boot project, it does not have `spring-boot-starter-parent` as its parent, because all the labs in this course already inherit from our course parent POM. However you will find the Web starter, `spring-boot-starter-web`, as well as `spring-boot-starter-test` for testing.

3. A simple JUnit test class (`BootLabApplicationTests.java`)

4. A Bootstrap class (`BootLabApplication.java`)
   This class contains the main method needed to start a Spring Boot application.
   The single `@SpringBootApplication` annotation is a meta-annotation combining the `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` annotations.

11.3.2. Verify everything is working

You already have a working Spring Boot application now. You can test your project by executing the `BootLabApplicationTests` using JUnit. Run the test and make sure it passes.

11.3.3. Start the application

Your application is now ready to start. But first, make sure you don't have a server running from a previous lab. If you do, stop it; it will cause a conflict on port 8080 if it is still running.

Tip

When running a Spring Boot web-applications, if you get an error about the `Tomcat Connector in a failed state` it just means the port is already in use.

A Spring Boot web application can run in any web container compliant with the Servlet 3.0 API, but we will run our project as a standard Java application. The application can run from the IDE, we don't have to package it in any way, launching a `main` class will do the trick! Right-click on the `BootLabApplication` and select `Run As / Spring Boot Application`. Once running, open http://localhost:8080 in your browser. You should get a 404, as we haven't created anything yet.

Note

When running outside of a container, there is no need to specify the application's context in the servlet container, since there is only one application. This is why the URL is *not* "http://localhost:8080/spring-boot".

Note

Spring Boot packages the embedded Tomcat JARs to allow it to run without a container. If you never intend to run outside of a container, you can simply exclude these JARs using Maven/Gradle.

11.3.4. Create a simple "Hello" controller and web page

Now create a new Spring MVC controller in the package `accounts`. You can pick any name for your controller class, as long as it resides in the `accounts` package.

The controller should only contain one method which is mapped to `/hello` and should return the String "hello".

Tip

Click here if you need detailed hints to create this controller.

Next, we'll need a web page to display. Navigate to `src/main/webapp` and create a "WEB-INF/views" folder structure. Within the `views` folder create a file named "hello.jsp". Within this file, add the text "Hello World from a JSP page!" - we don't need a formal web page, just something to confirm that all the pieces are working together. Save your work.

Next, you may recall from Spring MVC that we typically need a `InternalResourceViewResolver` setup. Spring Boot will take care of this for us, but does not know what to use for prefix and suffix. Boot has a default property file called `application.properties` that we can use to configure this kind of thing very easily. Open this file (it's located at the root of the project directory) and add the following lines:

```
# Control the InternalResourceViewResolver:
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

Next, open the `pom.xml` file and check the following dependencies are set up:

```
<!-- These dependencies enable JSP usage -->
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
</dependency>
```

As mentioned in the comment, these dependencies enable JSP web applications, which is not straightforward when running outside of a container. Save your work, stop the application (big red button in the console view), and start it again. Open http://localhost:8080/hello. You should see `Hello World from a JSP page!`.

Tip

When running outside of a container, you can use the `Relaunch Application` icon in the STS Toolbar to restart your application.

Figure 11.2. Relaunch a Java application in STS

At this point we have a working web application. Next we will add the same infrastructure we used in previous labs to have a full-fledged application.

11.3.5. Add Database, Data, and Account Management

In this section, we will add a database, some data, and the AccountManagement object that we used in prior labs. Then we will copy the JSPs and controllers from the last lab and re-create the account listing system.

11.3.5.1. Add additional dependencies.

Open the `pom.xml` file and check the following dependencies are set up:

```
<!-- These dependencies enable JPA and an in-memory DB -->
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
</dependency>

<!-- These dependencies bring in the classroom files -->
<dependency>
  <groupId>io.pivotal.education.core-spring</groupId>
  <artifactId>rewards-common</artifactId>
</dependency>
<dependency>
  <groupId>io.pivotal.education.core-spring</groupId>
  <artifactId>rewards-db</artifactId>
</dependency>
```

1. `spring-boot-starter-data-jpa` - This is a "Starter" POM that defines a typical set of dependencies needed to work with JPA. This single dependency eliminates the need for us to define > 10 separate dependencies related to Spring ORM, JPA, and Hibernate.
2. `hsqldb` - This is the HSQL in-memory database. On startup, Spring Boot will notice this JAR, notice that we have not defined a dataSource, and will define one for us.
3. `rewards-common / rewards-db` - These contain the lab classes that we have been working with throughout this course. Account, Beneficiary, AccountManager, etc. are all defined here.

In order to work with our Account objects we will need to tell Spring Boot where our JPA annotated classes are. Add the `@EntityScan("rewards")` annotation to the `BootLabApplication` class definition.

Note

If our JPA annotated classes were located under the "accounts" package structure, we would not need to use `@EntityScan("rewards")`.

Next, there is one bean that we will need to define manually; our account manager. Add this bean definition to

the `BootLabApplication` class:

```
@Bean
public AccountManager accountManager(){
  return new JpaAccountManager();
}
```

This bean simply instantiates our AccountManager, the same one used in previous labs. One interesting thing to note here is that we do NOT need to define a dataSource, TransactionManager, or JPA EntityManagerFactory - Spring Boot will deduce that we need these, based on the fact that we have JDBC data source and JPA JARs on our classpath!

Save your work. Next, open `application.properties` again and add the following lines:

```
# Control how Boot loads data when it starts:
spring.jpa.hibernate.ddl-auto=none
spring.datasource.schema=classpath:/rewards/testdb/schema.sql
spring.datasource.data=classpath:/rewards/testdb/data.sql
```

1. spring.jpa.hibernate.ddl-auto - Instruction to Hibernate on what its policy should be regarding automatically creating database tables based on annotated classes. Ordinarily it tries to do this for us when using an in-memory database, so we are asking it not to.
2. spring.datasource.schema / data - Locations of any SQL files to be executed on application startup.

Tip

There are many quick settings that you can control via application.properties. See Spring Boot Reference / Appendix A. Common application properties for details.

Before going further, let's make sure the last few steps were done correctly. Save all your work and restart your application. Open http://localhost:8080/hello. You should still see `Hello World from a JSP page!`. If you do not, you probably made a mistake on one of the previous steps. Be sure the application starts before proceeding. In the next section, we will add the account listing pages.

11.3.5.2. Add Account Web Artifacts

Next we want to re-create the account listing pages from the MVC lab. Rather than create them from scratch we will simply copy most of this.

COPY the following files / folders from the `mvc-solution project`:

1. Copy the entire `accounts.web` package to our spring-boot-start project
2. Copy the entire contents of `webapp/WEB-INF/views` folder to the `webapp/WEB-INF/views` folder in

spring-boot-start project

Save all your work and Restart your application. Open http://localhost:8080/accountList. You should see the account listing page from the last lab, and the links to the detail pages should all work.

Congratulations, you have completed this lab! But before moving on, consider all of the items that you did NOT have to do, because Spring Boot did them for you:

1. You did NOT define a dataSource - Spring Boot noticed that you did not, and noticed HSQL on the classpath, and created one for you.
2. You did NOT define an EntityManagerFactoryBean, or a JpaVendorAdapter - Spring Boot noticed a JPA implementation on the classpath and set this up for you, using the dataSource.
3. You did NOT define a PlatformTransactionManager - Spring Boot assumed you would need one when working with JPA.
4. There is no web.xml file. Spring Boot is using a Servlet Initializer.
5. You did NOT define the DispatcherServlet, or a Servlet Mapping - Spring Boot noticed Spring MVC JARs on the classpath and defined this for you.
6. You did NOT define an InternalResourceViewResolver - Spring Boot did this for you, and allowed you to easily define the prefix and suffix.

11.3.6. Adding devtools

One of the biggest problems when dealing with Java application in general is that, every time we change a Java class (for example the Controller), we need to manually restart the JVM process (or redeploy to web container). In order to make the development experience more pleasant, we are introducing the Spring Boot Devtools.

Devtools will monitor the application's classpath, and reloads the application automatically.

Now we need to add a dependency to our `pom.xml` file. Check the following dependency is set up in the POM:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

Now start the Run the application as Spring Boot. Once it is started, navigate to the index page. So let's change the "HelloController" by changing the name of its method to "home":

```
@RequestMapping("/")
public String home() {
  return "hello";
}
```

When you save the changes, you should notice that STS output tab starts printing some new log entries. Now,

go refresh the index page, instead of the rewards app, we see the helloworld jsp!

For more info, please refer to the Devtools Reference

11.3.7. Bonus: Adding Actuator

Now let's use Spring Boot Actuator to get some runtime information. To enable the Actuator, we just need to add a dependency for it to our `pom.xml` file. Check the following dependency is set up in the POM:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

There are several Spring Boot Actuator URLs, such as:

- The Spring Beans: http://localhost:8080/beans
- Environment: http://localhost:8080/env
- Controller mappings: http://localhost:8080/mappings

For more details, refer to the Actuator section of the Spring Boot documentation.

Congratulations, you have completed the lab.

The next two sections are for reference and are not part of the lab.

11.4. Reference: Creating a New Spring Boot Project

There are a couple of convenient ways to creare a Spring Boot project described below which you might find useful when you return to your workplace.

Warning

This section is for reference, *you do not need to do this as part of the lab*. Any project you create this way now will not compile because STS is in offline mode and you won't have all the dependencies you need.

In STS you can create a new Spring Boot project using `File -> New -> Spring Starter Project`. If `Spring Starter Project` is not there, use `File -> New -> Other -> Spring -> Spring Starter Project` instead:

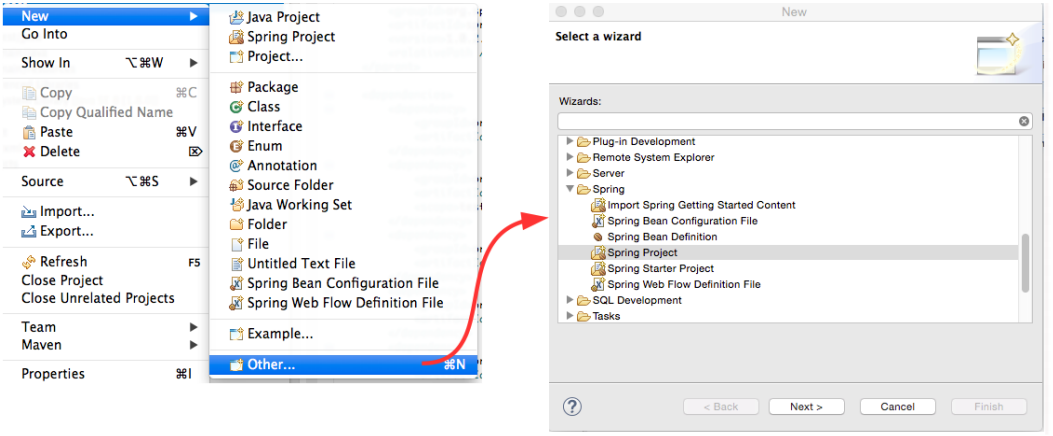


Figure 11.3. New starter project

Either runs a wizard that allows you to select the Spring modules that you want (such as Web, REST, Security, JPA and so on) and generates a customized project. We already did this for you to create the `spring-boot-offline` project used in the lab.

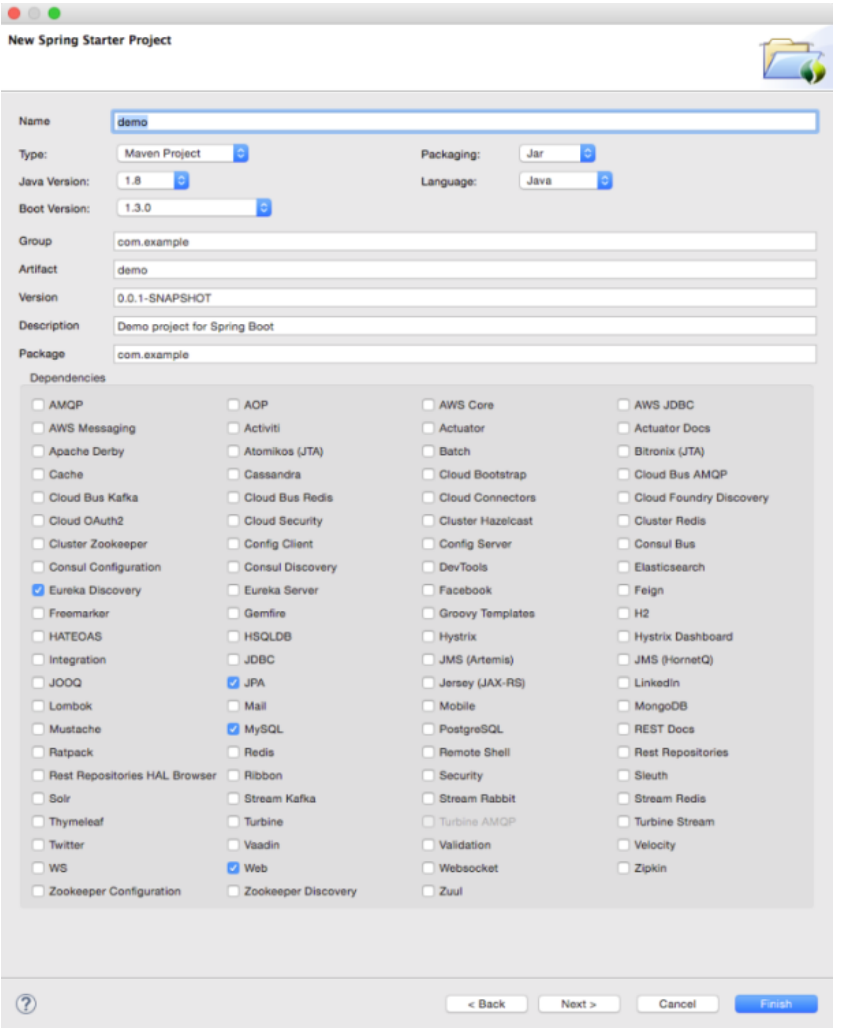


Figure 11.4. Define project and select Spring modules

If you don't normally use STS, you can do the same thing by going online to https://start.spring.io/:

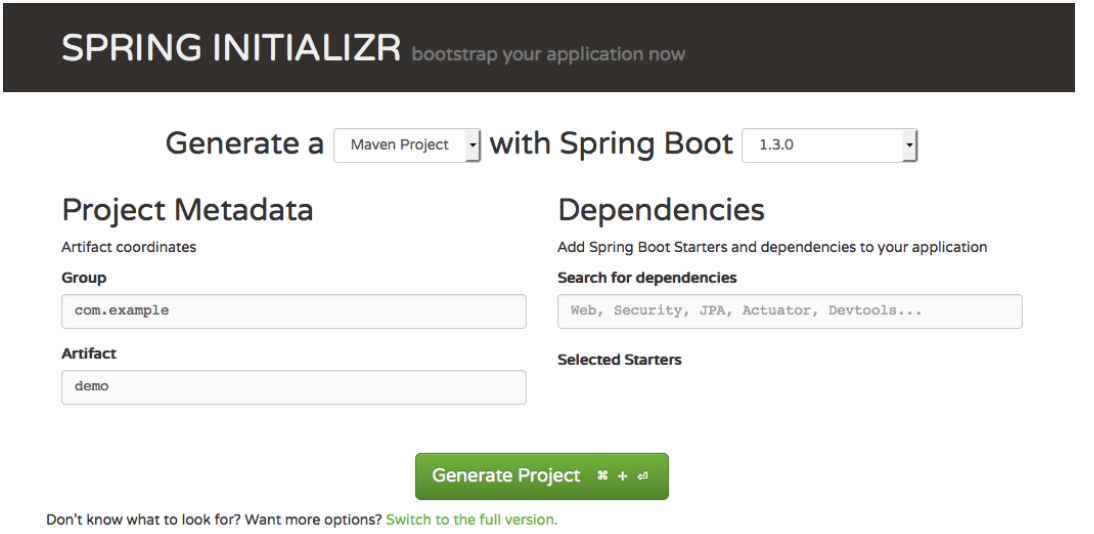


Figure 11.5. Spring Initializr web-page

Specify the modules you want (if you switch to the "full version" you can use checkboxes to select the modules you want just like in the STS wizard in Figure 11.4). When you hit `Generate Project` it creates a zip file to download. Unpack and import the project into your IDE.

When you return to your workplace you might like to try this for real.

11.5. Reference: Cheat sheet

This section contains code snippets, referenced by the lab instructions. There is no need to read through it unless you are stuck.

11.5.1. Create a "Hello" controller

Here is an example how a Hello controller could look like. It is an even simpler controller than the one you worked on in the Spring MVC lab.

Ask your instructor if you don't fully understand the code.

Copy and paste the following:


```
package accounts;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;

@Controller
public class HelloController {

  @RequestMapping("/hello")
  public String hi() {
    return "hello";
  }
}
```


Chapter 12. security: Securing the Web Tier

12.1. Introduction

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

What you will learn:

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

Estimated time to complete: 45 minutes

12.2. Quick Instructions

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

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

12.3. Detailed Instructions

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

12.3.1. Setting up Spring Security in the application

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

(TODO-01) Begin by starting the application for this project and navigate to the home page at http://localhost:8080/. This is a Spring Boot application so run by doing right click -> Run As -> Spring Boot App. You should see a link to 'View Account List' - click on this link and the list of accounts should appear. Be sure the application starts successfully before moving on to the next step. You may need to stop any application that could be already running. If the application will not run, ask your instructor now for help.

12.3.2. Enable Spring Security

(TODO-02) Remove the statement as indicated in the TODO. Spring Boot will automatically enable Spring Security for us. We explicitly disabled the Spring Security auto-configuration in the first TODO, that's why we managed to access the application in the first place. If we use Spring Boot auto-configuration, it will detect Spring Security on the classpath and protect the whole application. As soon as we'll set up our own security configuration, it will override Spring Boot's default auto-configuration.

12.3.3. Include Security Configuration

(TODO-03) Next, import the bean configuration class containing the security configuration into the `SecurityApplication` class. (The name of the class is `SecurityConfig` and is located in the config package). This will include those beans when bootstrapping the application context.

At this point, Spring Security should be fully enabled and ready to intercept incoming requests. Save all work, restart the application and navigate to the home page at http://localhost:8080/. 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 `/accounts/accountList.htm` is secured and you have not configured a real login page yet. The `SecurityConfig` class currently specifies `loginPage("/TODO-03")` and there is no such page as `TODO-03`.

Figure 12.1. Accessing Secured Resource

12.3.4. Configuring Authentication

In this section you'll use Java Configuration to configure a login form for our application.

12.3.4.1. Specify the Login Page

(TODO-04) Open the `SecurityConfig` class. Notice that the actual security constraints are defined inside the method `configure(HttpSecurity http)`. In particular notice that the *EDITOR* role is required to access the `accountList` page.

When we tried to access the restricted `accountList` page the application tried to redirect us to a login page. Time to configure one.

Back inside `SecurityConfig`, configure the login page to be `/login` by modifying the `loginPage()` method under the `formLogin()` method.

Note

The login page `/login` needs to be accessible to anyone, especially someone who has not logged in yet. So we specify `permitAll()` to allow fully unrestricted access.

Set also the access denied page so it uses '/denied'.

12.3.4.2. Handling Login

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

Figure 12.2. Implementing Login Page

Try logging in with a random username and password such as username "foo" and password "foo". You should be returned to the login page with an error message in red. Notice the URL at the top of the page `http://localhost:8080/login?error`. By default a failed login redirects back to the login URL with the `error` parameter defined. (You can customize this by specifying `formLogin().failureUrl("/some-url")`).

Open `login.jsp` under the `src/main/webapp/WEB-INF/views/` folder. See how the test for `param.error` is used to display the error message. Notice that the default input field names are `username` and `password` and that the form action is to POST the form to itself.

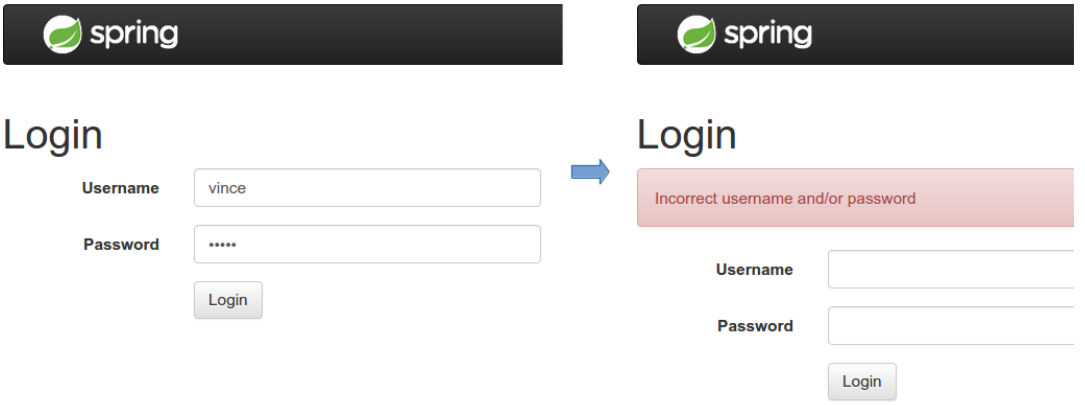


Figure 12.3. Handling Login Errors

12.3.4.3. Handling Denied Access

(TODO-05) To determine a valid username/password combination, you can explore the authentication configuration (further down within `SecurityConfig` you will find the `configureGlobal()` method). An in-memory authentication provider is being used. Notice that a single hard-coded user, password, and role have been setup to support testing.

Try logging in using the user called `vince`. You should see the "Access Denied" page.

Figure 12.4. Implementing "Access Denied" Page

12.3.5. Managing Users and Roles

(TODO 06) At the end of the previous section, vince could login, but was denied access to the list of accounts. In this section, you will modify the access rules and define additional users.

12.3.5.1. 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 `authorizeRequests` method and modify the rules for `/accounts/account*` to enable access for viewers as well.

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

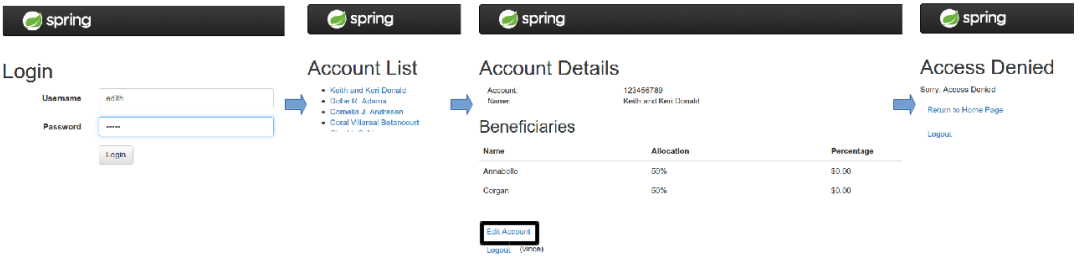


Figure 12.5. Configure Role-Based Access for Vince

12.3.5.2. Add a catch all

(TODO 07) Currently you secure URLs starting with `/accounts/edit` and `accounts/account` To get a more robust configuration, you should also enforce that people must at least be logged in to show anything else starting with `/accounts/`.

Log out by clicking on the 'log out' link. Then try to access http://localhost:8080/accounts/hidden. As you can see, this URL is currently not protected, but it should be.

Inside `SecurityConfig` class, add another `antMatchers` element at the bottom of the list with the pattern `/accounts/**` which enforces that the user should be fully authenticated. Be sure to do this using the method that will permit all users to access. Placing this entry at the end of the list causes the more specific entries to be evaluated first.

Save all work, restart the web application and check that attempts to access http://localhost:8080/accounts/hidden result in the login page being displayed.

12.3.5.3. Add a User

(TODO 08) Notice that the account list page provides a `logout` link. Open `accountList.jsp` within `WEB-INF/views` to see how it is implemented; by default Spring Security looks for a POST /logout request, which is implemented using Spring MVC form tag.

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

Revisit the globalSecurity method and add a user called `edith` with the EDITOR role. To do this, you will need to add an "`and()`" method on the end of the line that defines `vince`, then add `edith` on the following line. The new definition will be similar to that for `vince` except Edith's role must be `EDITOR`.

Note

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

Save all work, restart the web application, log in with the user `edith`. Navigate to the edit page by selecting an account from the account list and using the "Edit account" link on the details page. This time you should be able to access the `editAccount` page.

Our goal is to allow those with `EDITOR` role to edit account details while disallowing other roles, so we need to ensure that other users *cannot* access this page. Logout of the application and login again as vince. Repeat the navigation, but this time we expect vince to be redirected to the "Access Denied" page.

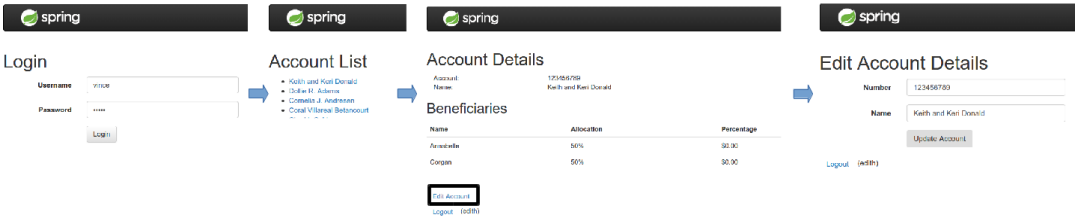


Figure 12.6. Configure Role-Based Access foe Edith

12.3.6. Using the Security Tag Library

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

12.3.6.1. Hide a Link Based on Role

(TODO-09) A fairly common requirement for web-tier security is to only display certain information and/or links to users with a specified role. For example, you could hide the 'Edit Account' link unless a user would be

able to perform an edit. This provides a much better user experience than constantly being redirected to the "Access Denied" page.

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

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

12.3.6.2. Hide Information Based on Role

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

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

Note

Notice the other available attributes on the `<security:authorize/>` tag. Feel free to apply the tag to other data and/or other JSPs. As you have seen, it's also trivial to define additional users and roles in order to have more options.

12.3.7. Bonus question: SHA-256 encoding

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

(TODO-11) Open `SecurityConfig` file and declare sha-256 encoding. embed a method call to `.passwordEncoder(new StandardPasswordEncoder())` in the chain before the users are defined. Now, passwords need to be encoded, note the encrypted values are already provided for you in the comment. Change the plain-text passwords into sha-256-encoded ones. You will not need to setup any salt source.

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

Note that the encoded password is based on SHA-256 with a random salt value introduced by the StandardPasswordEncoder, and actually includes the generated salt value in the first few bytes (8) of the encoded value. Although SHA-256 encoders are available in various websites, the easiest way to generate passwords loaded with this salt is to use (new StandardPasswordEncoder()).encode("thePassword")

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

Tip

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

Chapter 13. rest-ws: Building RESTful applications with Spring MVC

13.1. Introduction

In this lab you'll use some of the features of Spring 3.0 that 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. Use Spring MVC to implement server-side REST
4. 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: 50 minutes

13.2. Quick Instructions

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

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

13.3. Detailed Instructions

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

13.3.1. Exposing accounts and beneficiaries as RESTful resources

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

13.3.1.1. Inspect the current application

(TODO-01) First open the `RestWsApplication` class in the `accounts` package to see how the application is bootstrapped: it imports the `AppConfig` configuration class, which contains an `accountManager` bean that provides transactional data access operations to manage `Account` instances. As `RestWsApplication` uses the `SpringBootApplication` annotation, it will use component scanning and will define a bean for the `AccountController` class.

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

Finally, run this Spring Boot application (right click -> Run As -> Spring Boot App) and verify that the application deployed successfully by accessing http://localhost:8080 from a browser. When you see the welcome page, the application was started successfully. If you have a problem starting the application, the most likely cause is that you already have an application from a previous lab still running, so be sure to stop it.

13.3.1.2. Expose the list of accounts

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

Tip

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

When you've done that, save all work and restart the application. Now try to access

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

The reason is that the project includes the Jackson library on its classpath - check your Maven Dependencies folder for your project (in the Package Explorer).

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

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

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

Tip

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

Note

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

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

13.3.1.4. Expose a single account

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

Tip

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

Save your work and restart your application. To test your code, just try to access http://localhost:8080/accounts/0 to verify the result.

(TODO 05) When you're done with the controller, complete the `AccountClientTests` by retrieving the account with id 0.

Tip

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

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

If the time allocated for the lab is almost used up, this is a good place to stop.

13.3.2. Modifying data using POST and DELETE

In this section we will create and remove data.

13.3.2.1. Create a new account

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

(TODO 06) Update the `createAccount` method by adding a mapping to it from 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` parameter appropriately to let the request's body be unmarshaled into an account object. 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. We will write a test for this method in an upcoming step.

(TODO 07) RESTful clients that receive a `201 Created` response will expect a `Location` header in the response containing the URL of the newly created resource. Complete the TODO by implementing `entityWithLocation()`. Then save all work and restart the application.

Tip

To help you generate the full URL on which the new account can be accessed, you will need to use a `ServletUriComponentsBuilder` - refer back to the example in the slides to see how. This approach means you can avoid hard-coding the server name and servlet mapping in your controller code! You should use a `ResponseEntity` to generate the response.

(TODO 08 - 09) When you're done, complete the test method by POSTing the given `Account` to the `/accounts`

URL. The `RestTemplate` has two methods for this. Use the one that returns the location of the newly created resource and assign that to a variable. Then complete `TODO 09` by retrieving the new account on the given location. The returned `Account` will be equal to the one you POSTed, but will also have been given an `entityId` when it was saved to the database.

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

13.3.2.2. Seeing what happens at the HTTP level

If your create account test worked, and you are running short of time, you may skip this section and move on to "Create and delete a beneficiary". However the HTTP monitor discussed in this section is a useful debugging tool to know.

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

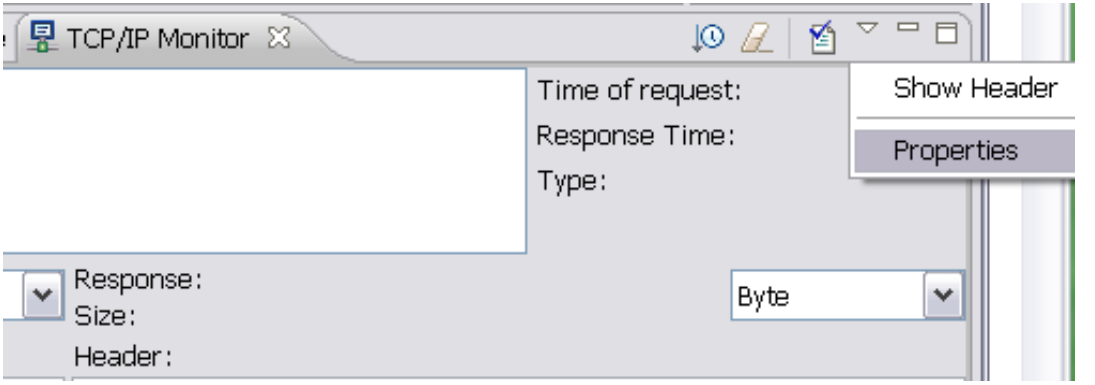


Figure 13.1. The "properties" menu entry of the TCP/IP Monitor view

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 your application 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.

13.3.2.3. Create and delete a beneficiary

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

(TODO 12) Create a ResponseEntity containing the location of the newly created beneficiary the new beneficiary. Save all work and restart the application.

Note

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

in the URL. You will need to use `entityWithLocation()` again.

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

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

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

(TODO 18) The current test ensures that we always create a new account using a unique number. Let's change that and see what happens. Edit the `createAccount` method in the test case to use a fixed account number, like `"123123123"`. Run the test: the first time it should succeed. Run the test again: this time it should fail. When you look at the exception in the JUnit View or at the response in the TCP/IP monitor, you'll see that the server returned a `500 Internal Server Error`. If you look in the Console View for the server, you'll see what caused this: a `DataIntegrityViolationConstraint`, ultimately caused by a `SQLException` indicating that the number is violating a unique constraint.

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

Tip

Have a look at the existing `handleNotFound` method or in the Advanced notes in the slides 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.

13.3.2.5. BONUS (Optional): Testing with a Mock HTTP Request

The `AccountController.createAccount()` method uses a `ServletUriComponentsBuilder` to generate the location URL of the new Account. It does this by initializing itself with the URL that invoked the `createAccount()` method. This is possible because Spring MVC has put the current request information in the current thread. This doesn't happen when running tests.

Follow the instructions in (TODO-20) to see how to mock up the necessary request for testing.

Chapter 14. spring-microservices: Microservices with Spring

14.1. Introduction

This lab refactors the MVC application to use a microservice.

14.1.1. What you will learn:

1. How to deploy a Eureka service-discovery server <ok>
2. How to write a microservice with a RESTful interface
3. How to access a microservice and interact with it.

14.1.2. Specific subjects you will gain experience with:

1. Spring Cloud projects
2. Eureka discovery-server
3. Using @DiscoveryClient
4. Spring Clouds enhanced `RestTemplate`

Estimated time to complete: 45 minutes

14.1.3. Overview

The original application was a simple, single process "monolith". It managed its own accounts database and fetched data directly.

We are going to refactor it into two processes:

The full system also requires a discovery-server, making three processes in all:

14.1.4. Notes

Since you will be running (eventually) 3 applications at the same time:

1. You will need to switch between console output (use the Display Selected Console icon on the right-hand side of the Console view.
2. Output to the console will keep taking the focus from the Task window. To avoid this, drag the Console view so it sits in its own pane. Ask your instructor if you don't know how to do this.

14.1.5. Instructions

14.1.5.1. Discovery Server

1. Go to the `ms-discovery-server` project.
2. Write the Discovery Server. `RegistrationServer.java` is the only file you need. Follow TODOs 01-02 to make it into a Eureka Discovery Server running as a Spring Boot application.
3. TODO-03: Run the `RegistrationServer` as a Spring Boot app and go to http://localhost:1111 to view its Dashboard.
4. TODO-04: Quiz: where is the port 1111 configured?
5. TODO-05: Leave the registration-server running and move on to the next project.

14.1.5.2. Accounts Microservice

1. Open the `ms-accounts-server` project.
2. TODO-06: Setup the `AccountsMicroservice.java` to be a Spring Boot application that registers itself with the Registration Server.
3. TODO-07: Run the AccountsMicroservice as a Spring Boot application and go to its home-page: http://localhost:/NNNN - you will need to work out the port number it is listening to (how? - similar to TODO-05). You should be able to fetch an account list as JSON by clicking the link.
4. TODO-08: Go back to the Registration Server dashboard http://localhost:1111. Has this microservice registered yet (you may need to wait a minute or so and refresh the screen a few times).

5. TODO-09: Leave accounts-microservice running and move on to the last project.

14.1.5.3. Accounts Web Application

1. Open the `ms-web-client` project.
2. TODOs 09-12: Setup `AccountsWebApplication.java` as a micro-service client. Note that TODO-12 is for information, there is nothing to do.
3. TODOs 13-15: Setup `RemoteAccountManager` to access the Accounts microservice using RESTful requests - you will need a `RestTemplate` to do the work.
4. TODO-16: Return to `AccountsWebApplication.java` and run it as a Spring Boot application. Once it is up and running properly it will also appear in the Registration Server dashboard - this may take a minute or so.
5. TODO-16: Goto http://localhost:8080 and see if you can list and view accounts.

Notice that setting up all three processes is a lot more complicated than running our original MVC application. But each process is actually very simple.

Congratulations, you have finished the lab.

Chapter 15. xml-dependency-best-practices: XML Dependency Injection Best Practices

15.1. Introduction

What you will learn:

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

Specific subjects you will gain experience with:

1. Bean Definition Inheritance
2. Importing Configuration Files

Estimated time to complete: 30 minutes

15.2. Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments. To display them, open the `Tasks` view (Window -> Show view -> Tasks (*not Task List*)). Use the view's small down arrow to select a `Configure Contents...` menu, you'll find the instructions are easy to follow if you configure TODOs to display `On any element in the same project`.

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

15.3. Detailed Instructions

15.3.1. Using Bean Definition Inheritance to reduce 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*.

(TODO 01) Inside the `rewards` package, open the test class called `RewardNetworkTests`. As you can see Spring is being configured by using `TestInfrastructureConfig` in the same package.

Open `TestInfrastructureConfig.java` and you will see an empty `@ImportResources` annotation. We will use this to load our two XML configuration files: `test-infrastructure-config.xml` and `application-config.xml`. Modify it now.

Note

You can use `CTRL-R` in STS to locate resources like XML files (use `COMMAND-R` on a Mac).
Keep in mind that you should include the file paths starting from inside the classpath root folders (`src/test/resources` and `src/main/resources` are part of the classpath). And yes, these are in different locations to the first XML lab. When done, run `RewardNetworkTests`. It will pass if the file paths are correct.

15.3.2. Define the `abstractJdbcRepository` bean

Bean definition inheritance is useful when you have several beans that should be configured the same way. It lets you define the shared configuration once, then have each bean inherits from it. In the `rewards` application, there is a case where bean definition inheritance makes sense. Recall there are three JDBC-based repositories, and each repository needs the same `dataSource`.

(TODO 02) Inside `src/main/java` within the `rewards.internal` package, open `application-config.xml`. Note how the `property` tag instructing Spring to set the `dataSource` is currently duplicated for each repository.

Now in `application-config.xml`, create an abstract bean named `abstractJdbcRepository` that centralizes the `dataSource` configuration. You will not need to define the class for this bean, but you should define the `dataSource` property and set it with a reference to the `dataSource` bean.

Next, update each repository bean so it extends from your `abstractJdbcRepository` bean definition. The repository beans will no longer need to set their own `dataSource` properties since this is now defined by the abstract bean definition.

Re-run `RewardNetworkTests`. It should still pass.

15.3.3. Externalizing values to a Properties file

(TODO 03) In this section, you'll gain experience with using the `<context:property-placeholder>`, element.

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

Create a file named `application.properties` in the root of the classpath. Add the following properties:

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

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

Within `test-infrastructure-config.xml`, replace each property value configured for your `embedded-database` with a placeholder. The placeholder name should match the respective property name in your properties file. The placeholders follow the syntax `${placeholder}`

If you run the `RewardNetworkTests` at this point it will fail. One more step left to complete...

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`, it should pass.

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.

15.3.4. Using the `<import/>` tag to combine configuration fragments

(TODO 04) Using the `<import/>` tag is often a good idea when working with multiple configuration files. Return to `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.

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.

Open `test-infrastructure-config.xml` and add an `<import/>` tag to import `application-config.xml`. Within `RewardNetworkTests`, remove the reference to `application-config.xml` from the array of

configuration files. Rerun the test, it should pass.

15.3.5. Using the `<jdbc/>` namespace

(TODO 05) The declaration of the embedded database in `test-infrastructure-config.xml` can be simplified by using the `<jdbc/>` namespace.

Make the change and rerun `RewardNetworkTests`, it should still pass.

Chapter 16. jms: Simplifying Messaging with Spring JMS

16.1. 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.

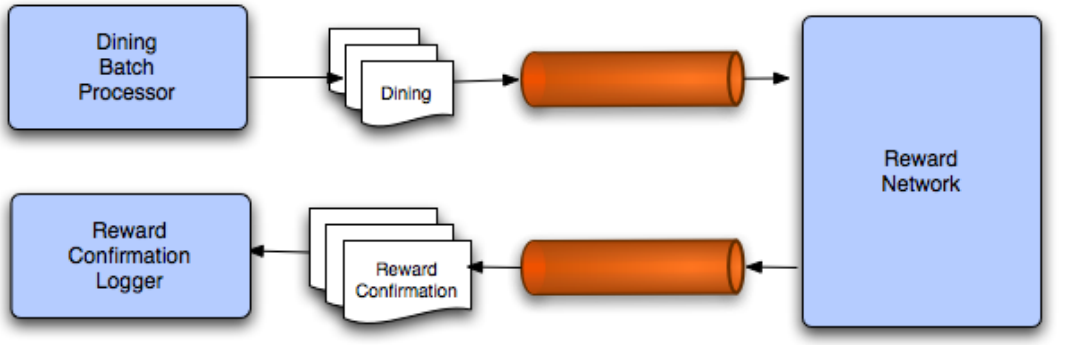


Figure 16.1. The batch processing of dining events with asynchronous messaging

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

Estimated time to complete: 45 minutes

Note

Refer to the diagram as you proceed through the lab - we get you to implement the code in the same order as the flow.
You will have to implement every step before you can run anything. Only at the end will you know if it all works.

16.2. Quick Instructions

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

16.3. Detailed Instructions

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

16.3.1. Providing the messaging infrastructure

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

16.3.1.1. Define the `ConnectionFactory`

(TODO-01) In JMS-based applications, the Connection is obtained from a `ConnectionFactory`. Spring's JMS support will handle the resources, but it does require a `ConnectionFactory` bean definition. In this step you will review what we have setup for you.

Open the `JmsInfrastructureConfig` class in the `config` package. There is a bean definition to create an instance of `org.apache.activemq.ActiveMQConnectionFactory`. Note the `brokerURL` property.

Tip

For this simple lab, you will be using an embedded broker with persistent queues disabled. Also

Spring will handle shutdown of the Broker, so ActiveMQ doesn't have to. Hence the `brokerURL` is `'vm://embedded?broker.persistent=false&broker.useShutdownHook=false'`.

Once you're happy you understand this bean definition, move on to the next step!

16.3.1.2. Define the message queues

(TODO-02) Now you will need to create two queues, one for handling dining notifications and the other for handling the reward confirmations. Create two bean definitions of type `org.apache.activemq.command.ActiveMQQueue` and call them "diningQueue" and "confirmationQueue". Provide a unique name for each queue using constructor injection. You can use any names you want (we suggest "`rewards.queue.dining`" and "`rewards.queue.confirmation`", which are the names used in the solution, or something similar if you prefer), but keep track of the names for use in a later step.

Warning

Remember the queue names you select, you will need them later. If you specify the wrong queue name later, the messages are quietly ignored. You DO NOT get an error. Using the wrong queue-names is the most common error in this lab.

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

16.3.2. Sending Messages with `JmsTemplate`

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

16.3.2.1. Establish a dependency on `JmsTemplate`

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

16.3.2.2. Implement the batch sending

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

Note

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

16.3.2.3. Define the template's bean definition

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

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

16.3.3. Configuring the `RewardNetwork` as a message-driven object

In the previous section you implemented the dining notification sending. In this section you will provide the necessary configuration for receiving those messages and delegating their content to the `RewardNetwork`. You will do this using an annotation-driven approach.

16.3.3.1. Define the JMS Message Listener

(TODO-06) Open the `RewardNetworkImpl` class within the `rewards.internal` package. Locate the `rewardAccountFor` method, this is the method that we want to use to process the JMS message and create and return a JMS response message. But to keep our code completely decoupled from the JMS API, we will use Spring annotations to simply indicate the incoming and outgoing JMS destinations.

Place the `@JmsListener` annotation above the `rewardAccountFor` method. Within this annotation, set the name of the destination that you created earlier, the one that will contain the Dining objects. (Recall that in JMS, the term destination refers to both where messages come from as well as where they go to.) Be sure to use the actual destination / queue name and NOT the bean ID. There are many other setting that you can control on this annotation, but the destination is the only one we need to set now.

Note that this method returns a `RewardConfirmation` object used to record information about the reward. We would like to take this returned object and send it back out to a separate JMS destination. Use the `@SendTo` annotation to define the destination to send this confirmation to. As before, be sure to use the actual destination / queue name, not the bean ID.

At this point, the `rewardAccountFor` method is "wired" to receive JMS traffic as Dining objects and return RewardConfirmation objects. But we still have to setup a few more pieces before our system will work. Move on to the next step.

16.3.4. Receiving the asynchronous reply messages

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

16.3.4.1. Define a second JMS Listener

(TODO-07) Open `RewardConfirmationLogger` in the `rewards.jms.client` package. This existing logic is designed to simply listen for RewardConfirmation objects on the confirmations queue and keep track of how many we receive. However, it is not yet "wired" into the JMS infrastructure to do this work.

Place the `@JmsListener` annotation above the `log` method. Within this annotation, set the name of the destination that you created earlier, the one that will contain the RewardConfirmation objects. Be sure to use the actual destination / queue name and NOT the bean ID. Note that this method does not return any return value, so you do not need to supply a `@SendTo`.

16.3.5. Enable Asynchronous Message Reception

In the previous sections, you configured two method to respond to JMS input. However there are still some housekeeping items that we need to setup to hook all the pieces together - we have to tell Spring to look for and process the `@JmsListener` annotations, and we need to setup a factory that can produce the listener containers needed to wrap the JMS processing endpoints.

16.3.5.1. Define a JMS Listener Container Factory Bean

(TODO-08) Return to the `JmsInfrastructureConfig` class and add a @Bean definition. The bean we want to create should have the ID of `jmsListenerContainerFactory`, and should instantiate and return a `DefaultJmsListenerContainerFactory` object. This Factory is used by Spring whenever it needs to create a "Listener Container" for one of our `@JmsListener`-annotated methods. We could provide separate beans for each `@JmsListener`-annotated method, but they would typically be configured exactly the same, so providing Spring a factory to use is much less work.

The `DefaultJmsListenerContainerFactory` has many properties that we can set, but for our exercise we really only need to provide a reference to the connection factory. You defined this bean in one of the first steps, so simply set this dependency before returning.

(TODO-09) Finally, we need to tell Spring to look for the `@JmsListener` and `@SendTo` annotations we've placed in the code, and to wrap these in proxies associated with the relevant JMS Listener Containers. Simply add a `@EnableJms` annotation on the top of any `@Configuration` class to turn this feature on.

16.3.6. 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.

16.3.6.1. Send the batch of dining notifications

(TODO-10) 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-11) Finally, provide an assertion to verify that the entire batch was sent and that the `confirmationLogger` has received the same number of replies. Run this test, it should pass at this point. If you receive a failure, examine the exception message carefully, backtracking your steps as needed to produce a successful run.

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 your tests pass, you have completed this lab. Congratulations!

Chapter 17. jmx: JMX Management of Performance Monitor

17.1. Introduction

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

What you will learn:

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

Specific subjects you will gain experience with:

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

Estimated time to complete: 30 mins

17.2. Quick Instructions

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

17.3. Detailed Instructions

17.3.1. Exposing the `MonitorFactory` via JMX

17.3.1.1. Check the Application Runs>

(TODO 01) This is already a complete, working application. Run it now by right-clicking on the project and selecting `Run As ... --> Run on Server` (as described Appendix D, *Using Web Tools Platform (WTP)*). Open a browser window to http://localhost:8080/jmx. If you are having trouble, please ask your trainer for help.

Once you are happy the application is working, you can stop it so we can modify the code.

17.3.1.2. 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

17.3.1.3. Add JMX metadata to the implementation class

(TODO 02) Add Spring's JMX annotations `@ManagedResource`, `@ManagedAttribute` and `@ManagedOperation` to the class as well as methods you want to expose via JMX. Use `statistics:name=monitorFactory` as name for the bean exposed.

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

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

17.3.1.4. Activate annotation driven JMX in application configuration

(TODO 03) Find and open the `AspectsConfig` class in the `config` package. In this file activate annotation driven JMX by adding the appropriate annotation.

17.3.1.5. Start the MBeanServer and deploy the web application

In this step, you will deploy the project as a web application as described Appendix D, *Using Web Tools Platform (WTP)*. However, before you can do that, you must tell the Java VM to start an `MBeanServer`.

IMPORTANT STEP: (TODO 04) To start our MBeanServer, 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.

(TODO 05) Now deploy the project as a web application. Once deployed, open http://localhost:8080/jmx in your browser. You should see the welcome page display containing a form that submits to the RewardServlet.

17.3.1.6. View the monitor statistics using JConsole

(TODO 06) 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.

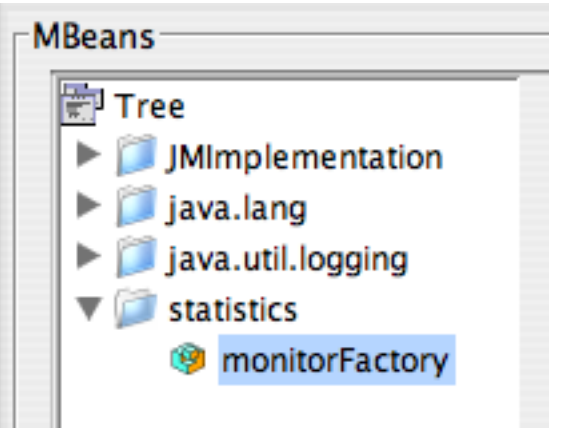


Figure 17.1. The `MonitorFactory` MBean

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

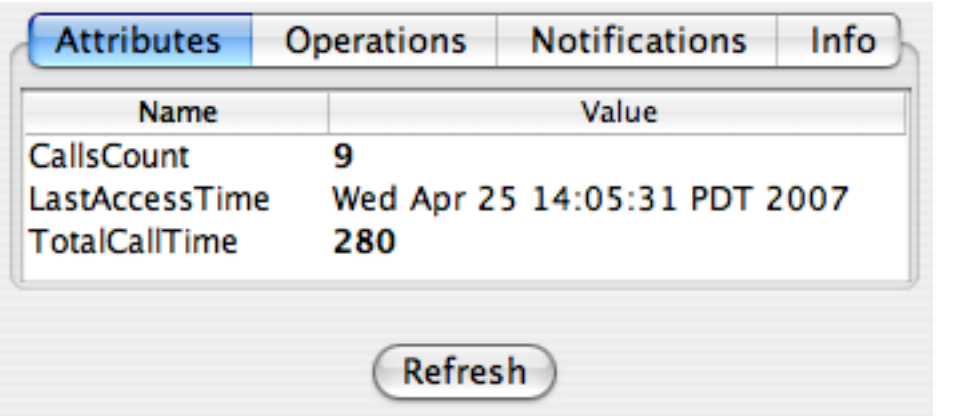


Figure 17.2. The `MonitorFactory` attributes

Tip

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

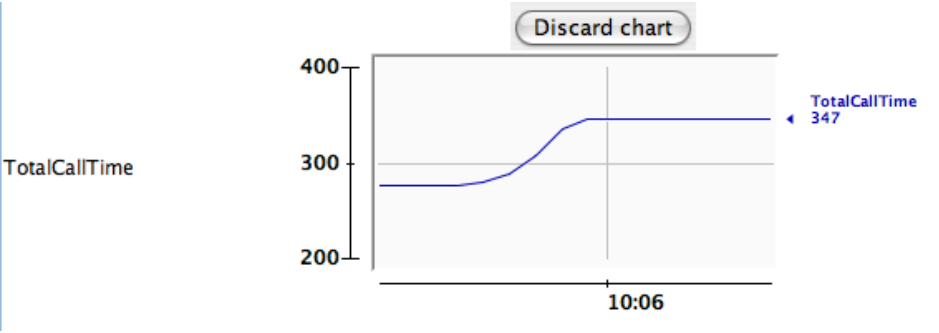


Figure 17.3. Scalar value graph

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

17.3.2. Exporting pre-defined MBeans

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

17.3.2.1. View the Hibernate statistics bean

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

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

Appendix A. Spring XML Configuration Tips

Note that auto-completion in Eclipse and STS is `CTRL+Space`, even on a Mac.

A.1. Empty Bean Definitions

Only the bean element is defined, no nested elements. If you do this first, then when you come to add contructor-arg or property sub-elements, auto-completion can be used to select the ref attribute value (see below).

```
<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>
```

Figure A.1. Examples of empty bean definitions

A.2. Bean Class Auto-Completion

The STS XML Editor is smart enough to know that the class attribute represents a fully-qualified Java classname. Type in the first 4 letters of the classname then use `CTRL+Space` to show all matching classes. Pick one and the FQCN becomes the class attribute value.

The example below shows using auto-completion by typing just the first letter of each component word in the classname. Thus typing "RNI" brings up `RewardNetworkImpl` and `RewardNetworkImplTests`.

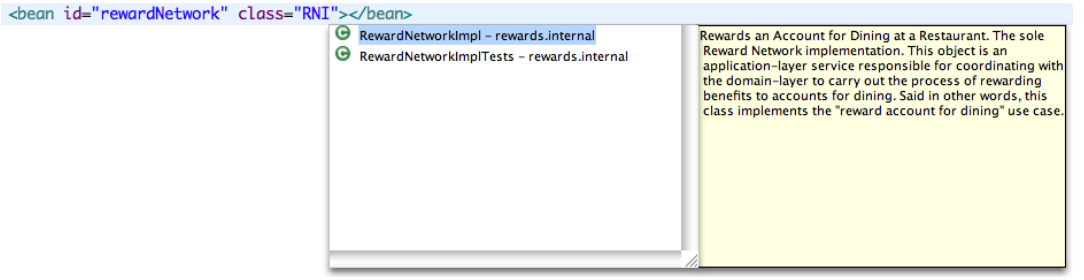


Figure A.2. Bean classname auto-completion

A.3. Ref Attribute Auto-Completion

Similarly using CTRL+Space in a ref attribute presents a list of all known bean ids. Works for both contructor-arg or property elements.

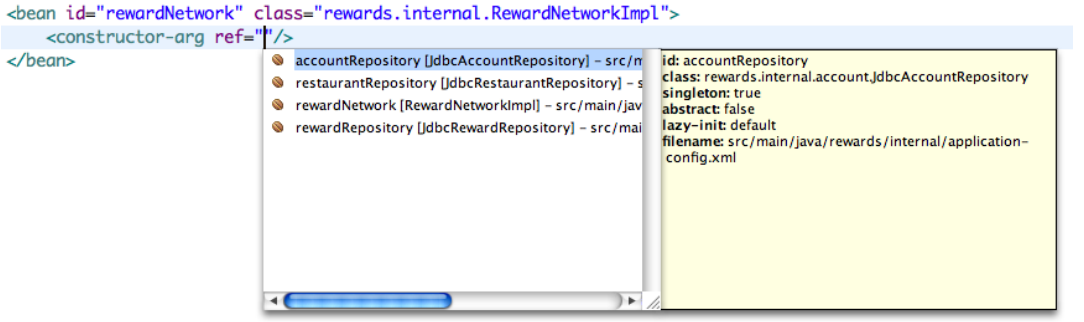


Figure A.3. Constructor argument reference auto-completion

A.4. Bean Properties Auto-Completion

This time CTRL+Space can be used to show available property names when defining a property element. Each name corresponds to a setter on the bean class.

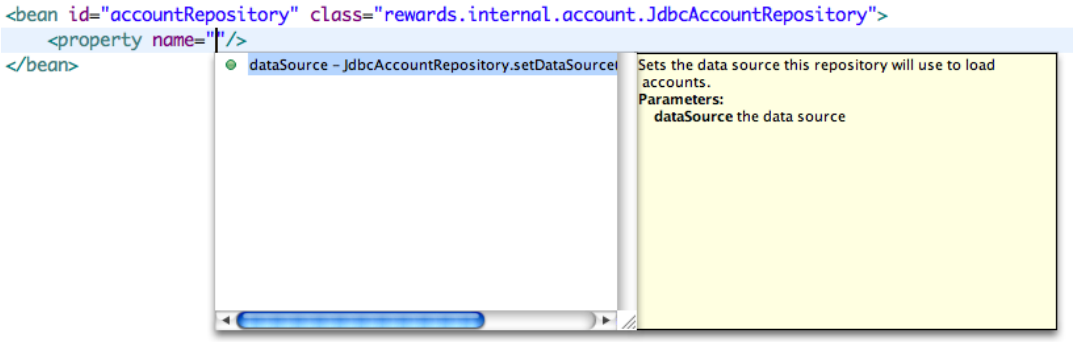


Figure A.4. Bean property name completion

Appendix B. Instructions for IntelliJ IDEA Users

B.1. Configuring the IDE

B.1.1. Configure a JDK

The first thing you have to do after installing IntelliJ IDEA is configure a JDK.

To see the list of preconfigured JDKs click the Configure button in the Quick Start panel of the Welcome screen.

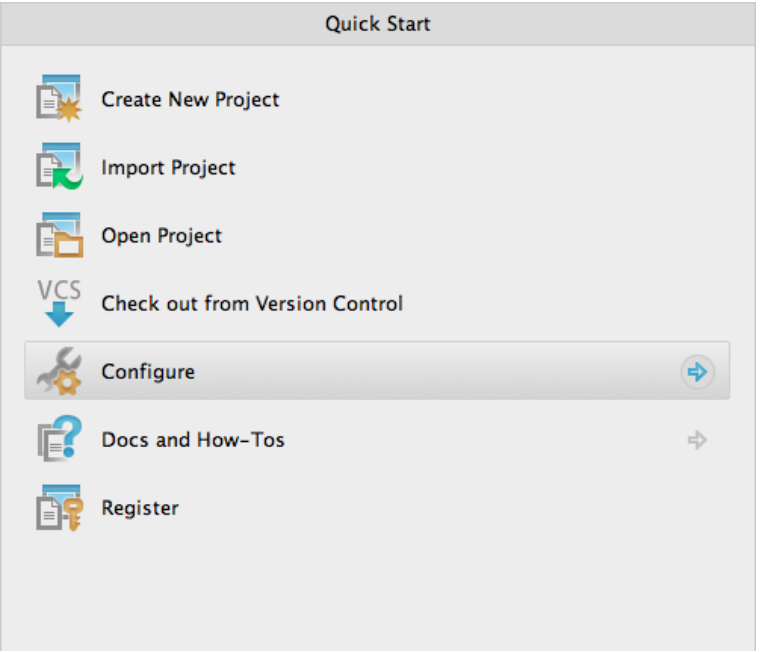


Figure B.1. Select JDK - Step 1

The window contents slide across.

Now choose Project Defaults

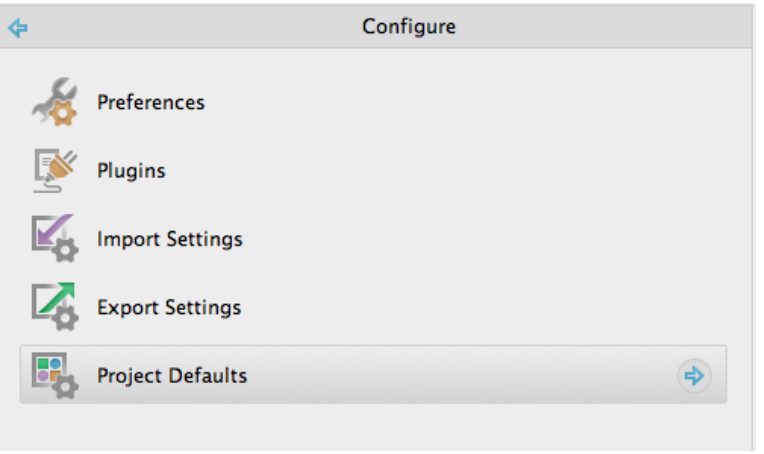


Figure B.2. Select JDK - Step 2

And finally Project Structure

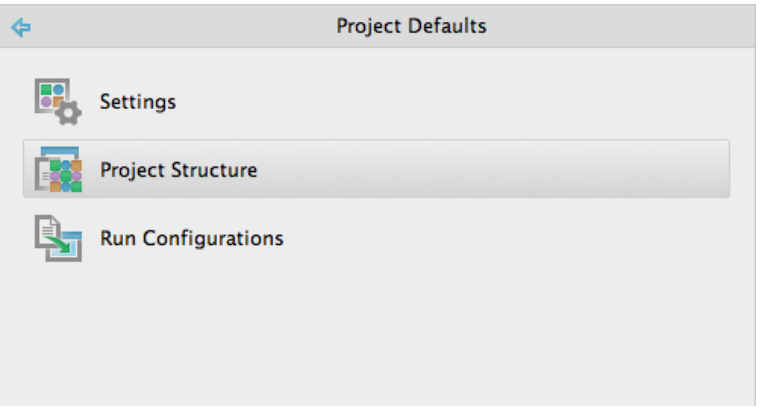


Figure B.3. Select JDK - Step 3

In the Project Structure dialog switch to the SDKs section. Here you can add a JDK by clicking the plus button:

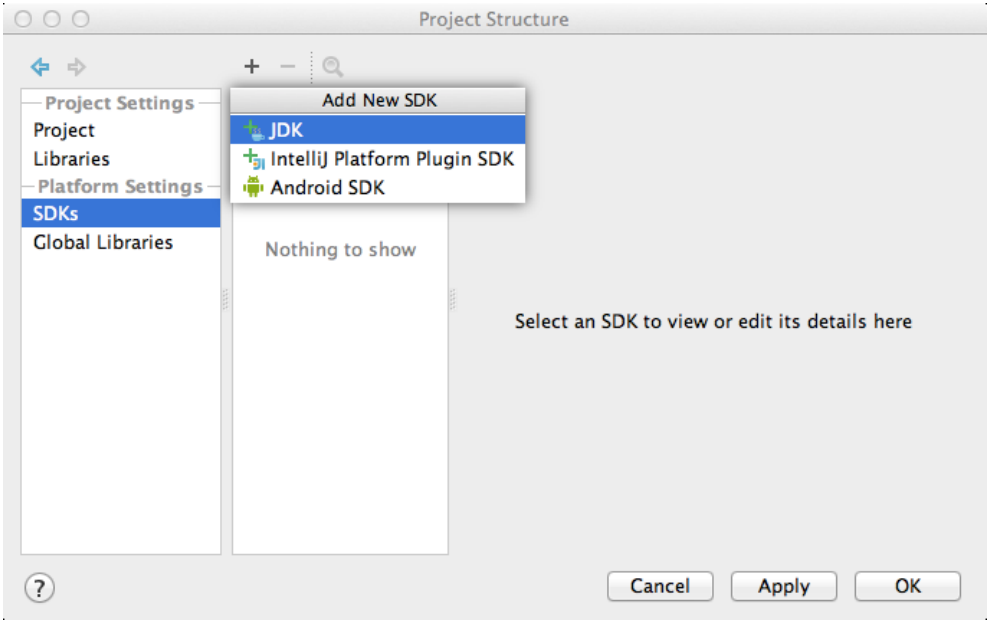


Figure B.4. Select JDK - Step 4

Make sure you have configured at least one JDK before you import the project.

B.1.2. Specify Maven local repository

Before you import the project, be sure to specify the Maven repository found within the courseware installation folder or directory. To do that, click the Configure button on the Welcome screen, and then choose Settings (or Preferences for MacOS).

In the Settings dialog:

1. Switch to the Maven tab;
2. Select the Override checkbox next to the Local repository setting; and
3. Specify the path to the Maven repository folder, a sub-folder of the course installation folder.

The default course installation folder is:

- MS Windows: `C:\<course-name>`
- MacOS: `/Applications/<course-name>`
- Linux: `/home/<user-name>/<course-name>`

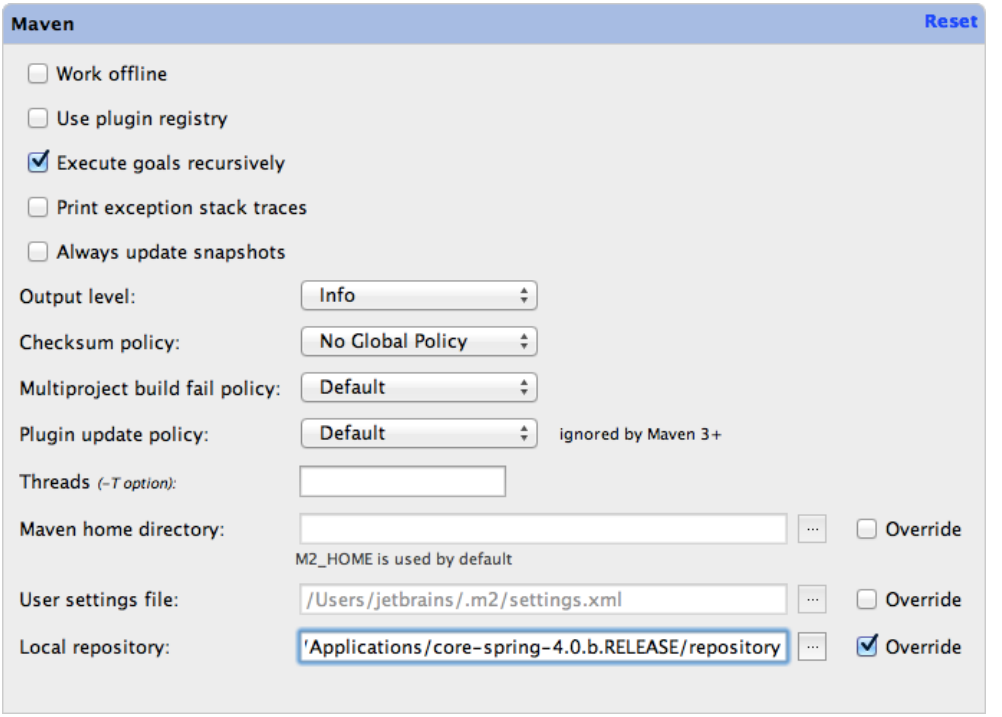


Figure B.5. Setup a Local Repository for Maven

This example shows `core-spring-4.0.b.RELEASE` on MacOS. The full path is:

```
/Application/core-spring-4.0.b.RELEASE/repository
```

Here are some more examples for different courses and releases:

- MS Windows: `C:\spring-web-4.0.a.RELEASE\repository`

- MacOS: `/Applications/enterprise-spring-4.0.a/repository`
- Linux: `~/core-spring-4.0.b.RELEASE/repository`

Remember this location if you are asked to configure M2_REPO later.

B.1.3. Configure a Tomcat application server

As you will need to run web applications during the course, make sure you've configured a Tomcat application server. To see the list of configured application servers, return to the Settings/Preferences dialog and switch to the Application Servers tab. Click the plus button to add an application server:

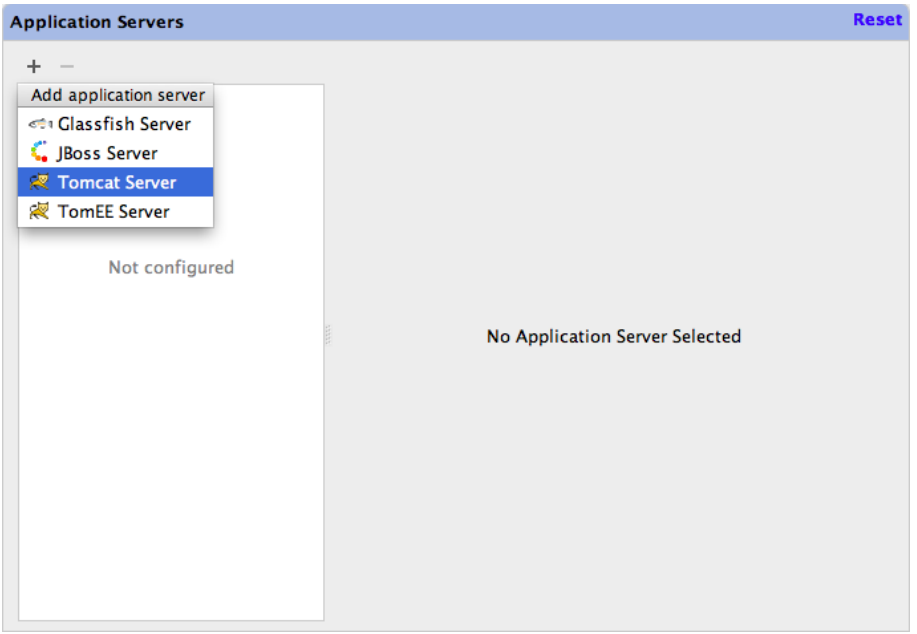


Figure B.6. Configure Tomcat

B.2. Importing the project into the IDE

To import a project into IntelliJ IDEA click the Import project button on the Welcome screen:

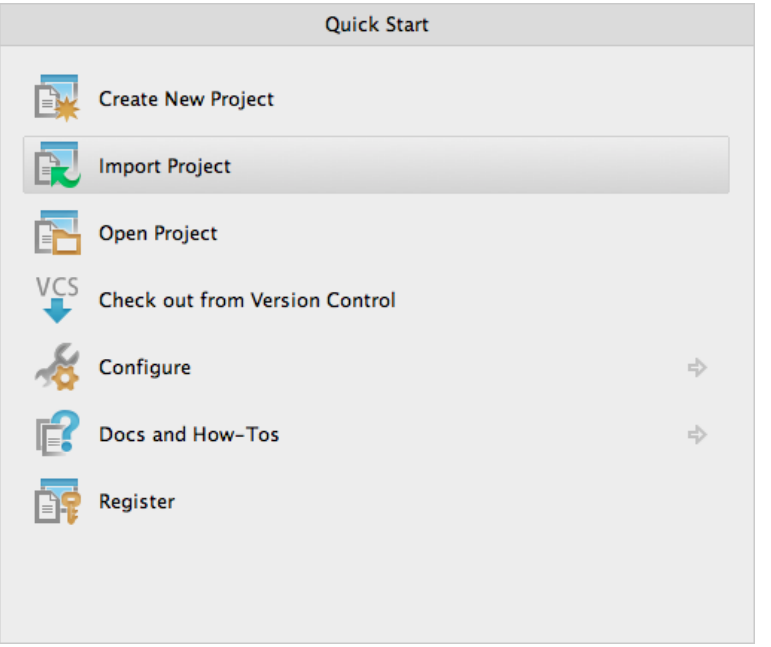


Figure B.7. Import Projects

B.2.1. Importing a Maven project

When the project you're trying to import has a root pom.xml file (which means this is a Maven project), then you have to choose this pom.xml file in the dialog that appears after you've clicked the Import project button.

Our courses contain many sub-projects, each with their own pom.xml. Make sure to pick the parent POM, located in the folder that holds all the projects as shown. The parent folder is:

- MS Windows: `C:\<course-name>\<course-name>`
- MacOS: `/Applications/<course-name>/<course-name>`
- Linux: `/home/<user-name>/<course-name>/<course-name>`

The example below shows The example below shows the location of the parent POM for `core-spring-4.0.b.RELEASE` on MacOS. Its full path is:


```
/Application/core-spring-4.0.b.RELEASE/core-spring-4.0.b.RELEASE/pom.xml
```

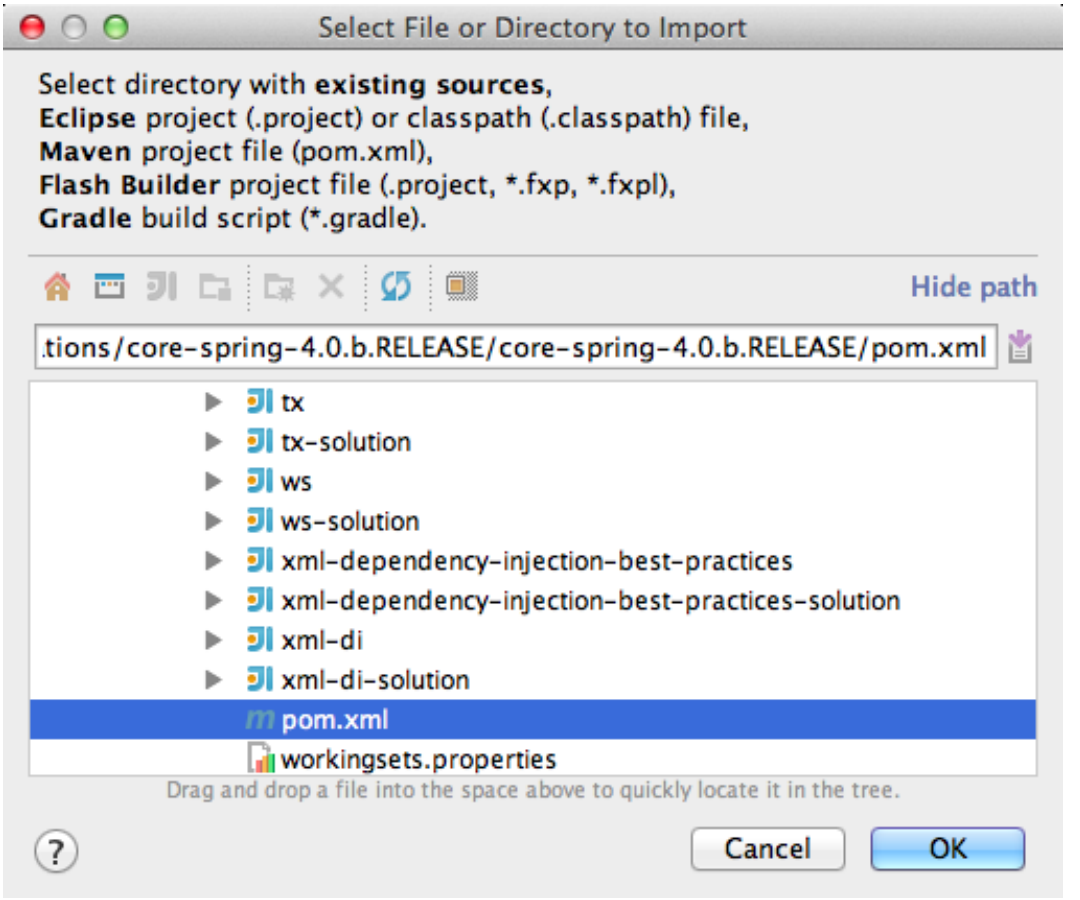


Figure B.8. Import using Maven POM

On MS Windows the same file would be at:

```
c:\core-spring-4.0.b.RELEASE\core-spring-4.0.b.RELEASE\pom.xml
```

B.2.2. Importing Eclipse projects

When the project comes with no root pom.xml file you can import it as an Eclipse project using its .classpath file. In our case there is more than one project, so choose the entire root folder:

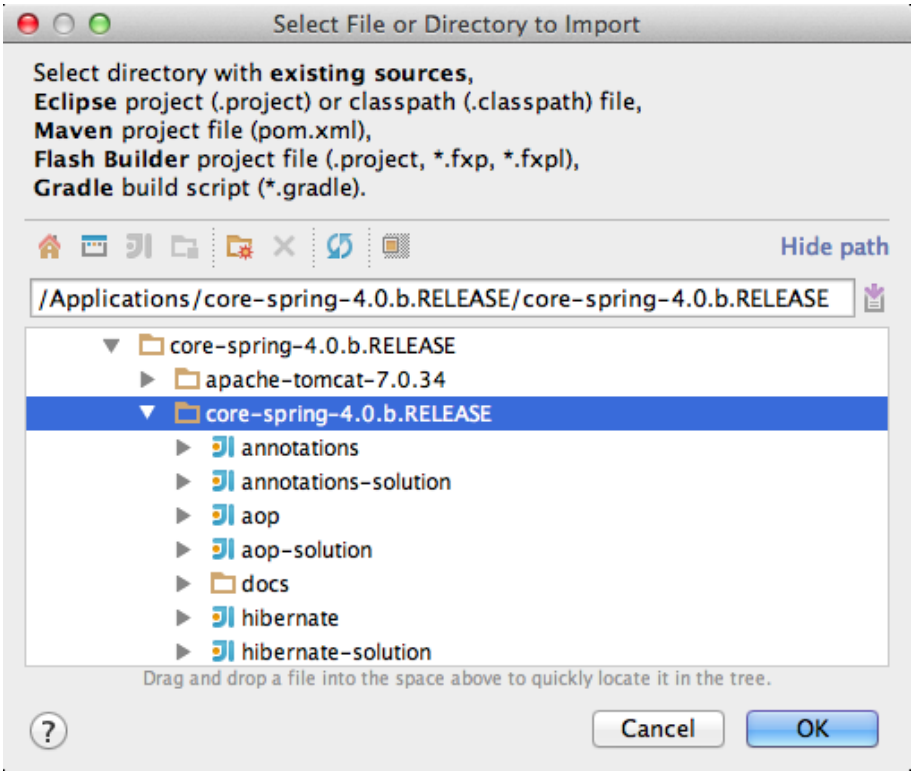


Figure B.9. Import Eclipse Projects - Step 1

After you've chosen the folder, the IDE will ask you which external model to use for the import. Make sure you've selected Eclipse:

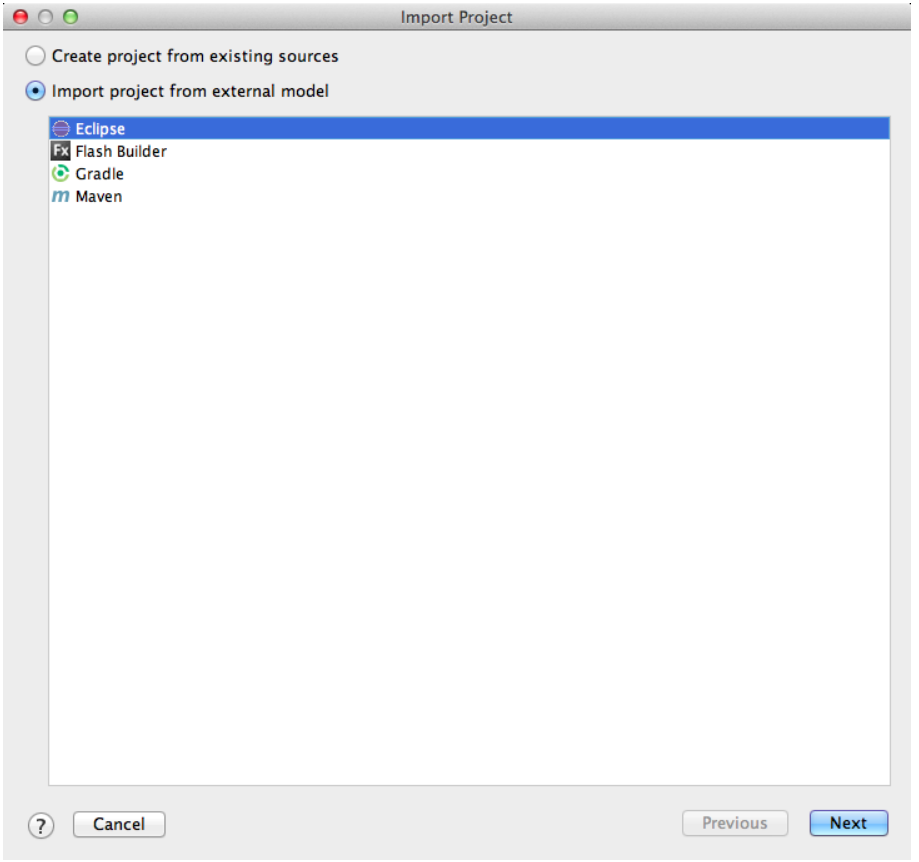


Figure B.10. Import Eclipse Projects - Step 2

If there are several projects in the folder, the IDE will ask you to select the projects to import:

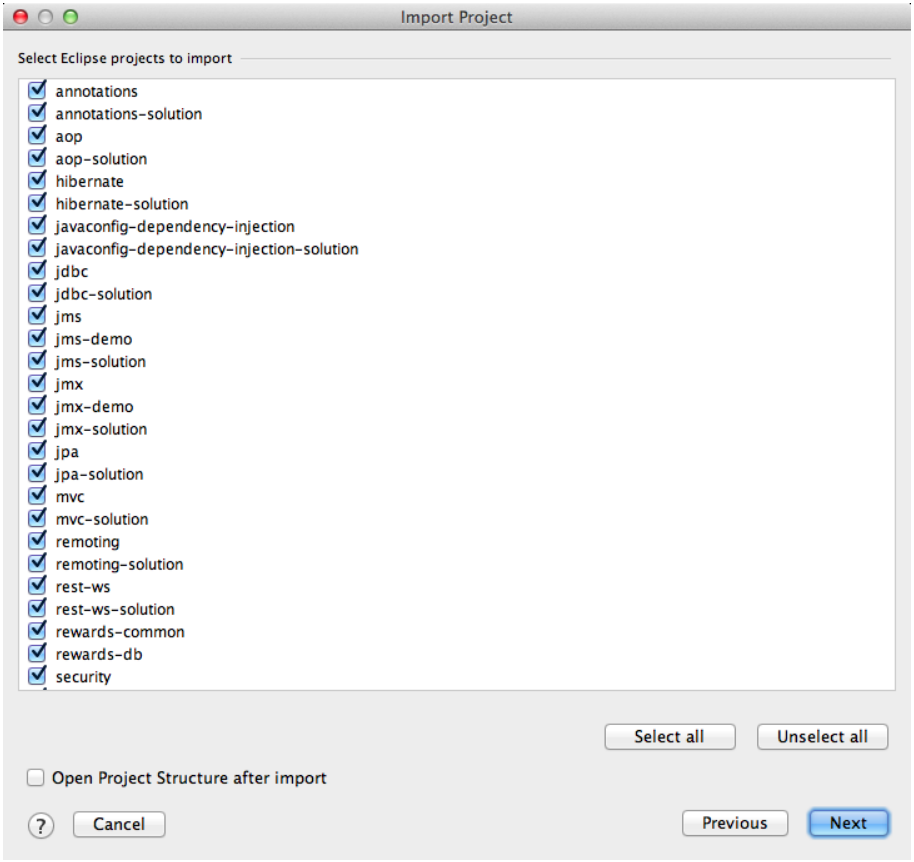


Figure B.11. Import Eclipse Projects - Step 3

At this point IntelliJ IDEA may prompt you to set the M2_REPO variable. Set this variable to point to the 'repository' folder within the install folder (the same location that we configured earlier). Once set, you should now see all of the Eclipse projects as modules within IntelliJ.

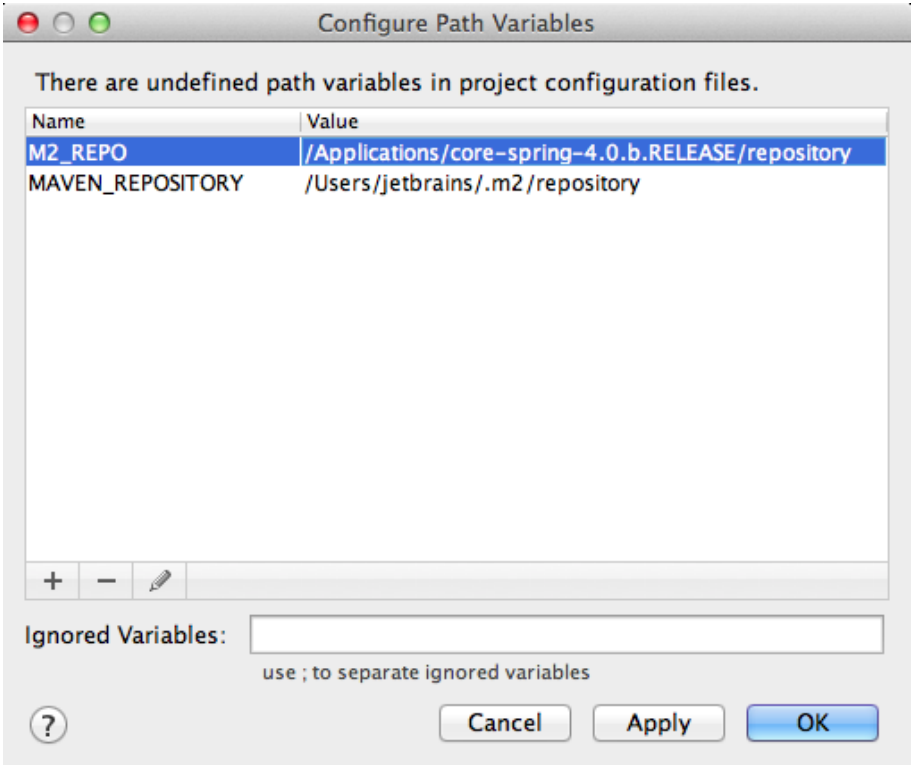


Figure B.12. Import Eclipse Projects - Step 4

Finally, after the IDE has imported the project, make sure it is compiled without errors. Once the import is finished the IDE may offer you to restart the IDE, please do it.

B.3. Running applications and tests

B.3.1. Deploying web applications

To run a web application, deploy the corresponding artifact to the application server. A Run configuration defines how artifacts are deployed to a server. Go to the Run # Edit Configurations menu, and add a Local Tomcat configuration. The Local run configuration will start a new instance of the configured server and deploy artifacts there.

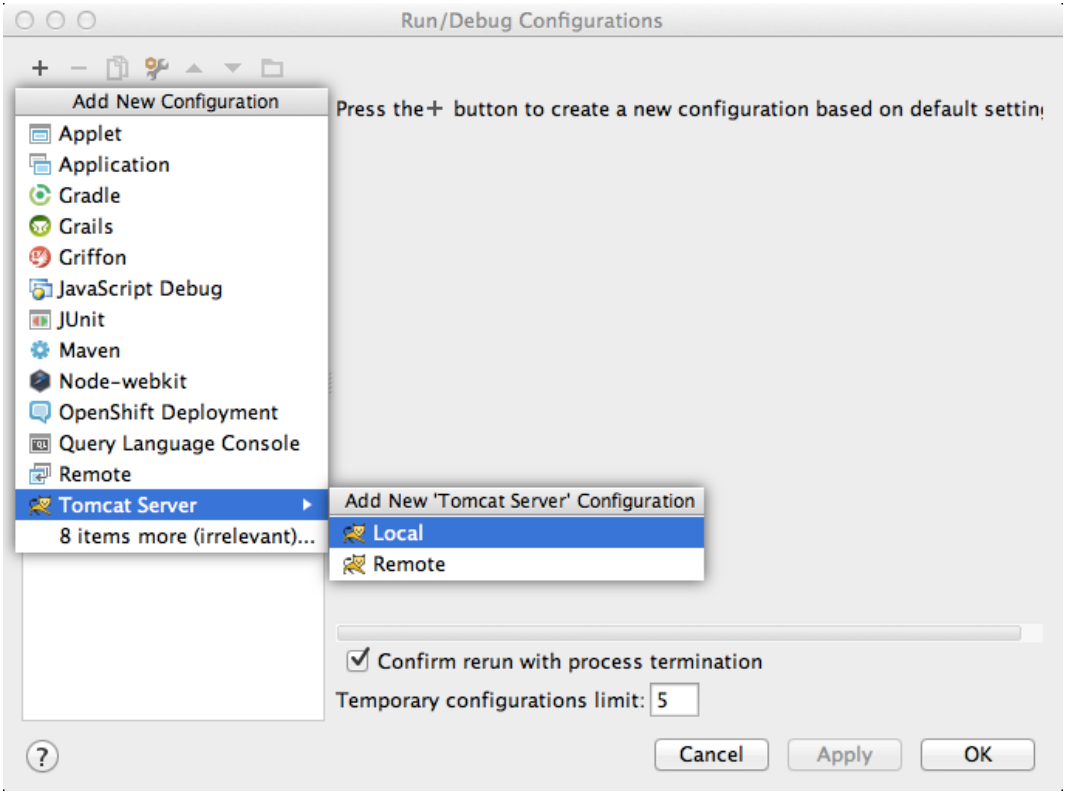


Figure B.13. Deploy Web Project - Step 1

Then switch to the Deployment tab and add the artifacts by clicking the plus button:

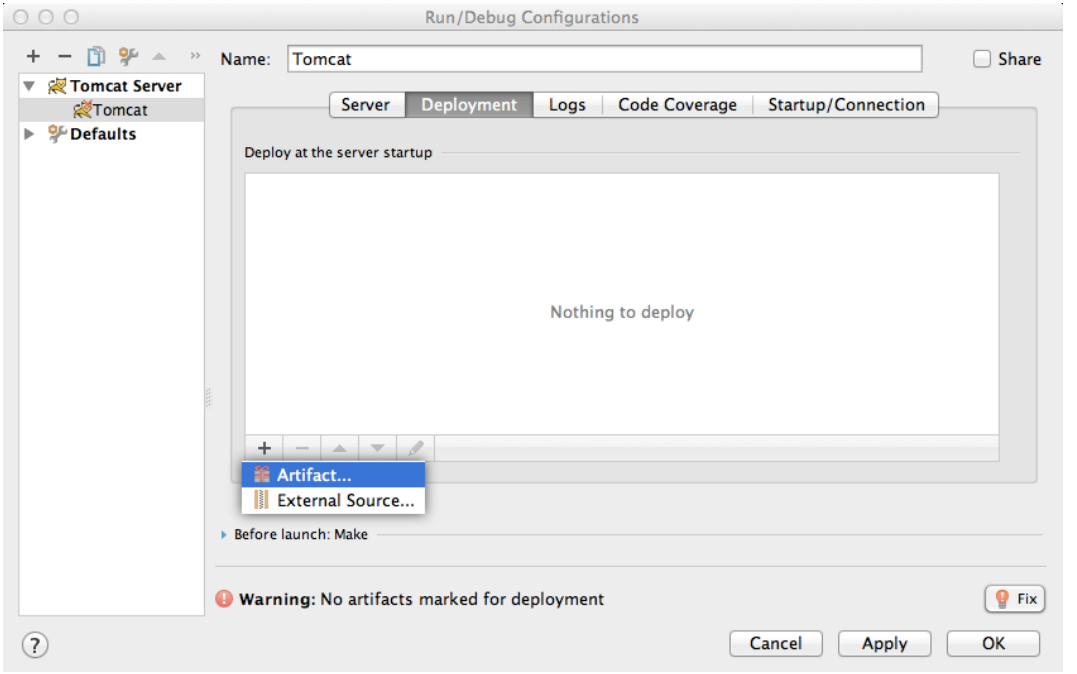


Figure B.14. Deploy Web Project - Step 2

Select the artifacts you'd like to deploy to the server:

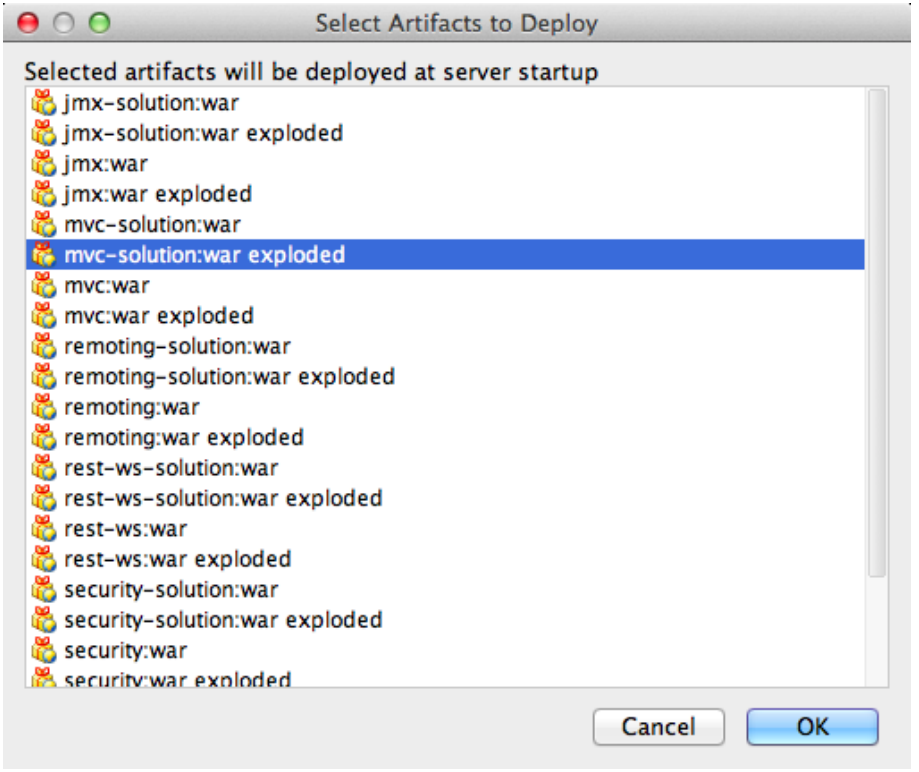


Figure B.15. Deploy Web Project - Step 3

B.3.2. Running tests

To run all tests from a package or the entire project, simply select Run 'All Tests' from the context menu in the Project tool window:

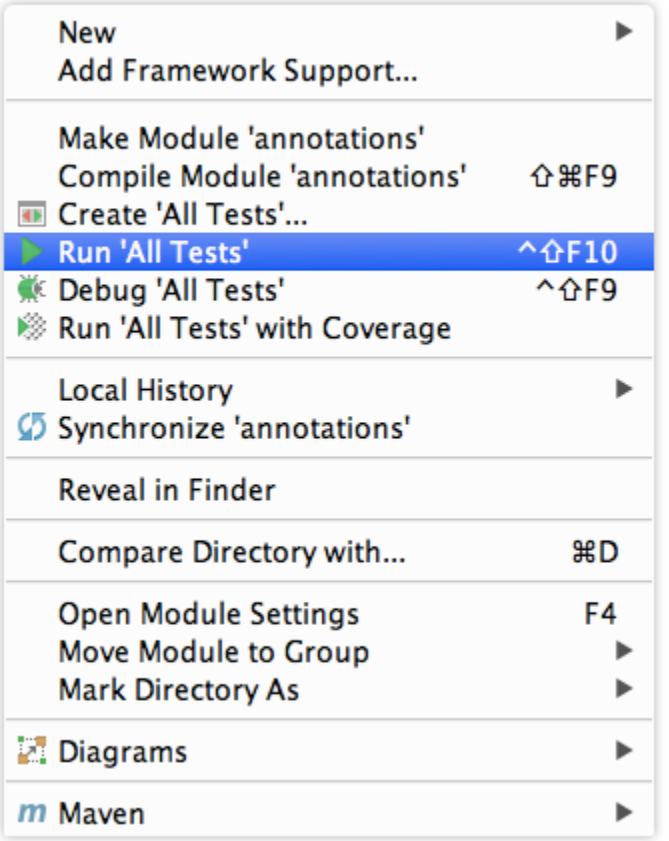


Figure B.16. Run Tests

If you want to run tests from a single class, use the corresponding action from the context menu for that particular class. To use specific parameters for running tests, you can create a run configuration manually via the Run # Edit Configurations menu.

B.3.3. Running applications

To run an application from its main method, use the corresponding context menu action:

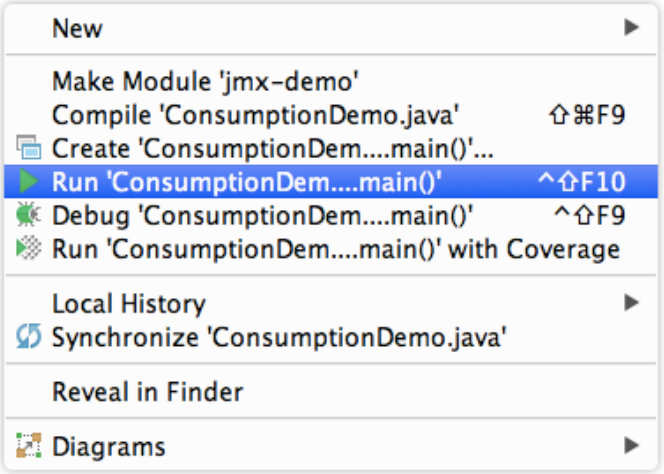


Figure B.17. Deploy Application

Or create a run configuration manually via the Run # Edit Configurations menu.

B.3.4. Working with TODOs

To see the list of TODO instructions, use the TODO tool window, which can be opened from the left-hand bottom corner of the IDE. Use the toolbar buttons to group items by module:

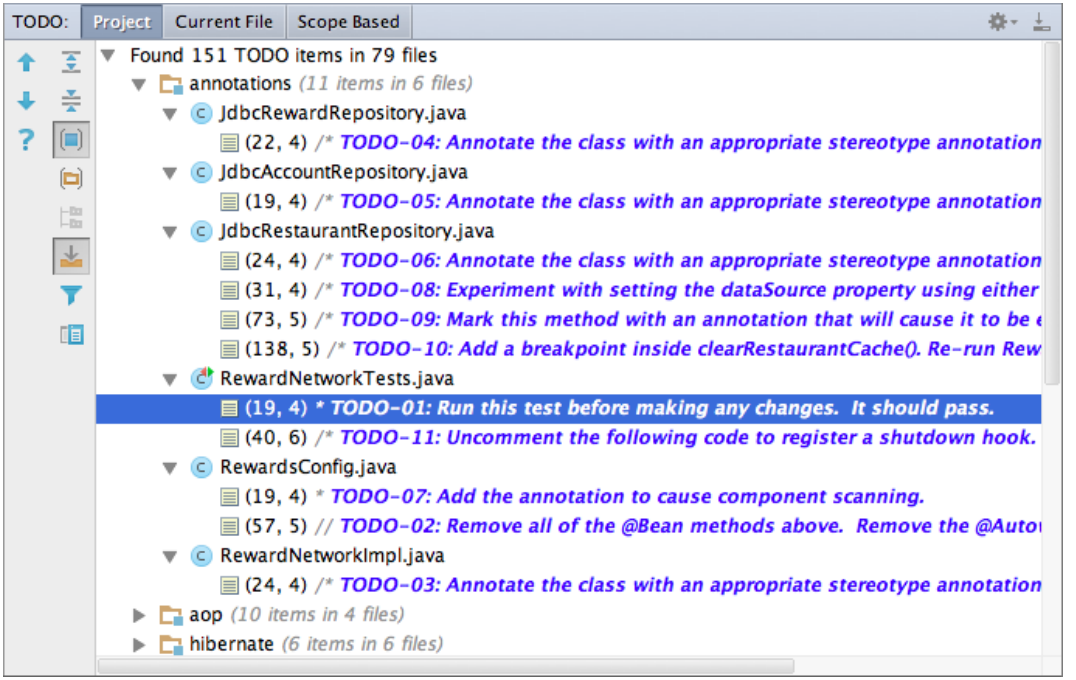


Figure B.18. View TODO Steps

B.3.5. Other Resources

Refer to the following resources to learn more about IntelliJ IDEA:

1. IntelliJ IDEA - quick start guide
2. How to migrate from Eclipse to IntelliJ IDEA
3. IntelliJ IDEA help

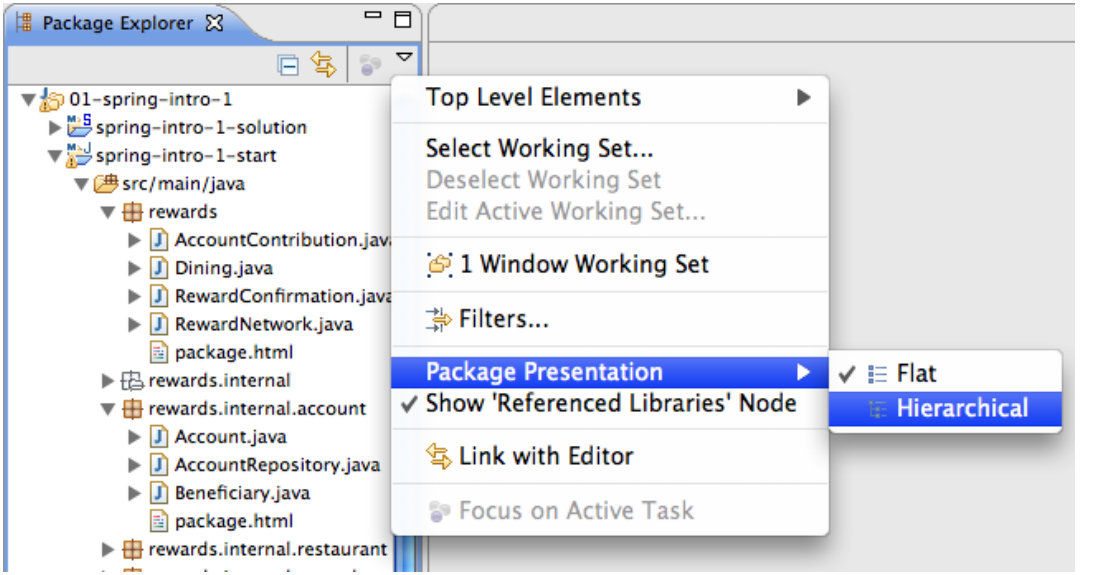
Appendix C. Eclipse Tips

C.1. Introduction

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

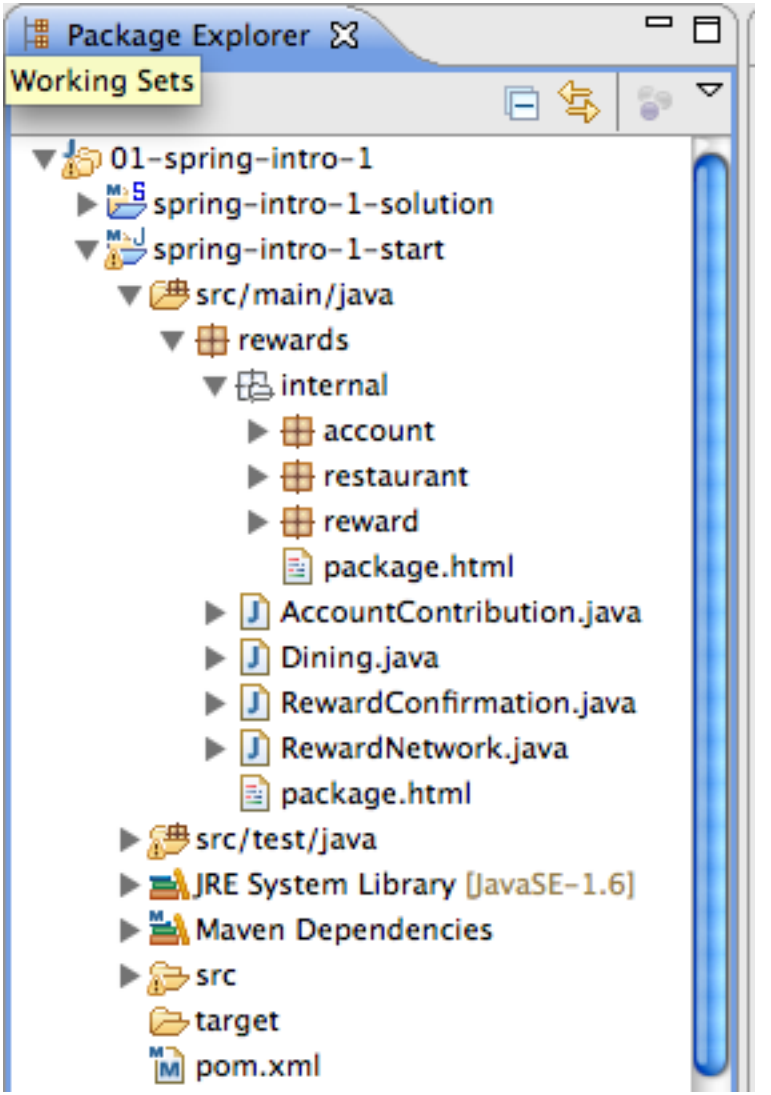
C.2. Package Explorer View

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

Figure C.1. Package explorer panel (package view)



The hierarchical view shows nested packages in a tree view

Figure C.2. Package explorer panel (hierarchical view)

C.3. Add Unimplemented Methods

If the cursor is on a line with an error, `CTRL+1` (use `COMAND+1` on a Mac) may popup a list of quick fixes

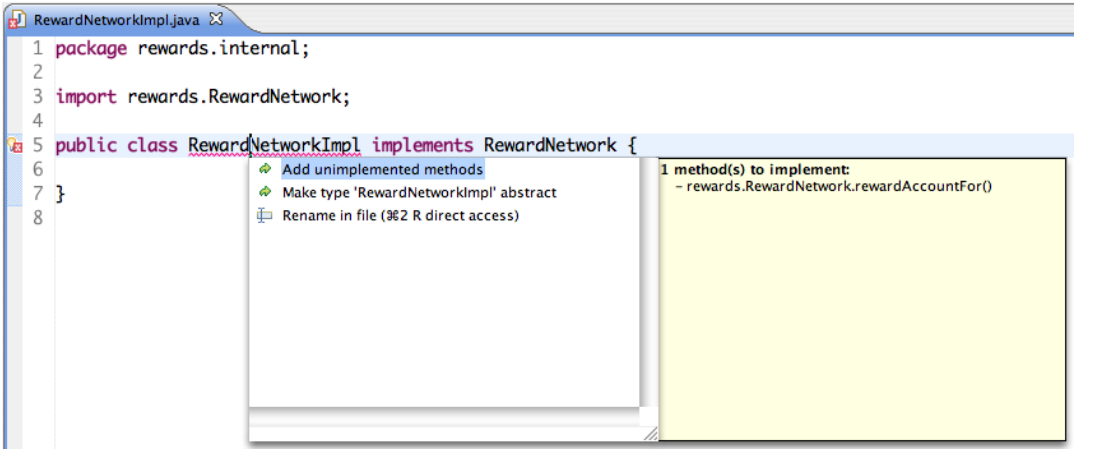


Figure C.3. "Add unimplemented methods" quick fix

C.4. Field Auto-Completion

Note that auto-completion in Eclipse and STS is `CTRL+Space`, even on a Mac.

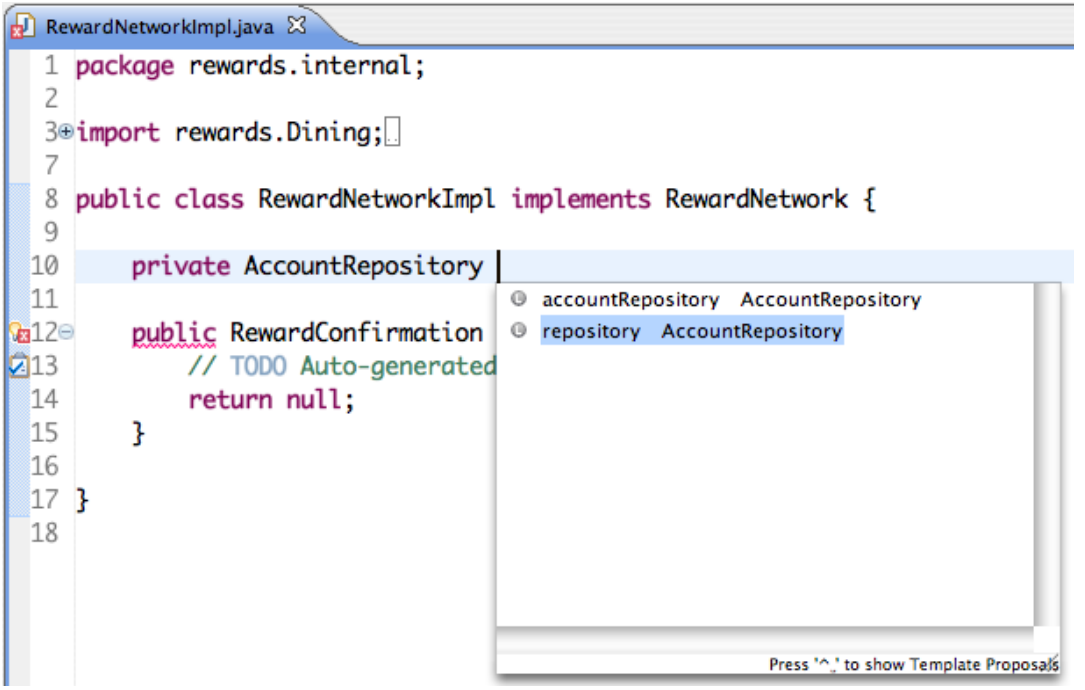


Figure C.4. Field name auto-completion

C.5. Generating Constructors From Fields

Right-click in a class and select "Source" on the popup menu. Alternatively use ALT-SHIFT-S (ALT-COMMAND-S on a Mac). Many options to generate source, including constructors.

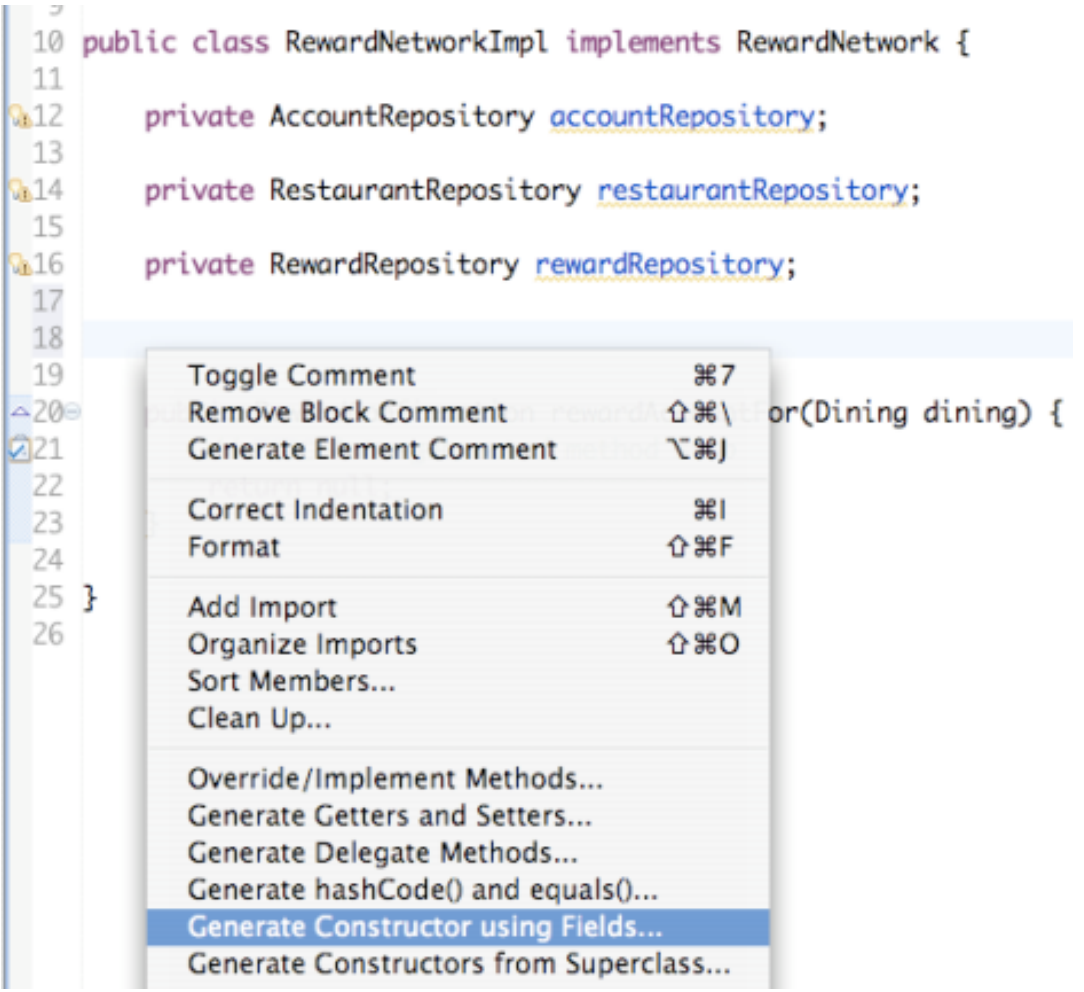


Figure C.5. Generate Constructor using Fields

C.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.

C.7. Tasks View

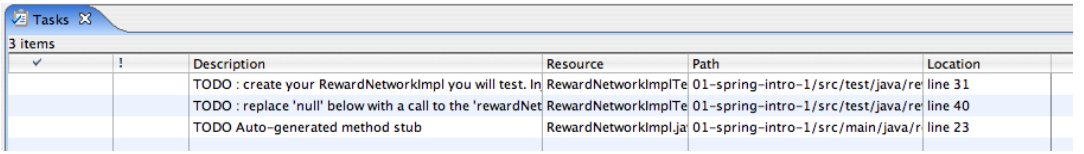


Figure C.6. The tasks view in the bottom right page area

To display the Tasks view, select the `Window` menu, then `Show View`. If `Tasks` is not in the list, select `Other ...` and then open the `General folder`. Select `Tasks` from the folder contents. You do not want the view called "`Task List`" (which is from the MyLyn plugin).

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.

C.8. Rename a File

It is not immediately obvious how to do this - it is actually a refactoring exercise.

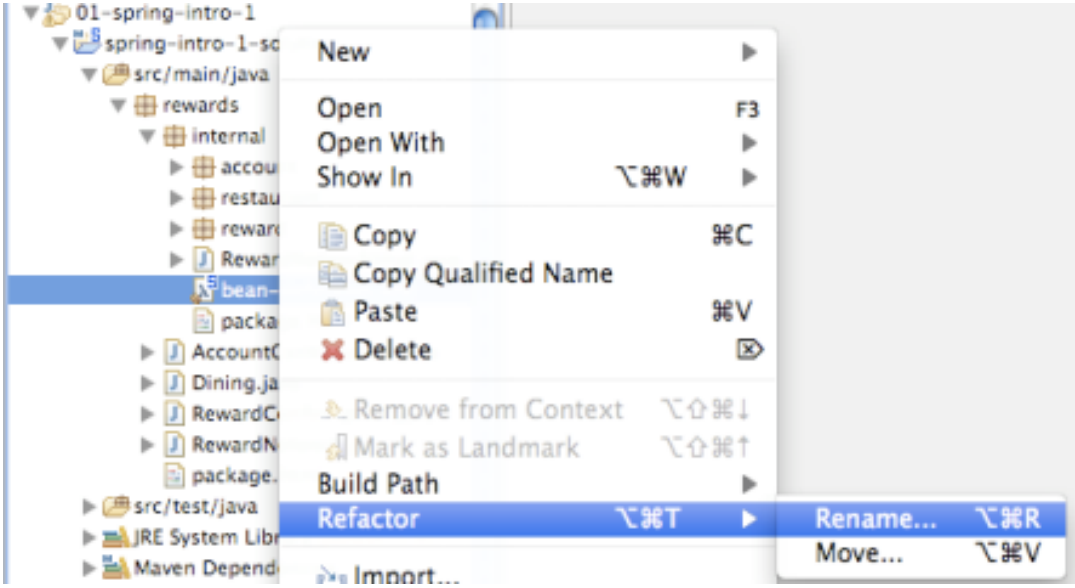


Figure C.7. Renaming a Spring configuration file using the Refactor command

Appendix D. Using Web Tools Platform (WTP)

D.1. Introduction

This section of the lab documentation describes the general configuration and use of the Web Tools Platform plugin for Eclipse to run applications on Tomcat or Pivotal tc Server. You will need to do this to run the course labs and samples.

D.2. Verify and/or Install the Server

D.2.1. Does A Server Exist?

The *Servers* view provided by the WTP plugin needs to be open, so that you can see the status of any existing servers. Verify that you can see the Servers view. The tab for this view will typically be at the bottom, either in the bottom-left hand corner or with other views such as *Problems*, *Progress* and *Console*. If the view is not open, open it now via the '*Window -> Show View -> Other ... -> Server -> Servers*' menu sequence.

1. Your workspace may already contain a pre-created entry for a Tomcat server instance, visible in the *Servers* view as '*Tomcat XX Server at localhost*' where XX is the version number. If it does, skip ahead to Section D.3, “Starting & Deploying the Server”
2. Alternatively there may already be a tc Server instance in the Servers view, called something like '*Pivotal tc Server v3.x*' or perhaps the older '*tc Server VMware vFabric tc Server V2.5 - V2.9*'. Again, skip ahead to Section D.3, “Starting & Deploying the Server”
3. Otherwise, you will need to install a new server runtime.

Since there are no servers at all, there is a link to click to create one. Click on that link now.

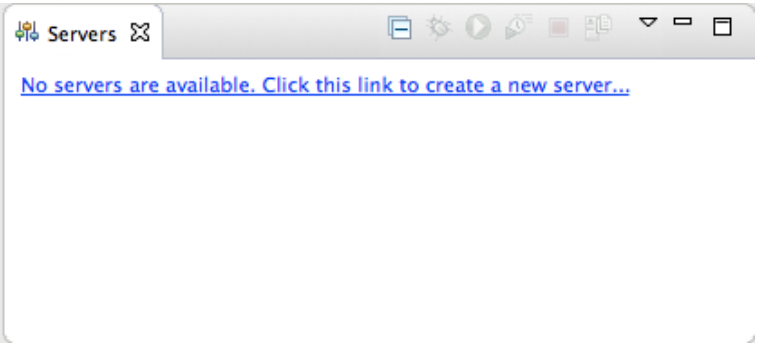


Figure D.1. Create a New Server

D.2.2. Creating a New Server Instance

The popup that appears has a long list of server products - the first in the list is Apache (for Tomcat) or, if you have it, you could install tc Server (this will be in the list under Pivotal or VMware depending on what brand is current when you take this course).

Note

The 'New Server' dialog supports tc Server right back thru version 2.5 to 2.9, when it was VMware branded, back to early versions 2.0/2.1, when it was SpringSource branded. Choose '*Pivotal tc Server v3.x*" if available, or '*tc Server VMware vFabric tc Server 2.x*' otherwise.

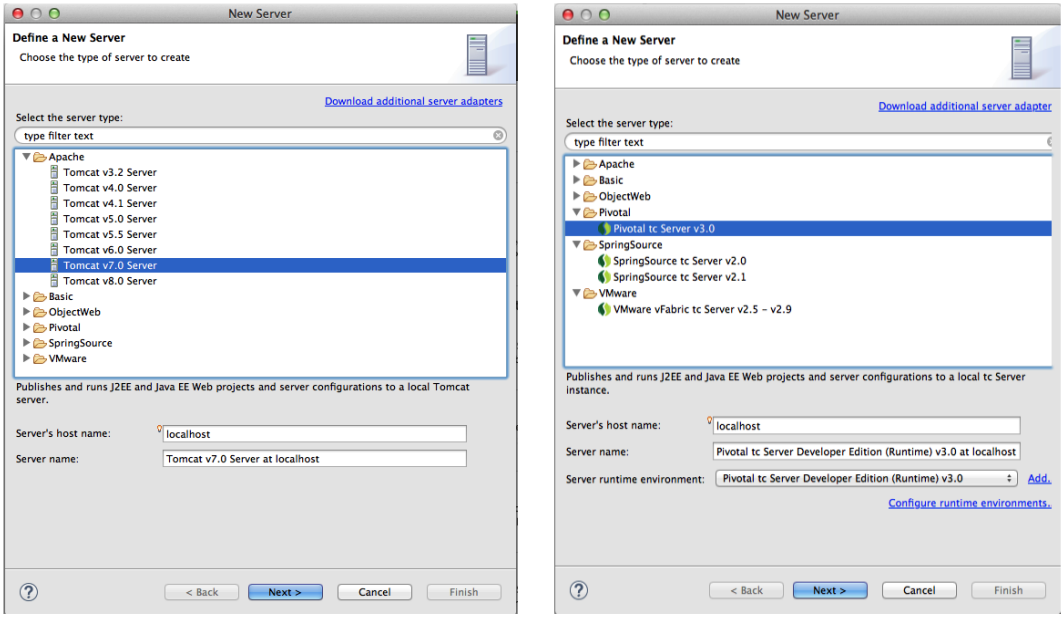


Figure D.2. Create a New Server

Either Apache Tomcat 7 (or later) or tc Server 3.0 (or later) can be used. Check in the course installation folder/directory to see what is provided (You should find a Tomcat or tc Server sub-directory):

The default course installation folder is:

- MS Windows: `C:\<course-name>`
- MacOS: `/Applications/<course-name>`
- Linux: `/home/<user-name>/<course-name>`

Pick the right server-type and follow the instructions to create a new server.

Option I - Creating a Tomcat Server

If Apache Tomcat is bundled with your course, perform the following steps. tc Server users, please skip to the next section

1. Select the Tomcat version you wish to use (you need at least Tomcat 7 for Servlet 3 support). If you aren't sure, use the latest version. If the `Finish` button is enabled, click it and you should be done - skip the next step.

2. However if `Finish` is disabled, click `Next`. You now need to tell STS where to find a Tomcat installation, so you will see this dialog:

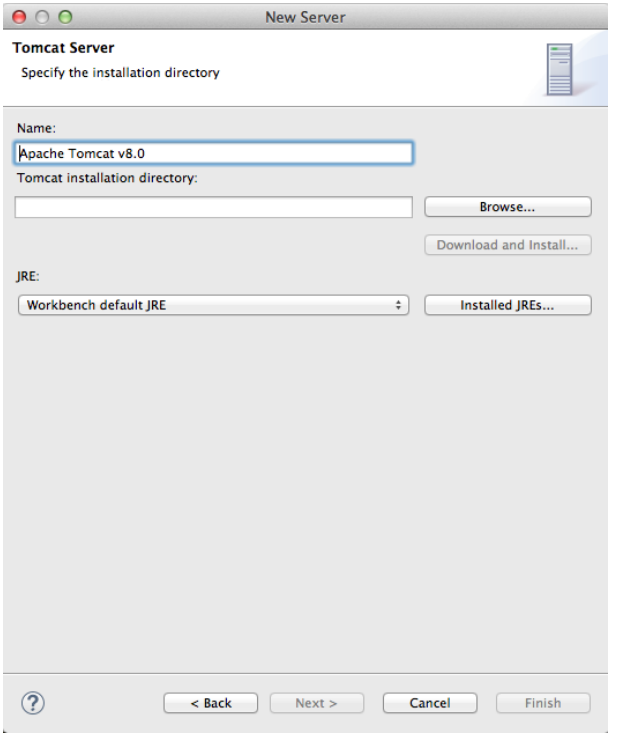


Figure D.3. Locate Tomcat

Tomcat *should* have been installed with the rest of the lab materials. Click on the `Browse...` button and locate the Tomcat installation. By default the file-explorer dialog opens in the current workspace directory. Your tomcat installation should be in an adjacent directory. Click `Open` to select the Tomcat directory and, when you return to the '*New Server*' dialog, click `Finish`.

3. Finally verify that a server runtime has appeared in the *Servers* view.
If you have not done this before, can't find Apache Tomcat or have any other problems, ask your instructor for help. Once this is setup, you won't have to do it again.

Skip to Section D.3, “Starting & Deploying the Server” below.

Option II - Creating a tc Server Instance

If Pivotal (or VMware) tc Server is bundled with your course, perform the following steps. Tomcat users, skip to Section D.3, “Starting & Deploying the Server” below.

1. Select the tc Server version you wish to use (the latest version in the Pivotal group is preferred). Failing that, select '*tc Server VMware vFabric tc Server V2.5 - V2.9*' in the VMware group. If the `Finish` button is enabled, click it and you should be done. Skip to the last of these steps.

2. However if `Finish` is disabled, click `Next`. You now need to tell STS where to find a tc Server installation, so you will see this dialog:

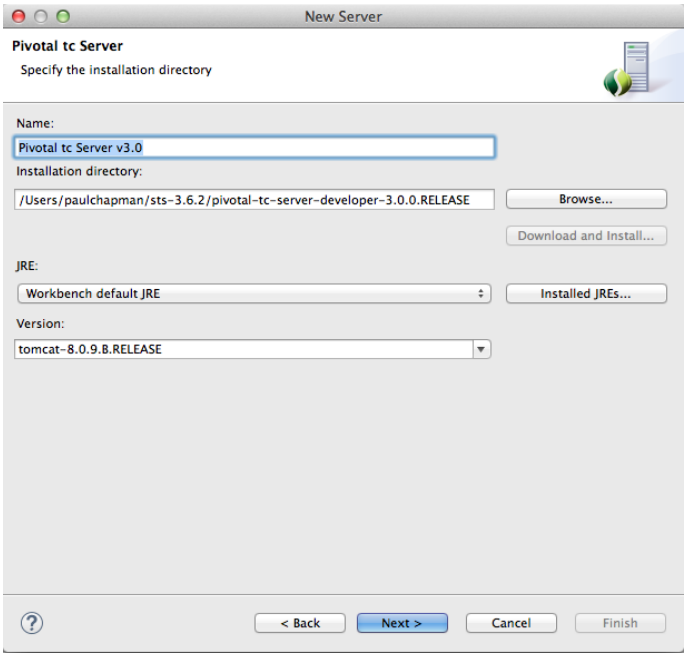


Figure D.4. Locate tc Server Installation

tc Server *should* have been installed with the rest of the lab materials. Click on the `Browse...` button and locate the tc Server installation. By default the file-explorer dialog opens in the current workspace directory. Your tc Server installation should be in an adjacent directory. Click `Open` to select the Tomcat directory and,

when you return to the '*New Server*' dialog, you should find the Version: field has been filled in automatically to show the Tomcat version tc Server is using. Click `Next`.

3. tc Server requires the instance to be created on disk as well as in STS. In the next dialog (on left in diagram below), select '*Create new instance*' and click `Next`

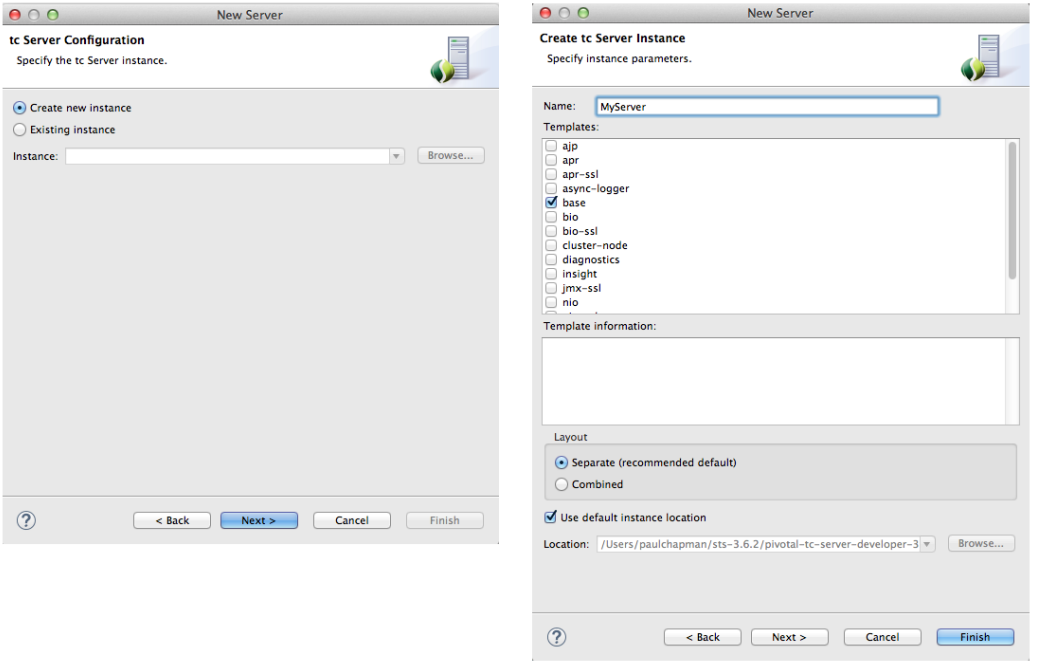


Figure D.5. New tc Server Instance

4. In the next dialog (on right in diagram above), give the instance a name (here we have used `MyServer`) and under templates select `base`. Retain the other defaults (*Separate* under Layout and *Use default instance location*). Click `Finish` to create the server in STS

5. Finally verify that a server runtime has appeared in the *Servers* view.
   If you have not done this before, can't find tc Server or have any other problems, ask your instructor for help. Once this is setup, you won't have to do it again.

D.3. Starting & Deploying 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*'.

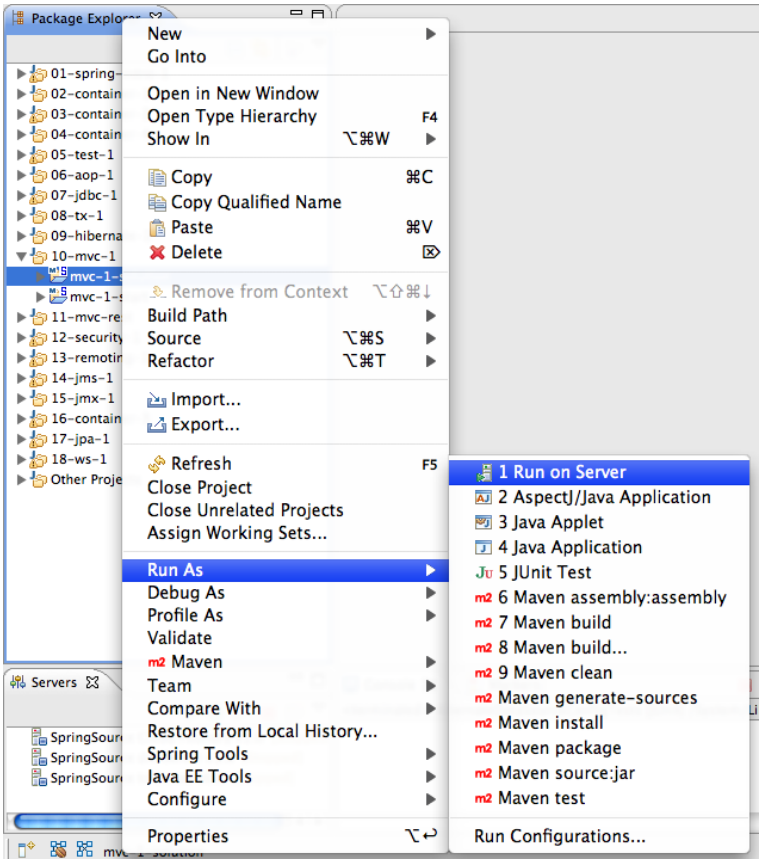


Figure D.6. Run On Server

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

Note

Tomcat or tc Server will not fail to start even if your project fails to load. The server will run, but your application cannot be accessed because it is not running.

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

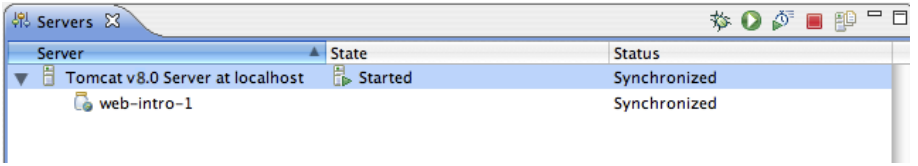


Figure D.7. Running On Server

Tip

Once you have deployed the first time, the server can be shut down (stopped by pressing the red box in the toolbar of the Servers tab. It can be started again using the green arrow button.

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. Changes to Java code will force the server to restart automatically (may take a few seconds).
However, changes to Spring Application Context definition files will require stopping and restarting the server manually 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. If you close it, you can get it back using the world icon in the STS main toolbar (you must be in the Spring Perspective). Alternatively, open the URL in an external browser (such Firefox or Chrome). You will need to use an external browser to access tools like Firebug or Web Developer.

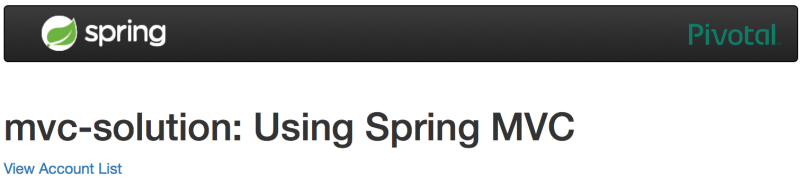


Figure D.8. Sample Application in Browser

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*'.

Tip

If you use `Run As -> Run On Server` the first time for each new project you can unselect the old project at the same time. After that use the stop and start buttons in the Servers view.

D.4. Adding More Servers

More servers can be added at any time. To do so, right-click in the blank area in the Servers View. There is a New option and, if you select it, Server is the only option.

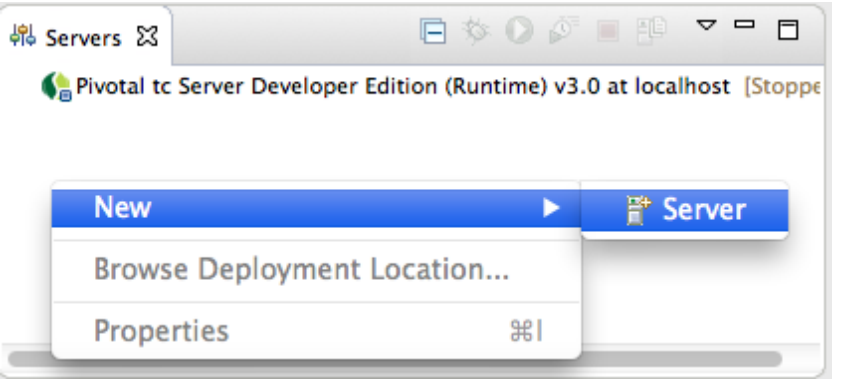


Figure D.9. Show in Browser

This will popup the `*New Server*` dialog and you can continue as described in Section D.2.2, “Creating a New Server Instance” above.

