spring

Enterprise Spring

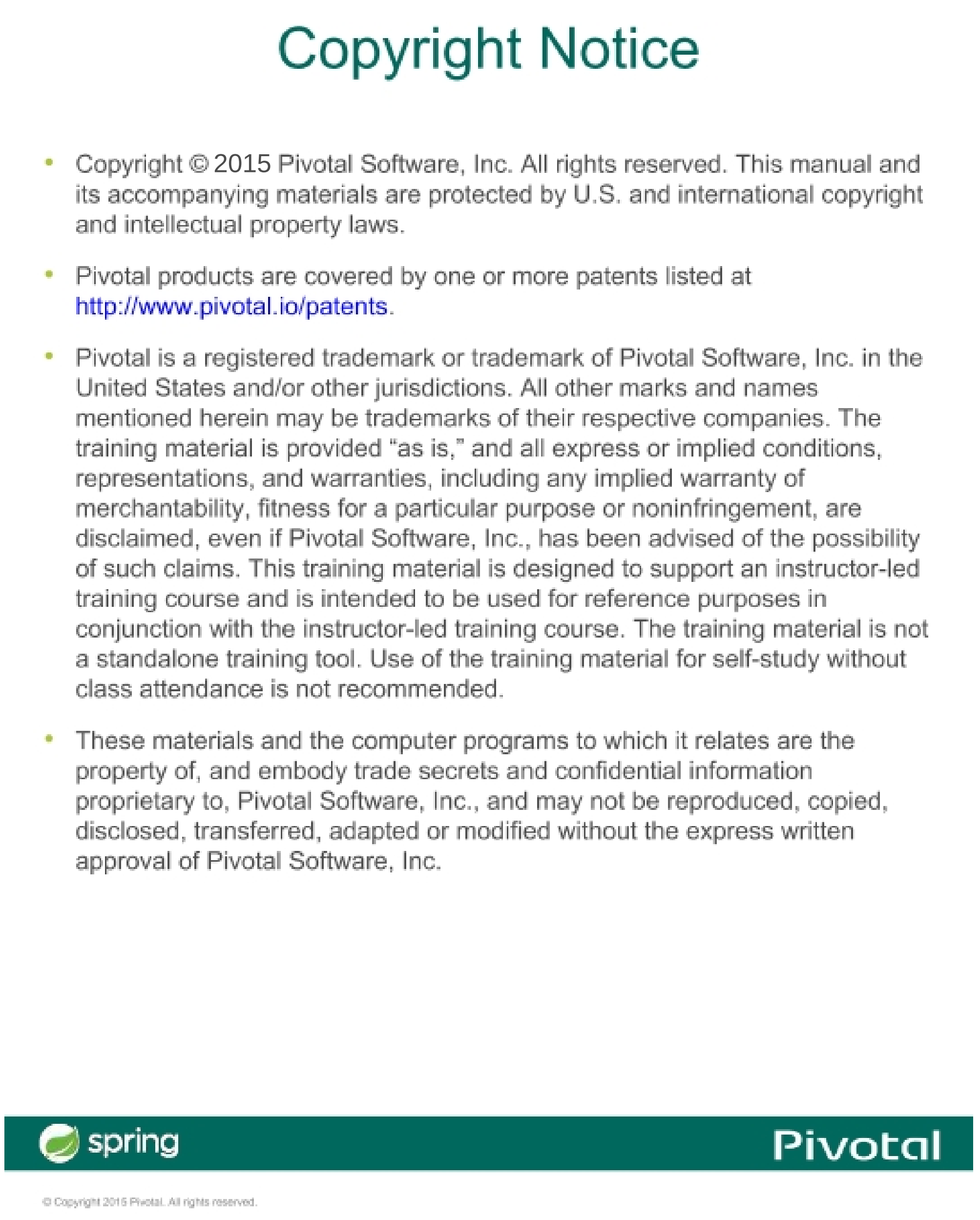
Lab Instructions

Building loosely coupled event-driven architectures

Version 4.2.a

spring Pivotal

ved.



Copyright Notice

This Page Intentionally Left Blank

Table of Contents

Reward Dining: The Course Reference Domain ... viii
- 1. Introduction ... viii
- 2. Domain Overview ... viii
- 3. Reward Dining Domain Applications ... x
  - 3.1. The Rewards Application ... x
- 4. Reward Dining Database Schema ... xiv

1. ei-course-intro: Introduction to Enterprise Spring ... 1
- 1.1. Introduction ... 1
- 1.2. Quick Instructions ... 1
- 1.3. Instructions ... 2
  - 1.3.1. Getting Started with the Spring Tool Suite ... 2
  - 1.3.2. Understanding the 'Reward Network' Application Domain and API ... 5

2. tasks: Tasks and Scheduling ... 17
- 2.1. Introduction ... 17
- 2.2. Quick Instructions ... 17
- 2.3. Instructions ... 18
  - 2.3.1. Refactoring Legacy Concurrent Batch Processing ... 18
  - 2.3.2. Automating Batch Processing ... 20

3. rest-intro: Introduction to RESTful Applications with Spring ... 22
- 3.1. Introduction ... 22
- 3.2. Quick Instructions ... 22
- 3.3. Instructions ... 24
  - 3.3.1. Use Case ... 24
  - 3.3.2. Inspecting the Existing Application Configuration ... 24
  - 3.3.3. Testing the Controller using a *RestTemplate* ... 27
  - 3.3.4. PART 2: Out-of-container testing ... 29

4. jms: Simplifying Messaging with Spring's JMS Support ... 30
- 4.1. Introduction ... 30
- 4.2. Quick Instructions ... 30
- 4.3. Instructions ... 31
  - 4.3.1. Sending Dining Messages to a Destination ... 31

5. jms-tx: Add Transactional Capabilities to the JMS Interaction ... 35
- 5.1. Introduction ... 35
- 5.2. Quick Instructions ... 35
- 5.3. Instructions ... 36
  - 5.3.1. Inspecting and Running the Current Application ... 36

5.3.2. Introducing errors in the application 42
6. tx-xa: Using Global JTA Transactions 47
6.1. Introduction: Using Global JTA Transactions for Atomic Behavior Across Multiple Transactional Resources 47
6.2. Quick Instructions 47
6.3. Instructions 48
6.3.1. Introducing JTA to have a global transaction in place for the *DiningListener* 49
6.3.2. Reviewing the result of the same errors under a global transaction 51
7. si-intro: Refactoring from JMS to Spring Integration 53
7.1. Introduction 53
7.2. Quick Instructions 53
7.3. Instructions 54
7.3.1. Inspect the current application 54
7.3.2. Refactor to Spring Integration 54
7.3.3. Log incoming Dining messages using a wire tap 55
7.3.4. Send emails for confirmations 55
7.3.5. Test the mail sending 56
7.3.6. BONUS STEP 1: Refactor the client as well 56
7.3.7. BONUS STEP 2: Simplify the Spring Integration configuration 57
8. si-configuration: Using an Idempotent Receiver and Filter invalid Dinings 58
8.1. Introduction 58
8.2. Instructions 58
8.2.1. Part 1. Make the RewardNetwork Idempotent 59
8.2.2. Part 2. Filtering broken messages 61
8.2.3. BONUS: Java configuration refactoring 63
9. si-advanced: Splitter and File System Integration 65
9.1. Introduction 65
9.2. Instructions 65
9.2.1. Part 1. Add XML Marshalling to the Input and Output 65
9.2.2. Part 2. Add the Ability to Deal with Batches of Dinings 66
9.2.3. Part 3. Add File System Integration as Input for the Dining XML 67
9.2.4. BONUS: Refactor the idempotency implementation 68
10. batch-intro: Introduction to Spring Batch 70
10.1. Introduction 70
10.2. Instructions 70
10.2.1. Use Case 70
10.2.2. Implementing the First Step and Defining the Job 71
10.2.3. Implementing the Second Step 74
11. batch-restart-recovery: Spring Batch Restart and Recovery 78
11.1. Introduction 78
11.2. Instructions 78
11.2.1. Completing the initial batch job 78

11.2.2. Skipping in case of errors 80
11.2.3. Restarting a failed job 82
11.2.4. Limiting the number of (re)starts allowed 84
11.2.5. BONUS: Java configuration refactoring 85
12. batch-admin: Spring Batch Admin 87
12.1. Introduction 87
12.2. Instructions 87
12.2.1. Exploring the Batch Admin UI 87
12.2.2. Stopping and Restarting a Batch Job 92
12.2.3. Configuring a Partitioned Batch Job 92
13. Introduction to Spring XD 95
13.1. Introduction 95
13.2. Installation (PART 1) 95
13.2.1. Generic Installation 95
13.2.2. Installation using Homebrew on Mac OSX 96
13.2.3. Installation on RedHat/CentOS 96
13.2.4. Verifying the Installation 96
13.3. Introduction to Streams (PART 2) 98
13.3.1. A Simple Time Logger 98
13.3.2. Using Processors 99
13.3.3. Solutions 100
13.4. Introduction to Jobs (PART 3) 102
13.4.1. Installing and Running a Basic Batch Job 102
14. amqp: Simplifying Messaging with Spring's AMQP Support 104
14.1. Introduction 104
14.2. Prerequisites 104
14.3. Quick Instructions 105
14.4. Instructions 106
14.4.1. Sending Dining Messages to a Destination 106
15. remoting: Distributing an application using RMI and Spring Remoting 109
15.1. Introduction 109
15.2. Quick Instructions 109
15.3. Instructions 110
15.3.1. Exporting and Consuming the Remote Service 110
15.4. Bonus 115
15.4.1. Bonus #1: Break the Application 115
15.4.2. Bonus #2: Make the Application More Robust 115
16. soap-ws: Exposing SOAP Endpoints using Spring Web Services 117
16.1. Introduction 117
16.2. Quick Instructions 117
16.3. Instructions 118
16.3.1. Defining the Message Contract 118

16.3.2. Step 4: Generate the classes with JAXB2 120
16.3.3. Exporting the *RewardNetwork* as a SOAP Endpoint 120
16.3.4. Bonus #1: Find the WSDL 124
16.3.5. Bonus #2: Plain Old XML (POX) 124
17. ws-advanced: Advanced Spring Web Services 125
17.1. Introduction 125
17.2. Quick Instructions 125
17.3. Instructions 125
17.3.1. Verifying successful handling of valid requests 126
17.3.2. Adding validation for incoming requests 127
17.3.3. Mapping certain exceptions to certain SOAP faults 128
A. Spring XML Configuration Tips 130
A.1. Bare-bones Bean Definitions 130
A.2. Bean Class Auto-Completion 130
A.3. Constructor Arguments Auto-Completion 130
A.4. Bean Properties Auto-Completion 131
B. Eclipse Tips 132
B.1. Introduction 132
B.2. Package Explorer View 132
B.3. Add Unimplemented Methods 133
B.4. Field Auto-Completion 134
B.5. Generating Constructors From Fields 135
B.6. Field Naming Conventions 136
B.7. Tasks View 136
B.8. Rename a File 136
C. Using Web Tools Platform (WTP) 138
C.1. Introduction 138
C.2. Verify and/or Install the Tomcat Server 138
C.3. Starting & Deploying to the Server 138

Reward Dining: The Course Reference Domain

1. Introduction

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

2. Domain Overview

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

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

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

3. Reward Dining Domain Applications

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

3.1. The Rewards Application

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

1. When they are hungry, members dine at participating restaurants using their regular credit cards.

2. Every two weeks, a file containing the dining credit card transactions made by members during that period is generated. A sample of one of these files is shown below:

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

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

3.1.1. Public Application Interface

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

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

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

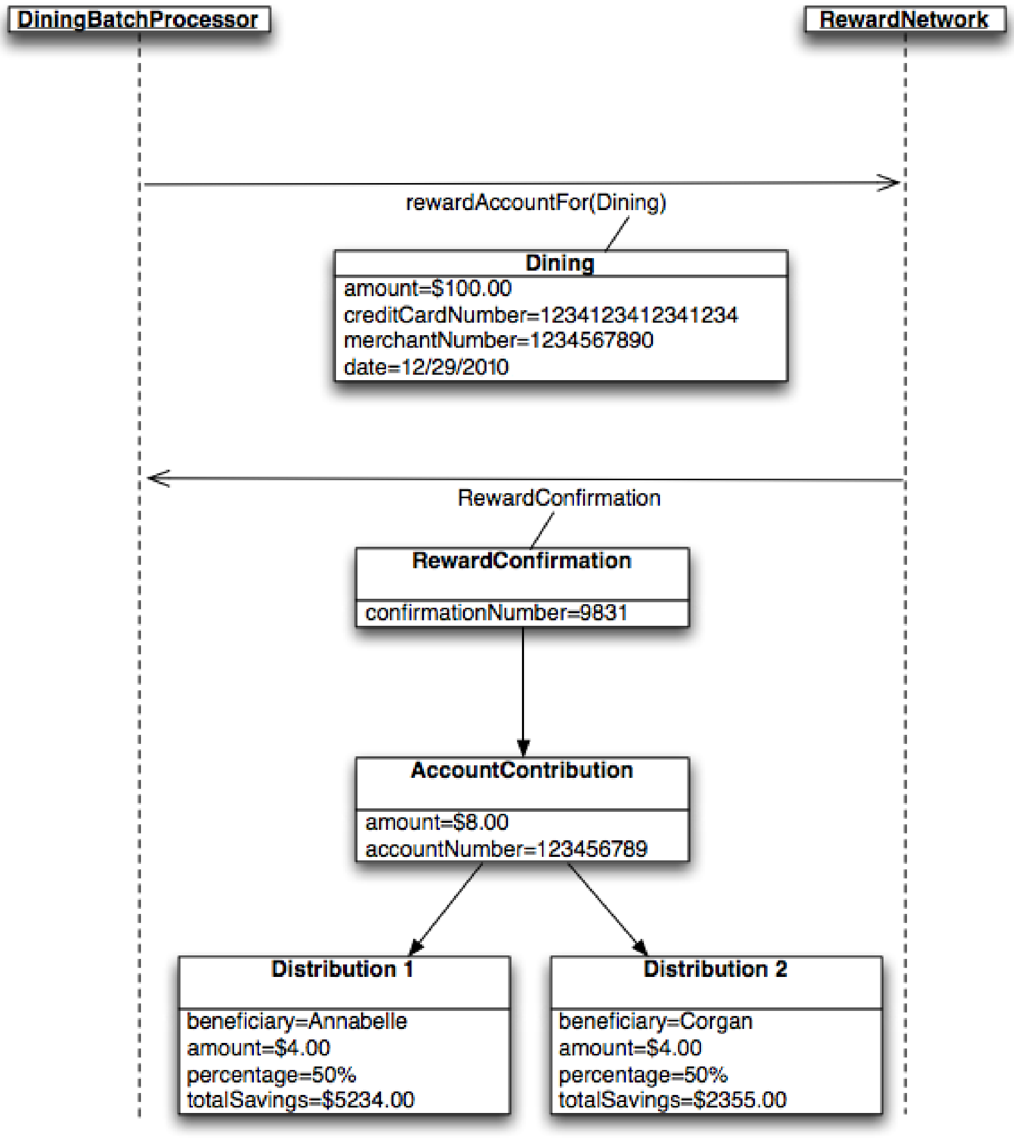


Figure 3: A client calling the `RewardNetwork` to reward an account for dining.

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

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

3.1.2. Internal Application implementation

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

This flow is shown below:

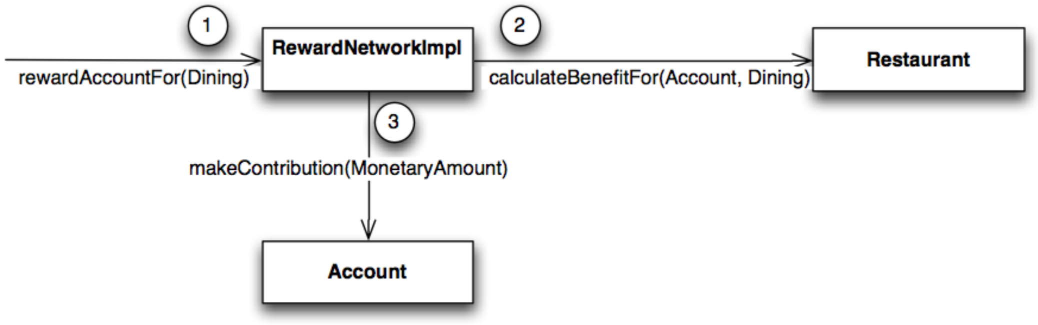

1
RewardNetworkImpl
rewardAccountFor(Dining)
2
Restaurant
calculateBenefitFor(Account, Dining)
3
makeContribution(MonetaryAmount)
Account

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

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

3.1.3. Supporting RewardNetworkImpl Services

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

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

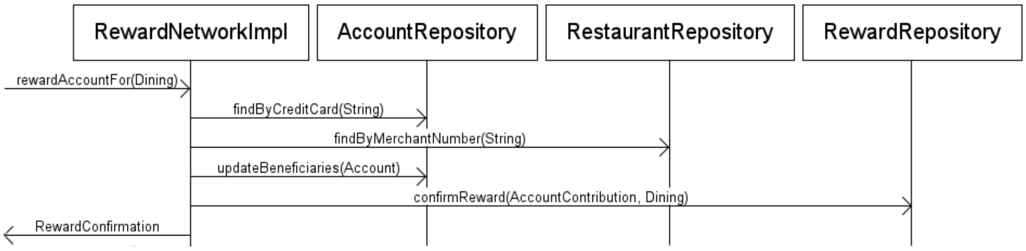


Figure 5: The complete `RewardNetworkImpl` `rewardAccountForDining(Dining)` sequence

4. Reward Dining Database Schema

The Reward Dining applications share a database with this schema:

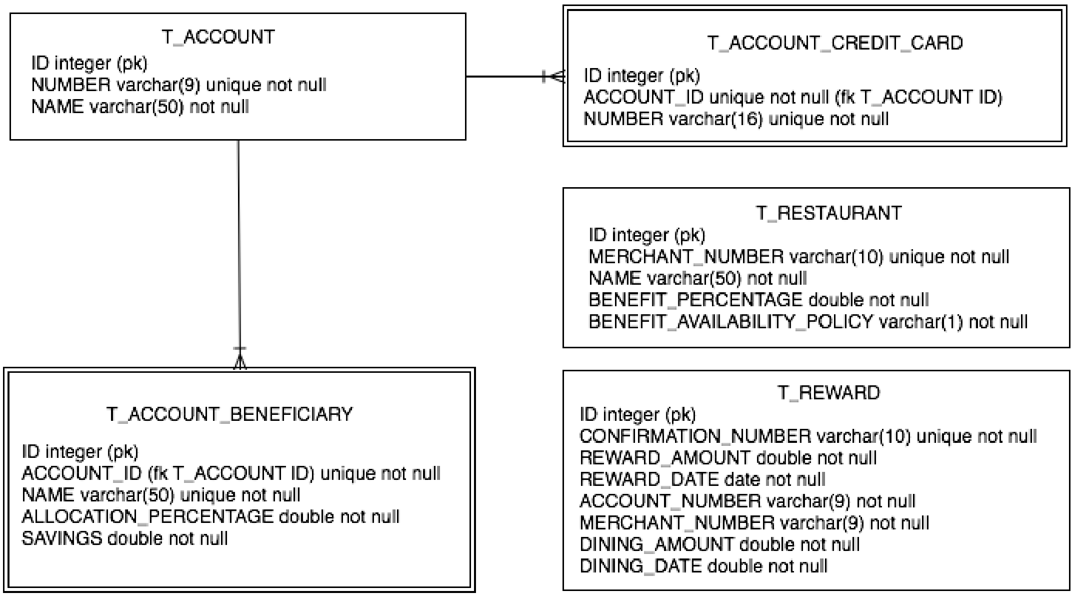

T_ACCOUNT
ID integer (pk)
NUMBER varchar(9) unique not null
NAME varchar(50) not null
T_ACCOUNT_CREDIT_CARD
ID integer (pk)
ACCOUNT_ID unique not null (fk T_ACCOUNT ID)
NUMBER varchar(16) unique not null
T_RESTAURANT
ID integer (pk)
MERCHANT_NUMBER varchar(10) unique not null
NAME varchar(50) not null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

Figure 6: The Reward Dining database schema

Chapter 1. ei-course-intro: Introduction to Enterprise Spring

1.1. Introduction

Welcome to *Enterprise 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.

The lab is broken into four major sections. In the first section you'll get up and running with the Spring Tool Suite and get to know the project structure and Maven-based build system.

In the second section, you'll get an overview of the 'Reward Network' application domain and API that will be used throughout the course.

In the third section, you'll finish testing and implementing a simple command-line user interface for the Reward Network application.

In the fourth section, you'll monitor UI application activity using Spring's support for JMX.

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

What you will learn

1. Basic features of the Spring Tool Suite

2. Core *RewardNetwork* domain and API

3. Review Spring JMX features and usage of JConsole

Estimated time to complete: 45 minutes

1.2. Quick Instructions

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

1. **Getting Started with the Spring Tool Suite (STS)** (details)

   Launch STS. Browse through the working sets and projects to make yourself familiar with the Tool Suite, and how the lab projects that you will be completing in this course are structured. You might also have to configure STS to view the TODOs in the Task pane.

2. **Understanding the 'Reward Network' Application Domain and API** (details)

   Please review *Reward Dining: The Course Reference Domain* in the preface to this handout for understanding the background of the Reward Network application domain. Once completed, execute the following steps:

   a. Visualize RewardNetwork bean graph with STS (navigate to 00-reward-network->rewards/src/main/resources and open app-config.xml).

   b. Follow TODO 01 and run the DiningEntryUI application. Enter the following details

      Dining Amount: 100

      Member credit card number: 1234123412341234

      Merchant number: 1234567890

      Would you like to enter another Dining transaction? [y/n]: n

   c. Complete TODO 02 - 03 and execute the JUnit test. The test will fail.

   d. Complete TODO 04 and execute the JUnit test. This time the test will pass.

   e. Redo TODO 01 to verify that the interactive dining entry works as expected. It should return a confirmation now.

   f. Complete TODO 05 - 07 to apply JMX metadata to DiningEntry. (details)

   g. Complete TODO 08 to enable automatic registration of DiningEntry as a JMX MBean. (details)

   h. Complete TODO 09 by launching DiningEntryUI and monitoring its statistics via JConsole. (details)

Congratulations, you've completed this lab!

1.3. Instructions

1.3.1. Getting Started with the Spring Tool Suite

The Spring Tool Suite (STS) is a free and enterprise-grade IDE built on the Eclipse Platform. In this section, you will become familiar with the Tool Suite and how the lab projects that you will be completing in this course are structured.

1.3.1.1. Step 1: Launch the Tool Suite

Launch the Spring Tool Suite by using the shortcut link on your desktop or from the *sts-xxx* directory where you installed the course content. You will see the STS splash image appear.

1.3.1.2. Understanding Eclipse/STS project structure

Tip

If you've just opened STS, it may still be 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.

1.3.1.3. Step 2: Browse Working Sets and projects

If it is not already open, expand the *01-ei-course-intro* Working Set. Within you'll find two projects: *ei-course-intro* and *ei-course-intro-solution*. This pair of projects is a common pattern throughout the labs in this course. Clearly one is the solution which we have completed for you, the other is the one you will work on - the *working* project. For each lab, you will follow the directions to complete the working project, and when finished, you can check your work against the *solution* to compare your implementation to that of the course designers.

All lab projects are imported as plain Eclipse projects, even the Maven POM file for each project is present. You can have a look at the POM files if you want, but don't modify them! Let's have a look to the dependencies of the projects.

Open the *working* project (*ei-course-intro*) and expand its *Referenced Libraries* node. Look at the dependencies. 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.

Note

Note that all dependencies have been pre-installed as part of the training environment, so no network access is required to resolve and download jar files. This is accomplished by setting the

M2_REPO classpath variable to point to the directory where the repository is located.

If you examine the project's build path, you will also notice that the *ei-course-intro* project has a dependency on the `rewards` project contained in the `00-reward-network` Working Set. In the next section you'll get to know the `rewards` project in detail.

1.3.1.4. Step 3: Configure the TODOs in STS

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:

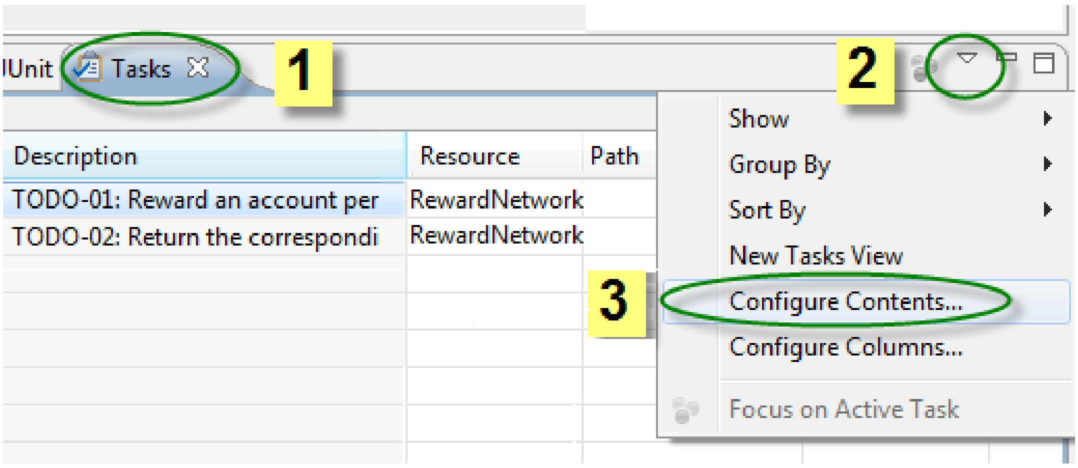
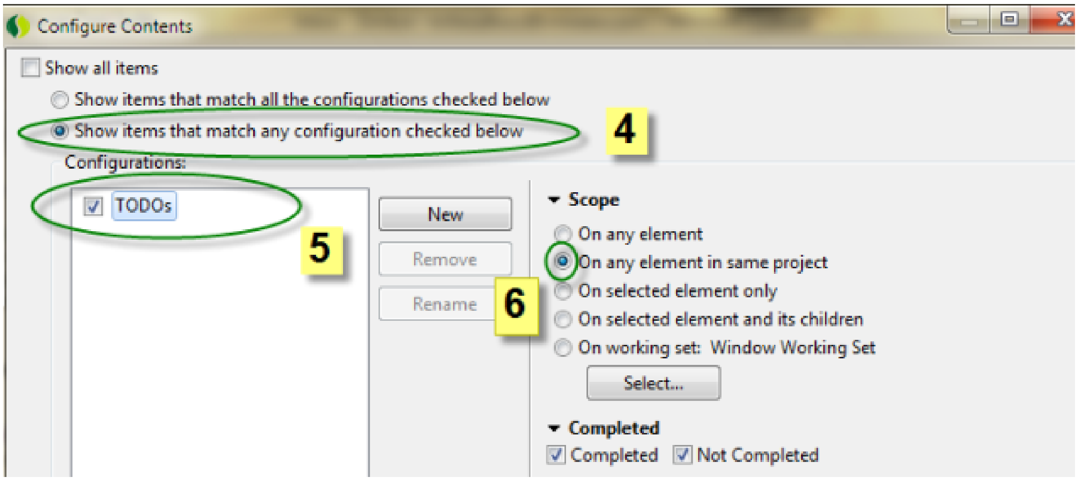


Figure 1.1. Configure TODOs



[ ] Show all items

( ) Show items that match all the configurations checked below

(●) Show items that match any configuration checked below

Configurations:

[✓] TODOs

Scope

( ) On any element

(●) On any element in same project

( ) On selected element only

( ) On selected element and its children

( ) On working set: Window Working Set

Completed

[✓] Completed [✓] Not Completed

Figure 1.2. 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.

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

Please review *Reward Dining: The Course Reference Domain* in the preface to this handout if you haven't done so already. This document will guide you through understanding the background of the Reward Network application domain and thus provide context for the rest of the course. There are no tasks for you to complete within this document, it is simply for your understanding. Once you are finished reading, return here and carry on with the rest of this section.

1.3.2.1. Step 1: Review the *RewardNetwork* business classes

In the Package Explorer view, navigate to *00-reward-network->rewards/src/main/resources* and open *app-config.xml*. Take a moment to review this Spring configuration file and notice that it configures a bean of type *RewardNetworkImpl* that depends on three *Repository* beans for data access.

We will make extensive use of this simple POJO application layer throughout this course. No matter what the integration task, this portable business and data access logic will remain the same.

1.3.2.2. Step 2: Review *DiningEntryUI* application

Open (*CTRL-SHIFT-T*) *rewards.ui.DiningEntryUI* from within the *ei-course-intro* project.

Notice that this very simple class consists of nothing more than a *main()* method that bootstraps a Spring *ApplicationContext* and invokes a `start()` method on a `DiningEntry` bean instance. In a moment, we'll run this application and interact with it using the console.

First, take a look at the Spring XML consumed by the *ApplicationContext*. Open (*CTRL-SHIFT-R*) *diningentry-config.xml* in the *ei-course-intro* project.

Notice that this file consists of two *<import>* statements and a bean definition that wires up a *DiningEntry* bean. The *DiningEntry* gets injected with the very same *RewardNetworkImpl* bean that we saw in the original *app-config.xml* file.

1.3.2.3. Step 3: Launch *DiningEntryUI* command-line application

Let's see what happens if we run the application in its current state.

Open (*CTRL-SHIFT-T*) the *DiningEntryUI* class once again. Right-click anywhere in the source file and navigate to *Run As->Java Application* (or use *ALT-SHIFT-X + J*). You should see the following at the console:

```
Welcome to the Reward Network dining entry UI

Please enter the following information to create and
                      process a new dining transaction

Dining amount:
```

Type *100* for the dining amount and hit Enter. Next, you'll be prompted for

```
Member credit card number:
```

Type *1234123412341234* and Enter. Finally you'll be asked for:

```
Merchant number:
```

Type *1234567890* and Enter.

You should now see:

```
Dining of $100.00 charged to '1234123412341234'
                      by '1234567890' on 4/24/09 12:00 AM

Rewarding account for dining via RewardNetwork...
Result: null

Would you like to enter another Dining transaction? [y/n]:
```

Notice the *Result: null* above? It appears that there is something wrong with the implementation.

Enter '*n*' to end the process and then move on to the next step where you'll diagnose and fix the problem.

1.3.2.4. Step 4: Test *DiningEntry*

After the last step, you know there's something wrong with the *DiningEntryUI* application that is resulting in a *null* value being printed out where there should be an indicator of a successfully rewarded dining transaction.

While it would be easy enough to begin browsing the implementation source and trying to find and fix the problem, the best thing to do is ensure that this problem never happens again with a unit test.

You're in luck, because just such a unit test has already been started for you. Your job in this step will be to finish it, and in the process, fix the *null* output above.

Note

Throughout this course, unit tests will be used to drive the process of completing labs. This promotes the best practice of unit testing in general and is a great way to let you know that you've successfully completed a given step. JUnit is the framework of choice, as it has wide adoption and great tooling support.

Take a moment now to locate *rewards.ui.DiningEntryTests* in the *src/test/java* source folder of the *ei-course-intro* project. Right click on that class from the Package Explorer view and select *Run As->JUnit Test* (or use *ALT-SHIFT-X + T*).

Notice that the bar is green? This is because the tests are not yet finished. Your first job is to complete the tests and make them fail.

Take a look at the source for *DiningEntryTests*, and in particular take a moment to review the *testValidDiningEntry()* method. It is heavily commented to aid your understanding.

The idea is to test what happens when a user enters perfectly valid information about a dining transaction. That information should be accepted by the program and turned into a *Dining* object. Then, that *Dining* object should be passed to the *RewardNetwork* service's *rewardAccountFor(Dining)* method to be processed.

Note

Understanding Mockito: As mentioned in the slide presentation, Mockito will be used as a mocking framework throughout this course. Mocking is a technique that greatly simplifies and speeds the process of unit testing.

Notice how in *testValidDiningEntry()* a *RewardNetwork* mock is created and given instructions about how to behave under current conditions.

A *DiningEntry* object is created and supplied with an *InputStream* containing valid dining transaction input. Finally, *diningEntry.start()* is called in order to begin processing the supplied user input. This 'turns on' the *DiningEntry* object for interactive use.

Complete the test by uncommenting the call to *Mockito.verify()* (*TODO 02*)

Tip

The "Tasks" view in the bottom half of the IDE shows all TODOs for the current project when you're using the default "Training" perspective. Just double-click on one of the TODOs to jump directly to the corresponding line in the containing file.

Also uncomment the verifications and assertions at the bottom of the *testInvalidDiningEntryFollowedByValidDiningEntry()* test method. (*TODO 03*).

Now, re-run the test by right clicking and selecting *Run As->JUnit Test*.

Notice that both tests fail this time! The error message for *testValidDiningEntry()* should read as follows:

```
org.mockito.exceptions.verification.WantedButNotInvoked:
Wanted but not invoked:

rewardNetwork.rewardAccountFor(
    Dining of $50.00 charged to '1234123412341234' by
                       '1234567890' on 4/24/09 12:00 AM
);

at rewards.ui.DiningEntryTests
        .testValidDiningEntry(DiningEntryTests.java:96)
...
```

This error message is Mockito's way of telling you that the *rewardNetwork* method never got called by

DiningEntry.start(). The *DiningEntry* object starts up and accepts user input correctly, but it never submits that user input to the *RewardNetwork* service.

In the next step you'll fix that problem.

1.3.2.5. Step 5: Finish implementing *DiningEntry*

Open *DiningEntry* and find *TODO 04*. Complete this task by passing the *dining* object as a parameter to *rewardNetwork.rewardAccountFor()*. Assign the return value to the *confirmation* variable that has already been declared for you.

Run *DiningEntryTests* once more. If *TODO 04* was completed correctly, you should see the green bar. You've now tested that *DiningEntry* functions properly.

1.3.2.6. Step 6: Verify interactive dining entry works as expected

Now that the core *DiningEntry* implementation has been tested and we know it works, you should be able to run the interactive *DiningEntryUI* application successfully.

Navigate to *DiningEntryUI* and from the context menu, select *Run As->Java Application*.

Repeat the data entry steps from above:

```
Dining amount: 100
Member credit card number: 1234123412341234
Merchant number: 1234567890
```

This time, you should see the following result:

```
Dining of $100.00 charged to '1234123412341234' by
                   '1234567890' on 4/10/09 12:00 AM

Rewarding account for dining via RewardNetwork...
Result: Result: Reward of $8.00 applied to account
                123456789. RewardConfirmation id is: 1
```

Finally, you'll be prompted to decide whether or not to enter another dining event. You can supply '*n*' to exit the program:

```
Would you like to enter another Dining event? [y/n]: n

Thank you, goodbye.
```

Notice that the amount of the *Reward* was *$8.00*. This *Restaurant* apparently has a generous 8% cash back policy. That money is now available in the *Beneficiary* savings accounts associated with this Reward Network member *Account*.

1.3.2.7. Monitor dining entry statistics via JMX

Throughout this course, you will use JMX to manage and monitor the systems being integrated. JMX provides a very convenient way of visualizing and manipulating objects at runtime. As a refresher on JMX and Spring's support for JMX, let's follow the two very simple steps below to register the *DiningEntry* bean as a JMX MBean.

1.3.2.8. Step 7: Apply JMX metadata to *DiningEntry*

Open (*CTRL-SHIFT-T*) *DiningEntry* once again. Complete *TODO 05* by adding Spring's *@ManagedResource* annotation on the class.

Tip

Remember to use *Content Assist* in Eclipse (*CTRL-SPACE*) to aid you when applying the *@ManagedResource* annotation.

Remaining within *DiningEntry*, complete *TODO 06* and *TODO 07* by adding the *@ManagedAttribute* annotation on the *getDiningEntryCount()* and *getDiningEntryErrorCount()* methods.

1.3.2.9. Step 8: Enable automatic registration of *DiningEntry* as a JMX MBean

Open (*CTRL-SHIFT-R*) *diningentry-config.xml*. You'll notice a TODO which asks you to enable registration of JMX MBeans.

Now, complete *TODO 08* by declaring the correct element from the *context:* namespace.

Tip

Remember that *Content Assist* works from within Spring XML files, as well. Type <*context:* followed by *CTRL-SPACE* to see a list of elements available in the *context:* namespace. From the *Content Assist* menu, you'll notice that you can read the XSD documentation about each element.

1.3.2.10. Step 9: Launch *DiningEntryUI* and monitor its statistics via *JConsole*

For the final step of the lab, let's launch *DiningEntryUI* once again. This time, however, we'll be able to monitor its statistics using JMX.

Right click on *rewards.ui.DiningEntryUI* and *Run As->Java Application*. The usual dining entry prompt should appear. Just leave it as is.

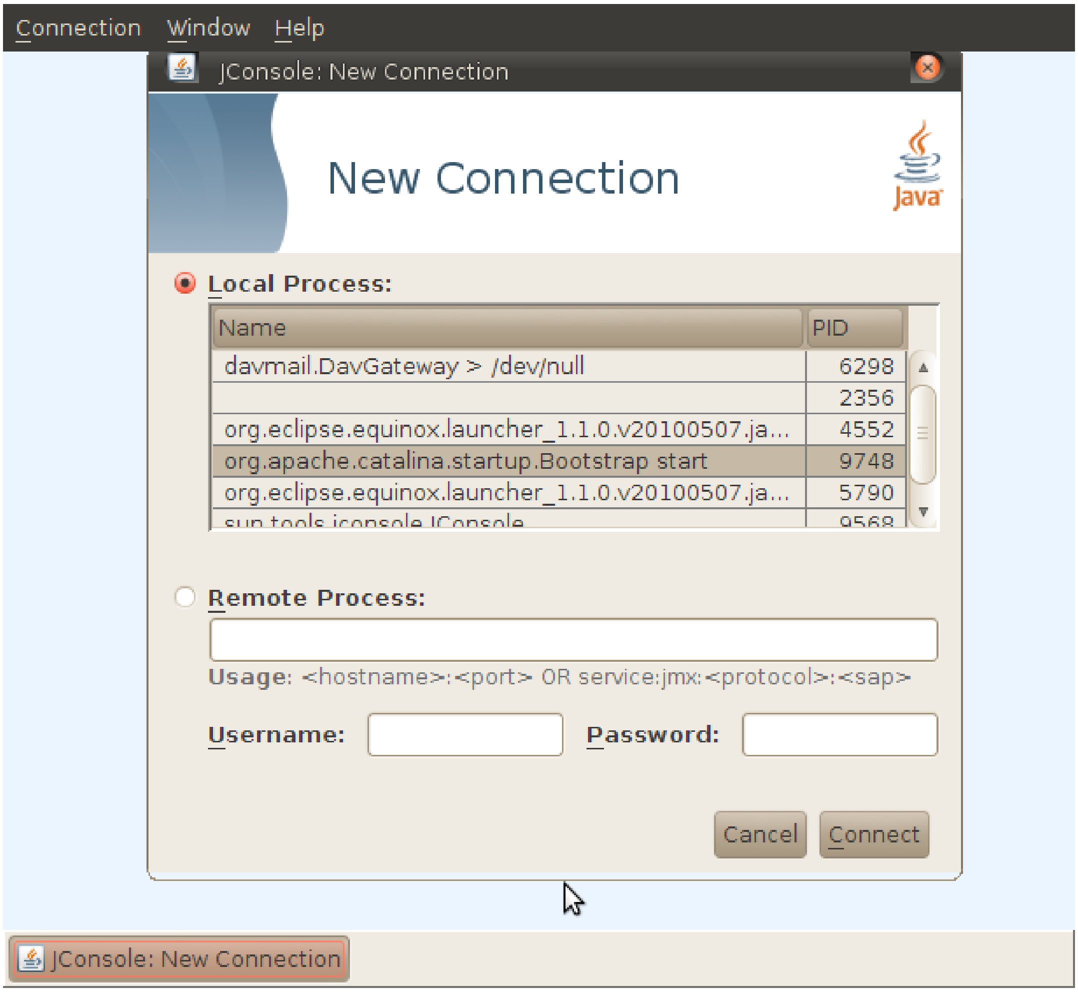
Now, open a windows command prompt (*Start->Run->'cmd'*), and type

```
C:\> jconsole
```

Tip

JConsole is the default JMX console that ships with Java, versions 5.0 and greater. It's located in the *bin* directory of the JDK home: if that's not on your path, then 'cd' into that directory first!

When the JConsole connection dialog appears, select the *rewards.ui.DiningEntryUI* process and click 'Connect'. After a few moments, you'll be presented with the following screen.



[● Local Process:]

Name	PID
davmail.DavGateway > /dev/null	6298
	2356
org.eclipse.equinox.launcher_1.1.0.v20100507.ja...	4552
org.apache.catalina.startup.Bootstrap start	9748
org.eclipse.equinox.launcher_1.1.0.v20100507.ja...	5790
sun.tools.jconsole.JConsole	9568

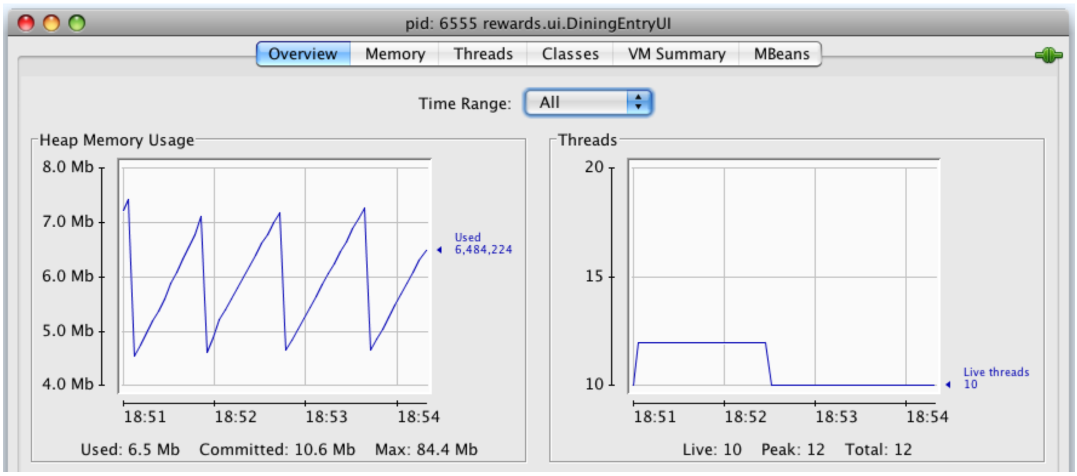
Remote Process:

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Username: Password:

Note

If you cannot connect, you need to set the `-Dcom.sun.management.jmxremote` VM argument within the Run Configurations... dialog for the DiningEntryUI Java Application. If you are unfamiliar with how to do this, ask your instructor.



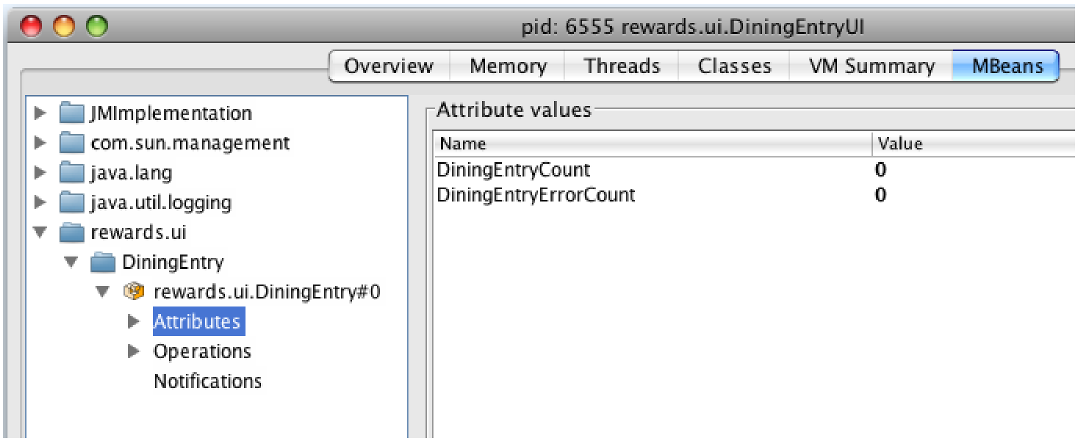
Connecting using JConsole

If you can not see the process you started in JConsole (in the 'Local Process' section) or the connect times out, 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
```

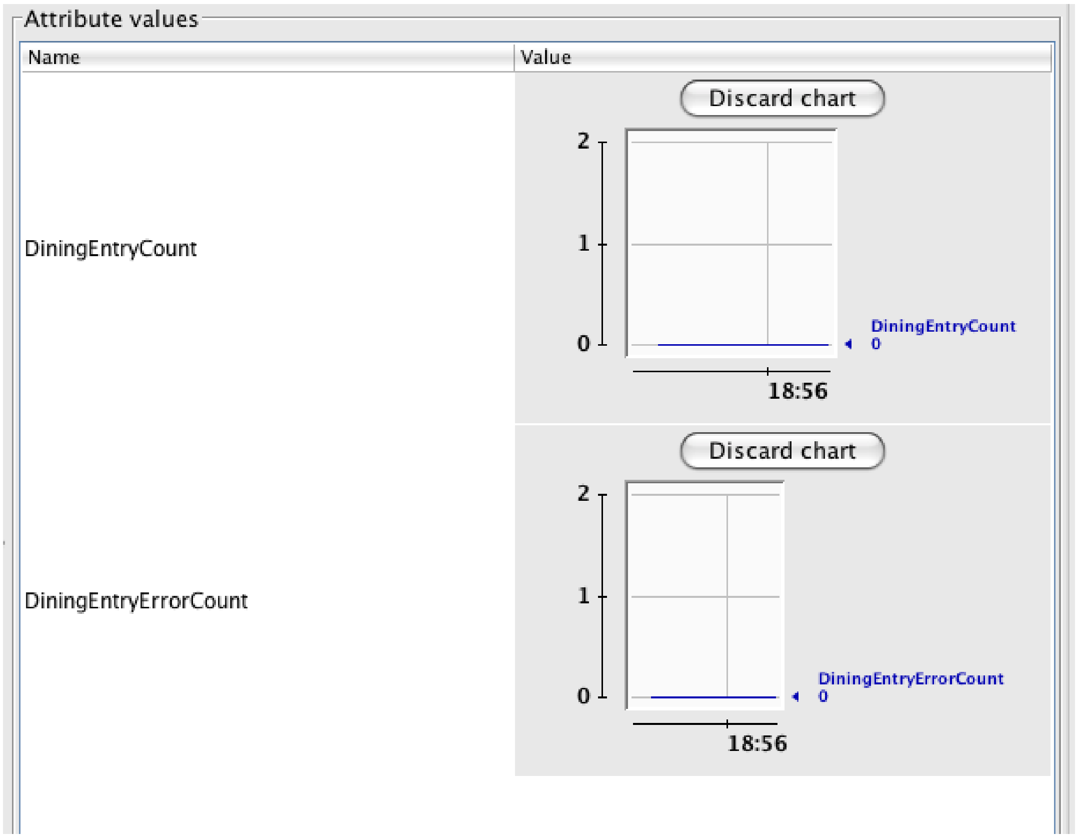
Then restart the process, and connect via JConsole by using the 'Remote Process' selection, specifying a host of *localhost* and port of *8181*.

Click on *MBeans* and navigate to the *rewards.ui->DiningEntry->rewards.ui.DiningEntry* MBean. Select 'Attributes' and notice that you're presented with the following:



Name	Value
DiningEntryCount	0
DiningEntryErrorCount	0

Now, double click on each of the *0* values under the *Value* column. This expands the values out to graphs as below:



Return to the console in STS, and complete a dining entry with valid values:

```
Dining amount: 100
Member credit card number: 1234123412341234
Merchant number: 1234567890
```

Notice that the *DiningEntryCount* graph has changed? It should now read '1'.

At the console, enter '*y*' to enter another dining record. This time, supply invalid values:

```
Dining amount: 100
Member credit card number: 123
Merchant number: 456
```

You'll notice an error in red in the STS console, and back in JConsole, the *DiningEntryErrorCount* graph should now read '1' as well.

1.3.2.11. Summary

You've successfully completed this lab! At this point, you should have an understanding of the following:

- The Spring Tool Suite and some of its features that aid in developing Spring-based applications
- The Reward Dining application domain and the core *RewardNetwork* API
- How to register any Spring-managed bean as a JMX MBean for easy monitoring and management

In the labs to come, we'll dive deeper into all of these topics. For now, you've got the basics and are ready to move on!

Chapter 2. tasks: Tasks and Scheduling

2.1. Introduction

This lab involves two use cases:

1. Refactoring a legacy concurrent batch processing solution to take advantage of modern *java.util.concurrent* facilities such as `Executors`. This is a development-centric use case and the intention is to improve maintainability and flexibility.
2. Automating and scheduling batch processing. Per an internal service level agreement (SLA), batch processing must occur at regular intervals in order to process all dining transactions in a timely fashion.

Topics covered

1. Proper use of the *java.util.concurrent* API introduced in Java 5.0 for easier management of multithreaded applications
2. Scheduling application execution with with the *org.springframework.scheduling* API

Specific subjects you'll gain experience with

1. Convenience methods in the `java.util.concurrent.Executors` class
2. Working with fixed-size thread pool `java.util.concurrent.ExecutorService`
3. The `java.util.concurrent.ExecutorService` lifecycle
4. The *task:* XML namespace introduced in Spring 3.0

Estimated completion time: 30 minutes

2.2. Quick Instructions

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

Use the following quick instructions as a guide.

1. Complete TODO 01 through TODO 05 by refactoring `DiningBatchProcessorImpl` to use an `ExecutorService`.
2. Complete TODO 06 through TODO 08 in `scheduled-tasks.xml` to establish scheduled execution of the diningBatchProcess.

Congratulations, you've completed this lab!

2.3. Instructions

2.3.1. Refactoring Legacy Concurrent Batch Processing

In the current code, you'll find a concurrent batch processing solution that uses manually-created `Thread` and `Runnable` instances. In this section, you'll refactor this solution to take advantage of the much more flexible and easy to use `java.util.concurrent.ExecutorService` API.

2.3.1.1. Before you start: about the existing application

Use Case

One of the key integration points in the Reward Network application is the nightly file transfer of batches of dining transactions processed by various Credit Card Processors (CCPs) such as Visa, MasterCard and Discover. Per agreement with the CCPs these files are transferred via FTP and other protocols in batches of 1000 records or less per file. Because the volume of incoming batches can be quite high, the Reward Network development team has an internal service level agreement (SLA) that any batch file of 1000 records or less must be processed in sixty (60) seconds or less.

To simulate a slow service, the code uses a `SlowRewardNetwork` which simply sleeps some milliseconds before calling the real `RewardNetwork`. This means that non-parallel execution would be really slow, but parallel execution will speed things up considerably since the time is mostly spent waiting.

2.3.1.2. Step 1: Review batch processing tests

Open `DiningBatchProcessorPerformanceTests` from within the *tasks* project. This test ensures that processing both a small and a large batch of input runs fast enough, by reading lines from a comma-separated values (CSV) file. Run the test to make sure it passes (TODO 01). Because this step involves only refactoring, your job is to improve the implementation of `DiningBatchProcessorImpl` without breaking this test.

2.3.1.3. Step 2: Refactor `DiningBatchProcessorImpl`

Open `DiningBatchProcessorImpl`. This implementation is already using concurrency, but in a sub-optimal way: `Thread` instances are manually created and joined. In this step, you'll refactor to a simpler solution.

Complete *TODOs 02-04* to remove low-level threading control and replace it with a `java.util.concurrent.ExecutorService`.

Tip

Remember that `java.util.concurrent.Executors` provides a number of convenience methods for creating various kinds of `ExecutorService` instances. For example, `Executors.newCachedThreadPool()`.

Tip

When calling *ExecutorService.awaitTermination()*, it's fine to wait forever. Use *Long.MAX_VALUE* and *TimeUnit.SECONDS* as parameters.

Tip

Just catch the `InterruptedException` and call `Thread.currentThread().interrupt()` when invoking the `awaitTermination` method.

When finished, run the test suite once again (TODO 05). If you've got green, then you've successfully refactored.

Note

Consider the advantages of using `ExecutorService` instead of `Thread`. In order to change from single-threaded to multi-threaded execution, you need only to switch the implementation of `ExecutorService` being used. In this case, you're using an `ExecutorService` backed by a cached thread pool, but upon exploring the other `ExecutorService` implementations, you'll find that others are backed by a single thread and execute all tasks synchronously and serially. *'ExecutorService' provides an abstraction that hides the details of threading from your application.*

2.3.2. Automating Batch Processing

Now that the `DiningBatchProcessor` implementation is complete and refactored to satisfaction, in this section you'll automate the batch processing to occur at regularly scheduled intervals.

Per the SLA mentioned above, batch processing should kick off every five seconds, seven days a week.

To implement this requirement, you'll use the new *task:* namespace, introduced in Spring 3.

2.3.2.1. Step 3: Review application design

Many scheduled batch applications are nothing more than headless java processes that are kicked off from unix cron or another similar scheduling mechanism. Our scheduled process will be similar in simplicity, but instead of using an external scheduler, the scheduling will be built-in to the java process and kicked off in familiar fashion - via a Spring `ApplicationContext`.

The design of the scheduling application couldn't be simpler. You'll be starting up a regular Java process that will be long-running and will periodically invoke the `processBatch()` on a `DiningBatchProcessor` instance.

This application has been started for you. Open the `ScheduledDiningBatchProcessorBootstrap` class in the *rewards.batch* package.

Notice that this class consists of nothing more than a main method that bootstraps a Spring `ApplicationContext`. Open *scheduled-tasks.xml*, where you'll find a spring bean definition and a couple of *TODO* markers.

The *diningBatchProcessorInvoker* bean is a very simple wrapper around the *diningBatchProcessor* bean you worked with in the first section. It accepts as a constructor argument a `Resource` path pointing to a batch input file. When it's *invoke()* method is called, it then passes that resource on to the *processBatch(Resource batchInput)* method on the *diningBatchProcessor* instance.

Go ahead and take a look at `DiningBatchProcessorInvoker` to see how simple it is.

Once you feel comfortable with what you've seen so far, move on to the next step, where you'll use the *task:* namespace to configure scheduling.

2.3.2.2. Step 4: Complete scheduling configuration

As mentioned above, the scheduling requirement is simple. Batch processing should kick off every five seconds, 24 hours a day, seven days a week.

Complete *TODO 06* by adding a Spring task scheduler using the *task:* namespace. XML auto-completion will work to show you what elements are available. Give the scheduler a thread pool size of *1*.

Complete *TODO 07* by configuring a *task:scheduled-tasks* element containing a single *task:scheduled* element that calls the *invoke* method of the *diningBatchProcessorInvoker* bean every 5 seconds. Consider which is more appropriate: should you use a *fixed-delay* of five seconds or a *fixed-rate*? What are the up- and downsides of each?

2.3.2.3. Step 5: Kick off automated processing

(TODO 08) From the Package Explorer, right click on `ScheduledDiningBatchProcessorBootstrap`, and select *Run As->Java Application*. If you've configured everything correctly, you should begin to see output like the following:

```
INFO : rewards.batch.DiningBatchProcessorImpl - processed 1000 batch records
INFO : rewards.batch.DiningBatchProcessorImpl - processed 1000 batch records
INFO : rewards.batch.DiningBatchProcessorImpl - processed 1000 batch records
```

If you have any errors, work through them until you get the desired outcome.

2.3.2.4. Step 6 (BONUS): Introduce a more sophisticated scheduling requirement

Let's consider a more sophisticated, and perhaps more real-world scheduling scenario. In practice, it turns out that dining batches are only delivered on weekdays between 9:00 am and 5:00 pm. Therefore, our attempts at batch processing only need to be kicked off between those hours. We'll keep the interval at every five seconds just so that you can get rapid feedback.

Fortunately, you'll only need to make one simple change to meet this requirement. Open *scheduled-tasks.xml* once again, and find your *task:scheduled* element. Instead of a *fixed-rate* or *fixed-delay*, this time you'll supply a *cron* expression.

Tip

If you need help with the cron expression, refer back to the slides!

Once you've finished with this update, run the `ScheduledDiningBatchProcessorBootstrap` once more. If you're running this on a weekday between 9am and 5pm, you should see the very same output. If not, you should see nothing.

Finally, try updating the cron expression such that it excludes today (for example: if you're doing this lab on a Tuesday, update the cron expression to only run on Fridays). Run the application a final time, and you should notice that there is no output.

You've completed this lab!

Chapter 3. rest-intro: Introduction to RESTful Applications with Spring

3.1. Introduction

In this lab you will gain experience with using Spring 3.0's new REST support to build a RESTful application and a client that uses the *RestTemplate* to access the application.

What you will learn:

1. Routing HTTP Requests to controller methods
2. Extracting data from requests and populating response bodies
3. Using HTTP Return codes and headers to communicate the result of an HTTP operation
4. Consuming RESTful resources

Specific subjects you will gain experience with:

1. Using *@RequestMapping* and *@ExceptionHandler* for controller method routing
2. using URI Templates and *HttpMessageConverters* with *@Request-* and *@ResponseBody*
3. using the *RestTemplate*

Estimated time to complete: 45 minutes

3.2. Quick Instructions

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

1. **Annotate the method with @RequestMapping, configured to be invoked for POSTs to /dining/{txId} (TODO 01)** (details)

Annotate the reward method of the DiningController class with @RequestMapping. This annotation takes two attributes: set value to the URI Template for this method, which is "/dining/{txId}", and set method to RequestMethod.POST to indicate that this method should only be used for POST requests.

2. **Annotate the diningRequest param with @RequestBody and the txId param with @PathVariable (TODO 02)** (details)

   In the reward method, annotate the dining parameter with @RequestBody and the txId parameter with @PathVariable.

3. **Add the annotation-driven element from the mvc namespace to auto-configure a Jaxb2RootElementHttpMessageConverter (TODO 03)** (details)

   Add the annotation-driven element from the mvc namespace within rest-servlet.xml file.

4. **Add a Location header to the response with the absolute URI of the new reward, using the UriComponentsBuilder to determine the URI based on the current request URI (TODO 04)** (details)

   In the DiningController class, compute the Location header of the response to the absolute URI of the newly created reward.

5. **Return the appropriate ResponseEntity for the creation of a resource (TODO 05)** (details)

   Create and return a *ResponseEntity* that sets a 201 Created status code.

6. **Test the Controller using a RestTemplate. (TODO 06 - TODO 11)** (details)

   Open the CcpClient class. In the sendDiningRequest method, implement TODO 06 by posting the diningRequest to BASE_URI + "dining/{transactionId}", and assign to the location. Deploy the application, and start the CcpClientTestsForServer test class. This class contains two tests: they will both fail, as you haven't finished the implementation of the server and the client yet.

7. **Annotate with @RequestMapping to be invoked for GETs to /rewards/{confirmationNumber} (TODO 07)** (details)

   Open the RewardController class, and annotate the getReward method with an @RequestMapping annotation that uses a URI template of /rewards/{confirmationNumber} and responds to GET requests.

8. **Return the mapped Reward with the given confirmationNumber and annotate the method with @ResponseBody to indicate that the reward should be marshalled (TODO 08)** (details)

   In the getReward method, annotate the method with @ResponseBody and return the found Reward object.

9. **GET the URI at the returned location and assign to reward (TODO 09)** (details)

In the CcpClient class, complete TODO 09 by using the RestTemplate to GET the reward from the returned location. Then run the test again: this time, the processDining() test method should succeed, but the processDiningTwice() will still fail.

10. **Annotate DiningAlreadyProcessedException with @ResponseStatus to return a 409 Conflict when thrown from a controller method (TODO 10)** (details)

    Open the DiningAlreadyProcessedException and implement TODO 10 by annotating the exception with @ResponseStatus, configured to return a 409 Conflict.

11. **Annotate handleEmptyResult method in the RewardController with @ExceptionHandler to let this method be invoked when an EmptyResultDataAccessException occurs and @ResponseStatus to return a 404 Not Found (TODO 11)** (details)

3.3. Instructions

The instructions for this lab are organized into sections. In the first section, you'll complete the steps required to issue a POST request containing a *<dining>* element that will result in a new reward being created. In the second section, you'll add support for issuing a GET request to obtain an XML representation of the reward.

3.3.1. Use Case

Before we start with writing the lab code, we'll first provide you with a bit of context. The Reward Network organization has decided it wants to offer a RESTful API for its clients to send in new dining requests that lead to the creation of a reward. Because HTTP does not offer guaranteed delivery, clients will provide a unique transaction id whenever they send in a dining, and the server will store this id with the resulting reward. This way, a client can simply re-attempt to send a dining if it isn't sure if processing was successful by using the same transaction id. Clients should also be able to see created rewards. To send a new dining to the reward network, a client will POST an XML representation to the following URI: *BASE-URI/dining/{transactionId}*, where *BASE-URI* would be something like *http://reward-network.com/rest*. In our lab it will simply be *http://localhost:8080/rest-intro/rest*.

3.3.2. Inspecting the Existing Application Configuration

The server-side application has already been started. Let's have a look!

3.3.2.1. Step 1: Inspect *web.xml*

First, open the *web.xml* file (under *src/main/webapp/WEB-INF*). The middle tier of the application is bootstrapped by the *ContextLoaderListener,* which uses the *system-test-config.xml* file that's defined in the *rewards* project in the *00-reward-network* Working Set. This simply creates the *RewardNetwork* bean and all the necessary infrastructure. The REST tier is based on Spring MVC, so that's bootstrapped in a different *WebApplicationContext* created by the *DispatcherServlet*. This uses a default location for its configuration file, which is */WEB-INF/<servlet-name>-servlet.xml*. That means the *rest-servlet.xml* file will contain all the necessary beans for our MVC-based REST tier.

3.3.2.2. Step 2: Inspect *rest-servlet.xml*

Open the *rest-servlet.xml* file. Notice that it enables component-scanning for all classes under the *rewards.rest* package (incl. any subpackages). This means that classes that are annotated with *@Component* or one of Spring's stereotype annotations like *@Controller* will automatically result in a bean definition. For controllers, this is a very convenient way to work that saves you from having to write a lot of straightforward XML configuration. You also see a *TODO* tag: we'll get to that in a moment.

3.3.2.3. Handling the first request

Now that you understand how the application is bootstrapped, you'll first perform the steps required to handle the incoming POST request containing the dining.

3.3.2.4. Step 3: Configure the *DiningController*

Open the *DiningController* class. This class will be responsible for handling POST requests to URIs like */dining/1432535435* containing an XML representation of a *Dining* object. The number is a transaction id and will be determined by the client: it serves as a unique identifier for the passed-in dining, which has no unique qualifiers of its own. When such a request is made, the *reward* method of the controller should be called. First implement *TODO 01* by annotating that *reward* method with *@RequestMapping*. This annotation takes two attributes: set *value* to the *URI Template* for this method, which is "/dining/{txId}". This means the value of the transaction id will be available as a method parameter later on. Set *method* to *RequestMethod.POST* to indicate that this method should only be used for POST requests.

Note: Import annotation types

Like always, remember to *import* the annotation types you're using or you'll get a compiler error! We won't use the fully qualified names of all the annotations you'll use in these instructions, as they are all uniquely identified by just their short class names.

Now look at the method implementation: it first uses the passed in *txId* to determine if a reward has already been created for this transaction id and then calls the *RewardNetwork.rewardAccountForDining* method to create a new reward. The passed in *rewards.oxm.Dining* object is transformed into a *rewards.Dining* first: like in earlier labs, we're using JAXB 2 to map the XML Schema for a <*dining*> element to a class, which is this *rewards.oxm.Dining*. Don't confuse it with the regular *rewards.Dining* class that's passed as a parameter to the *rewardAccountFor* method.

However, there's no way for Spring to know how it needs to pass in the unmarshalled dining and the transaction id as specified in the URI template to the controller method. To let Spring know how to do this, you'll annotate the method parameters. Implement *TODO 02* by annotating the *dining* parameter with *@RequestBody* and the *txId* parameter with *@PathVariable*. The first tells Spring to take to contents of the HTTP request and to transform it into a *rewards.oxm.Dining* instance. The second tells Spring to apply the URI template you specified as the value of the *@RequestMapping* and to extract the *txId* variable.

Note: @PathVariable annotated parameters

You don't have to name the URI template variable in this case. This works because the parameter name is the same as the template variable, and you're using compiler settings that add debug information to the generated class files. If that's not the case, you'd have to name the template variable in the annotation!

3.3.2.5. Step 4: Register a *Jaxb2RootElementHttpMessageConverter*

There's a catch, though: the so-called *HttpMessageConverters* that are registered by default don't know how to unmarshal the XML from the incoming request using JAXB 2. However, there's an easy way to configure a number of extra converters, including one that supports JAXB2 with annotated classes: use the new MVC namespace that was introduced in Spring 3.0. Switch back to the *rest-servlet.xml* file. Finish *TODO 03* by first adding the *mvc* namespace using the Namespaces tab, and then adding the *<mvc:annotation-driven/>* element. This element causes a number of MVC-specific objects to be registered, including a *Jaxb2RootElementHttpMessageConverter* that's perfect for our purpose. This class is smart enough that it doesn't need additional configuration like the marshallers from previous labs.

3.3.2.6. Producing the first response

Now that we can process the request, it's time to think about the response. In a regular web application, we might simply send the reward that was created back in the response. In a RESTful application, however, a POST returns the URI of the new resource that was created in the *Location* header of the response. A client can then GET that content at the given location. The contents of the response will often just echo the request data, as that's what is typically created with a POST, or be empty.

3.3.2.7. Step 5: Set the *Location* header

Our sample application is a bit different, as a POST to */dining/txId* doesn't create a new dining: it creates a new reward! We'll simply return an empty response body, but it's critical that we inform the client where it can find the new reward. Switch back to the *DiningController* and complete *TODO 04* to calculate the *Location* header of the response. The *UriComponentsBuilder* is there to help.

Note: A note on request URIs

A request URI like *http://localhost:8080/rest-intro/rest/dining/1234* consists of a number of elements. The *path info* of this request would be */dining/1234*, which follows the *servlet path* of */rest*, which follows the *context path* of */rest-intro*. We want to change only the path info for new requests, not the rest.

3.3.2.8. Step 6: Return the response with the appropriate status code

If we would now use our controller, it would return a 200 OK status code when the dining was processed successfully. That's not what a RESTful application should do after creating a new resource in response to a POST: that should return a 201 CREATED to indicate that a new resource has become available, and its location is set as a response header. The easiest way to specify both an header and a status is to use a *ResponseEntity*. Complete the controller implementation by implementing *TODO 05* and return a *ResponseEntity* created from one of the *ResponseEntity* factory methods.

3.3.3. Testing the Controller using a *RestTemplate*

Before taking on the second controller, let's first test your work.

3.3.3.1. Step 7: Fix the *CcpClient* to call the *DiningController* and test the application

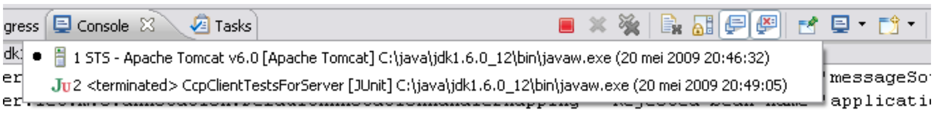
Open the *CcpClient* class. Implement *TODO 06* by POSTing the passed in *diningRequest* to the server using the *RestTemplate* instance field. The full URI to use consists of the *BASE_URI*, followed by *dining/* plus the *transactionId*. You can use URI templates in the *RestTemplate* just like you can on the server. Assign the returned location to the given variable. The *RestTemplate* will automatically support object to XML marshalling using JAXB2 if that's available on the classpath, you don't need to configure anything for that.

Now deploy the application by dragging the project onto the local Tomcat server and start the server. When the application started successfully, start the *CcpClientTestsForServer* test class. This class contains two tests: they will both fail, as you haven't finished the implementation of the server and the client yet, but if you switch the Console view to show the output for the test you should see the following lines in the output:

```
DEBUG: org.springframework.web.client.RestTemplate -
       Created POST request for
       "http://localhost:8080/rest-intro/rest/
            dining/4c61418d-6477-4269-880f-a381fb809cea"
DEBUG: rewards.client.rest.CcpClient -
       Received location of newly created reward:
    http://localhost:8080/rest-intro/rest/rewards/1
```

Tip: Switching Consoles

If you have multiple processes running, you'll only see the output of one process in the Console view. If that's the Tomcat process, you'll have to switch the Console to the other process: click on the small arrow next to the monitor icon and choose the test process.



```
1 STS - Apache Tomcat v6.0 [Apache Tomcat] C:\java\jdk1.6.0_12\bin\javaw.exe (20 mei 2009 20:46:32)
2 <terminated> CcpClientTestsForServer [JUnit] C:\java\jdk1.6.0_12\bin\javaw.exe (20 mei 2009 20:49:05)
```

This tells you that the POST did indeed result in the creation of a reward, with the URI *http://localhost:8080/rest-intro/rest/rewards/1*. This is communicated to the client through the Location header of the response, but performing a GET on this URI will still result in a 404 Not Found. That's what you'll fix next.

3.3.3.2. Handling the second request

The next step is to ensure that the client can now request the reward with the given URI.

3.3.3.3. Step 8: Finish the *RewardController.getReward* method

Open the *RewardController* class. This class will be responsible for handling GET requests to URIs like */rewards/1*. Because this is a GET, the request won't have any content like the POST did. First finish *TODO 07* by annotating the *getReward* method with an *@RequestMapping* annotation that uses a URI template of */rewards/{confirmationNumber}* and responds to GET requests. Then implement *TODO 08* by returning the found *Reward* object and annotating the method with *@ResponseBody*. Since you already configured an appropriate *HttpMessageConverter* in step 4, that's all you need to do! You could annotate the method with *@ResponseStatus*, but in this case the default status code 200 OK is exactly what we want already.

3.3.3.4. Step 9: Finish the *CcpClient* and run the test again

Finish the *CcpClient* by using the *RestTemplate* to GET the reward from the returned location (*TODO 09*). Then run the test again: this time, the *processDining()* test method should succeed, but the *processDiningTwice()* will still fail. Let's fix that next.

3.3.3.5. Step 10: Make the exception for duplicate dinings result in a 409 return code

When a client sends a dining request using a transaction id that has already been processed successfully, the *RewardController* throws a *DiningAlreadyProcessedException*. Currently, that results in a 500 Internal Server Error code being returned. However, this should indicate a client error: resubmitting dinings that have already been processed is not allowed. We want to indicate this to the client by sending a 409 Conflict error code instead. Fortunately, this is trivial to do. Open the *DiningAlreadyProcessedException* and implement *TODO 10* by annotating the exception with *@ResponseStatus*, configured to return a 409 Conflict. Notice that this is the same annotation you used on the controller method in step 6.

Now run the test again and make sure that the *processDiningTwice()* test method now passes as well.

3.3.3.6. Step 11: Make requests for non-existent rewards result in a 404 Not Found return code

Open a web browser and request the url [http://localhost:8080/rest-intro/rest/rewards/1]. This will return the XML for the reward you created earlier. Now change the reward confirmation number in that URI to some non-existent number. Notice that this results in a 500 Internal Server Error again, with a stack trace. This should be a 404 Not Found error. However, this time we can't simply annotate the exception, as the exception that's thrown is Spring's *EmptyResultDataAccessException*. Fortunately, Spring 3 adds another mechanism for handling exceptions thrown from your controller methods. Look at the *handleEmptyResult* method in the *RewardController*: this method should be called whenever an *EmptyResultDataAccessException* is thrown from a method in that controller while handling a request. To indicate that to Spring, annotate the method with *@ExceptionHandler* and provide *EmptyResultDataAccessException.class* as the value (*TODO 11*). You can use the *@ResponseStatus* annotation again, so add another annotation to the method to indicate that we want to return a 404 Not Found. Since the response doesn't need a body, our method needs no further implementation.

Refresh your browser and check that you now receive a 404. When you see that error code, you have successfully completed the lab!

3.3.4. PART 2: Out-of-container testing

Estimated time to complete: 30 minutes

Follow the TODOs in the code to implement an out-of-container test of our REST application. This means we'll be able to test the logic of the controllers, as well as the Spring MVC part (e.g. XML/JSON serialization), without deploying the application in a web container. This makes the testing effort much easier and faster.

Chapter 4. jms: Simplifying Messaging with Spring's JMS Support

4.1. Introduction

In an effort to improve the performance and robustness of the application, the reward network team wishes to process incoming customer dining events in an asynchronous fashion. Remember that the fundamental use case of the reward network application is *rewarding an account for dining* - that won't change. The goal of this lab is to decouple the process of rewarding an account from the process of receiving a new dining event.

JMS is a good fit for achieving this goal. As *Dining* objects arrive in the system, they will be sent to a JMS *Destination* where they will wait to be consumed and routed through the existing *RewardNetwork.rewardAccountFor(Dining)* method. Recall that *rewardAccountFor(Dining)* returns a *RewardConfirmation* - this object will need to be placed on its own JMS *Destination* as well, so that it can be consumed by a *RewardConfirmationLogger* (more on this object later).

When finished, the benefit will be that any number of *Dining* events can arrive in the system, regardless of whether the *RewardNetwork* endpoint is available. When the *RewardNetwork* does come online, it will consume however many *Dining* events have queued up. This is *asynchronous processing*.

This lab is broken into two sections. In the first section you will complete the steps necessary to send incoming *Dining* objects as messages to a JMS Queue. In the second section you will consume those messages and process them with the *DiningProcessor* object.

What you will learn

1. How to configure and use Spring's *JmsTemplate*
2. How to configure and use Spring's *MessageListenerContainer* support, via the *@JmsListener* annotation.

Estimated time to complete: 30 minutes

4.2. Quick Instructions

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

1. **Process five `Dining` objects through the diningProcessor (TODO 01)**

   Open `DiningProcessorTests` and process five Dining objects through the `diningProcessor`. Now run the test. You will see an `UnsupportedOperationException`.

2. **Convert the `dining` to a message and send it to its destination (TODO 02)**

   Click on the exception from the stack trace and implement the `process(Dining)` method on `JmsDiningProcessor`. Run the test again to see the effect. The test will still fail. You should see a `NullPointerException`.

3. **Create a `JmsTemplate` and inject the `cachedConnectionFactory` as well as the defaultDestination (diningQueue) (TODO 03)**

   Define a `JmsTemplate` bean and inject it into the `diningProcessor` bean. Now run `DiningProcessorTests` once again. The Test should pass.

4. **Wait for the batch and assert it's size is 5 (modify the assertion) (TODO 04)**

   Use the `waitForBatch()` to wait 10 seconds for the 5 dinings to get sent, consumed and routed through the `RewardNetwork`. Run `DiningProcessorTests` again. It should fail.

5. **Activate the detection of JMS annotations and declare a `JmsListenerContainerFactory` (TODO 05)**

   Open the `JmsInfrastructureConfig` configuration class. Add the `@EnableJms` annotation on top of it and declare a `jmsListenerContainerFactory` bean of type `DefaultJmsListenerContainerFactory`. Don't forget to inject the `cachingConnectionFactory` bean into it.

6. **Wire the `rewardNetwork` with the `@JmsListener` annotation (TODO 06)**

   Open up the `RewardNetworkImpl` class at the `rewardAccountFor` method level. Use the `@JmsListener` annotation to listen on the `rewards.queue.dining` queue. Use also the `@SendTo` annotation to send the method output to the `rewards.queue.confirmation` queue.

4.3. Instructions

4.3.1. Sending Dining Messages to a Destination

In this section you will see how to convert a *Dining* instance into a JMS *Message* and send it to a *Destination*.

4.3.1.1. Step 1. Modify the test

Open *DiningProcessorTests* and complete *TODO 01*. Now run the test. You will see an *UnsupportedOperationException*.

4.3.1.2. Step 2. Resolve the error

Click on the exception from the stack trace and implement the *process(Dining)* method on *JmsDiningProcessor* (*TODO 02*).

Tip

When converting and sending the *Dining* object as a *Message*, you will not need to specify a *Destination* explicitly. A *default destination* will be set up later in the configuration for *JmsTemplate*.

When you're finished making the change, run the test again to see the effect. The test will still fail, but you should see a *NullPointerException* this time.

4.3.1.3. Step 3. Set up the *JmsTemplate*

The root cause of the *NullPointerException* above is that the *JmsTemplate* was not injected into our *DiningProcessor* object.

To fix this problem, open *client-config.xml* and look at the *diningProcessor* bean definition. Notice that it is indeed missing its *jmsTemplate* property. Complete *TODO 03* by defining a *JmsTemplate* bean and injecting it into the *diningProcessor* appropriately.

Tip

Be sure to configure the *connectionFactory* and *defaultDestination* properties on your *JmsTemplate* bean definition. Beans named *cachingConnectionFactory* and *diningQueue* have already been defined for this purpose. See *jms-infrastructure-config.xml* to understand how these beans are created. Notice that the *diningQueue* bean is created with a *constructor-arg* indicating that the name of this queue in the JMS broker is *rewards.queue.dining*.

Now run *DiningProcessorTests* once again. You should have a green bar, meaning that you've successfully processed the five *Dining* objects and sent them to the *rewards.queue.dining* destination.

4.3.1.4. Receiving dining messages from a destination using JMS listener

In this section you will implement the receiving end of the application. The goal is to watch the *rewards.queue.dining* destination and route *Dining* objects through our existing *RewardNetwork* implementation as they arrive. In this way, the *RewardNetwork* object will become a participant in an *event-driven application* without requiring you to write any extra code.

A note about *RewardConfirmationLogger*

Notice in *DiningProcessorTests* that an object of type *RewardConfirmationLogger* is injected using *@Autowired*. This object is configured to listen on a second JMS destination named *rewards.queue.confirmation*, which has already been set up for you. Every time a *RewardConfirmation* gets placed on that queue, the *RewardConfirmationLogger* consumes it and adds it to its *confirmations* collection.

4.3.1.5. Step 4. Modify the test

Go back to *DiningProcessorTests* and complete *TODO 04*. Use the *waitForBatch()* to wait 10 seconds for our 5 dinings to get sent, consumed and routed through the *RewardNetwork*.

Tip

Look at the implementation of *waitForBatch()*. Notice that it polls the size of the *RewardConfirmationLogger*'s *confirmations* collection for the specified number of seconds. The method returns as soon as the desired number of *RewardConfirmations* have been processed or the specified number of seconds to wait runs out, whichever comes first.

Run *DiningProcessorTests* again. It should fail, which makes sense because we haven't yet wired up the *RewardNetwork* as a listener on the *rewards.queue.dining* destination. *waitForBatch()* just times out after 10 seconds - no *Dining* objects have been passed through *RewardNetwork*, thus no confirmation messages have shown up for the logger to record.

4.3.1.6. Step 5. Activate the detection of JMS annotations and declare a *JmsListenerContainerFactory*

Open the `JmsInfrastructureConfig` configuration class. Add the `@EnableJms` annotation on top of it to detect `@JmsListener` annotations. Then, declare a `jmsListenerContainerFactory` bean of type `DefaultJmsListenerContainerFactory`. Don't forget to inject the `cachingConnectionFactory` bean into it. The `jmsListenerContainerFactory` bean will create the actual listener objects when `@JmsListener` annotations are used.

4.3.1.7. Step 6. Wire the *rewardNetwork* with the *@JmsListener* annotation

Open up the `RewardNetworkImpl` class at the `rewardAccountFor` method level. Use the `@JmsListener` annotation to listen on the `rewards.queue.dining` queue. Use also the `@SendTo` annotation to send the method output to the `rewards.queue.confirmation` queue.

Tip: Understanding *destination*

When using the *@JmsListener* annotation, you'll need to specify the *destination* attribute. This signifies what JMS destination the listener should poll. Earlier in the lab, you provided a *destination* property to the *JmsTemplate* bean, which was a reference to the bean named *diningQueue*. In this case, you're specifying not a reference to a bean, but rather the *name of the destination* as it exists in the JMS broker. So, use *rewards.queue.dining* for the value of *destination*.

Now run the test again to confirm that this is working. If you've got the green bar, you've successfully put together an asynchronous processing pipeline for *Dining* objects. Congratulations!

Chapter 5. jms-tx: Add Transactional Capabilities to the JMS Interaction

5.1. Introduction

What you will learn:

1. Problems caused by non-transactional JMS message reception
2. How local JMS transactions can help to avoid some, but not all of these problems

Specific subjects you will gain experience with:

1. Local JMS transaction management in Spring

Estimated time to complete: 30 minutes

5.2. Quick Instructions

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

1. Start the server by running the `StartServer` class, enabling JMX by adding `'-Dcom.sun.management.jmxremote'` to the VM arguments.
2. Run the `StartSender` class in the `rewards.messaging.client.sender` package to send some messages.
3. Inspect the result in JConsole. The dining queue should have 0 messages, and the confirmation queue should have 5 messages. The ConfirmationLister should show 5 confirmations in the database. Run the `StartReceiver` process to consume all the 5 confirmation messages.
4. Cause an error after receiving the message. In the JConsole, navigate to the rewards/DiningListener MBean and set its `CauseErrorAfterReceiving` attribute to true. Run `StartSender` again to send 5 new dining request messages. Switch back to JConsole, and verify that all the messages are lost (check the dining queue, confirmation queue, and database to observe the results). Stop the server by pressing Enter in the

Console view.

5. Use a local JMS transaction for receiving the messages. Open the `JmsRewardsConfig` configuration class, and find the `jmsListenerContainerFactory` bean. Set the `sessionTransacted` flag to true (TODO 01). Start the server again, connect using JConsole and set the `CauseErrorAfterReceiving` attribute of the `DiningListener` to true again. Run `StartSender` again to send 5 more new messages. Use JConsole to verify the results (check dining queue, confirmation queue, database, and notice a new dead letter queue that has the 5 messages).

6. Cause an error after processing the message. Using JConsole, reset the `CauseErrorAfterReceiving` flag by setting it to false again, but this time set the `CauseErrorAfterProcessing` flag to true. Run `StartSender` again. Observe the effect in JConsole (check the queues and database).

7. Cause an error after sending the confirmation message. Reset the `CauseErrorAfterProcessing` flag back to false and set the `CauseErrorAfterSending` flag to true. Run `StartSender`, and observe the results in JConsole again. After you finished, stop the server by pressing Enter in the Console window.

5.3. Instructions

The instructions for this lab are organized into sections. In the first section, you'll look at an existing application that doesn't use JMS transactions which receives message from a JMS queue, stores them in a database and sends some response messages. You will learn what errors this can cause. In the second section, you'll rework the application to use local JMS transactions to solve some of these errors. You'll see in what scenarios local JMS transactions can help to solve all issues, and when global transaction might be necessary instead.

5.3.1. Inspecting and Running the Current Application

The existing application performs the following steps:

1. Picks up `Dining` request messages from a JMS queue

2. Processes these requests using the `RewardNetwork`

3. Sends the resulting `RewardConfirmation` as a response message to another queue

Ideally, these operations should be *atomic*: either all should succeed, or none of them should. Let's see why this is currently not the case.

5.3.1.1. Step 1: Inspect the current application

Open the *jms-tx* project. This project consists of three parts: a *server* that implements the use case as described above, a *sender* that sends a couple of new `Dining` requests as JMS messages to the request queue, and a *receiver* that takes confirmation messages from the confirmation queue and prints them. Our interest is in the server. Inspect the `InfrastructureConfig` configuration class. Notice how it defines a local transaction manager for an embedded Derby database (the `DataSourceTransactionManager`) and an embedded ActiveMQ connection factory with a broker that can be accessed from other applications using a TCP connector.

Now open the `JmsRewardsConfig` class. This config class imports the bean definitions for the reward network application and defines a `JmsListenerContainerFactory` which creates `DefaultMessageListenerContainer` instances. But where is the JMS listener? Open the `DiningListener` and look at the `onMessage` method: it's annotated with the `JmsListener` annotation. The `DiningListener` processes messages from a queue called *rewards.queue.dining*. Have a look at the content of the `onMessage` method and confirm that the code implements the flow described before. Responses are sent to a queue called *rewards.queue.confirmation*, as configured in the `JmsTemplate` that's injected into the `DiningListener`. Notice that this class has various flags that can be set to cause errors at runtime: you'll use those in the next section. When you understand the setup of the server application, move to the next step.

5.3.1.2. Step 2: Run the application

Start the server by running the `StartServer` class. This will bootstrap the database and JMS broker, and the server will wait for messages to come in on the *rewards.queue.dining* queue.

Warning

To stop the server when you're done with it, click in the Console view and press Enter. *Don't* press the Terminate stop button, as that will not properly close down the Derby database! In contrast to most other labs, we use persistent data in this lab and killing the VM will cause issues when restarting the database.

Since we want to use JMX to examine the server instance, stop it now, as we just described. Then, go to Run... | Run Configurations... Select Start Server under Java Application, and add '-Dcom.sun.management.jmxremote' to the VM arguments; click Run.

To send some messages to the server, run the `StartSender` class in the *rewards.messaging.client.sender* package. This class simply sends five messages to the dining queue for processing and then exits. In the Console view for the `StartServer` process, you should now see output similar to this:

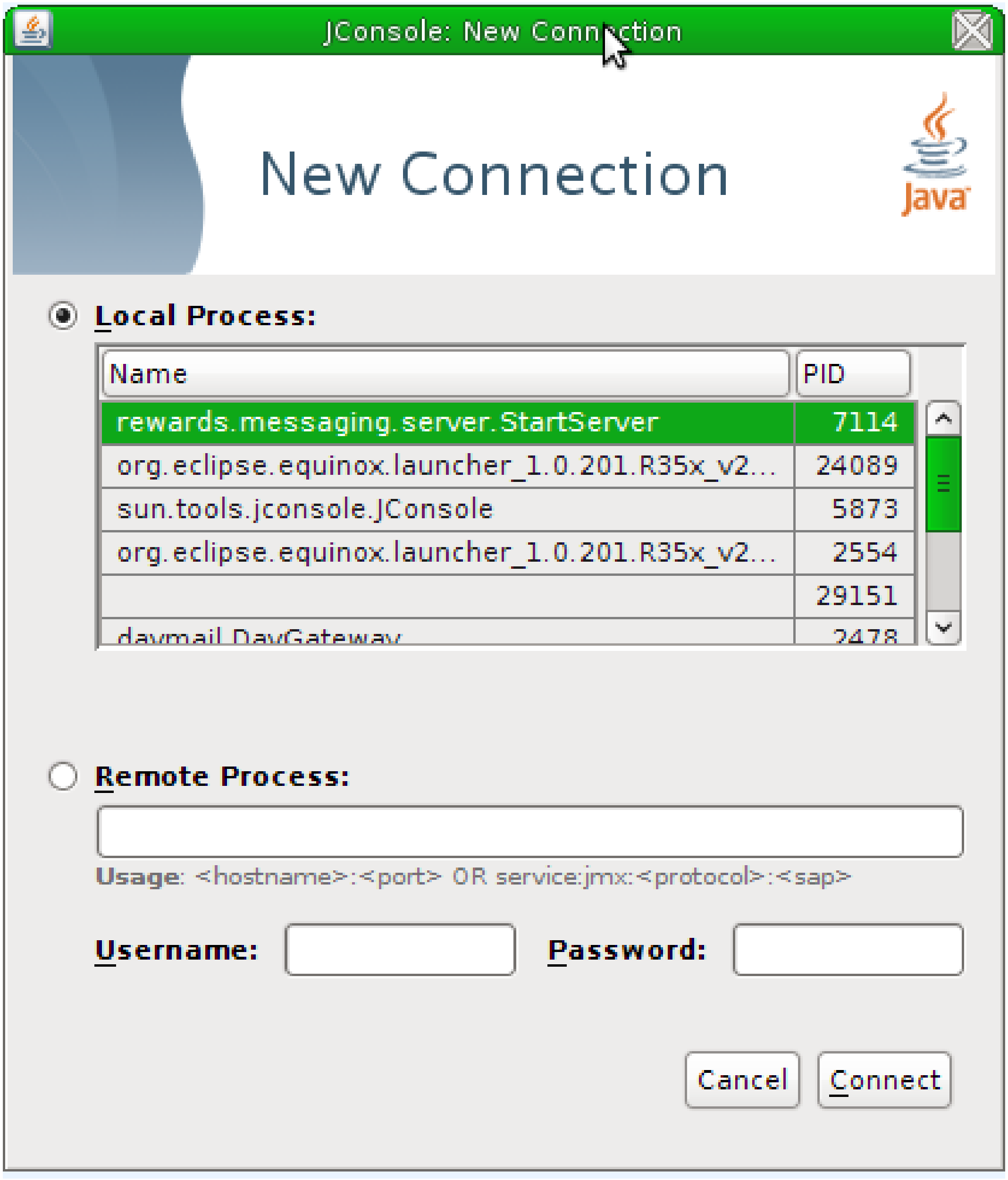
```
...
Started server, press Enter to stop
DEBUG: rewards.messaging.server.DiningListener -
         Received Dining with amount $80.93
DEBUG: rewards.messaging.server.DiningListener -
         Sent response with confirmation nr 1
DEBUG: rewards.messaging.server.DiningListener -
         Received Dining with amount $56.12
```

```
DEBUG: rewards.messaging.server.DiningListener -
         Sent response with confirmation nr 2
DEBUG: rewards.messaging.server.DiningListener -
         Received Dining with amount $32.64
DEBUG: rewards.messaging.server.DiningListener -
         Sent response with confirmation nr 3
DEBUG: rewards.messaging.server.DiningListener -
         Received Dining with amount $77.05
DEBUG: rewards.messaging.server.DiningListener -
         Sent response with confirmation nr 4
DEBUG: rewards.messaging.server.DiningListener -
         Received Dining with amount $94.50
DEBUG: rewards.messaging.server.DiningListener -
         Sent response with confirmation nr 5
```

That means the server started successfully, after populating the database with some initial schema and data, and then received and processed five messages.

5.3.1.3. Step 3: Inspect the result

What we expect now is that the database contains five reward confirmations, and that the *rewards.queue.confirmation* queue contains five messages. Let's confirm this by connecting to the server using JConsole. From the *bin* directory of your local Java SDK, start *jconsole* and connect to the running `StartServer` process.



JConsole: New Connection

New Connection

(●) Local Process:

Name	PID
rewards.messaging.server.StartServer	7114
org.eclipse.equinox.launcher_1.0.201.R35x_v2...	24089
sun.tools.jconsole.JConsole	5873
org.eclipse.equinox.launcher_1.0.201.R35x_v2...	2554
	29151
davmail.DavGateway	2478

( ) Remote Process:

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Username: Password:

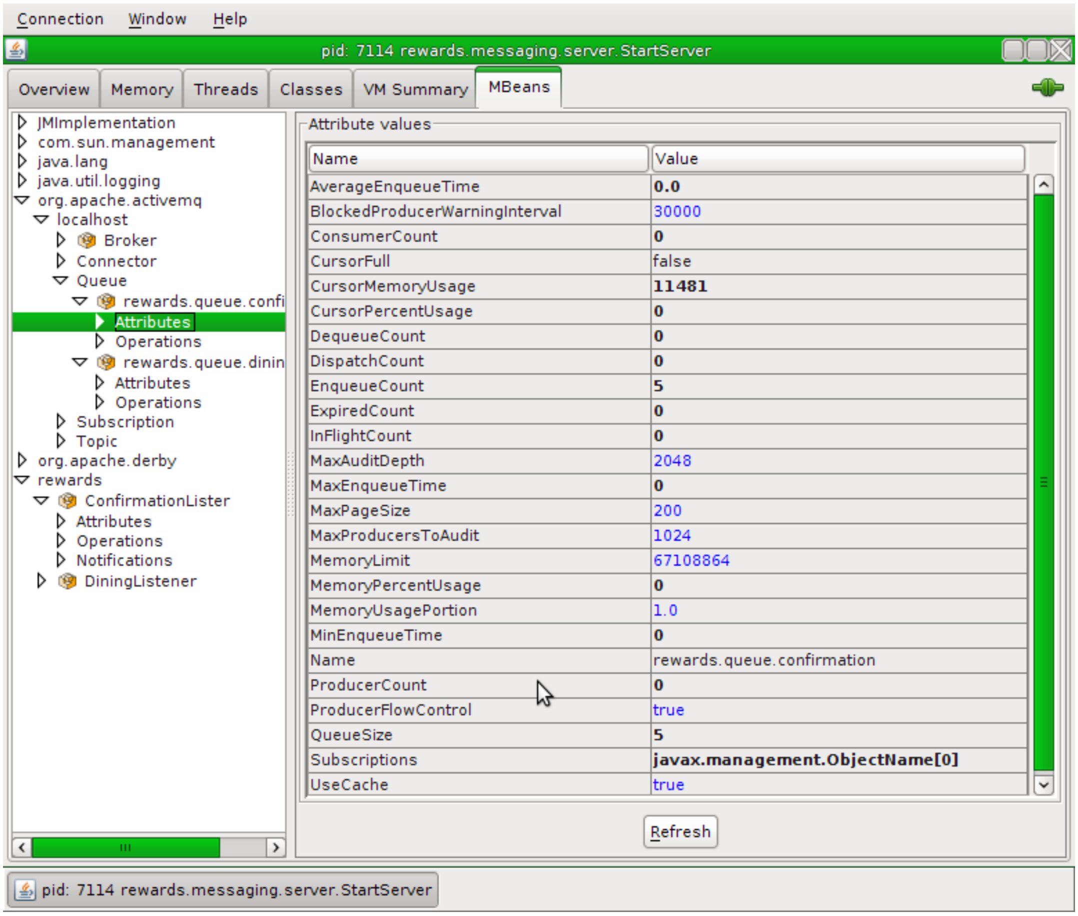
Tip: Connecting using JConsole

If you can not see the process you started in JConsole (in the 'Local Process' section) or the connect times out, 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
-Djava.rmi.server.hostname=127.0.0.1
```

Then restart the process, and connect via JConsole by using the 'Remote Process' selection, specifying a host of *localhost* and port of *8181*.

Switch to the MBeans tab. Notice that there is an *org.apache.activemq* node. Navigate to its *localhost/Queue* subnode and you will see the two queues that our application is using. Click on the *Attributes* node under each queue and inspect the *QueueSize* attribute. The dining queue should have 0 messages, as they're all processed, and the confirmation queue should have 5.



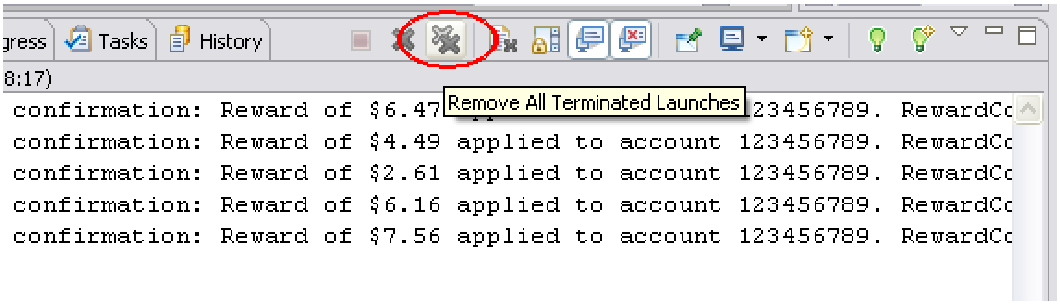
Name	Value
AverageEnqueueTime	0.0
BlockedProducerWarningInterval	30000
ConsumerCount	0
CursorFull	false
CursorMemoryUsage	11481
CursorPercentUsage	0
DequeueCount	0
DispatchCount	0
EnqueueCount	5
ExpiredCount	0
InFlightCount	0
MaxAuditDepth	2048
MaxEnqueueTime	0
MaxPageSize	200
MaxProducersToAudit	1024
MemoryLimit	67108864
MemoryPercentUsage	0
MemoryUsagePortion	1.0
MinEnqueueTime	0
Name	rewards.queue.confirmation
ProducerCount	0
ProducerFlowControl	true
QueueSize	5
Subscriptions	javax.management.ObjectName[0]
UseCache	true

Now navigate to the *rewards* node and check the *Attributes* of the *ConfirmationLister* you see under that. It should tell you that there are 5 confirmations in the database. You can also invoke the *listAllConfirmations* operation to see their contents.

Note

If you're wondering how the ConfirmationLister and DiningListener are exported as MBeans without using the `@ManagedResoruce` annotation you saw earlier in the course, have a look at the `JmxConfig` configuration class: the `MBeanExporter` in there exposes the given beans under the `ObjectName` given by the entry keys by making all public methods part of the managed interface.

So far so good: as a final step, run the `StartReceiver` process to consume the 5 confirmation messages. When you're done, remove all terminated processes from the Console view by pressing the double X button.



```
confirmation: Reward of $6.47 ... 23456789. RewardCo
confirmation: Reward of $4.49 applied to account 123456789. RewardCo
confirmation: Reward of $2.61 applied to account 123456789. RewardCo
confirmation: Reward of $6.16 applied to account 123456789. RewardCo
confirmation: Reward of $7.56 applied to account 123456789. RewardCo
```

5.3.2. Introducing errors in the application

When everything works as expected, the current application behaves just fine. However, with various JMS and JDBC operations involved, there are chances that things don't go as planned at runtime. In a typical production environment, both the database server and message broker would be running on a remote server, for example: that means that a network error could cause some operations to fail. In this section you'll simulate those failures to observe the result in the application.

5.3.2.1. Step 4: Cause an error after receiving the message

Currently, the `jmsListenerContainerFactory` bean is in charge of creating the message listener container used by the `DiningListener`. This message listener container performs the reception of the messages.

Notice that there's no transaction management configured: that means the receive will not be performed in a transaction managed from the application. To show how this can cause errors, switch back to JConsole again. Navigate to the *rewards/DiningListener* MBean and click on its Attributes node. Here, you can set the various error flags to *true* to cause failures at particular points in the process implemented by the `DiningListener`.

Set its *CauseErrorAfterReceiving* attribute to *true* and then start the `StartSender` application again to send 5 new dining request messages.

In the Console view, you'll see something like this for each message:

```
DEBUG: rewards.messaging.server.DiningListener -
         Received Dining with amount $80.93
WARN : org.springframework.jms.listener
                        .DefaultMessageListenerContainer -
         Execution of JMS message listener failed,
             and no ErrorHandler has been set.
java.lang.RuntimeException: error after receiving dining
         with amount $80.93
```

```
        at rewards.messaging.server.DiningListener
             .onMessage(DiningListener.java:48)
...
```

Now switch back to JConsole and look at the number of confirmations in the database by inspecting the *NrOfConfirmations* attribute of the *ConfirmationLister*: you'll see that it hasn't changed. This is of course exactly what we expected: we threw an error before the processing started. However, also have a look at the *QueueSize* of the *rewards.queue.dining*: that's still 0. This means that we have *lost all the messages*!

In most applications, this is simply not acceptable: if something goes wrong during the processing of a received message, the message should simply be put back on the queue so it can be reprocessed later. There's a number of ways to ensure this, but Spring makes this particularly easy. Stop the server by pressing Enter in the Console view and move to the next step to see how this is done.

5.3.2.2. Step 5: Use a local JMS transaction for receiving the messages

Open the `JmsRewardsConfig` and find the `jmsListenerContainerFactory` bean. Set the `sessionTransacted` flag of this bean to `true` (*TODO 01*). This tells Spring to start a local JMS transaction for the reception of the message. Moreover, this transaction will only be committed *after* the local JDBC transaction that's used for processing the message and updating the database. That means that if the processing fails, the message will not be lost.

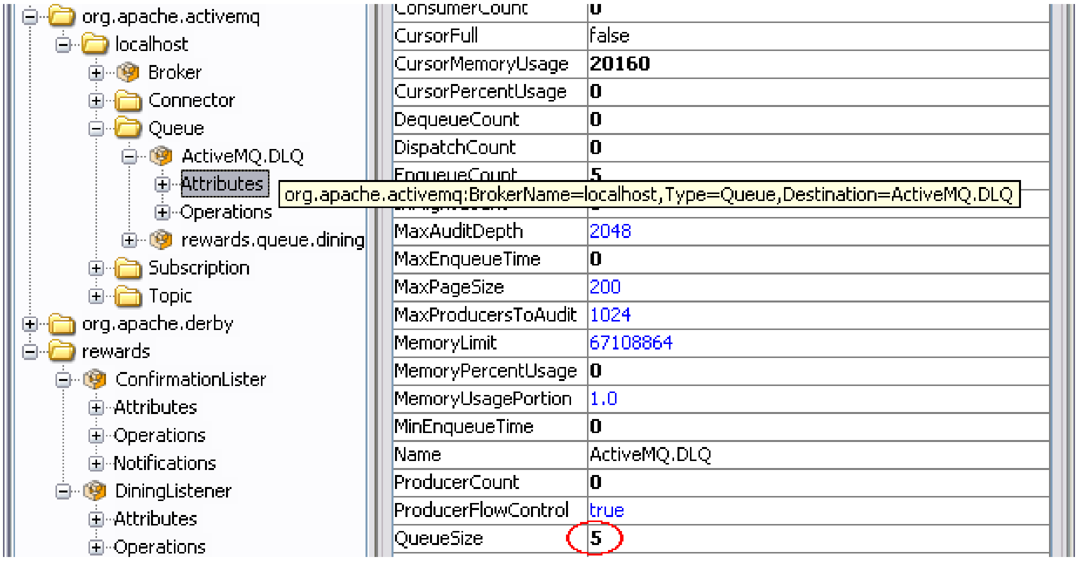
Start the server again, connect using JConsole and set the *CauseErrorAfterReceiving* attribute of the *DiningListener* to *true* again. Now send 5 more messages. This time, the output you see in the Console will be different and looks like this for each message:

```

DEBUG: rewards.messaging.server.DiningListener -
         Received Dining with amount $80.93
WARN : org.springframework.jms.listener
                        .DefaultMessageListenerContainer -
         Execution of JMS message listener failed,
                         and no ErrorHandler has been set.
java.lang.RuntimeException: error after receiving dining
               with amount $80.93
        at rewards.messaging.server.DiningListener
                        .onMessage(DiningListener.java:48)
...
DEBUG: rewards.messaging.server.DiningListener -
  Received Dining with amount $80.93 (redelivered 1 times)
WARN : org.springframework.jms.listener
                        .DefaultMessageListenerContainer -
         Execution of JMS message listener failed,
                         and no ErrorHandler has been set.
java.lang.RuntimeException: error after receiving dining
                              with amount $80.93
        at rewards.messaging.server.DiningListener
                        .onMessage(DiningListener.java:48)
...
DEBUG: rewards.messaging.server.DiningListener -
   Received Dining with amount $80.93 (redelivered 2 times)
WARN : org.springframework.jms.listener
                         .DefaultMessageListenerContainer -
        Execution of JMS message listener failed,
                          and no ErrorHandler has been set.
java.lang.RuntimeException: error after receiving dining
                              with amount $80.93
        at rewards.messaging.server.DiningListener
                         .onMessage(DiningListener.java:48)
...
DEBUG: rewards.messaging.server.DiningListener -
   Received Dining with amount $80.93 (redelivered 3 times)
WARN : org.springframework.jms.listener
                         .DefaultMessageListenerContainer -
         Execution of JMS message listener failed,
                          and no ErrorHandler has been set.
java.lang.RuntimeException: error after receiving dining
                              with amount $80.93
        at rewards.messaging.server.DiningListener
                         .onMessage(DiningListener.java:48)
...
```

Notice how the same message is delivered a total of four times? Look back at the messages and note the *(redelivered # times)* comment on each of the last three lines.

What you see happening here is that after the error, the message is put back in the queue. If this error was caused by a temporary situation, like a network hiccup that prevented the application from communicating with the database, a redelivery would probably succeed. This is typically called a *transient* error. You can configure ActiveMQ on how it should do redeliveries: how often should it retry before giving up, how long should it wait between redeliveries, etc. If you're interested, have a look in `InfrastructureConfig` at the properties that are available on the `org.apache.activemq.RedeliveryPolicy` that's configured.

However, our error is not transient, so the redelivery doesn't help. To prevent a message that cannot be processed from blocking the entire application, ActiveMQ will move the message to a so-called *dead letter queue* after the maximum number of redeliveries it's configured with (3, in our case). Use JConsole to verify that there's a new dead letter queue defined now that has our 5 messages.



```
org.apache.activemq
  localhost
    Broker
    Connector
    Queue
      ActiveMQ.DLQ
        Attributes
        Operations
      rewards.queue.dining
    Subscription
    Topic
org.apache.derby
rewards
  ConfirmationLister
    Attributes
    Operations
    Notifications
  DiningListener
    Attributes
    Operations
```

org.apache.activemq:BrokerName=localhost,Type=Queue,Destination=ActiveMQ.DLQ

ConsumerCount	0
CursorFull	false
CursorMemoryUsage	20160
CursorPercentUsage	0
DequeueCount	0
DispatchCount	0
EnqueueCount	5
MaxAuditDepth	2048
MaxEnqueueTime	0
MaxPageSize	200
MaxProducersToAudit	1024
MemoryLimit	67108864
MemoryPercentUsage	0
MemoryUsagePortion	1.0
MinEnqueueTime	0
Name	ActiveMQ.DLQ
ProducerCount	0
ProducerFlowControl	true
QueueSize	5

This is much better, as messages don't get lost anymore. However, it doesn't take care of all problems, as we'll see in the next step.

5.3.2.3. Step 6: Cause an error after processing the message

Using JConsole, reset the *CauseErrorAfterReceiving* flag by setting it to *false* again, but this time set the *CauseErrorAfterProcessing* flag to *true*. As you can see in the code for the *DiningListener*, this causes an exception to be thrown *after* the call to `rewardAccountFor(Dining)`, which means that the database has already been updated. An example of how this could happen in a real system is when a database transaction commits successfully, but the subsequent JMS transaction commit fails because of a network issue.

Run the `StartSender` application again to send 5 new messages. The output in the console will be very similar to what you just saw; each message will be redelivered three times and then sent to the dead letter queue. However, check the *NrOfConfirmations* attribute of the *ConfirmationLister* again: it has just increased by 20! What happened is that each message was received, processed, and then rolled back to be put back in the queue four times (1 delivery plus 3 redeliveries). The rollback of the JMS transaction happens *after* the local database transaction has already been committed!

Using local transactions, there's not much we can do to avoid these duplicate messages. Had processing been idempotent, then this wouldn't matter, but in our case we want to avoid processing a message more than once. One way to deal with this is to detect redeliveries and don't process them if they have been processed successfully already. To completely avoid these duplicates however, we need a *global transaction*: only this guarantees so-called *once-and-once-only* delivery.

5.3.2.4. Step 7: Cause an error after sending the confirmation message

Also try what happens if you reset the *CauseErrorAfterProcessing* flag back to false and set the *CauseErrorAfterSending* flag to *true*. In this case, the sending happens with a transactional `JmsTemplate` which will automatically use the same JMS `Session` as was used for receiving the `Dining` message: this means that the JMS transaction will still roll back, including the sending of the confirmation, but only after the database transaction has been committed already. The net effect is the same as with the previous step. Had we not used a transactional template, then the confirmations would be sent while the receive would be rolled back. That would be pretty bad: you'd inform clients that you've processed their requests once, where in reality you'll see them again because they were put back on the queue.

Stop the server now by pressing *Enter* in the Console window. In the next module, you'll see how a global transaction can help us out if we require once-and-once-only delivery guarantees.

Chapter 6. tx-xa: Using Global JTA Transactions

6.1. Introduction: Using Global JTA Transactions for Atomic Behavior Across Multiple Transactional Resources

In this lab you will gain experience with using JTA for global transactions. This enables atomic behavior for operations across multiple transactional resources in a single transaction. In this case, these resources are a messaging provider and a database.

What you will learn:

1. The need for XA for truly atomic operations across multiple transactional resources
2. Configuring a JTA transaction manager in Spring

Specific subjects you will gain experience with:

1. Atomikos Transaction Essentials
2. Spring's *JtaTransactionManager*
3. Spring's support for using an external transaction manager in your *DefaultMessageListenerContainer*

Estimated time to complete: 30 minutes

6.2. Quick Instructions

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

1. Add JTA support to the server's infrastructure. In the `InfrastructureConfig` configuration class, replace the type of `transactionManager` to `org.springframework.transaction.jta.JtaTransactionManager`. Set the two properties for this bean: `transactionManager` and `userTransaction`, using the following Java code (TODO 01):

```
@Bean PlatformTransactionManager transactionManager() throws Exception {
        JtaTransactionManager transactionManager = new JtaTransactionManager();
        transactionManager.setTransactionManager(userTransactionManager());

        UserTransactionImp userTransaction = new UserTransactionImp();
        userTransaction.setTransactionTimeout(300);
        transactionManager.setUserTransaction(userTransaction);

        return transactionManager;
}

@Bean(destroyMethod="close") UserTransactionManager userTransactionManager() throws Exception {
        UserTransactionManager userTransactionManager = new UserTransactionManager();
        userTransactionManager.init();
        return userTransactionManager;
}
```

2. Replace `EmbeddedDataSource40` with its XA-aware variant `EmbeddedXADataSource40`, and wrap it inside an AtomikosDataSourceBean (TODO 02).

3. Replace `ActiveMQConnectionFactory` with `ActiveMQXAConnectionFactory`, and wrap it inside an AtomikosConnectionFactoryBean (TODO 03).

4. Open the `JmsRewardsConfig` configuration class, and set up the `transactionManager` property on the `DefaultJmsListenerContainerFactory` bean (TODO 04).

5. Start the server by running the `StartServer` application.

6. Run the `StartSender` application. Start JConsole and verify that there are 5 new confirmations in the database and 5 new messages on the confirmation queue.

7. In JConsole, set the `CauseErrorAfterReceiving` flag of the `DiningListener` to `true` and verify that the behavior is the same as before when you send new messages.

8. Reset the `CauseErrorAfterReceiving` flag to `false` again and now set the `CauseErrorAfterProcessing` flag to `true`. Run the `StartSender` application again and verify that no new confirmations were added to the database this time.

9. Reset the `CauseErrorAfterProcessing` flag to `false` again and now set the `CauseErrorAfterSending` flag to `true`. Send new messages again, and verify that this time both transactional resources are rolled back.

Congratulation, you have successfully completed the lab!

6.3. Instructions

In this lab you'll rework the application from the previous lab to use global XA transactions with the use of a JTA transaction manager to solve the errors that still exist when using local transactions only.

6.3.1. Introducing JTA to have a global transaction in place for the *DiningListener*

Adding support for JTA, which is the Java API for global transaction management, to a Spring application typically requires no code changes: you only need to change some configuration. This is one of the many benefits of using Spring for transaction management. In this section we'll show you how to do this by using a stand-alone, open-source JTA implementation called *Atomikos*. Most Enterprise Java applications that use JTA will use the JTA support that ships with the application server they're deployed on, but in our case we're not using a full-blown Java EE server with JTA support.

6.3.1.1. Step 1: Add JTA support to the server's infrastructure

Open the `InfrastructureConfig` configuration class. Complete *TODO 01* by replacing the `transactionManager` bean of type `DataSourceTransactionManager` with a bean of the same name of type `org.springframework.transaction.jta.JtaTransactionManager`. This bean allows Spring to let your transactional resources participate in global transactions managed by an external JTA transaction manager. You need to set two properties for this bean: *transactionManager* and *userTransaction*. The configuration of these properties is highly Atomikos-specific, so we'll simply provide you with the Java configuration you need to use.

```
@Bean PlatformTransactionManager transactionManager() throws Exception {
        JtaTransactionManager transactionManager = new JtaTransactionManager();
        transactionManager.setTransactionManager(userTransactionManager());

        UserTransactionImp userTransaction = new UserTransactionImp();
        userTransaction.setTransactionTimeout(300);
        transactionManager.setUserTransaction(userTransaction);

        return transactionManager;
}

@Bean(destroyMethod="close") UserTransactionManager userTransactionManager() throws Exception {
        UserTransactionManager userTransactionManager = new UserTransactionManager();
        userTransactionManager.init();
        return userTransactionManager;
}
```

Note on the transactionManager configuration

In a Java EE server, you typically wouldn't need this: just adding a `<tx:jta-transaction-manager/>` would be enough to define the *transactionManager*. If you were using Java configuration, you would directly use the appropriate transaction manager implementation, e.g. `WebLogicJtaTransactionManager`. We only need this configuration because we're using a stand-alone JTA transaction manager not provided by our deployment environment.

Now implement *TODO 02* by replacing the current `EmbeddedDataSource40` with its XA-aware variant `EmbeddedXADataSource40`. To make this `DataSource` participate properly in the Atomikos-managed global transactions, Atomikos provides a wrapper for the data source called `com.atomikos.jdbc.AtomikosDataSourceBean`. Wrap the current *dataSource* bean definition with this type and set its *xaDataSource* property to the existing bean by turning that into an *inner bean*. Because this is again very Atomikos-specific, we'll provide you with the complete bean definition here:

```
@Bean(destroyMethod="close") DataSource dataSource() throws Exception {
        AtomikosDataSourceBean atomikosDataSourceBean = new AtomikosDataSourceBean();
        atomikosDataSourceBean.setUniqueResourceName("XADBMS");

        EmbeddedXADataSource40 dataSource = new EmbeddedXADataSource40();
        dataSource.setDatabaseName("rewardsdb");
        dataSource.setCreateDatabase("create");

        atomikosDataSourceBean.setXaDataSource(dataSource);
        atomikosDataSourceBean.init();
        return atomikosDataSourceBean;
}
```

Now implement *TODO 03* by doing the same thing for the `ActiveMQConnectionFactory`. The complete bean definition for that looks like this:

```
@Bean(destroyMethod="close") ConnectionFactory connectionFactory() throws Exception {
        AtomikosConnectionFactoryBean atomikosConnectionFactoryBean =
        new AtomikosConnectionFactoryBean();
        atomikosConnectionFactoryBean.setUniqueResourceName("QUEUE_BROKER");

        ActiveMQXAConnectionFactory connectionFactory = new ActiveMQXAConnectionFactory();
        connectionFactory.setBrokerURL("vm:broker:(tcp://localhost:61616)?persistent=false");
        RedeliveryPolicy redeliveryPolicy = new RedeliveryPolicy();
        redeliveryPolicy.setMaximumRedeliveries(3);
        connectionFactory.setRedeliveryPolicy(redeliveryPolicy);

        atomikosConnectionFactoryBean.setXaConnectionFactory(connectionFactory);

        atomikosConnectionFactoryBean.init();
        return atomikosConnectionFactoryBean;
}
```

Note on the dataSource and connectionFactory configurations

In a Java EE application, these resources would be configured in the server and not in your application. You wouldn't need to wrap the `EmbeddedXADataSource40` and `ActiveMQXAConnectionFactory` then, as the container would take care of everything. In your Spring application, you'd simply use a JNDI lookup. Again, we only need to do this because we're using a stand-alone JTA transaction manager.

6.3.1.2. Step 2: Make the `DefaultMessageListenerContainer` participate in the JTA transaction

You're almost done with the JTA configuration; we just need to tell the JMS listener container to use the JTA transaction manager. We don't need to do this for the JDBC operations performed by the `RewardNetwork` implementation, as these will automatically use the `Connection` enlisted with the current JTA transaction.

Open the `JmsRewardsConfig` and complete *TODO 04* by setting the `transactionManager` property on the `jmsListenerContainerFactory`. Leave the *sessionTransacted* attribute in place: in a Java EE setting you wouldn't need it, as the value of the boolean flag passed into `javax.jms.Connection.createSession(flag, ackMode)` is simply ignored with JTA, but Atomikos requires it to be *true* to start a global transaction!

That's it: you've now switched from using two local transactions to using a global transaction managed by a JTA transaction manager! Let's have a look at the result. Notice that no code changes were required.

6.3.2. Reviewing the result of the same errors under a global transaction

Now that you've configured the use of a JTA transaction, start the server by running the `StartServer` application. Make sure you use the right project, don't start the server from the jms-tx lab again by accident!

6.3.2.1. Step 3: Run the application without errors

First run the `StartSender` application to see if the application still works as expected when no errors occur. As before, be sure to add the VM argument '-Dcom.sun.management.jmxremote' to enable JMX monitoring. Start JConsole and verify that there are 5 new confirmations in the database and 5 new messages on the confirmation queue.

6.3.2.2. Step 4: Cause an error after receiving the message

Set the *CauseErrorAfterReceiving* flag of the *DiningListener* to *true* and verify that the behavior is the same as before when you send new messages: they will be redelivered three times and then forwarded to the dead letter queue.

6.3.2.3. Step 5: Cause an error after processing the message

Reset the *CauseErrorAfterReceiving* flag to *false* again and now set the *CauseErrorAfterProcessing* flag to *true*. Remember that this previously caused the redelivered messages to be processed over and over again. This time, you should see once-and-once-only delivery. Run the `StartSender` application again and verify that no new confirmations were added to the database this time.

6.3.2.4. Step 6: Cause an error after sending the confirmation message

Reset the *CauseErrorAfterProcessing* flag to *false* again and now set the *CauseErrorAfterSending* flag to *true*. Remember that this previously caused the JDBC transaction to commit, effectively ignoring the error, while the JMS transaction rolled back, just like with *CauseErrorAfterProcessing*. Send new messages again, and verify that this time both transactional resources are rolled back.

When you're done with this last step, you have successfully completed the lab!

Chapter 7. si-intro: Refactoring from JMS to Spring Integration

7.1. Introduction

In this lab you will learn how to set up a basic messaging application using Spring Integration. We will start off with a use case fully implemented using JMS (based on the `jms` lab) and refactor it into a Spring Integration based solution.

What you will learn:

1. How to define basic Spring Integration components using the spring-integration namespace
2. How to use Gateways and Adapters to decouple invoking logic from Spring Integration
3. How to use a Service Activator to decouple invoked business code from Spring Integration
4. How to run a Spring Integration application from a JUnit test
5. How to visualize a Spring Integration flow with STS

Specific subjects you will gain experience with:

1. Spring Integration Channels
2. Spring Integration Gateway
3. Spring Integration Service Activator
4. Spring Integration Adapters
5. Spring Integration Editor

Estimated time to complete: 45 minutes

7.2. Quick Instructions

If you feel you have a good understanding of the material, follow the TODOs listed in the `Tasks` view in Eclipse/STS.

7.3. Instructions

7.3.1. Inspect the current application

(TODO 01) The start of this lab is basically the solution of the `jms` lab: we have a JMS-based `DiningProcessor` that sends `Dining` messages to the server, which then uses a message listener to send the payload to the `rewardNetwork`. The resulting `RewardConfirmation` is then placed on another queue, from where it's picked up by the client using another message listener which passes the payload to a `RewardConfirmationLogger`.

Check out the code and configuration to make sure you understand it, and then run the `DiningProcessorTests` to verify that it passes.

7.3.2. Refactor to Spring Integration

(TODO 02) You're going to replace the server side's JMS configuration with Spring Integration, to make it easier to add aditional processing components to the incoming message flow. First look at `jms-rewards-config.xml` to see what the current application does: it delegates straight to the `rewardNetwork` using a message listener.

Now open up `spring-integration-rewards-config.xml`. Notice that it uses the `integration` namespace as the default. This makes it easier to add Spring Integration components, but you'll need to prefix regular bean definitions with "`beans:`". It also defines the Spring Integration jms and mail namespaces.

Add a `<int-jms:inbound-gateway>` element that will replace the message listener defined in the previous file. Add a `request-destination-name` and `default-reply-queue-name` attribute with the names of the request and response queues. Also add a `request-channel` attribute that you set to `dinings`.

(TODO 03) Define a simple direct channel called `dinings`. Note that we're not specifying a reply channel for now, so a temporary one will be created automatically when a new JMS message is received.

(TODO 04) We need to define a <service-activator /> that would call `rewardNetwork.rewardAccountFor(...)` when there is a message on the `dinings` channel. You don't have to specify the method in this case, since the `RewardNetwork` only has a single method. Note that we're not specifying an output channel, so the resulting `RewardConfirmations` will be sent back to the inbound gateway as reply messages automatically, based on the `replyChannel` header of each message.

Compare the contents of the integration configuration file with that of the `jms-rewards-config.xml`. When you've verified that you've configured the right queue names, rename the `jms-rewards-config.xml` file to `jms-rewards-config.xml.old` since we shouldn't need it anymore.

Note

Renaming the file removes it from use but allows you to refer to it later if you've made a mistake. The `DiningProcessorTests` test class simply loads all configuration files in the `rewards.messaging` package using a special wildcard syntax ("classpath*:rewards/messaging/*-config.xml") and doesn't refer to the filename explicitly.

On the bottom of the editor for the integration configuration file, click the 'integration-graph' tab to see a graphical representation of what you've configured so far. Note that this is not just a read-only view, but an actual editor that shows components for all the namespaces that you have configured in your file.

Run the `DiningProcessorTests` again, it should still pass. From the client's point of view nothing has changed. However, it has now become much easier to make changes and add new components to the application when new JMS messages arrive!

7.3.3. Log incoming Dining messages using a wire tap

(TODO 05) Now that that Spring Integration is in place, we're going to add some logging. As seen in the slides, it is possible to log information using a `wire tap`. Based on the example shown in the slides, complete `TODO 05` and `TODO 06`. Messages should be logged in full (not just the payload).

Run the test again and check that you're seeing the output of the `LoggingHandler` appearing in the Console. Note the `replyChannel` and JMS-related headers that were added automatically.

7.3.4. Send emails for confirmations

(TODO 07) We now have a new requirement: the server should send an email whenever a reward has been processed successfully. Luckily we're already using Spring Integration, so adding this functionality to the application is easy!

First define a new `<publish-subscribe-channel>` called `confirmations`. Make sure to update the inbound gateway and the service activator to use it as their `reply-channel` and `output-channel`, respectively. Run the test again to make sure you didn't break anything.

(TODO 08) Now that you're using a pub sub channel it's easy to add aditional endpoints. First, define a new Spring bean of type `rewards.RewardMailMessageCreator` and name it `rewardMailMessageCreator`. Note that you'll have to prefix the tag with `beans:`. Add a `<constructor-arg>` that references the `accountRepository` bean. Have a look at the class's implementation to see how it creates mail messages based on a `RewardConfirmation`.

(TODO 09) Then add another `<service-activator>` that passes messages from the `confirmations` channel to your new bean and puts the output on a channel called `mails`: make sure to also add a channel definition for that.

(TODO 10) To actually send the email, finally define a `<int-mail:outbound-channel-adapter>` element that send messages from the `mails` channel to `localhost` as the mail host. That's it: your application is now sending emails for every `RewardConfirmation` that's generated by the `rewardNetwork`!

Warning

The adapter will use the default SMTP port 25 by default to talk to a server you'll see in the next step: if you're running a local mail server on that port already then this won't work. In that case, please specify a non-default port number that doesn't conflict with things already running on your machine, like 2525, using the `port` property.

7.3.5. Test the mail sending

To actually test the email sending we're going to use a local embedded mail server that's just for testing purposes: have a look at the `rewards/mail-server-config.xml` file under `src/test/resources` to see how that test server is defined. When you used a non-default port number in the previous step, then please add a `<property name="port" value="2525"/>` to the mail server bean using the same port number!

(TODO 11) Now switch back to the test and add `/rewards/mail-server-config.xml` to the list of files configured in the `@SpringApplicationConfiguration` annotation.

(TODO 12) Then add an @Autowired field of type `Wiser` (the class defined in the configuration file). In the test method, verify that 5 emails have been received after sending the dinings and that the email address of the receiver of the first message is 'keith@example.com' (this mail address is called the 'envelope receiver' in Wiser, just use code assist in Eclipse to find the property).

When your updated test passes, you've succesfully completed the lab! If you have some time left, you can continue with the bonus steps.

7.3.6. BONUS STEP 1: Refactor the client as well

The `DiningProcessor` used by the test is still implemented using plain Spring JMS, with a `JmsTemplate` and a message listener. Refactor this solution to also use Spring Integration instead by updating the `client-config.xml`. We won't spell out the detailed steps this time, but this is what you'll need:

1. A regular gateway for the `DiningProcessor` interface (you can remove the `JmsDiningProcessor class`)

2. A JMS outbound gateway to send the dinings and receive the confirmations
3. A service activator or outbound channel adapter to invoke the `log` method of the `confirmationLogger`

Note

Make sure to use unique channel names, since the client and server are running as a single application in the test!

Note

When you're using a JMS outbound gateway, it will add a JMSReplyTo header to your JMS message (you can verify this by looking at the logging channel adapter's output). That means that you can remove the `default-reply-queue-name` attribute on the JMS *inbound* gateway in the server-side config file, since Spring will automatically send the confirmation reply message to the right queue now!

When you're done, the test should still pass unchanged.

7.3.7. BONUS STEP 2: Simplify the Spring Integration configuration

Open the `spring-integration-rewards-config.xml` file again. Notice that the `mails` channel is only coupling the service activator to the outbound gateway. If that's the case, there's a shortcut to define the channel implicitly: delete the channel definition and update the mail adapter to have `id="mails"` now instead of `channel="mails"`. This will automatically define a direct channel with the name of the adapter's id that will act as its input channel.

If you did the first bonus step, you can make similar changes in your client's config file. The `request-channel` of the JMS outbound gateway will be defined automatically, so you don't need an explicit channel definition for it, and if you give your outbound channel adapter an id with the name of the `reply-channel` configured in the outbound gateway you can remove the channel definition for those confirmations as well.

Chapter 8. si-configuration: Using an Idempotent Receiver and Filter invalid Dinings

8.1. Introduction

This lab involves the following use cases:

1. Refactor the messaging configuration to ensure idempotency
    a. Prevent duplicate messages from reaching the Reward Network
    b. Send confirmations no matter what
2. Filter out invalid Dinings

Topics covered

1. Idempotent Receiver pattern
2. Filtering Messages
3. Working with the global errorChannel for polling receivers

Specific subjects you'll gain experience with

1. The *chain* element
2. Implementing a custom handler
3. Defining a filter using SpEL
4. Bridging a channel for testing purposes

Estimated completion time: 40 minutes

8.2. Instructions

In this lab you will improve the Reward Network application to allow the Credit Card Processors to hand off their Dinings faster.

An Idempotent Receiver is a component that ensures that it will be in the same state after each reception of a certain message. So no matter how many times you receive a certain Dining, after processing there should be exactly one Reward for it. This is something we can guarantee even without using distributed transactions. The only thing you need to do to guarantee this is to check if you have already processed the Dining and only process it if you haven't. After ensuring that the Dining was processed you will send a confirmation no matter what.

8.2.1. Part 1. Make the RewardNetwork Idempotent

The Credit Card Processor will send a Dining. What does this dining represent?

A Dining describes a real life event: someone has paid for a dinner with his creditcard. It tells you at what time this event happened (timestamp), where it happened (merchant number) and finally who did the dining (creditcard number). These three bits of information are *universally unique*. For your convenience this has been captured in the Dining transactionId by the Credit Card Processor.

This allows you to check if you have already processed this Dining and act accordingly. The Credit Card Processor can send the Dining as many times as it likes and the RewardNetwork can confirm the reward for it as many times as requested, as long as we ensure the reward isn't duplicated.

8.2.1.1. Set up the messaging system

We're going to work with a simplified setup that doesn't involve JMS or email, since it's not needed for this lab. The context has been split up in three parts. You will be working on *spring-integration-idempotent-receiver-config.xml* first. Open the file and examine its contents: we're using a *dinings* and *confirmations* channel with a *service-activator* that ties them together by calling the *rewardNetwork* again. However, there's no inbound gateway that enters *Dining* messages into the system, the testing code will do that directly instead.

Open up *IdempotentRewardNetworkIntegrationTests* and complete *TODO 1* within the *idempotence* test method. Run the test case.

The second assertion should fail. This indicates that the *RewardNetwork* is being invoked multiple times for the same dining event. In other words, our reward processing is *not* idempotent.

Note

Note that this test doesn't use the real *rewardNetwork* and *rewardRepository*, but mocks. Check the *test-context.xml* file to see how the mocks are set up.

8.2.1.2. Prevent duplicate dining submissions

Open up *spring-integration-idempotent-receiver-config.xml* if you haven't already.

The check if a Dining for the transactionId at hand has already been processed needs to happen before calling the *rewardAccountFor(Dining)* method. To accomplish this you're going to wrap the relevant part of the message flow in a <*chain/*> with some extra endpoints. A chain will automatically wire all its children together linearly using direct channels. Wrap a chain around the rewardNetwork service activator (*TODO 02*).

Tip

The chain should have input and output channels, but the service-activator will no longer need any explicit channel configuration.

8.2.1.3. Send the confirmation

Whenever a message is received a confirmation needs to be sent, even if nothing has happened. Otherwise our service wouldn't be idempotent. More importantly, how else would the Credit Card Processor know that they don't have to retry?

Note

It is possible that the Credit Card Processor already has decided to retry a Dining when you are processing it but before you've sent a RewardConfirmation. This shouldn't be a problem, but you should be aware of this when you're correlating messages in the Credit Card Processors log files, otherwise you might think there is a bug. Just remember what is unique: an actual dining transaction - all communication about it could be duplicated.

Now add another service-activator to the chain *before* the rewardNetwork. For your convenience an AlreadyRewardedConfirmer has been implemented that will side track the message out of the chain in case it was already processed. The only thing you need to do is point the service activator to the *alreadyRewardedConfirmer* bean (*TODO 03*).

Look at the code and configuration for the *alreadyRewardedConfirmer* to see how it works: it returns the received *Dining* when no confirmation was found so that the chain passes that to the *rewardNetwork*, but short-circuits the chain by returning *null* when a confirmation was found. To ensure that this confirmation is still sent to the *confirmations* channel it uses a simple gateway.

Run the *IdempotentRewardNetworkIntegrationTests* again. It should now pass. Once it does, open the *RewardMessagingIntegrationTests*, remove the *@Ignore* on the *sendDiningTwice* method (*TODO 04*) and run the test. Make sure that it passes before you start with the next part of the lab.

8.2.2. Part 2. Filtering broken messages

So far we've worked with valid input for the *rewardNetwork* (and the newly added *alreadyRewardedConfirmer*). But what happens when the input is broken? If handling the Dining happens in the same thread, the caller simply receives the exception that's thrown as a result of the invalid data. To test this, open *InvalidDiningsIntegrationTests* and complete *TODO 05* by sending the *invalidDining* to the "dinings" channel using the provided template. Run the test and check the exception that's thrown.

8.2.2.1. Filter out invalid messages

Instead of allowing invalid Dinings to reach the *rewardNetwork*, we want to filter them out before they reach the service activator. Since you have a chain in place already, that only requires an additional filter as the first endpoint in the chain. Add a filter to the chain that filters out Dinings that have one of their properties set to *null* (*TODO 06*).

Tip

This is a good candidate for using an expression instead of a Java implementation: something in the form of *payload.firstProperty != null and payload.secondProperty != null* ... will do the trick.

Note

If you see an error on the service-activator after adding the filter that says it must be followed by attribute specifications, ">" or "/>" then please ignore it: that's a bug in the tooling.

Run the test again and make sure there's no exception this time.

In some scenarios silently dropping invalid messages is a fine solution. In other cases you might want to forward the message to some discard channel or throw an exception instead: enable the latter by setting the *throw-exception-on-rejection* attibute of the filter to *true* (*TODO 07*). Run the test again to see the difference (it should fail.)

8.2.2.2. Test error handling for asynchronous message handling

With synchronous endpoints, it's easy to tell the difference between successful processing or an exception as you just saw. The situation is very different when the invalid Dining is processed asynchronously on a different thread, however. To see this, open *spring-integration-idempotent-receiver-config.xml* and change the "dinings" channel to a QueueChannel by adding a `<queue>` subelement to it (*TODO 08*). Note that a default poller is already configured in *spring-integration-infrastructure-config.xml*, so no additional configuration is needed.

Run the *InvalidDiningsIntegrationTests* again. Notice that there's no exception this time, even though the filter throws one. That's because the exception was thrown from a different thread, so Spring Integration has wrapped it and sent it to the "errorChannel" channel. A logging channel adapter that logs the exception message is already set up, so the Console view should show the exception. What we want is a good way to test for the exception message to arrive on the *errorChannel*: just looking at the Console is no good for an automated test of course.

A common technique for checking for expected messages in a test is to *bridge* the channel that will contain the expected message to a dedicated test queue channel. You can then receive from that test channel in your test and check the message.

Open *test-context.xml* and complete *TODO 09* by creating a new QueueChannel called "errorTestChannel" and define a bridge that forwards messages from the default "errorChannel" to this "errorTestChannel".

Note

Note that bridging or subscribing to a point-to-point instead of a publish-subscribe channel with an existing receiver in the test context would not be a good idea, as it interferes with the intended receiver. You could use a global wire tap element then instead of a bridge:

```
<int:wire-tap channel="errorTestChannel" pattern="errorChannel" />
```

Then switch to the *InvalidDiningsIntegrationTests* and complete *TODO 10* by using the provided MessagingTemplate to check that there's a message on the "errorTestChannel" with a payload of type *MessageRejectedException*; this is the exception thrown by the filter when it rejects the message with the invalid Dining.

Note

The template is configured with a receive timeout already, so it will simply return *null* when there's no message on a channel that you try to receive from.

When the test passes, switch to the *RewardMessagingIntegrationTests* and remove the *@Ignore* on the *sendInvalidDining* method (*TODO 11*). Make sure the test passes now.

Note

Note that this test uses a diffent approach to check for the expected message: it subscribes a handler to the "errorChannel" and waits to give the system time to process the message before checking the result (after all, it's asynchronous). In general the bridging approach is preferable, as it relies on the receive call to wait for you, whereas this approach requires manual waits. The approach in this test does not require a dedicated test configuration file with the QueueChannel and bridge, though.

When the test passes, you've successfully completed the lab. You might like to tidy the output by suppressing the stack-traces from the exceptions that we are expecting as part of the tests. We don't care about them and they make the tests look like they are failing. (TODO 12).

8.2.3. BONUS: Java configuration refactoring

If you have some time left, you can work on refactoring the XML configuration files to Java configuration. For this purpose a new dependency `spring-integration-java-dsl` has been added to your project. See the project https://github.com/spring-projects/spring-integration-java-dsl/wiki/Spring-Integration-Java-DSL-Reference for more informations. We'll use the same tests, without any modification in the test logic, to validate the configuration refactoring works. *Before* moving on to the refactoring, ensure the `InvalidDiningsIntegrationTests` and `RewardMessagingIntegrationTests` pass.

8.2.3.1. Refactor the jobs configurations to Java

`TODO 13`: Change the `@SpringApplicationConfiguration` annotation in the `RewardMessagingIntegrationTests` class to use the `SpringIntegrationIdempotentReceiverConfig`, `SpringIntegrationInfraConfig` and `SystemTestConfig` configuration classes. The `@SpringApplicationConfiguration` has a `classes` attribute to specify this.

`TODO 14`: Take a look at the configuration classes. `SpringIntegrationIdempotentReceiverConfig` is the configuration entry point for the spring integration workflow configuration. `SystemTestConfig` imports the infrastructure configuration like the datasource and business beans. And `SpringIntegrationInfraConfig` contains Spring Integration infrastucture.

`TODO 15`: The point here is to enable the Spring Integration Java configuration support. To do that, add the `@EnableIntegration` and `@IntegrationComponentScan` annotations into `SpringIntegrationIdempotentReceiverConfig`. This annotations declare the Spring Integration infrastructure and enable component scanning for Spring Integration components. For more informations, see documentation http://docs.spring.io/spring-integration/docs/latest-ga/reference/html/history.html#4.0-enable-configuration.

`TODO 16`: It's time now to configure the `gateway` the Java way. Here, step in `ConfirmationProcessor` and use the `@MessagingGateway` and `@Gateway` annotations to replace the XML configuration (`<int:gateway ...`).

`TODO 17`: Now you're going to create your first channel. Let's go in `SpringIntegrationIdempotentReceiverConfig` and create a method named *'dinings'*. Use for this the factory builder `MessageChannels`. Be careful, the method name will be used to reference the channel elsewhere.

`TODO 18`: move on few lines above in method `diningsFlow` and add this channel as the input channel to the dinings integration flow. As it's the first time we manipulate this flow, use the factory builder `IntegrationFlows` to declare it. Don't hesitate to go back to the slides if you don't remember the exact syntax.

`TODO 19`: Now create the output channel. In `SpringIntegrationIdempotentReceiverConfig`, create a method named *'confirmations'*. Be careful, the method name will be used to reference the channel later.

`TODO 20`: move on few lines above in method `diningsFlow` and add this channel as the output of the dinings integration flow.

`TODO 21`: Now, let's configure the filter. Define the filter with the appropriate method and the same filter logic as in `spring-integration-idempotent-receiver-config.xml` (to filter empty dinings). Don't forget to set the *throwExceptionOnRejection* property to *true* (with a Java 8 lambda.) Don't hesitate to go back to the slides if you don't remember the exact syntax.

`TODO 22`: We're getting close to the end of the refactoring. We have to define the service activators. Add these 2 service activators. The first one uses the `alreadyRewardedConfirmer` bean and its `sendConfirmationForExistingDining` method. The second one uses the `rewardNetwork` bean and its `rewardAccountFor` method.

`TODO 23`: Configure a poller. Step in java config class `SpringIntegrationInfraConfig`. Just like in the XML configuration file 'spring-integration-infrastructure-config.xml' we need to configure a default poller with a fixed delay of 250 ms.

`TODO 24`: Run the `RewardMessagingIntegrationTests`, it should pass if the Java configuration is correct.

Congratulations, you refactored the jobs configurations to Java DSL!

Chapter 9. si-advanced: Splitter and File System Integration

9.1. Introduction

First you will implement support for xml input and output. Second you will add support for xml messages that contain multiple dinings. Finally you will add support for receiving these messages from the file system.

Topics covered

1. XML (un)marshalling
2. XPath-based splitters
3. File system channel adapter

Estimated time to complete: 40 minutes

9.2. Instructions

9.2.1. Part 1. Add XML Marshalling to the Input and Output

As we have discussed before it is common practice to integrate with other systems using a common contract. We've already agreed upon an xml contract with the credit card processors, so we will honor that. In this step you will wire an inbound unmarshaller and outbound marshaller that have been provided already. You will be working on *spring-integration-marshalling-config.xml*.

9.2.1.1. Wire the DiningRequestUnmarshaller

Open *MarshallingIntegrationTests* and complete `TODO 01` in `inboundDiningXml()`.

Tip

Notice that the input is generated from the contents of *dining-sample.xml* within the same package as *MarshallingIntegrationTests*.

Run the test and see that it fails. From the failure you can see that the XML File you're sending hasn't been unmarshalled. Open *spring-integration-marshalling-config.xml* and replace the bridge between xmlDinings and dinings with an unmarshaller (`TODO 02`). Use an *int-xml:unmarshalling-transformer* element to do the job and reference the predefined `diningRequestUnmarshaller`.

Note

The *int-xml* namespace has already been configured.

Rerun the test. It should now pass. If you're interested in the code that does the actual unmarshalling, have a look at the `DiningRequestUnmarshaller`: it simply extracts some attributes using XPath from the given XML to create a `Dining` instance.

9.2.1.2. Wire the RewardConfirmationMarshaller

Now that the dining is properly unmarshalled you'll run into problems when the confirmation comes back. Complete `TODO 03` in `outboundConfirmation()` in *MarshallingIntegrationTests* to reproduce this problem.

You need to replace the bridge between *confirmations* and *xmlConfirmations* with a proper transformation. You can use an *XmlPayloadMarshallingTransformer* to wrap the marshaller and a *ResultTransformer*. Lucky for you the rewardConfirmationMarshaller and resultToStringTransformer beans are already wired, you only have to configure the wrapping transformer. Furthermore, Spring Integration provides namespace support for the *XmlPayloadMarshallingTransformer*, so you just need to define an <int-xml:marshalling-transformer/> and provide references via the "marshaller" and the "result-transformer" attributes (`TODO 04`).

When you have properly configured the confirmation marshaller, the test should pass again.

9.2.1.3. Wrapping up

Open up the *RewardMessagingIntegrationTests*. There is an ignored `sendSingleXmlDiningTwice()` test that should now pass when you enable it (`TODO 05`). If it does, you are done with this part of the lab.

9.2.2. Part 2. Add the Ability to Deal with Batches of Dinings

In this step you will implement the logic in *spring-integration-xml-splitting-config.xml*. Messages of two types may now be sent to the *mixedXmlDinings* channel. First there may be single dining messages:

```
<dining transaction-id="some id">
  ...
</dining>
```

But now we must also support messages of the form:

```
<dinings>
        <dining ... />
        <dining ... />
</dinings>
```

To pull individual dinings out, but leave the singular messages unmodified, we're going to use an xpath splitter.

9.2.2.1. Finish *BatchedDiningSplitterIntegrationTests*

In the *BatchedDiningSplitterIntegrationTests* test class, finish `TODO 06-08` in the *inboundMultipleDiningXml* test method. The test will fail because in *spring-integration-xml-splitting-config.xml* we've simply used a file to string transformer to wire the channels together, which only makes the *inboundSingleDiningXml* test succeed.

9.2.2.2. Fix the configuration by adding the splitter

Now you are going to fix the configuration by replacing the current transformer with an xpath splitter. Xml namespace support is added to the file already for your convenience under the *int-xml* prefix.

Tip

If you need to do a lot of xml configuration it might be more convenient to add http://www.springframework.org/schema/integration/xml as the default namespace.

You need to configure a splitter that has the appropriate input and output channels and additionally has an *<int-xml:xpath-expression ... />* as a sub element (`TODO 09`). Use code completion to figure out the details.

If you get in trouble designing the appropriate xpath expression ask your instructor for help. If you have the right expression, both the first and second test should be green *without requiring any changes to the first test*.

When this works remove the @Ignore from *RewardMessagingIntegrationTests.sendMultipleDinings* and ensure that it passes (`TODO 10`). Congratulations, the second feature is implemented!

9.2.3. Part 3. Add File System Integration as Input for the Dining XML

Some Credit Card Processors have asked to be able to send files with Dinings as a backup option. This is a use case that might be better solved with Spring Batch, but for small files we can still use Spring Integration alone.

9.2.3.1. Add an inbound Channel Adapter for Files

Start by reviewing the *filesReceived* test method within *InboundFileDiningIntegrationTests*. It is currently ignored, but since you will be implementing the functionality in this step, go ahead and remove the *@Ignore* annotation (`TODO 11`). Then run the test. Of course at this point it will fail.

Open *InboundFileDiningIntegrationTests-context.xml*. Currently it is only importing the other configuration files, also used by the full system integration test. You will be adding a component so that dining xml files can be retrieved from a directory. For your convenience, the 'int-file' namespace has already been configured.

Add a *<int-file:inbound-channel-adapter/>* element and specify the directory as a relative path to the location containing the *dining-sample.xml* and *dinings-sample.xml* files (`TODO 12`).

Note

Unlike many other resources you've been configuring, here we'll be specifying the file location so it should start with *src/test/resources* rather than the root of the classpath. For a relative file path, do not begin with a forward slash.

Because you only want to pick up those files and not the other files within that directory, you should also add the *filename-pattern* attribute with an ant-style pattern that would match those files but not the others.

Tip

Use a `*` character to match 0 or more characters.

Finally, be sure to set the "channel" attribute to the *mixedXmlDinings* channel. The message payload will be `java.io.File` instance, but fortunately the `XPathMessageSplitter` that you configured earlier uses a `DefaultXmlPayloadConverter` that can deal with files containing XML directly.

Once you've configured the inbound file adapter, try running *InboundFileDiningIntegrationTests*. If it passes, you have completed this lab.

9.2.4. BONUS: Refactor the idempotency implementation

In the previous lab you used a chain with an `AlreadyRewardedConfirmer` to check if an incoming dining had already been rewarded, in which case it would short-circuit the chain and forward the `RewardConfirmation` directly to the `confirmationChannel` using a gateway. In this section you'll refactor this implementation to a much simpler solution using a non-loadbalancing dispatcher with failover support.

This is the idea: first, simply assume a confirmation does exist and let a service activator return it. Let this service activator throw an exception if it doesn't exist, and then fail over to the `rewardNetwork` to create it by calling the `rewardAccountFor` method.

9.2.4.1. Simplify the spring integration configuration

In the `spring-integration-idempotent-receiver-config.xml` file, remove the gateway and the `alreadyRewardedConfirmer`. You can delete the corresponding .java files as well. Then remove the wrapping `<chain>` tags and change the first service activator to call the `rewardRepository.findConfirmationFor` method. To ensure that an exception is thrown when no confirmation exists, set its `requires-reply` attribute to `true`. Add the same input- and output-channels to both service activators. Finally, add a `<dispatcher>` to the `dinings` channel and disable load balancing, as we want to ensure that the `rewardRepository` is always invoked first. The ordering in the XML already ensures this, but you might want to add explicit `order` attributes to each service activator to make it clear that you're relying on their ordering.

When you're done, run the `IdempotentRewardNetworkIntegrationTests` (which uses mocks) and `RewardMessagingIntegrationTests` (which uses the real service and repository) to make sure your application still works as expected.

Chapter 10. batch-intro: Introduction to Spring Batch

10.1. Introduction

In this lab you will gain experience with using Spring Batch to define and run batch jobs. You'll define the job itself, consisting of two distinct steps, and will implement the code to receive and send JMS Messages and to update and query the database.

What you will learn:

1. Defining a job using the Spring Batch namespace
2. Implementing *ItemReaders* and *ItemWriters* that contain the batch's logic
3. Using JDBC batched statements through the *JdbcTemplate*
4. Running a batch job with a launcher

Specific subjects you will gain experience with:

1. Spring Batch
2. Spring's *JmsTemplate*
3. Spring's *JdbcTemplate*

Estimated time to complete: 60 minutes

10.2. Instructions

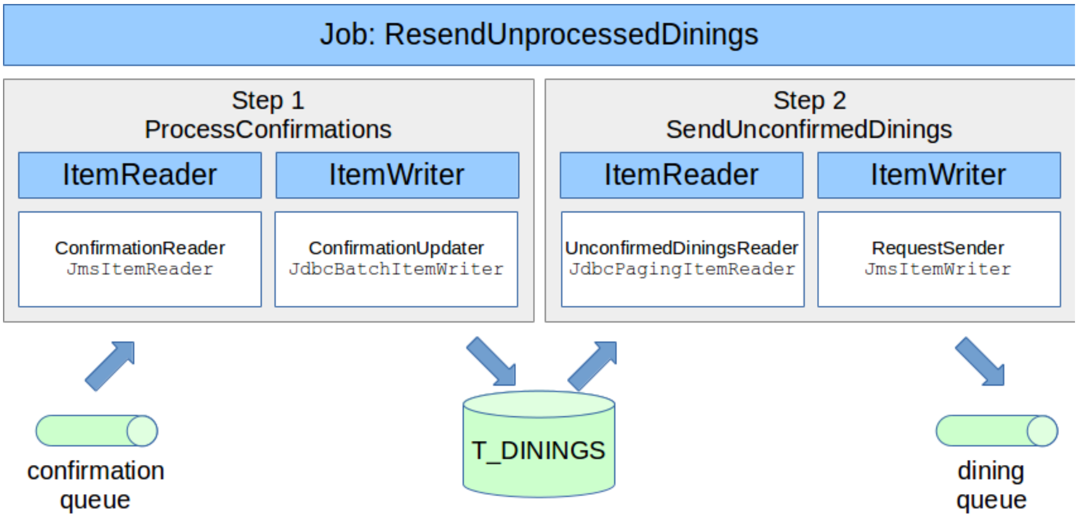
The instructions for this lab are organized into sections. In the first section, you'll define a batch job and implement the first step of the job: processing incoming confirmation messages by updating the dinings in the database. In the second section, you'll implement the second step: sending new dining request messages for each dining that has not yet been confirmed.

10.2.1. Use Case

Before we start with writing the lab code, we'll first provide you with a bit of context. The Credit Card Processor (CCP) sends many Dining requests to the Reward Network. However, depending on the communication mechanism that's used and the implementation of the Reward Network itself, not all requests will be received and processed successfully by the Reward Network application. For requests that have been processed successfully, the Reward Network sends back confirmations to the CCP.

Once a day, the CCP processes all these confirmations and updates its database that holds the requests that have been sent to mark them as confirmed. The requests that are still marked as unconfirmed in the database are then sent again. This way, we can guarantee that a request will eventually be processed. Notice that the Reward Network will automatically ignore requests it has already processed (it's *idempotent*), so it's safe for the CCP to simply resend a request for which no confirmation was received. Processing all of the confirmations and resending all the unconfirmed requests could interfere with the regular real-time operations, so it's performed once a day by a batch job that runs during the night.

An overview of the complete job can be found in the following diagram:



10.2.2. Implementing the First Step and Defining the Job

In this section you'll implement the first step of the batch job: processing confirmation messages from a queue by updating the corresponding dinings in the database. As can be seen in the diagram, this involves creating an *ItemReader* that receives JMS Messages and an *ItemWriter* that updates the database based on the received confirmations. You'll then start the job definition using this first step.

10.2.2.1. Step 1: Define the *confirmationReader*

The first step starts with receiving incoming JMS Messages. These messages contain Confirmation objects, each containing the dining ID to be updated.

Tip

Because we're implementing an offline batch-process and not a real-time system, we can simply pull these messages of the queue synchronously ourselves, instead of using a Spring *MessageListenerContainer* that listens for incoming messages and then calls our application. That means Spring's *JmsTemplate* is fine for our purposes.

TODO 01: Open *batch-processors-config.xml* and define a *JmsItemReader*. Give this bean the ID *confirmationReader* (you can use any ID, but this ID will be referred to in a later step). JmsItemReaders use JmsTemplates to read from JMS, so we will need to provide one for it to use. We've already configured such a JmsTemplate for you named *receivingJmsTemplate*. If you want, you can examine how this template is configured by finding this bean definition within the *jms-ccp-config.xml* file. It is setup to read *confirmation.queue*, which is the queue in which our test data will be populated when we run our test below.

Tip

If the JMS messages contained XML, we would probably unmarshal them first into Confirmation objects. This can be done by the JmsTemplate used to read the queue by using a MarshallingMessageConverter.

10.2.2.2. Step 2: Define the *ConfirmationUpdater*

The Spring Batch framework will call our *confirmationReader* a specified number of times and collect the results in a *List*. It then passes this list to an *ItemWriter* that can then process them, for example by updating data in a database. This is exactly what you need to do: inside the T_DINING table, each dining that has been confirmed should have "1" in the CONFIRMED column. Fortunately, Spring Batch already provides a special item writer that performs JDBC batched updates against a database. We will configure this off-the-shelf component to update the T_DINING table for us.

TODO 02: Within *batch-processors-config.xml*, define a `JdbcBatchItemWriter`. Give this bean the ID *confirmationUpdater*. To perform JDBC batched updates, this bean will need a dataSource set, and also the actual SQL statement to be executed. For the SQL statement provide *update T_DINING set CONFIRMED=1 where ID=?*. This statement will set a flag on the database for any row that matches the dining ID (which is exactly what our Confirmation object provides).

The `JdbcBatchItemWriter` also requires a way to set the placeholders (? symbols) within our SQL statement. To do this, set the *itemPreparedStatementSetter* property to reference the bean *confirmationPreparedStatementPreparer* defined later in the file. This bean will be explained next.

10.2.2.3. Step 3: Understand the PreparedStatementPreparer

TODO 03: All ItemWriters in Spring Batch receive a List of items to process. In our case, we wish to perform a single JDBC batch update rather than sending separate update statements to the database. JDBC batch updating is much more efficient than repeating single updates. (Full understanding of JDBC batching operations is outside the scope of this exercise. For further information consult Java's JDBC documentation.)

To issue a batch update, the PreparedStatement used by the `JdbcBatchItemWriter` needs to prepared once for each item in the List (chunk). The `JdbcBatchItemReader` does this by delegating to an object that we provide, an object coded with understanding of the structure of the SQL statement and how it relates to the incoming item. We have provided an object that does this for you and defined it as a bean with id `confirmationPreparedStatementPreparer`.

First examine the SQL statement that we have provided. There is a single placeholder (? symbol) that needs to be 'prepared' with the ID (the confirmation's transaction ID) for each item (Confirmation) in the List.

Next, open and examine the `ConfirmationPreparedStatementPreparer` class. First, note that this class implements `ItemPreparedStatementStetter` and it is typed to Confirmation. Next, note the *setValues* method; this method is called by the `JdbcBatchItemWriter` for each item in the batch, passing the Confirmation with the PreparedStatement to be set. Our SQL statement has only a single parameter to be set, so a single setString() call on the PreparedStatement is used to establish the value as the transactionID for the Confirmation object passed. Note that this code is called once for each Confirmation in the List (chunk), so in our lab there will be 150 updates in each chunk.

TODO 04: When you have added the two bean definitions, run the `ConfirmationIntegrationTests` and make sure that it passes. When it does, move on with the next step to start to define our batch job!

10.2.2.4. Step 4: start the definition of the batch job

Spring Batch has its own domain language for batch-related concepts with a dedicated namespace that allows you to define your jobs using that language directly, instead of through generic *<bean>* definitions.

TODO 05: Open the *batch-job-config.xml* in the *src/main/resources* source folder. Notice how the default namespace is set to the batch namespace URI. That means you don't have to prefix your batch-related elements and you have full code completion support.

Define a *job* with id *resendUnprocessedDiningsJob*. Inside of that job, define a *step* with id *processConfirmationsStep*. This is going to be the first step of our two-step job. Inside the *step* element, create *tasklet* with a *chunk* child element. This syntax allows you to easily configure the typical case of a step that consists of reading, (optionally) processing and writing some data.

This *chunk* requires four arguments to be set: *reader* and *writer* need to refer to the names of the beans you just defined for the *confirmationReader* and *ConfirmationUpdater*, respectively.

Tip

You can use code assist (Ctrl + Space) to complete the names of the beans for your reader and writer defined in the other XML file!

commit-interval needs to be set to the chunk size we want to use. In your case, that value is defined in an external properties file, *batch.properties*, which is read by a *PropertyPlaceholderConfigurer* already defined in the *batch-processors-config.xml* you edited in the previous step. That means you can simply use *${chunk.size}* as the value.

Finally, set the attribute *reader-transactional-queue* to *true*. Transactional readers will be explained more fully in the next chapter, but the short explanation is that Spring Batch will understand that our reader obtains input from a transactional source that will be rolled-back should an error occur (as opposed to a non-transactional source, like a file, that does not exhibit transactional behavior). This affects its behavior during "retry" scenarios, but since this exercise will only deal with non-failure scenarios this setting is not critical for us.

10.2.2.5. Step 5: test the job you have defined

TODO 06: We have created a simple integration test for your job already. Open the *BatchTests* class and implement the test logic. First, use the provided *JobLauncherTestUtils* to run your job: its `launchJob()` method will automatically create unique job parameters, which are used to identify multiple runs of the same job. This guarantees that every test run starts a new job execution.

The `launchJob()` method will block until the entire batch job is complete, which is fine for our test scenario. After the job completes, assert that the *ExitStatus* of the returned *JobExecution* is *ExitStatus.COMPLETED*. Also use the provided *JdbcTemplate* to query the database table for the number of *confirmed* rows. (You can use *queryForObject(String sql, Integer.class)* for this, the query is defined in the *NR_OF_CONFIRMED_DININGS* variable). Assert that 150 rows are effected, which is the number of confirmation messages that are present on the confirmation queue when you start the test. This demonstrates that you've successfully updated the rows in the database based on the incoming messages.

Run the test and make sure it passes. If your test fails but you don't see any error, try setting the log level for *org.springframework.batch* to *debug* in the *log4j.xml*: Spring Batch might swallow exceptions from your reader or writer that it doesn't consider severe enough to fail the step; you will only see these exceptions using the debug log level.

10.2.3. Implementing the Second Step

Since the first step has marked all of the dinings in the database that are confirmed, the second step can create new dining requests for those remaining unconfirmed. We can safely send those requests to the server again, as the server will make sure not to process two requests with the same transaction id twice (it's *idempotent*).

Just like with the first step, this involves implementing an item reader and writer. As before, we will try to use off-the-shelf *ItemReader*s and *ItemWriter*s to eliminate coding whenever possible. Our second step will read from the database (JDBC Reader) and write to JMS (JMS Writer).

10.2.3.1. Step 6: finish the definition of the *unconfirmedDiningsReader*

Reading records from a database using a query as input for a batch step is very common: so common in fact, that we don't need to implement a custom *ItemReader* to do it. We can reuse an existing reader implementation provided by the Batch framework. There are several options, including a reader that scrolls through an open *ResultSet* and one that uses *pagination* to get the data we need in chunks. For this lab we have chosen the latter.

TODO 07: Open the *batch-processors-config.xml* file again. Finish the bean definition for the *unconfirmedDiningsReader* by setting the *queryProvider* property using an inner bean of type *SqlPagingQueryProviderFactoryBean*. Refer back to the slides for an example. You will need to set separate properties for selectClause, fromClause, whereClause, and sortKey as well as the dataSource. For the selectClause you can select all columns using "select *" and the from clause can simply indicate the T_DINING table by indicating "from T_DINING". For the where clause, we want to select all Dining rows that are NOT confirmed so look for the confirmed flag set to 0: "where CONFIRMED=0". Finally, the sortKey must be a unique value to sort the SQL result set by so that distinct queries can be created, and we have such a column in "ID".

This *FactoryBean* will determine the DBMS type being used and will assemble SQL queries that use pagination. You'll need to inject five properties. The first one, *dataSource*, needs to refer to the existing *dataSource* bean. This identifies the *type* of DBMS being used.

The other four are related to the SQL query you want to run:

1. *selectClause*, which is the SELECT part of your query (everything before the FROM)
2. *fromClause*, which is the FROM *tablename* part of your query
3. *whereClause*, which is the WHERE *someColumn* = *someValue* part of your query
4. *sortKey*. The column(s) to sort on. Your SQL query might not sort yet, but to do pagination we need to sort on a column to get repeatable results. You can simply use ID here, as the test data contains incrementally numbered transaction IDs

This SQL query will select all columns of the T_DINING table that have their CONFIRMED column set to *0*.

The reader will execute this query repeatedly, for a distinct "page" of data at a time, until there is no more data.

TODO 08: But how does the *JdbcPagingItemReader* know how to map each row to an instance of *Dining*? For that we must provide a *RowMapper*, just as we would if using a *JdbcTemplate*. An implementation has been provided for you already: examine the *DiningMapper* class to see how it works. This is simple, boring code to write, so we've provided it for you already. Finish by configuring the *rowMapper* property of the *unconfirmedDiningsReader* by injecting it with an instance of `DiningMapper`.

Tip

It's easy to use an *inner bean* for this again. Do this by instantiating the `DiningMapper` within the *rowMapper* property tag.

TODO 09: When you're done, run the `UnconfirmedDiningsReaderTests`, it should pass. When it does, move on to the next step.

10.2.3.2. Step 7: create the definition of the *requestSender*

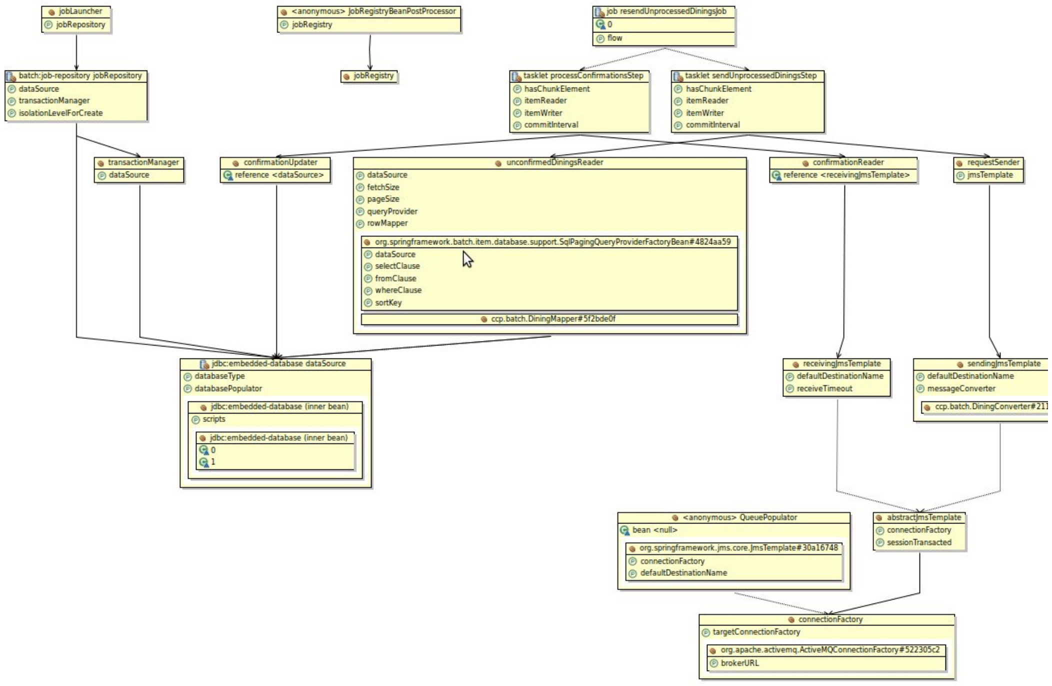
TODO 10: The `Dining` items read in the last step need to be sent to a JMS queue for processing through the Reward Network. Create a bean called *requestSender* of type `JmsItemWriter` that takes care of the sending. It needs its *jmsTemplate* set with a *JmsTemplate* that has a default destination or destination name configured. You can use the existing *sendingJmsTemplate* defined in *jms-ccp-config.xml*.

10.2.3.3. Step 8: create the second step definition for the job

TODO 11: Return to *batch-job-config.xml* and add a second step called "*sendUnprocessedDiningsStep*" with a tasklet and chunk. Use the correct names for the new reader and writer and don't include the *reader-transactional-queue* attribute for this *<chunk>*. Set the *chunk-interval* attribute as you did before.

Also, set the *next* attribute of the first step to the id of this second step to make sure the new step is executed after the first one! Spring Batch requires you to be explicit about the step ordering and doesn't assume sequence based on the position in the job. We will see the reason later; Spring Batch allows steps to be run in parallel.

When you've done this, the job definition is complete. If you would like to see a graphical representation of all the components, go to the Spring Explorer View. Open the `batch-intro` project and navigate to the *system-test-config.xml* file. Right-click it and select Open Graph. The result should look like this:



If your job looks good, now let's test it to see if it works!

10.2.3.4. Step 9: finish the *BatchTests* and run it

TODO 12: Return to the *BatchTests* class again and implement a check that there were really 150 messages sent. To make this easier, we have provided a *QueueViewMBean* which has a `getQueueSize()` method. We're using a little trick here: ActiveMQ defines a JMX MBean for each queue that will let you check some attributes and perform some operations. We've used Spring's JMX support to obtain a reference to the MBean for the dining queue: you can see how this is done in the *system-test-config.xml* configuration file.

Run the test and make sure it passes. When it does, you have successfully completed this lab!

Chapter 11. batch-restart-recovery: Spring Batch Restart and Recovery

11.1. Introduction

In this lab you'll see how Spring Batch allows dealing with errors during batch processing, both by skipping invalid items and by allowing you to restart failed job instances where the job resumes where it left off.

What you will learn:

1. Configuring skipping and restarting
2. How Spring Batch keeps track of your job executions to allow resuming a jailed job instance
3. How to be informed of skipped input items
4. How to limit the amount of allowed restarts for a step

Specific subjects you will gain experience with:

1. Batch namespace support for skiping and restarting
2. Implementing a `SkipListener`

Estimated time to complete: 45 minutes

11.2. Instructions

The instructions for this lab are organized into sections.

11.2.1. Completing the initial batch job

You're going to use a batch job that reads lines of dining information from a comma-separated value file, calls our `RewardNetwork.rewardAccountFor()` method from an `ItemProcessor`, and then writes a simple report to the Console. The first thing to do is to complete the job definition.

11.2.1.1. Inspect the job configuration so far

Start with the `batch-execution-config.xml` file in the `src/main/resources` folder. Here the job registry and repository are defined, as well as a simple job launcher and some other infra-level components. Like in the previous lab, the actual batch job is defined in a separate file. Open the `batch-job-config.xml` file. Notice that it includes the `app-config` from the common `rewards` project, which means that the `rewardNetwork` bean and its dependencies are available for the job. There's also the start of a reader and a simple bean for a writer, but you still have to define the actual batch job.

If you look at the `db-config.xml` file, you'll notice that the application uses an embedded HSQLDB database that's stored on disk. Labs so far have only used in-memory databases, but in this lab you'll need to persist state between various runs.

If you inspect the `integration-config.xml` file, you'll see that a simple file-based inbound channel adapter has been configured to watch a `spool` directory for `.csv` files, which are then picked up by a `DiningRequestsJobLauncher`. Open this class and see how it launches the injected job using the injected launcher by creating some `JobParameters` containing a resource path for the file that the Spring Integration adapter picked up from the spool directory.

11.2.1.2. Define the job

`TODO 01`: Open `batch-job-config.xml` and define a job called `diningRequestsJob`. Give the job a step and in that step, configure a tasklet and a chunk. Let the chunk's reader attribute refer to the existing `diningRequestsReader` bean and its writer attribute to the `reportWriter`. Set its commit-interval to `10`.

Although we could create a dedicated class that implements `ItemProcessor` and invokes the `rewardAccountFor` method on the `rewardNetwork` bean, Spring Batch can actually generate that code for you. Inside the `<chunk>` element, add a `<batch:processor>` element. Provide it with a `<ref>` child element that refers to the `rewardNetwork` bean and an `adapter-method` attribute that refers to the `rewardAccountFor` method.

11.2.1.3. Complete the item reader configuration

`TODO 02`: Although a start has been made with configuring the `diningRequestsReader` bean, it's not finished yet. Please complete it by setting the `Resource` from which the reader should read the input file from. Since the reader shouldn't hard-code the path of its input file, we can use a SpEL expression to obtain the value of a job parameter called `input.resource.path`.

Tip

The parameters map can be accessed using the `jobParameters` variable. You can access the value of a map entry in SpEL using the `map['some.key']` syntax for keys containing dots.

Tip

When placing SpEL expressions within a value tag, you should enclose the expression within `#{ }` tags.

Since this value is only available when the job is launched, give the reader bean a `step` scope! This means the expression will be evaluated when the step is being executed, so when the job parameters actually exist.

`TODO 03`: As you can see, the `diningRequestsReader` is already configured to read a number of fields from each line of the field, and has even defined their names. You'll have to write the code to convert the resulting `FieldSet` into an actual `Dining` instance. Modify the `DiningFieldSetMapper` class within the `rewards.batch` package. Extract the fields from the `FieldSet` by name and use them to create and return a new `Dining` instance.

Tip

The `Dining` class provides a convenient static factory method that you can use.

The date field uses the `yyyy-MM-dd` date format.

Now switch back to the job config xml file and complete the `diningRequestsReader` by injecting an instance of the new `DiningFieldSetMapper` into the inner `DefaultLineMapper` bean.

11.2.1.4. Test your work

`TODO 04`: Open the `BatchTests` class in the `rewards.batch` package of the test source folder. This file contains a number of tests that are currently all marked with `@Ignored`. The test is autowired with a job launcher and the job you just defined. It uses an in-memory database, so it won't interfere with the application that you'll run later which stores its database on disk.

Remove the `@Ignored` from the `regularJobSucceedsWithValidInput` method and run the test. Make sure that the test passes: if it doesn't, have a good look at the output and stacktraces in both your JUnit View as well as the Console to spot the cause of the error. If you cannot get the test to pass, ask your instructor for help.

11.2.2. Skipping in case of errors

In the real world, batch input isn't always under your control and can therefore contain invalid data. Spring Batch allows you to deal with the resulting errors in various ways, including *skipping* it.

11.2.2.1. Test what happens with broken input

`TODO 05`: You've now tested that your job works when its input is valid. But what happens if the input contains invalid items? To test that we've provided you with another input file that contains a single error. Open the `diningRequests-broken.csv` file and see if you can spot the line with the invalid input. Note its line number.

Tip

If you have Excel installed, then right-click the file and choose "Open With -> Text Editor" to avoid opening the file in Excel!

Now remove the `@Ignored` annotation from the `regularJobFailsWithInvalidInput` method and run the tests again. Notice that the tests asserts the `FAILED` exit status for the job when the input is invalid. Check the console to see how Spring Batch reports the cause. The original exception is wrapped in a `FlatFileParseException` and this exception and its stacktrace are reported by the `FlatFileItemReader` and also the `AbstractStep`, ultimately causing the job to fail.

11.2.2.2. Define an error-skipping job

`TODO 06`: Create another job definition by copying your existing one and renaming the job to `skippingDiningRequestsJob`. Also change the step name, since the step id has to be unique across jobs. Now set the chunk's `skip-limit` attribute to `1` to indicate that we'll tolerate at most one error. To indicate which errors are considered skippable, add a `<batch:skippable-exception-classes>` child element to the chunk and include `org.springframework.batch.item.file.FlatFileParseException`.

Note

Use code assist (Ctrl+Space) inside the skippable-exception-classes element to see what's available. Also, note that we don't include the `IncorrectTokenCountException` even though that's the root cause: when Spring Batch determines if the exception is skippable it's already wrapped.

11.2.2.3. Test the skipping

`TODO 07`: Switch back to the tests. First add an `@Autowired` annotation to the `skippingDiningRequestsJob` field to have both jobs injected.

Note

You might wonder why this works: both fields are of the same type, and Spring performs autowiring by type. Why won't it complain about the ambiguity of having two matching beans? Well, it turns out that Spring has added an extra mechanism for these type of cases: if the name of the dependency (the field, in this case) matches the name of one of the candidate beans then it will use that bean. Therefore, make sure that the names of the fields match the names you chose for your job ids!

`TODO 08`: Remove the @Ignore on the `skippingJobSucceedsWithInvalidInput` test and run the tests again. Note how this tests uses the same invalid input as the previous test, but checks for an exit state of COMPLETED instead of FAILED. Make sure all tests pass before you move on to the next step.

11.2.2.4. Add a `SkipListener`

`TODO 09`: The last test showed that the job is now skipping the invalid input. The `FlatFileItemReader` logs the `FlatFileParseException` as an error, but you might want to act on the skipping of an invalid item in your own code as well; for example, by writing the invalid item to a separate file.

In order to get a callback when skipping occurs, look at the `DiningSkipListener` class under the `rewards.batch` package. This class implements the `SkipListener` interface, but as we're only interested in one of the three methods that interface defines it's easiest to simply extend the `SkipListenerSupport` class provided by the framework. The type is annotated with `@SuppressWarnings("rawtypes")`: you can use the raw type here, since the method you'll implement doesn't use the type information.

Within the `onSkipInRead` method it simply logs a warning using a Log4J logger that includes the message from the `Throwable` that gets passed. In a real-life application you could append the original line to a file, send a JMS message, an SNMP trap, or use a Spring Integration gateway to put a message on a channel for further consumption.

Switch back to the job configuration and complete it by configuring a `<batch:listeners>` element for the tasklet inside the step of the `skippingDiningRequestsJob` with a `<batch:listener>` element that contains an inner bean of type `DiningSkipListener`. Then run your tests again and check the output in the Console View for your new log statement.

11.2.3. Restarting a failed job

Not all errors can simply be skipped and ignored: sometimes you simply want to stop the batch job, report the error and have someone investigating the cause. If that person is able to fix the issue (s)he might want to restart the job, starting from the point where the input items weren't processed yet. Let's see how that would work for our job.

11.2.3.1. Start the batch job with invalid input

Open the `rewards.batch.Bootstrap` class. This class starts an application context that includes the Spring Integration configuration you reviewed earlier. After that, it simply waits while the spool directory is being polled for new files until you press Enter in the Console View, after which it closes down. Closing down properly will ensure a clean shutdown of the database that we're using, so please use this method for all following steps where you want to stop the current application. Now start the application.

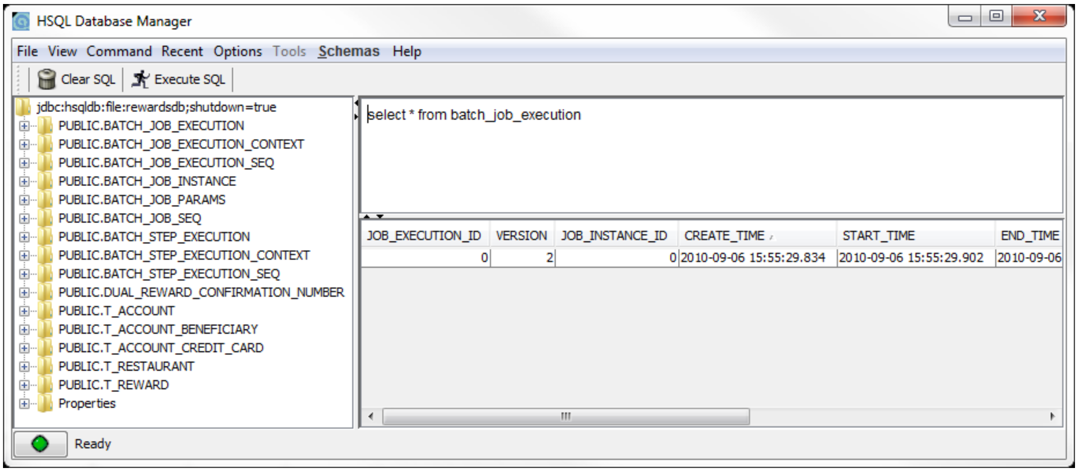
At this point, you won't see much output yet. Now copy the `diningRequests-broken.csv` file from the root of the project into the `spool` directory. This will trigger your batch job to run, and since this is the regular non-skipping job it will fail. Stop the application by pressing Enter in the Console View. Refresh the project and you'll see two new files, `rewardsdb.properties` and `rewardsdb.script`, that make up your new database.

Tip

If you want to start with a clean database again later, simply delete these two files.

11.2.3.2. Inspect the database

Let's see what Batch has written in the database so far. Right-click the `HSQLDB Manager.launch` file in the root of your project and select "Run As -> HSQLDB Manager". You should now see the following application with all the database tables listed:



jdbc:hsqldb:file:rewardsdb;shutdown=true
PUBLIC.BATCH_JOB_EXECUTION
PUBLIC.BATCH_JOB_EXECUTION_CONTEXT
PUBLIC.BATCH_JOB_EXECUTION_SEQ
PUBLIC.BATCH_JOB_INSTANCE
PUBLIC.BATCH_JOB_PARAMS
PUBLIC.BATCH_JOB_SEQ
PUBLIC.BATCH_STEP_EXECUTION
PUBLIC.BATCH_STEP_EXECUTION_CONTEXT
PUBLIC.BATCH_STEP_EXECUTION_SEQ
PUBLIC.DUAL_REWARD_CONFIRMATION_NUMBER
PUBLIC.T_ACCOUNT
PUBLIC.T_ACCOUNT_BENEFICIARY
PUBLIC.T_ACCOUNT_CREDIT_CARD
PUBLIC.T_RESTAURANT
PUBLIC.T_REWARD
Properties

```
select * from batch_job_execution
```

JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	CREATE_TIME	START_TIME	END_TIME
0	2	0	2010-09-06 15:55:29.834	2010-09-06 15:55:29.902	2010-09-06

Perform a couple of "select * from *tablename*" queries to inspect the contents of the database, in particular the `_EXECUTION` tables. Can you understand the numbers listed for the various COUNT columns in the `BATCH_STEP_EXECUTION` table?

Tip

Remember that the commit-interval is set to 10!

When you're done, close the manager client and proceed with the next step.

11.2.3.3. Fix the error and restart the job

Open the `diningRequests-broken.csv` file from the spool directory in the text editor and fix line 123 by removing the extra value at the end of the line.

Tip

Use Ctrl+L in Eclipse to quickly jump to a given line number

Save the file and then restart the job by running the `Bootstrap` class again. Read the output in the Console: does the job indeed restart by only processing the items that weren't processed successfully yet?

`TODO 10`: When you think that restarting works OK, switch back to the tests and enable the `restartRegularJobAfterFixingInputSucceeds` test and see what it does. Run the tests and make sure they all pass.

Note

In order to test restarting with repaired input the test uses two different files. Since the filenames are part of the `JobParameters`, that would normally mean that Spring Batch would not consider the second launch to be a restart of an existing job instance, but the start of a new job instance instead. To fool the framework into thinking it's the same job instance we've used a little trick that hides the relevant parameter from the parameter map's key set. Don't worry too much about how it works, as long as you understand what we're testing here that's OK.

11.2.4. Limiting the number of (re)starts allowed

The current jobs are allowed to be restarted as often as you like. In a real life situation, you might want to limit the amount of restarts allowed for a given job. Spring Batch easily allows you to do just that.

11.2.4.1. Add a start-limit

`TODO 11`: Switch to the job configuration and add a `start-limit` attribute with a value of `3` to the tasklet of the `diningRequestsJob`. This is the total number of starts allowed, including the initial start, so this limits the number of restarts allowed to 2.

`TODO 12`: Open the tests and remove the @Ignored from the `exceedingRestartLimitPreventJobFromRunningAgain` method. Check what the method does: it runs the job three times, checking for failure on each run, and then asserts that another attempt at restarting results in a

`StartLimitExceededException` as the root cause of the failure exception returned in the `JobExecution`. Run the tests and make sure that they all pass.

When all tests pass you have completed this lab successfully. Congratulations!

You might like to tidy the output by suppressing the stack-traces from the exceptions that we are expecting as part of the tests. We don't care about them and they make the tests look like they are failing. (TODO 13). Warnings are output instead, so you can still see what is going on.

11.2.5. BONUS: Java configuration refactoring

If you have some time left, you can work on refactoring the jobs XML configurations to Java configuration. We'll use the same test, without any modification in the test logic, to validate the configuration refactoring works. Before moving on to the refactoring, ensure the `BatchTests` passes.

11.2.5.1. Refactor the jobs configurations to Java

`TODO 14`: Change the `@SpringApplicationConfiguration` annotation in the `BatchTests` class to use the `SystemTestConfiguration` configuration class. The `@SpringApplicationConfiguration` has a `classes` attribute to specify this.

`TODO 15`: Take a look at the `SystemTestConfiguration` configuration class. It is the configuration entry point for the tests. It imports the infrastructure configuration class (`BatchExecutionConfig`) and the jobs configurations class (`BatchJobConfig`). It also declares an embedded `DataSource` bean.

`TODO 16`: Add the `@EnableBatchProcessing` annotation to `BatchExecutionConfig`. This declares the Spring Batch infrastructure (job repository, job launcher, etc.) See `batch-execution-config.xml` for the XML way: the annotation replaces all that XML code.

`TODO 17`: It's time now to configure the `diningRequestsJob` the Java way (`BatchJobConfig` class). Some of the work is already done for you: skeleton of the configuration class, most of the batch artifacts (reader, processor, writer, etc.) Here, you just need to refer to the appropriate step in the `flow` method of the `diningRequestsJob` method.

`TODO 18`: You must now configure the `diningRequestsStep`. It needs its reader, its processor, and its writer. Fortunately, all of these have bean already declared, you just need to wire them. And don't forget the start limit!

`TODO 19`: Time now to configure the skip behavior in the `skippingDiningRequestsStep` bean. You need to setup the exception to skip (`FlatFileParseException`) and the skip limit. Don't hesitate to go back to the slides if you don't remember the exact syntax.

`TODO 20`: At last, add the skip listener to the step. The configuration is now complete, let's check if everything works fine.

`TODO 21`: Run the `BatchTests`, it should pass if the Java configuration is correct.

Congratulations, you refactored the jobs configurations to Java!

Chapter 12. batch-admin: Spring Batch Admin

12.1. Introduction

In this lab you'll see how Spring Batch Admin provides a UI to launch, manage and inquire status of batch jobs. You will also see how you can partition input data to improve batch throughput.

What you will learn:

1. Starting, stopping, and restarting batch jobs using Batch Admin
2. Examining the status of running and completed jobs with batch admin
3. Configuring a job for parallel execution using partitioning

Specific subjects you will gain experience with:

1. The Batch Admin UI
2. Implementing a partitioned job

Estimated time to complete: 45 minutes

12.2. Instructions

The instructions for this lab are organized into sections.

12.2.1. Exploring the Batch Admin UI

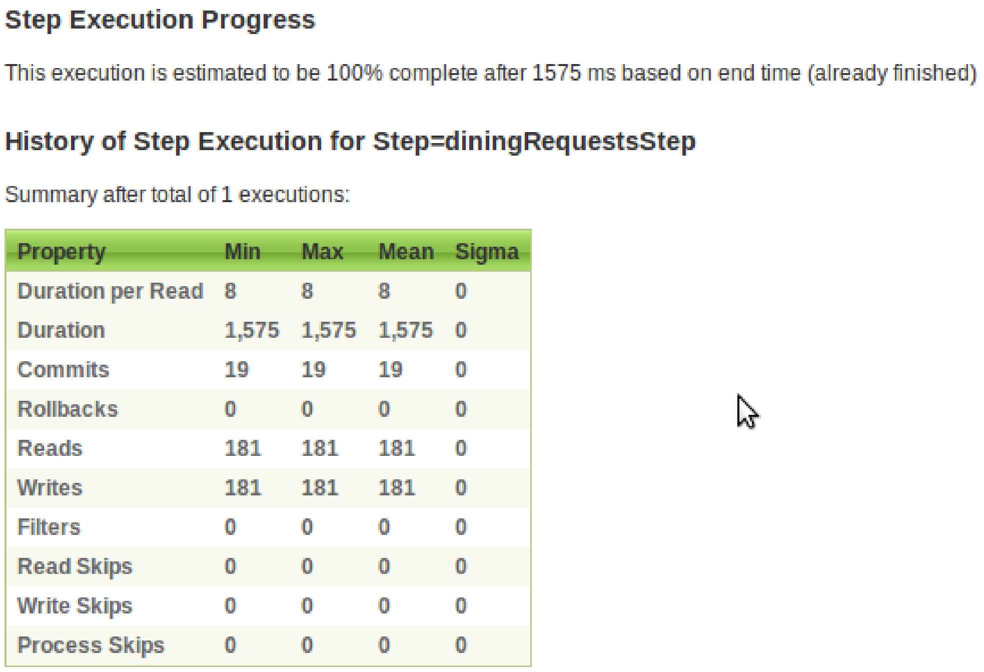
The `batch-admin` lab consists of the Batch Admin sample UI application, with a version of the `diningRequestsJob` job from the `batch-restart-recovery` lab deployed.

12.2.1.1. Step 1: Deploy the Project and Explore the UI

(`TODO 01`): Right click the `batch-admin` project and select Run As...->Run on Server. STS will open a browser view containing the home page; use that view or a stand-alone browser with URL http://localhost:8080/batch-admin/ to explore the UI.

Select Jobs; you should see a page showing the diningRequestsJob; click the job name. In the Job Parameters field, enter `input.resource.path=diningRequests.csv`; click `Launch`.

Drill down to the Step Execution details (use the slides for reference); you should see that the job executed 181 reads and writes, like this:



Step Execution Progress

This execution is estimated to be 100% complete after 1575 ms based on end time (already finished)

History of Step Execution for Step=diningRequestsStep

Summary after total of 1 executions:

Property	Min	Max	Mean	Sigma
Duration per Read	8	8	8	0
Duration	1,575	1,575	1,575	0
Commits	19	19	19	0
Rollbacks	0	0	0	0
Reads	181	181	181	0
Writes	181	181	181	0
Filters	0	0	0	0
Read Skips	0	0	0	0
Write Skips	0	0	0	0
Process Skips	0	0	0	0

12.2.1.2. Step 2: Explore the Batch Admin Project Structure

(`TODO 02`): The Batch Admin Sample UI application loads each job in a separate application context, to avoid clashes between bean names etc. Each batch job is placed in a separate config file under src/main/resources/META-INF/spring/batch/jobs. Take a look at the job file there; notice that we have added an `incrementer` to the job. Take a look at the implementation of that incrementer. This allows us to submit multiple executions of the same job, with the same job parameters, without Spring Batch attempting to restart a previous job instance.

Also notice the src/main/resources/META-INF/override folder. Beans declared in files in this folder override those in the batch admin jars.

12.2.1.3. Step 3: Add Test Failure to Writer

(`TODO 03 - TODO 04`): Open the `ReportWriter` class and expose the writer as an MBean and expose the `failOnConfirmationNumber` as an MBean Attribute.

Stop Tomcat and restart it with JMX enabled.

Connecting using JConsole

Either in the launch configuration for Tomcat, or by setting it as a default in the 'Installed JREs' under Window | Preferences, you need to set the following VM argument

```
-Dcom.sun.management.jmxremote
```

If you are unfamiliar with how to do this, ask your instructor.

Now, open a windows command prompt (*Start->Run->'cmd'*), and type

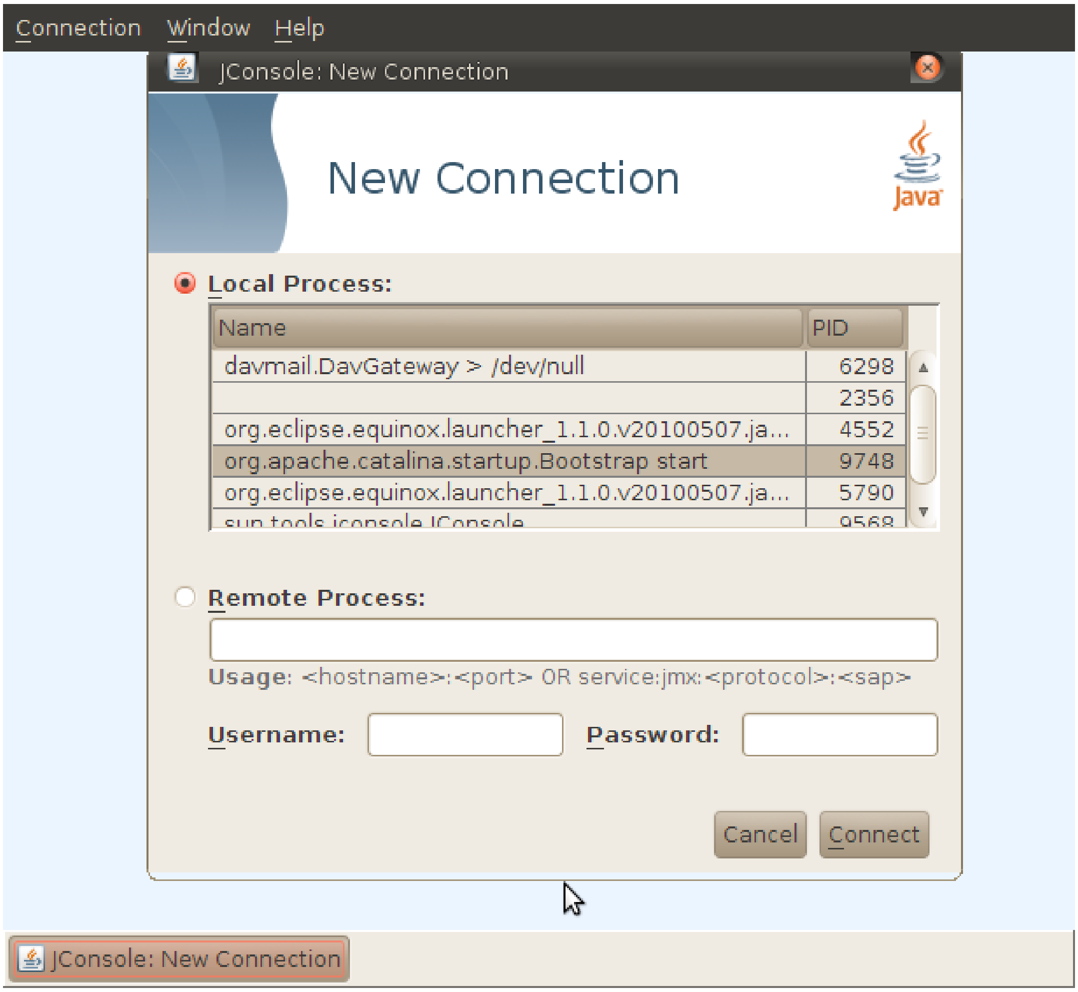
```

    C:\> jconsole

```

Tip

JConsole is the default JMX console that ships with Java, versions 5.0 and greater. It's located in the *bin* directory of the JDK home: if that's not on your path, then 'cd' into that directory first!

You may find it more convenient to double click jconsole.exe in the bin directory.

When the JConsole connection dialog appears, select the *org.apache.catalina.startup.Bootstrap* process and click 'Connect'. After a few moments, you'll be presented with the following screen.



Name	PID
davmail.DavGateway > /dev/null	6298
	2356
org.eclipse.equinox.launcher_1.1.0.v20100507.ja...	4552
org.apache.catalina.startup.Bootstrap start	9748
org.eclipse.equinox.launcher_1.1.0.v20100507.ja...	5790
sun.tools.jconsole.JConsole	9568

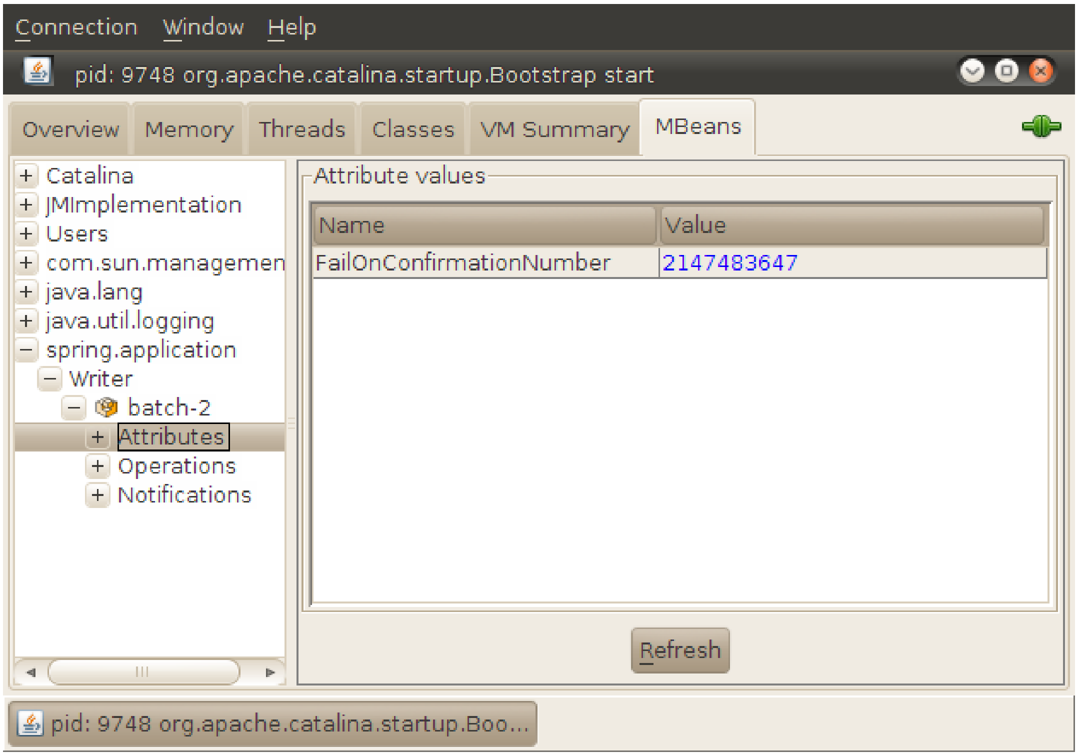
Tip

If you cannot see the catalina process in JConsole (in the 'Local Process' section) or the connect times out, 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 Tomcat, and connect via JConsole by using the 'Remote Process' selection, specifying a host of *localhost* and port of *8181*.

Navigate to the MBeans tab and open 'spring.application'; navigate to the FailOnConfirmationNumber attribute, shown here.



Name	Value
FailOnConfirmationNumber	2147483647

Set the attribute to, say, 55, and click Refresh to confirm.

12.2.1.4. Step 4: Fail and Recover the Job

Return to the browser and start a new execution of the job.

Navigate to the step execution details as before and note that the job failed to complete and look at the number of rows processed.

Return to JConsole and reset the attribute to at least 200.

Return to the browser, navigate to the page for the failed execution and restart the job.

Navigate to the step execution results and see that the remaining rows were processed.

Congratulations! You have successfully tested job restart and recovery.

12.2.2. Stopping and Restarting a Batch Job

In this section you will see how we can manually stop a batch job, and restart it from where it was stopped.

12.2.2.1. Step 5: Slowing Down the Job

Our test batch job completes very quickly. Here we will introduce a pause so the job takes several minutes to complete.

(`TODO 05`): Open the ReportWriter and introduce a delay, simulating that it takes 1 second to process each item.

Redeploy the project using Run As...->Run on Server...

Navigate to the Job and launch it, again with job parameter 'input.resource.path=diningRequests.csv'.

In the browser, navigate to the step execution and note that the job is progressing (slowly).

Navigate to the job execution and stop the job. Navigate to the step execution; confirm the step and job are stopped and observe how many items were processed.

Navigate to the job execution and click the 'Restart' button. Allow the job to complete and observe that the total items processed is 181 for the job instance (across both executions).

Congratulations! You have successfully stopped and restarted a job.

12.2.3. Configuring a Partitioned Batch Job

In this section, you will gain experience configuring and running a partitioned batch job. The partitioning uses a simple `Partitioner` that finds the minimum and maximum primary keys in a table to obtain a count of rows and then calculates the number of rows in the partition.

Tip

This technique only works if the primary keys are uniformly distributed; a more sophisticated partitioning technique may be necessary for other situations.

12.2.3.1. Step 6: Adding Additional Tables to the Schema

The job has several steps; the first step deletes all data from temporary tables; the second step loads the data from the flat file into a temporary table (T_DINING_REQUEST); the third step partitions the data and processes each partition, with the results being written to a temporary table that can be used later to generate a report.

(`TODO 06`): Take a look in src/main/resources and open the file `partition-schema.sql`. We need to add this script to the database initializer. Open `src/main/resources/META-INF/spring/batch/override/data-source-context.xml` and include the additional schema script.

Tip

Files in `src/main/resources` end up on the classpath in the war file. Given this is the location of the schema file, no folders are needed when using the `classpath:` modifier in the `location` attribute.

12.2.3.2. Step 7: Preparing the Partitioned Batch Job Configuration File

(`TODO 07`): As mentioned above, Spring Batch Admin looks for job definitions under `META-INF/spring/batch/jobs` on the classpath. A start for the partitioned job has been provided for you, under `src/main/resources`.

Move `partition-rewards-job-config.xml` from `src/main/resources` to `src/main/resources/META-INF/spring/batch/jobs` (alongside the existing `batch-job-config.xml`).

(`TODO 08`): Review the `ColumnRangePartitioner` class; open the `partion-rewards-job-config.xml` and complete the configuration for the `ColumnRangePartitioner`.

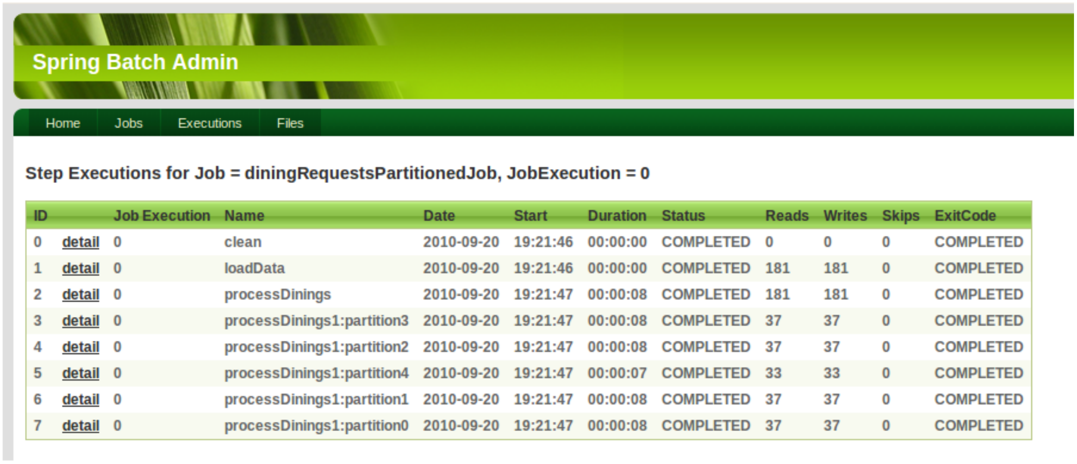
Tip

Some of the information you need for this task is in the schema file you just added to to the database initializer.

12.2.3.3. Step 8: Testing Partition Sizes

Redeploy the project on Tomcat; when the home page is displayed, select Jobs. You should now see a second job `diningRequestsPartitionedJob`. Select that job and launch it with Job Parameters `input.resource.path=diningRequests.csv`.

Wait for the execution to complete; and drill down to the Step Executions. You should see something similar to this:



Step Executions for Job = diningRequestsPartitionedJob, JobExecution = 0

ID		Job Execution	Name	Date	Start	Duration	Status	Reads	Writes	Skips	ExitCode
0	detail	0	clean	2010-09-20	19:21:46	00:00:00	COMPLETED	0	0	0	COMPLETED
1	detail	0	loadData	2010-09-20	19:21:46	00:00:00	COMPLETED	181	181	0	COMPLETED
2	detail	0	processDinings	2010-09-20	19:21:47	00:00:08	COMPLETED	181	181	0	COMPLETED
3	detail	0	processDinings1:partition3	2010-09-20	19:21:47	00:00:08	COMPLETED	37	37	0	COMPLETED
4	detail	0	processDinings1:partition2	2010-09-20	19:21:47	00:00:08	COMPLETED	37	37	0	COMPLETED
5	detail	0	processDinings1:partition4	2010-09-20	19:21:47	00:00:07	COMPLETED	33	33	0	COMPLETED
6	detail	0	processDinings1:partition1	2010-09-20	19:21:47	00:00:08	COMPLETED	37	37	0	COMPLETED
7	detail	0	processDinings1:partition0	2010-09-20	19:21:47	00:00:08	COMPLETED	37	37	0	COMPLETED

See that there are 5 partitions (4 of 37 rows and 1 of 33 for a total of 181 rows as before).

Make a note of the elapsed time of the `processDinings` step - the 'master' step that ran the partitions.

(`TODO 09`): Open the `partion-rewards-job-config.xml` file and find the `grid-size` attribute on the `processDinings` step. Change the grid-size to, say, 10. Save, redeploy and re-run the test.

Drill down to the Step Executions and note the new elapsed time for the master step.

Congratulations! You have completed the lab.

Chapter 13. Introduction to Spring XD

13.1. Introduction

In this lab, you'll get an overview of Spring XD. This lab is made of 3 parts. In part 1, you'll install Spring XD. In part 2, you'll be creating streams. In part 3, you'll be creating jobs.

What you will learn

1. Installing and running a single-node instance of Spring XD
2. Connecting to the Spring XD instance with the Spring XD shell
3. Creating, deploying, and running some basic streams in Spring XD
4. Deploying a simple batch process in Spring XD, without custom coding

Specific subjects you will gain experience with:

1. Setting up Spring XD from scratch.
2. Using the XD shell to connect to Spring XD
3. Deploying streams and jobs with the shell

Estimated time to complete: 45 minutes

13.2. Installation (PART 1)

In this part you will single-node Spring XD instance, use the xd-shell to connect to it.

For classroom purposes, you can run Spring XD in standalone mode on any platform that supports Java, such as Windows, Mac, Linux, etc.. For production instances, Pivotal recommends running on Linux-based operating systems. For Mac OSX, there is a Homebrew installation available, and Pivotal also provides a yum install for RedHat/CentOS. These latter repositories (Homebrew and Yum) may not contain the most recent release - the latest version can always be obtained from the Spring XD project website.

13.2.1. Generic Installation

Since Spring XD is written in Java, it can be installed on essentially any platform that supports a JVM. Also, performing a generic installation (as opposed to using Homebrew or Yum) will result in getting the latest version of the product running on your system.

1. Download the latest version of Spring XD from the Spring website, http://projects.spring.io/spring-xd/. All releases can be found at http://repo.spring.io/release/org/springframework/xd/spring-xd/, there you will find subdirectories containing various builds. In this case we demonstrate installation of the 1.2.1 release: `spring-xd-1.2.1.RELEASE-dist.zip`.
2. Extract the archive into the directory of your choice.
3. Set the `XD_HOME` environment variable to the `xd` subdirectory of the installation:

   `<installation-directory>/spring-xd-<version>.RELEASE/xd`. Ensure this environment variable is always set up when you start Spring XD!

13.2.2. Installation using Homebrew on Mac OSX

A homebrew installation of Spring XD is available for installing on OSX. Again, note that the version may not be the latest current release.

1. To install Spring XD using homebrew simply add the `pivotal/tap` using the `brew tap pivotal/tap` command.
2. Execute the `brew install springxd` command.

13.2.3. Installation on RedHat/CentOS

Spring XD can be installed from the Pivotal App-Suite yum repository on RedHat or CentOS. However, this requires a distributed mode configuration. As a result, this won't be using this installation.

13.2.4. Verifying the Installation

Now that you've completed the installation, let's verify that you can start the server and connect to it.

13.2.4.1. Starting the Spring XD Server

1. Start the server by running the `xd-singlenode` server startup script. On Windows you'll have to switch to the `<installation-directory>/spring-xd-<version>.RELEASE/xd/bin` directory first or provide the full path to the `xd-singlenode.bat` batch file. On a Linux-based system you can simply execute `xd-singlenode`, assuming the script has execute permissions (chmod 777).

   You should see the server start up, with the Spring XD ASCII-art logo output onto your console.

```
$XD_HOME/xd/bin>xd-singlenode

 _____                           __   _______
/  ___|          (-)             \ \ / /  _  \
\ `--. _ __  _ __ _ _ __   __ _   \ V /| | | |
 `--. \ '_ \| '__| | '_ \ / _` |  / ^ \| | | |
/\__/ / |_) | |  | | | | | (_| | / / \ \ |/ /
\____/| .__/|_|  |_|_| |_|\__, | \/   \/___/
      | |                  __/ |
      |_|                 |___/
1.2.1.RELEASE                    eXtreme Data


Started : SingleNodeApplication
Documentation: https://github.com/spring-projects/spring-xd/wiki

2015-08-17T10:22:30+0200 1.2.1.RELEASE INFO main singlenode.SingleNodeApplication - Starting SingleNodeApplication v1.2.1.RE
2015-08-17T10:22:30+0200 1.2.1.RELEASE INFO main singlenode.SingleNodeApplication - Started SingleNodeApplication in 1.255 s
10:22:34,691  INFO HSQLDB Server @4981d95b HSQLDB4F3AC0BF9B.ENGINE - Checkpoint start
10:22:34,691  INFO HSQLDB Server @4981d95b HSQLDB4F3AC0BF9B.ENGINE - checkpointClose start
10:22:34,746  INFO HSQLDB Server @4981d95b HSQLDB4F3AC0BF9B.ENGINE - checkpointClose end
10:22:34,747  INFO HSQLDB Server @4981d95b HSQLDB4F3AC0BF9B.ENGINE - Checkpoint end - txts: 1
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD Home: /home/acogoluegnes/Downloads/spr
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Transport: local
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Hadoop version detected from classpath 2.
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD config location: file:/home/acogoluegn
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD config names: servers,application
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD module config location: file:/home/aco
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD module config name: modules
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Admin web UI: http://acogoluegnes-xps:939
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Zookeeper at: localhost:22854
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Zookeeper namespace: xd
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Analytics: memory
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO LeaderSelector-0 zk.DeploymentSupervisor - Leader Admin admin:default,admin,sing
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO main admin.AdminServerApplication - Started AdminServerApplication in 6.754 seco
2015-08-17T10:22:44+0200 1.2.1.RELEASE INFO DeploymentSupervisor-0 zk.ContainerListener - Path cache event: type=INITIALIZED
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD Home: /home/acogoluegnes/Downloads/spr
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Transport: local
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Hadoop version detected from classpath 2.
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD config location: file:/home/acogoluegn
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD config names: servers,application
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD module config location: file:/home/aco
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - XD module config name: modules
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Container IP address: 10.0.1.6
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Container hostname:   acogoluegnes-xps
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Zookeeper at: localhost:22854
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Zookeeper namespace: xd
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main util.XdConfigLoggingInitializer - Analytics: memory
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main container.ContainerRegistrar - Container {ip=10.0.1.6, host=acogoluegnes-xp
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO main container.ContainerServerApplication - Started ContainerServerApplication i
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO DeploymentSupervisor-0 zk.ContainerListener - Path cache event: path=/containers
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO DeploymentSupervisor-0 zk.ContainerListener - Container arrived: Container{name=
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO DeploymentSupervisor-0 zk.ContainerListener - Scheduling deployments to new cont
```

Find the installation directory (based on where you installed it above; see class slides for background). Take a look in the `<installation-directory>/spring-xd-<version>.RELEASE/xd/logs` directory. You should find a `singlenode-xxxx.log` log file there. If you don't, verify that you set the `XD_HOME` environment variable, and check the directory permissions. Examine the log file, noting the locations of the configuration files.

13.2.4.2. Connecting to the Server

1. Let's start up the Spring XD shell client now. Open a separate command prompt, and change to the `<installation-directory>/spring-xd-<version>.RELEASE/shell/bin` directory, and launch the `xd-shell` client. You should again see the Spring XD ASCII-art logo, and the server you connected to.

```
$XD_HOME/../shell/bin>xd-shell
 _____                           __   _______
/  ___|          (-)             \ \ / /  _  \
\ `--. _ __  _ __ _ _ __   __ _   \ V /| | | |
 `--. \ '_ \| '__| | '_ \ / _` |  / ^ \| | | |
/\__/ / |_) | |  | | | | | (_| | / / \ \ |/ /
\____/| .__/|_|  |_|_| |_|\__, | \/   \/___/
      | |                  __/ |
      |_|                 |___/
eXtreme Data
1.2.1.RELEASE | Admin Server Target: http://localhost:9393
Welcome to the Spring XD shell. For assistance hit TAB or type "help".
xd:>
```

2. From within the XD command shell, examine the list of commands available by typing `help` at the prompt. Note that the shell offers tab-complete, so you can start to type a command and then hit <tab> to see a list of options. Try a couple simple commands, such as `admin config info`, `version`, and `stream list`. What port, by default, does the single-node instance listen on?

```
xd:>admin config info
  -------------  -------------------------------------------
  Result         Successfully targeted http://localhost:9393
  Target         http://localhost:9393
  Timezone used  Central European Time (UTC 1:00)
  -------------  -------------------------------------------

xd:>version
1.2.1.RELEASE
xd:>stream list
  Stream Name  Stream Definition  Status
  -----------  -----------------  ------

xd:>
```

3. Once you are finished, type `exit` to leave the Spring XD shell.

13.3. Introduction to Streams (PART 2)

13.3.1. A Simple Time Logger

Create, deploy and run simple streams and jobs without specific coding. Let's begin by creating a very simple time logger stream.

1. Go back to the Spring XD shell that is connected to your running server. If you get stuck at any time, remember that you can enter `help`, or hit <tab> in the shell for autocomplete.
2. Create a stream consisting of a time source connected directly to a log sink. Don't forget to give your stream a name. Verify the stream was created by running the `stream list` command. Note the status of the stream.
3. Make sure that you can see the Spring XD server output in the background, and go ahead and deploy your stream. You should observe the time logging in the server window. What is the default frequency of the time source?

   Run the `stream list` command again. What is the status of the stream now?
4. Undeploy and delete the stream by executing the `stream destroy` command. Once again, verify the stream is gone by checking the status.
5. Now, let's demonstrate how to pass in options to the the modules. Create a new stream with the time source and log sink, but this time change the interval from 1 second to 10 seconds. If you don't remember how to do this, go back to the slide notes to check, or check the sources documentation found at http://docs.spring.io/spring-xd/docs/current/reference/html/#time. Deploy and test your new stream, verifying that it logs every 10 seconds now. When you are finished you can destroy it.

13.3.2. Using Processors

Now that you are familiar with deploying a simple stream, and specifying options for the stream modules, let's add some processing to a stream. This time we will use an HTTP source that will receive a JSON order summary payload, and filter out orders without a country code of "US". All orders with a "US" countryCode will get written to a file.

1. From the Spring XD shell, create and deploy a stream with an HTTP source, a filter Spring Expression Language (SpEL) processor, and a file sink. You will use the jsonPath SpEL method to extract the `countryCode` field, and discard any order payloads that do not have a country code of US.

   The HTTP source should listen on port 9090.

   To specify an SpEL on a filter processor, use the `--expression` option. The payload of the HTTP post is represented by the `payload` variable. So the format of the processor module you need to create will be:

```
filter --expression=#jsonPath(payload,'$.countryCode').equals('US')
```

   Change the directory of the output file to a temporary location (eg. /tmp), and be sure to append orders to the file. Refer to the online documentation for the file sink found here: http://docs.spring.io/spring-xd/docs/current/reference/html/#file-sink
2. The Spring XD shell includes a utility that will post an HTTP payload to a URL. You can use the following example to post some sample orders to your stream for testing:

```
http post --target http://localhost:9090 \
    --data "{\"id\":\"2773\",\"countryCode\":\"US\",\"orderAmt\":\"100\"}"
```

Post several order payloads, changing the order number and alternating between US and non-US country codes.

Examine the contents of the output file created in the temporary directory. What is the name of the file that was created? You should only see orders with US country codes in the file, if your stream is functioning correctly.

If you have time, feel free to experiment with some of the different options for the modules.

13.3.3. Solutions

1. Simple Time Logger - Spring XD shell input:

```
xd:>stream create --definition "time | log" --name timer
Created new stream 'timer'
xd:>stream list
  Stream Name  Stream Definition  Status
  -----------  -----------------  ----------
  timer        time | log         undeployed

xd:>stream deploy --name timer
Deployed stream 'timer'
xd:>stream list
  Stream Name  Stream Definition  Status
  -----------  -----------------  --------
  timer        time | log         deployed

xd:>stream undeploy --name timer
Un-deployed stream 'timer'
xd:>stream list
  Stream Name  Stream Definition  Status
  -----------  -----------------  ----------
  timer        time | log         undeployed

xd:>stream destroy timer
Destroyed stream 'timer'
```

```
xd:>stream create --definition "time --fixedDelay=10 | log" --name timer --deploy
Created and deployed new stream 'timer'
xd:>stream destroy timer
Destroyed stream 'timer'
xd:>
```

Simple Time Logger - Spring XD server log output:

```
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO DeploymentSupervisor-0 zk.ContainerListener - Path cache event: path=/containers
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO DeploymentSupervisor-0 zk.ContainerListener - Container arrived: Container{name=
2015-08-17T10:22:46+0200 1.2.1.RELEASE INFO DeploymentSupervisor-0 zk.ContainerListener - Scheduling deployments to new cont
2015-08-17T10:35:06+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
2015-08-17T10:35:06+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Deploying module '
2015-08-17T10:35:06+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Deploying module [
2015-08-17T10:35:07+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
2015-08-17T10:35:07+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Deploying module '
2015-08-17T10:35:07+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Deploying module [
2015-08-17T10:35:07+0200 1.2.1.RELEASE INFO task-scheduler-1 sink.timer - 2015-08-17 10:35:07
2015-08-17T10:35:07+0200 1.2.1.RELEASE INFO DeploymentSupervisor-0 zk.ZKStreamDeploymentHandler - Deployment status for stre
2015-08-17T10:35:08+0200 1.2.1.RELEASE INFO task-scheduler-1 sink.timer - 2015-08-17 10:35:08
2015-08-17T10:35:09+0200 1.2.1.RELEASE INFO task-scheduler-2 sink.timer - 2015-08-17 10:35:09
2015-08-17T10:35:10+0200 1.2.1.RELEASE INFO task-scheduler-1 sink.timer - 2015-08-17 10:35:10
.
.
.
2015-08-17T10:35:31+0200 1.2.1.RELEASE INFO task-scheduler-10 sink.timer - 2015-08-17 10:35:31
2015-08-17T10:35:32+0200 1.2.1.RELEASE INFO main-EventThread container.DeploymentListener - Undeploying module [ModuleDescri
2015-08-17T10:35:32+0200 1.2.1.RELEASE INFO main-EventThread container.DeploymentListener - Undeploying module [ModuleDescri
2015-08-17T10:35:32+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
2015-08-17T10:35:32+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
```

```
2015-08-17T10:36:48+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
2015-08-17T10:36:48+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Deploying module '
2015-08-17T10:36:48+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Deploying module [
2015-08-17T10:36:49+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
2015-08-17T10:36:49+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Deploying module '
2015-08-17T10:36:49+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Deploying module [
2015-08-17T10:36:49+0200 1.2.1.RELEASE INFO task-scheduler-5 sink.timer - 2015-08-17 10:36:49
2015-08-17T10:36:49+0200 1.2.1.RELEASE INFO DeploymentSupervisor-0 zk.ZKStreamDeploymentHandler - Deployment status for stre
2015-08-17T10:36:59+0200 1.2.1.RELEASE INFO task-scheduler-9 sink.timer - 2015-08-17 10:36:59
2015-08-17T10:37:09+0200 1.2.1.RELEASE INFO task-scheduler-9 sink.timer - 2015-08-17 10:37:09
2015-08-17T10:37:19+0200 1.2.1.RELEASE INFO task-scheduler-9 sink.timer - 2015-08-17 10:37:19
.
.
.
2015-08-17T10:39:29+0200 1.2.1.RELEASE INFO task-scheduler-4 sink.timer - 2015-08-17 10:39:29
2015-08-17T10:39:31+0200 1.2.1.RELEASE INFO main-EventThread container.DeploymentListener - Undeploying module [ModuleDescri
2015-08-17T10:39:31+0200 1.2.1.RELEASE INFO main-EventThread container.DeploymentListener - Undeploying module [ModuleDescri
2015-08-17T10:39:31+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
2015-08-17T10:39:31+0200 1.2.1.RELEASE INFO DeploymentsPathChildrenCache-0 container.DeploymentListener - Path cache event:
```

2. Using Processors - Spring XD shell input:

```
xd:>stream create --name orderFilter --definition "http --port=9090 |
filter --expression=#jsonPath(payload,'$.countryCode').equals('US') |
file --dir=/tmp --mode=APPEND" --deploy
Created and deployed new stream 'orderFilter'
xd:>http post --target http://localhost:9090 --data "{\"id\":\"2773\",\"countryCode\":\"US\",\"orderAmt\":\"100\"}"
> POST (text/plain;Charset=UTF-8) http://localhost:9090 {"id":"2773","countryCode":"US","orderAmt":"100"}
> 200 OK

xd:>http post --target http://localhost:9090 --data "{\"id\":\"2774\",\"countryCode\":\"CA\",\"orderAmt\":\"125\"}"
> POST (text/plain;Charset=UTF-8) http://localhost:9090 {"id":"2774","countryCode":"CA","orderAmt":"125"}
> 200 OK

xd:>http post --target http://localhost:9090 --data "{\"id\":\"2775\",\"countryCode\":\"US\",\"orderAmt\":\"110\"}"
> POST (text/plain;Charset=UTF-8) http://localhost:9090 {"id":"2775","countryCode":"US","orderAmt":"110"}
> 200 OK

xd:>
```

Using Processors - Output:

```
$ more /tmp/orderFilter.out
{"id":"2773","countryCode":"US"}
{"id":"2775","countryCode":"US"}
$
```

13.4. Introduction to Jobs (PART 3)

13.4.1. Installing and Running a Basic Batch Job

This simple example demonstrate how a Spring XD job can read a txt file, replace the pipes with commas, and output the result into a new file.

1. Before installing the custom module, let's modify the location of the custom module registry. Create a subdirectory in your user home directory called `xd-custom-modules`. Open the Spring XD `servers.yml` file (located within `XD_HOME/config`), and set the `xd.customModule.home` property to the location of the new directory.

   Restart the Spring-XD server to pick up the changes.
2. Now, examine the contents of the `xd-intro-jobs/dining-requests-in.txt` file - note it contains pipe-delimited lines of text. Copy the file from the course labs directory, into your home directory `$HOME`.
3. Open the `xd-intro-jobs/xd-job-basic.xml` file and find the `itemReader` bean. Change the value of the `resource` property to point to the `dining-requests-in.txt` file that you just copied into your home directory.

```
...
        <bean id="itemReader" class="org.springframework.batch.item.file.FlatFileItemReader">
                <property name="resource" value="file:$HOME/dining-requests-in.txt" />
                <property name="lineMapper" ref="lineMapper" />
        </bean>
...
```

   Find the `itemWriter` bean, and change the `resource` output file location to your home directory (leaving the filename the same).

```
...
        <bean id="itemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
                <property name="resource" value="file:$HOME/dining-requests-1-out.csv" />
                <property name="lineAggregator" ref="lineAggregator" />
        </bean>
...
```

4. Use the following procedure to upload the module. Note that if this were a JAR file, the `module upload` command could be used.
   a. Under the `xd.customModule.home` directory, create the `job/xd-job-basic/config` subdirectories.
   b. Copy the `xd-intro-jobs/xd-job-basic.xml` file from the course labs installation directory, to the `<xd.customModule.home>/job/xd-job-basic/config` directory.
5. From the Spring XD shell, run the `module list` command and verify the module appears in the list of Job modules.
6. From the XD shell, create a new Spring XD batch job using the `job create` command.

```
xd:>job create --name myBatchJob --definition "xd-job-basic" --deploy
Successfully created and deployed job 'myBatchJob'
xd:>
```

7. Launch the job, using the `job launch` command.

```
xd:>job launch myBatchJob
Successfully submitted launch request for job 'myBatchJob'
xd:>
```

8. Verify the job completed successfully.

   Open the admin UI (http://localhost:9393/admin-ui/), navigate to the Job Executions page, and verify the status of the job is "COMPLETED". Click on the "Details" icon (magnifying glass) of your job and examine the information shown such as the duration and steps. Feel free to check out other areas such as the Modules, Definitions, and Deployments tabs.

   Navigate to your `$HOME` directory, and examine the contents of the `dining-requests-1-out.csv` file. You should see that the pipes have been replaced with commas.

Chapter 14. amqp: Simplifying Messaging with Spring's AMQP Support

14.1. Introduction

In an effort to improve the performance and robustness of the application, the reward network team wishes to process incoming customer dining events in an asynchronous fashion. Remember that the fundamental use case of the reward network application is *rewarding an account for dining* - that won't change. The goal of this lab is to decouple the process of rewarding an account from the process of receiving a new dining event.

AMQP is a good fit for achieving this goal. As *Dining* objects arrive in the system, they will be sent to an AMQP *Exchange* for delivery by the broker to a *Queue*, where they will wait to be consumed and routed through the existing *RewardNetwork.rewardAccountFor(Dining)* method. Recall that *rewardAccountFor(Dining)* returns a *RewardConfirmation* - this object will need to be routed to its own AMQP *Queue* as well, so that it can be consumed by a *RewardConfirmationLogger* (more on this object later).

When finished, the benefit will be that any number of *Dining* events can arrive in the system, regardless of whether the *RewardNetwork* endpoint is available. When the *RewardNetwork* does come online, it will consume however many *Dining* events have queued up. This is *asynchronous processing*.

This lab is broken into two sections. In the first section you will complete the steps necessary to send incoming *Dining* objects as messages to an AMQP Exchange. In the second section you will consume those messages from a Queue and process them with the *DiningProcessor* object.

What you will learn

1. How to configure and use Spring's *RabbitTemplate*
2. How to configure and use Spring's *MessageListenerContainer* support, via the *rabbit:* namespace.

Estimated time to complete: 60 minutes

14.2. Prerequisites

In order to complete this lab, you must first install RabbitMQ on your workstation. You can find the download and installation guides *here*. Choose the appropriate download and installation guide for your OS.

Once you have completed the installation, start the RabbitMQ broker and leave it running for the duration of the lab.

14.3. Quick Instructions

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

1. **Process five `Dining` objects through the diningProcessor (TODO 01)**

   Open `DiningProcessorTests` and process five Dining objects through the `diningProcessor`. Now run the test. You will see an `UnsupportedOperationException`.

2. **Convert the `dining` to a message and send it to its destination (TODO 02)**

   Click on the exception from the stack trace and implement the `process(Dining)` method on `AmqpDiningProcessor`. Run the test again to see the effect. The test will still fail. You should see a `NullPointerException`.

3. **Define a `RabbitTemplate` bean and inject the `rabbitConnectionFactory` bean defined in the `amqp-infrastructure-config.xml`. (TODO 03)**

   Inject the `RabbitTemplate` bean into the `diningProcessor` bean. Now run `DiningProcessorTests` once again. The Test should pass.

4. **Wait for the batch and assert it's size is 5 (modify the assertion) (TODO 04)**

   Use the `waitForBatch()` to wait 10 seconds for the 5 dinings to get sent, consumed and routed through the `RewardNetwork`. Run `DiningProcessorTests` again. It should fail.

5. **Wire the `rewardNetwork` in an amqp-listener inside an `amqp-listener-container` (TODO 05)**

   Open up `amqp-rewards-config.xml`. Use the `rabbit:` namespace to create a `listener-container` with a single nested listener that references the `rewardNetwork` bean. Configure it to take its input from `rewards.queue.dining` and send its output to the default `response-exchange` (empty string), with a `response-routing-key` of `rewards.queue.confirmation`.

   Note that when you run the test now for the first time, it will fail because it picks up messages that were previously published. However, once those messages have been consumed, from that point on the test should pass.

14.4. Instructions

14.4.1. Sending Dining Messages to a Destination

In this section you will see how to convert a *Dining* instance into an AMQP *Message* and send it to an *Exchange* for delivery to a *queue*.

14.4.1.1. Step 1. Modify the test

Open *DiningProcessorTests* and complete *TODO 01*. Now run the test. You will see an *UnsupportedOperationException*.

14.4.1.2. Step 2. Resolve the error

Click on the exception from the stack trace and implement the *process(Dining)* method on *AmqpDiningProcessor* (*TODO 02*).

Tip

When converting and sending the *Dining* object as a *Message*, you will need to specify the destination queue *rewards.queue.dining* as the routing key. The destination *queue* has already been set up in the *amqp-infrastructure-config.xml* file.

When you're finished making the change, run the test again to see the effect. The test will still fail, but you should see a *NullPointerException* this time.

14.4.1.3. Step 3. Set up the *RabbitTemplate*

The root cause of the *NullPointerException* above is that the *RabbitTemplate* was not injected into our *DiningProcessor* object.

To fix this problem, open *client-config.xml* and look at the *diningProcessor* bean definition. Notice that it is indeed missing its *amqpTemplate* property. Complete *TODO 03* by defining a *RabbitConnectionFactory* bean, and injecting it into a *RabbitTemplate* bean. Finally, inject the *RabbitTemplate* bean into the *diningProcessor* appropriately.

Tip

Use the rabbit:connection-factory namespace to declare the *rabbitConnectionFactory* bean.

Inject the *onnectionFactory* property on your *RabbitTemplate* bean definition. The *rabbitConnectionFactory* has already been defined for this purpose in the *amqp-infrastructure-config.xml*. Refer to this file to understand how the bean and queues are declared.

Now run *DiningProcessorTests* once again. You should have a green bar, meaning that you've successfully processed the five *Dining* objects and sent them to the *rewards.queue.dining* destination.

14.4.1.4. Receiving dining messages from a destination using AMQP listener

In this section you will implement the receiving end of the application. The goal is to watch the *rewards.queue.dining* destination and route *Dining* objects through our existing *RewardNetwork* implementation as they arrive. In this way, the *RewardNetwork* object will become a participant in an *event-driven application* without requiring you to write any extra code.

A note about *RewardConfirmationLogger*

Notice in *DiningProcessorTests* that an object of type *RewardConfirmationLogger* is injected using *@Autowired*. This object is configured to listen on a second AMQP destination named *rewards.queue.confirmation*, which has already been set up for you. Every time a *RewardConfirmation* gets placed on that queue, the *RewardConfirmationLogger* consumes it and adds it to its *confirmations* collection.

14.4.1.5. Step 4. Modify the test

Go back to *DiningProcessorTests* and complete *TODO 04*. Use the *waitForBatch()* to wait 10 seconds for our 5 dinings to get sent, consumed and routed through the *RewardNetwork*.

Tip

Look at the implementation of *waitForBatch()*. Notice that it polls the size of the *RewardConfirmationLogger*'s *confirmations* collection for the specified number of seconds. The method returns as soon as the desired number of *RewardConfirmations* have been processed or the specified number of seconds to wait runs out, whichever comes first.

Run *DiningProcessorTests* again. It should fail, which makes sense because we haven't yet wired up the *RewardNetwork* as a listener on the *rewards.queue.dining* destination. *waitForBatch()* just times out after 10 seconds - no *Dining* objects have been passed through *RewardNetwork,* thus no confirmation messages have shown up for the logger to record.

14.4.1.6. Step 5. Wire up *rewardNetwork* in a listener container

Open up *amqp-rewards-config.xml*. You can see that it's empty. Complete *TODO 05* by using the *rabbit:* namespace to create a *listener-container* with a single nested *listener* that references the *rewardNetwork* bean. Configure it to take its input from *rewards.queue.dining* and send its output to *rewards.queue.confirmation*.

Tip: Understanding *queue-names*

When creating your *rabbit:listener* element, you'll need to specify the *queue-names* attribute. This signifies what AMQP queue the listener consumes messages from. In this case, you're specifying the *name of the queue* as it exists in the AMQP broker. So, use *rewards.queue.dining* for the value of *destination*.

Tip

Recall that the method to invoke on *RewardNetwork* is *rewardAccountFor(Dining)*. The *rabbit:listener* element provides a *method* attribute where you can specify this.

Tip: Understanding *response-exchange* and *response-routing-key*

If the *response-exchange* is left as an empty string, then the message will be published to the AMQP default exchange. Note that every queue declared is automatically bound to the AMQP default direct exchange with a *routingKey* of the queue name. So for the *response-routing-key*, it is necessary to specify *rewards.queue.confirmation* so that the messages are routed to the *rewards.queue.confirmation* queue.

Tip: Don't forget content assist!

Use *ctrl-space* to help you while filling out the *rabbit:* namespace elements. STS is aware of the *rabbit:* namespace and makes content assist available in a number of places.

Now run the test again to confirm that this is working. It may fail the first time due to unprocessed messages left on the queue. If so, rerun it as the messages will have been consumed by the initial test run. If you've got the green bar, you've successfully put together an asynchronous processing pipeline for *Dining* objects. Congratulations!

Chapter 15. remoting: Distributing an application using RMI and Spring Remoting

15.1. Introduction

In this lab you will learn how to distribute a client-server application across the network using Spring's support for the *Remoting* style of integration.

What you will learn

1. How to configure service exporters
2. How to configure client side proxies
3. Serialization
4. Recovering from broken connections

Estimated time to complete: 30 minutes

15.2. Quick Instructions

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

1. **Define a bean of type RmiServiceExporter (TODO 01)**

   Find and open rmi-exporter-config.xml in the rewards.remoting package of the remoting-server project. Complete TODO 01 by creating a bean definition of type RmiServiceExporter. Optionally set 'alwaysCreateRegistry' to 'true' to avoid the time-consuming registry lookup.

2. **Define the 'rewardNetwork' bean (as an RmiProxyFactoryBean) (TODO 02)**

   Open diningentry-config.xml in the rewards.ui package of the remoting-client project. Complete TODO 02 by creating a bean named rewardNetwork of type RmiProxyFactoryBean and provide the serviceInterface and serviceUrl properties.

3. **Assert that 'dining' & 'confirmation' is serializable (TODO 03 and TODO 04)**

   Complete the test implementation by calling the assertSerializable() method where appropriate.

4. **Make Dining serializable (TODO 05)**

   Find and open the rewards.Dining class within the remoting-distributed-api project. Complete TODO 05 by enhancing the class definition to implement the java.io.Serializable interface..

5. **Make MonetaryAmount, SimpleDate, RewardConfirmation, AccountContribution, Percentage serializable (TODO 06-10)**

   Make all shared types are serializable and both tests in SerializabilityTests pass with a green bar

6. Run the distributed application.

15.3. Instructions

15.3.1. Exporting and Consuming the Remote Service

15.3.1.1. Step 1: Review the client-side *DiningEntryUI* application

If you are not already familiar with the command-line *DiningEntryUI*, take a moment to review it by opening *rewards.ui.DiningEntryUI* in the *src/main/java* folder of the *remoting-client* project.

Notice it consists of a simple *main* method that bootstraps a Spring *ApplicationContext* for the *diningentry-config.xml* file. Once the DiningEntry instance is instantiated, its *start()* method is called in order to start a simple command-line interface.

Open *diningentry-config.xml*, where you'll see that there is one bean definition and a *TODO*. You'll complete the *TODO* later, but for now take a look at the *rewards.ui.DiningEntry* class to make sure you understand how it functions. Its `start()` is called from within the `main()` method of the `DiningEntryUI` class.

Notice that *DiningEntry* requires a *RewardNetwork* instance as a constructor argument. Take a look at the *acceptDiningInput()* method and you'll see that *DiningEntry* simply accepts user input through a series of prompts at the command line, builds up a new *Dining* instance based on that input and then calls the *rewardAccountFor(Dining)* method on the *RewardNetwork* object.

Once you're clear about how the application works, move on to the next step!

15.3.1.2. Step 2: Define the server-side RMI service exporter

Spring provides *exporters* that allow you decorate existing POJOs in order to expose them as remote endpoints. In this step, you will configure an exporter to expose the existing *RewardNetworkImpl* bean via RMI. This will be done without touching a single line of code - only Spring configuration will change.

Find and open *rmi-exporter-config.xml* in the *rewards.remoting* package of the *remoting-server* project. Complete *TODO 01* by creating a bean definition of type *RmiServiceExporter*. You will need to provide the following properties:

- *service* (the reference to the bean to be exported)
- *serviceInterface* (the interface that the POJO implements)
- *serviceName* (the name used when binding to the RMI registry, e.g.: 'rewardNetwork')

When setting the *service* property, the bean reference should be to the actual 'rewardNetwork' bean that is imported.

Tip

You can set the *alwaysCreateRegistry* property to *true* in order to save time on startup (there is no need to search for an existing RMI registry during testing)

15.3.1.3. Step 3: Start the server-side RMI endpoint

In this step you'll verify that the exporter configuration is correct by starting the remote endpoint.

Find and run the *RmiExporterBootstrap* class in the *rewards.remoting* package of the *remoting-server* project. Right-click on the class and choose *Run As* -> *Java Application* to start the application.

At the bottom of the console, you should see the following, letting you know that the *rewardNetwork* has been successfully exported.

```
INFO: Binding service 'rewardNetwork' to RMI registry:
 RegistryImpl_Stub[UnicastRef [liveRef:
  [endpoint:[192.168.1.3:1099](local),objID:[0:0:0, 0]]]]
```

Note

Notice that the process continues to run indefinitely. This server process needs to be long-running in order to accept client connections.

Once this is complete, leave the server running and move on to the next step!

15.3.1.4. Step 4: Define the client-side RMI proxy

In this step you'll wire up the existing *DiningEntryUI* application to communicate over the wire to the remote endpoint you set up in the previous step.

Just as configuring the remote endpoint required no code changes, neither will consuming the endpoint.

Briefly open *rewards.ui.DiningEntry* in the *src/main/java* directory of the *remoting-client* project. Recall from Step 1 that the *DiningEntry* object needs just one dependency: an implementation of *RewardNetwork*. Your job in this step is to provide an implementation of *RewardNetwork* capable of communicating over the RMI to the remote endpoint.

To do that, open *diningentry-config.xml* in the *rewards.ui* package of the *remoting-client* project. Complete *TODO 02* by creating a bean named *rewardNetwork* of type *RmiProxyFactoryBean* and provide the *serviceInterface* and *serviceUrl* properties.

Tip

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

15.3.1.5. Step 5: Run the client-side *DiningEntryUI* application

In this step you'll verify that RMI communication between the client and server endpoints works as expected.

Locate the *DiningEntryUI* class in the Package Explorer view. Right-click and select *Run as->Java Application*.

As the application starts up, you'll be presented with the following prompt:

```
Welcome to the Rewards Network dining entry UI

Please enter the following information to create and
                               process a new dining event

Dining amount:
```

Enter a valid dining event by supplying the following values when prompted:

- *Dining amount: 100*
- *Member credit card number: 1234123412341234*

- *Merchant number: 1234567890*

After entering the merchant number, you should see the following error:

```
dining entry error! message was: Could not access remote
             service [rmi://localhost:1099/rewardNetwork];
nested exception is
   java.rmi.MarshalException: error marshalling arguments;
nested exception is:
   java.io.NotSerializableException: rewards.Dining
```

The problem is that you're trying to send objects across the wire that do not implement *java.io.Serializable*. In the next step, you'll fix this problem.

15.3.1.6. Step 6: Make shared types serializable

One of the constraints of using RMI as a remoting protocol is that all types that are distributed across the wire must be 'serializable', i.e. must implement the *java.io.Serializable* marker interface.

In this step, you'll go through the process of making a set of classes serializable. To aid in this process, a unit test has been started for you. In the *remoting-distributed-api* project, find and open the *rewards.SerializabilityTests* class. Go ahead and run the test by right-clicking and selecting *Run as->JUnit Test*. Notice that the bar is green, but this is only because the test does not yet perform any assertions.

Complete the test implementation by calling the *assertSerializable()* method where appropriate (*TODO 03*, *TODO 04*)

Now, run the test once more and notice that the bar is red. Look at the failure message for the *testDiningIsSerializable()* test method. It should read:

```
java.io.NotSerializableException: rewards.Dining
        ...
```

To fix this, find and open the *rewards.Dining* class within the *remoting-distributed-api* project. Complete *TODO 05* by enhancing the class definition to implement the *java.io.Serializable* interface.

Run *SerializabilityTests* once more. Notice that this time, the bar is still red, but it no longer complains about *rewards.Dining* - this time it's *common.money.MonetaryAmount*. This type is also in the *remoting-distributed-api* project. Update this class to implement *Serializable* as well (*TODO 06*), and repeat this process until all shared types are serializable and both tests in *SerializabilityTests* pass with a green bar (*TODOs 07-10*).

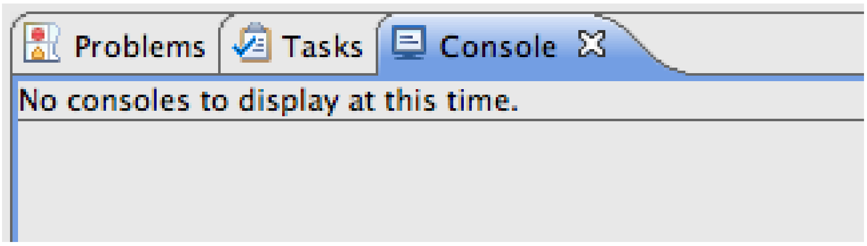
Note

As you can see from the process above, for an object to be *Serializable*, all of its fields' classes must be serializable too. This cascading effect means that we can't just mark *Dining* as *Serializable*, we have to walk the entire network of classes associated with *Dining* and make them implement *Serializable* as well.

Now you're ready to move on to the final step and see the distributed application in action.

15.3.1.7. Step 7: Run the distributed application

TODO 11: Open or make visible the *Console* view in STS. Make sure that any running processes are terminated by clicking the red *Terminate* and gray *Remove All Terminated Launches* buttons until the Console reads *"No consoles to display at this time"*.



No consoles to display at this time.

Now, as before, start the *RmiExporterBootstrap* using *Run As->Java Application*. Again, you should see

```
INFO: Binding service 'rewardNetwork' to RMI registry:
 RegistryImpl_Stub[UnicastRef
  [liveRef:
   [endpoint:[192.168.1.3:1099](local),objID:[0:0:0, 0]]]]
```

Leave that process running, and start *DiningEntryUI* using *Run As->Java Application*. As before, you'll be prompted to enter dining information. Enter the valid information below:

- *Dining amount: 100*
- *Member credit card number: 1234123412341234*
- *Merchant number: 1234567890*

This time, the transaction should succeed and you'll see the following:

```
Dining of $100.00 charged to '1234123412341234' by
```

```
                         '1234567890' on 4/22/09 12:00 AM
Reward of $8.00 applied to account 123456789.
                              RewardConfirmation id is: 1

Would you like to enter another Dining event? [y/n]:
```

Congratulations! You've just created a distributed application using Spring Remoting. While some code changes were necessary to the rewards object model in order to support serialization, no code changes were required for either our service layer POJO (*RewardNetwork/RewardNetworkImpl*), nor for the client (*DiningEntry/DiningEntryUI*) classes.

15.4. Bonus

15.4.1. Bonus #1: Break the Application

It's quite easy to see how fragile a distributed application can be. Simply terminate the *RmiExporterBootstrap* process and attempt to enter another dining via the currently running *DiningEntryUI* process. You'll see the error:

```

dining entry error! message was: Could not connect to
     remote service [rmi://localhost:1099/rewardNetwork];
nested exception is
 java.rmi.ConnectException: Connection refused to host:
                                              192.168.1.3;
nested exception is:
 java.net.ConnectException: Connection refused
```

15.4.2. Bonus #2: Make the Application More Robust

Restart the server by running the *RmiExporterBootstrap* again and then attempt to enter another dining through the *DiningEntryUI* console. Notice that you still get the same "Connection refused" message?

This is because by default, once the connection from the client-side proxy to the RMI server is broken, that connection never gets re-established. You can, however, configure Spring to attempt to reconnect and thus be a bit more resilient to these kinds of failures.

Open the *diningentry-config.xml* file once again. On the *rewardNetwork* bean defintion, provide the *refreshStubOnConnectFailure* property with a value of *true*.

Now complete the following steps:

1. Start *DiningEntryUI* and enter a valid dining (use the values above)
2. Stop the *RmiExporterBootstrap* server process
3. Enter another valid dining. Notice the *ConnectException* occurs as expected.
4. Restart *RmiExporterBootstrap*. The server side is now up again.
5. Enter another dining from *DiningEntryUI* console. Notice that this time, the dining succeeds! The Spring-generated RMI proxy has been configured to try to reconnect if it receives a connection failure, and that's just what happened.

```
WARN : org.springframework.remoting.rmi.RmiProxyFactoryBean
        - Could not connect to RMI service
           [rmi://localhost:1099/rewardNetwork] - retrying

Result: Reward of $8.00 applied to account 123456789.
    RewardConfirmation id is: 1
```

Chapter 16. soap-ws: Exposing SOAP Endpoints using Spring Web Services

16.1. Introduction

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

What you will learn:

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

Specific subjects you will gain experience with:

1. XML Schema Definition (XSD)
2. JAXB2
3. The *WebServiceTemplate* template class

Estimated time to complete: 45 minutes

16.2. Quick Instructions

If you feel comfortable with the material, you can work with the TODOs listed in the `Tasks` view in Eclipse/STS.

1. Complete TODO 01 - 04 to generate an XSD and matching Java classes.
2. Complete TODO 05 to setup the MessageDispatcherServlet.
3. Complete TODO 06 to process the dining requests and return confirmation responses.

4. Complete TODO 07 to enable WS annotation-driven programming.
5. Complete TODO 08 deploying the application to the server. http://localhost:8080/soap-ws/ should now be accessible.
6. Complete TODO 09 - 11 by implementing a test method.

16.3. Instructions

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

16.3.1. Defining the Message Contract

When designing SOAP services the important thing to keep in mind is that the SOAP services are meant to be used by disparate platforms. To effectively accomplish this task, it is important that a contract for use of the service is designed in a way that is accessible to all platforms. The typical way to do this is by creating an XML Schema Definition (XSD) of the messages that will be passed between the client and the server. In the following step you will define the message contract for the rewards application you created earlier.

16.3.1.1. Step 1: Create a sample message

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

Warning

Make sure to use attributes, not subelements, or you'll run into problems later on when you test your code!

16.3.1.2. Step 2: Infer the contract

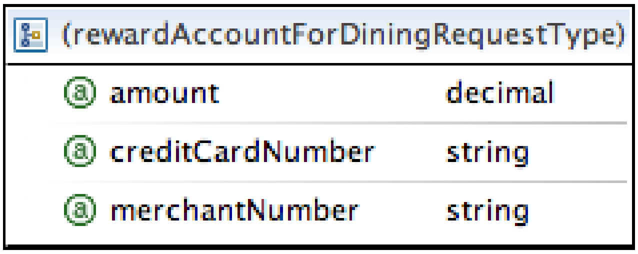
(TODO 02) You now need to infer a contract out of your sample message, in our case an XML Schema (XSD). If you are already experienced with XSDs you could of course also skip the sample message part, and write your schema yourself. But it often saves some time if you start with the sample message and use tools to create a corresponding XSD. You will use *Trang* in this lab, which is a open source schema converter. You already have a working Run Configuration in Eclipse. Just right-click on the file *ws-1 Trang.launch* in your project root and select *Run As->ws-1 Trang*. Trang will create a XSD named *trang-schema.xsd* in *src/main/webapp/WEB-INF/schemas*.

Tip

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

Open the file and inspect it (click on the 'Source' tab if the editor opened with the 'Design' tab). Trang should have generated a definition for the element *rewardAccountForDiningRequest* of type *complexType* with the 3 attributes in it from the previous step. Trang has also generated the types for the attributes, but you will probably have to tweak them. *amount* should be of type *xs:decimal* and the other two of type *xs:string*, *not xs:integer*.

When you've finished defining the *rewardAccountForDiningRequest* element, select the 'Design' tab from the lower left of the editor window and double-click the *rewardAccountForDiningRequest* element. If you have properly created the XSD, your (*rewardAccountForDiningRequestType*) will look like Figure 1.



(rewardAccountForDiningRequestType)	
amount	decimal
creditCardNumber	string
merchantNumber	string

Figure 1: (rewardAccountForDiningRequestType) structure

16.3.1.3. Step 3: Combine with response message

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

16.3.2. Step 4: Generate the classes with JAXB2

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

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

16.3.3. Exporting the *RewardNetwork* as a SOAP Endpoint

16.3.3.1. Step 5: Add the *MessageDispatcherServlet*

(TODO 05) Much like Spring MVC, Spring WS uses a single servlet endpoint for the handling of all SOAP calls. Open the *web.xml* file in the *src/main/webapp/WEB-INF* directory. Add a new servlet named *rewards* with a servlet class of *org.springframework.ws.transport.http.MessageDispatcherServlet*. Next define an initialization parameter for the servlet called *contextConfigLocation* that has a value that points to the servlet configuration file defined in the same directory.

16.3.3.2. Step 6: Create the SOAP endpoint

(TODO 06) Now that the Spring WS infrastructure has been set up, you must create an endpoint to service the *RewardNetwork* requests. You will use the annotation style mapping in this lab, including the latest annotations introduced in Spring Web-Services 2.0.

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

Now you have to implement the logic inside of the method. As the generated classes are not your domain classes you must convert them to the classes which are used in the service. Create a new *Dining* object with

Dining.createDining(String amount, String creditCardNumber, String merchantNumber). You will get the needed values out of *RewardAccountForDiningRequest*. Then call the method *rewardAccountFor* on the *rewardNetwork*. Finally create a *RewardAccountForDiningResponse* object and return it.

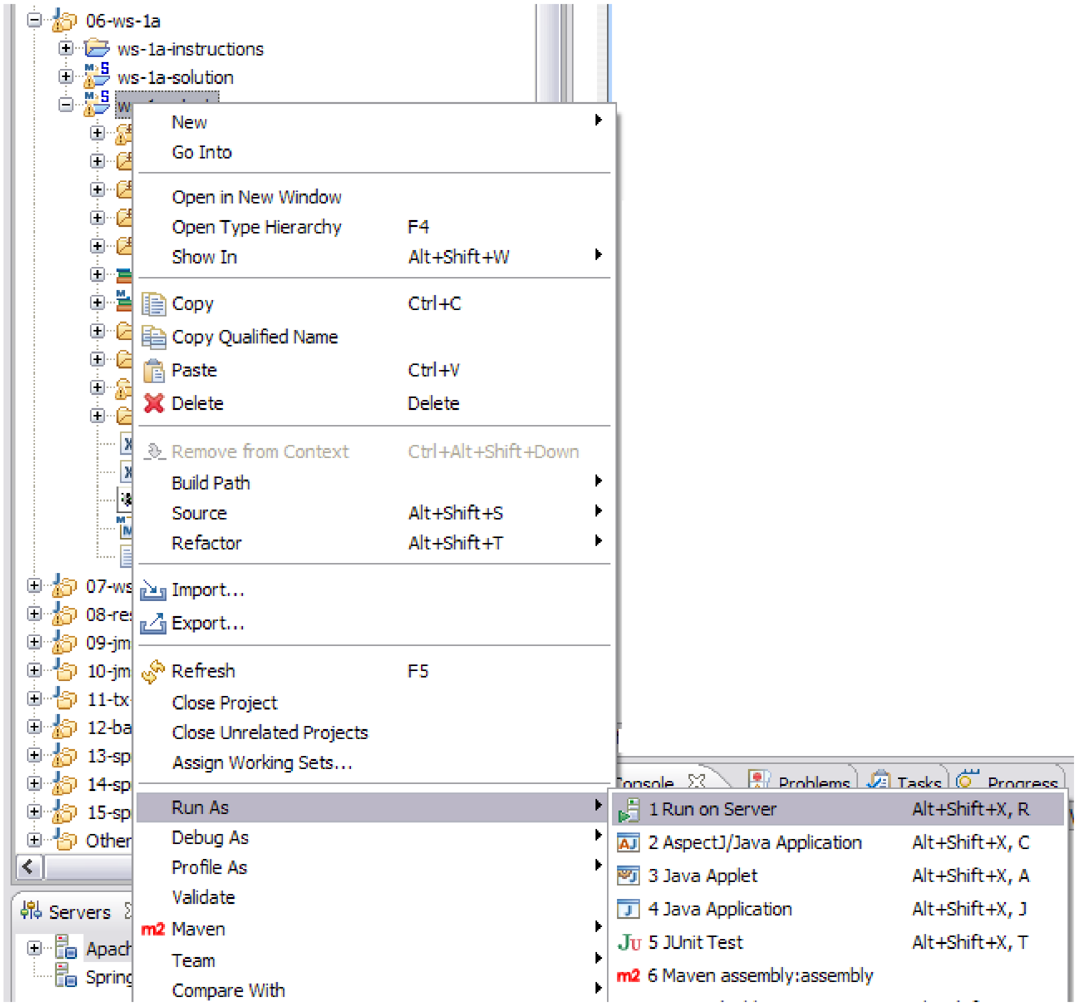
Complete your endpoint now by mapping the method to the correct request by placing an annotation on the method that uses the payload root's element name.

16.3.3.3. Step 7: Complete the Spring WS configuration

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

16.3.3.4. Step 8: Start the web application

(TODO 08) Now that the SOAP endpoint has been wired properly you must start the web application to export it. Start the web application for this project as shown below.



Once started, the welcome page (just a static index page at the context root) should be accessible as http://localhost:8080/soap-ws/

16.3.3.5. Consuming services from a SOAP endpoint

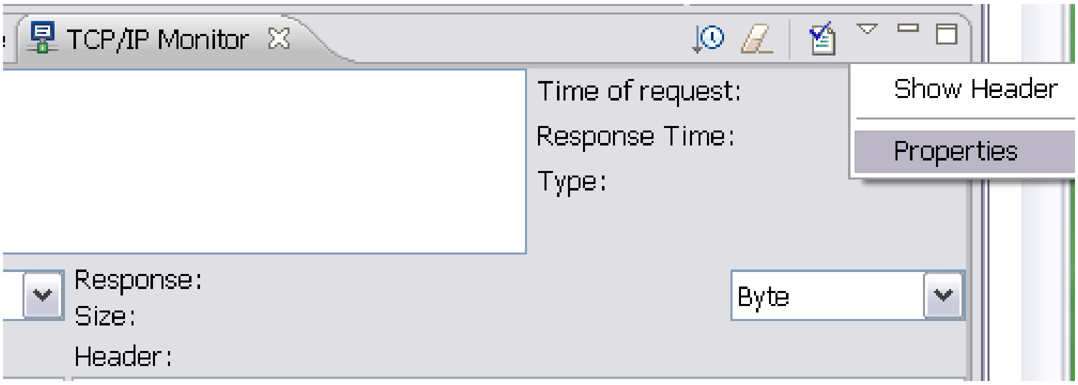
At this point you've successfully exported a service to a SOAP endpoint without changing the original class. If you are acting as a provider of services to other clients this would be all that you need to do. But there are many cases where you need to consume SOAP services as well. When doing this, it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

16.3.3.6. Step 9: Test the web service

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

16.3.3.7. Step 10: Using the TCP/IP monitor to see the SOAP messages

(TODO 10) Whether your test ran OK or not, you've probably noticed that there's not much to see when you run it: the actual content of the SOAP request and response is not available. When writing web services or web services clients, it's nice to see what XML is actually sent from the client to the server and vice versa. Several tools exist to help you with this. One of these tools is built in with Eclipse's *Web Tools Plugin* (WTP) and is called the *TCP/IP Monitor*. It is a view that you can add to your perspective. Type *Ctrl-3* and enter *TCP* followed by *Enter* to add the TCP/IP Monitor view to your perspective. Click the small arrow pointing downwards and choose "properties".



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

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

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

16.3.3.8. Step 11: Using *WebServiceTemplate* with JAXB2

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

16.3.4. Bonus #1: Find the WSDL

(TODO 12) We don't need to write a WSDL ourselves; Spring-WS can generate one for us based on some conventions.

If you have time, refer to the slides to determine the URL where the WSDL is exposed by this lab; enter the URL in a browser and examine the WSDL; note that it contains the schema you created earlier.

16.3.5. Bonus #2: Plain Old XML (POX)

(TODO 13) In this section, if you have time, you will see how we can make some simple configuration changes and then just exchange the payload of the SOAP message we have been using so far (omitting the SOAP Envelope).

Take a look at rewardNetwork-servlet-pox-config.xml; you will notice that it imports rewardNetwork-servlet-config.xml and overrides the definition of two beans which are provided by default. To change the server, edit the web.xml file and change the contextConfigLocation for the MessageDispatcherServlet to point to the rewardNetwork-servlet-pox-config.xml file. Redeploy the project (*Run As->Run on Server...*).

We also need to change the messageFactory on the client side so it, too, will use POX. We do this by adding another property to the WebServiceTemplate defined in client-config.xml (in src/test/resources/rewards/ws/client).

```
<property name="messageFactory">
   <bean class=
    "org.springframework.ws.pox.dom.DomPoxMessageFactory"/>
</property>
```

Again, this overrides the default message factory. Rerun the client through the tcp/ip monitor and notice the content is now simply the payload.

Chapter 17. ws-advanced: Advanced Spring Web Services

17.1. Introduction

In this lab you'll see how interceptors and error handling can be applied to send specific faults back to a SOAP client and how this can be tested without the need to deploy an application to a local server.

What you will learn:

1. Adding logging and validation
2. Configuring SOAP-specific error handling
3. Running out-of-container integration tests for your Spring-WS applications

Specific subjects you will gain experience with:

1. Logging and validating interceptors
2. Exception to SOAP fault mapping
3. Spring-WS 2.0's new integration testing support

Estimated time to complete: 30 minutes

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*)).

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.

17.3. Instructions

The instructions for this lab are organized into sections. First you'll add exception handling for requests that do not pass validation against the published XML Schema and test that using the new integration testing support in Spring-WS 2.0. Then you'll add exception handling for when the given input doesn't match data that's stored in the database and test that as well.

17.3.1. Verifying successful handling of valid requests

Your first assignment is to add validation to incoming requests. Before you add the corresponding interceptor, it would be good to establish that valid requests result in an expected response first: that way you can check if you didn't introduce any regressions.

17.3.1.1. Run an end-to-end test

(TODO 01) First deploy the project to the local server instance. This project is basically the same as the solution to the Spring-WS (soap-ws) lab, so if you did that lab, you already understand how it works. Once the server has started, run the `SoapRewardNetworkTests` to verify that the application works as expected. It would be nice if you could see what a valid request made by the client looks like, so you can test with both valid and invalid requests after adding validation support.

(TODO 02) Add a logging interceptor to the application. Open `rewardNetwork-servlet-config.xml` (it's in the `WEB-INF` directory). Add a `<ws:interceptors>` element with an inner bean definition for a `PayloadLoggingInterceptor`. Restart the application and run the end-to-end test again. You should now see the payload for both the request and the response in the Console View that contains the server output.

Note

Note that the output is logged at `DEBUG` level: in the `log4j.xml` config file you can check that logging for the relevant category has indeed been set to `DEBUG`.

Now that we know what input to test with, it would be preferable if we could use an out-of-container integration test so that you don't need to deploy to the server and restart all the time. This is what you'll do in the next section.

17.3.1.2. Write an out-of-container integration test

(TODO 03) A start for the test has been given already: open `ServerIntegrationTests` and autowire an `ApplicationContext` into a new field for the class. Use this to create a new `MockWebServiceClient` that you assign to the pre-defined field variable.

(TODO 04) The next step is to write a test that checks for the correct response payload given a valid request payload. Using the information from the Console, complete the test method by using the `mockClient` to check that a request with the given payload results in a response with the given payload. Refer back to the slides for details: note that the static methods of the `ResponseMatchers` class are already imported.

Note

Unfortunately Eclipse will not suggest any of these methods when you use code assist (Ctrl-Space) unless you type at least one letter of the method you want to invoke, so make sure to do that in order to see the available methods.

You can stop the server now, since we no longer need it.

Now run the test and notice that it fails. Check the output in the Console View to see why: the application context created by the test is not a `WebApplicationContext`, so it cannot find the referenced XML schema.

(TODO 05) To fix this, move the `schemas` directory from the `WEB-INF` folder to the `src/main/resources` folder and updating the `location` attribute of the `<ws:xsd>` element in `rewardNetwork-servlet-config.xml` accordingly.

Tip

Remember that Spring uses a `classpath:` prefix for resources located on the classpath.

Run the test again and ensure that it passes. Check the log output to verify that the regular handler chain including interceptors is still invoked, even though you're running the test out-of-container.

17.3.2. Adding validation for incoming requests

We want invalid incoming requests to result in a SOAP Client fault.

17.3.2.1. Write a test that verifies validation

(TODO 06) To see what currently happens for invalid requests, first switch back to the `ServerIntegrationTests` and complete the test method by sending a request without a `creditCardNumber` attribute and verify that you receive a 'Client' or 'Sender' fault in the `invalidRequestWithoutCreditCardNumber` test method.

Note

'Sender' is the term used by SOAP 1.2, but most applications still use SOAP 1.1.

Run the test and notice that it fails: the client is getting a 'Server' instead of a 'Client' fault code, which is the result of the server not knowing how to handle empty credit card numbers. This can easily be fixed by validating the incoming request against the schema you created in the previous lab.

17.3.2.2. Add the validating interceptor

(TODO 07) Switch to `rewardNetwork-servlet-config.xml` and add a `PayloadValidatingInterceptor` with a `schema` property that refers to the schema in the `schemas` directory that's on the classpath.

Run your test again and make sure that it passes now.

17.3.3. Mapping certain exceptions to certain SOAP faults

You've made sure that requests that don't pass validation will result in a 'Client' fault and will not be processed by the endpoint. But what about requests that simply contain invalid attribute values?

17.3.3.1. Write a test that checks for a Client error for invalid attribute values

(TODO 08) To test that, switch back to `ServerIntegrationTests` and implement the `invalidRequestWithUnknownCreditCardNumber` test method. Just reuse the request source from the first test method and change the credit card number into something else, and then make sure that the client receives a 'Client' fault again.

Run the test and notice that it fails: the server is sending a 'Server' fault. Look at the output in the Console View to note the type of the exception that caused this. We'd like to map this standard Spring exception onto a 'Client' fault, since it's not the server's fault that the client is specifying non-existing data.

17.3.3.2. Add an exception resolver to map to a Client fault

(TODO 09) To make it so, switch back to `rewardNetwork-servlet-config.xml` and add a bean of type `SoapFaultMappingExceptionResolver` that maps the thrown exception to a 'Client' fault and defaults to 'Server' faults for other exceptions. Check the slides for details if you need any help with this.

Note

The system defines two other resolvers by default - `SoapFaultAnnotationExceptionResolver` and `SimpleSoapExceptionResolver`. The second is the default catch-all so the server always returns a `SERVER` fault unless otherwise configured. The order that resolvers are used is defined by their `order` bean property. The defaults are 0 and MAXINT respectively. We need to make sure our resolver is used first, so set its order bean property to the value -1: `<property name="order" value="-1"/>`. Any value less than zero will do.

Now run the test again and make sure it succeeds this time. When you've completed this step, you're done with lab.

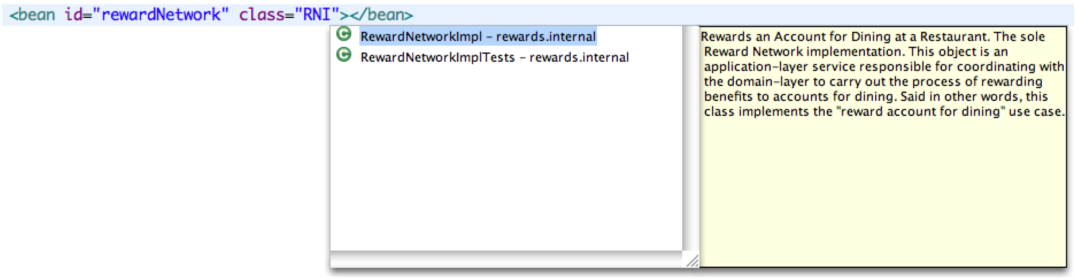
Appendix A. Spring XML Configuration Tips

A.1. Bare-bones Bean Definitions

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

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

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

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

Bare-bones bean definitions

A.2. Bean Class Auto-Completion



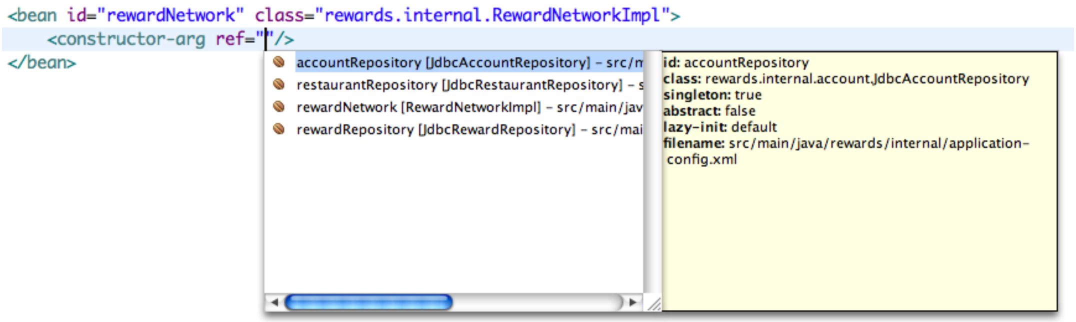
```
<bean id="rewardNetwork" class="RNI"></bean>
```

RewardNetworkImpl – rewards.internal
RewardNetworkImplTests – rewards.internal

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

Bean class auto-completion

A.3. Constructor Arguments Auto-Completion



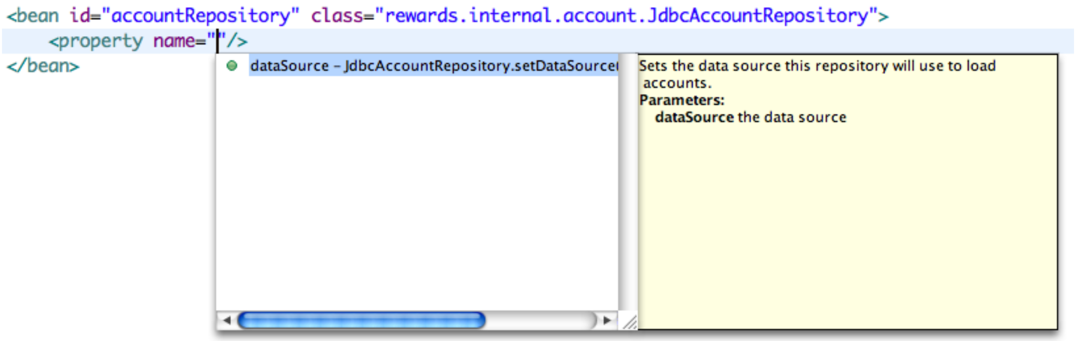
```
<bean id="rewardNetwork" class="rewards.internal.RewardNetworkImpl">
    <constructor-arg ref=""/>
</bean>
```

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

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

Constructor argument auto-completion

A.4. Bean Properties Auto-Completion



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

```
dataSource - JdbcAccountRepository.setDataSource(
```

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

Bean property name completion

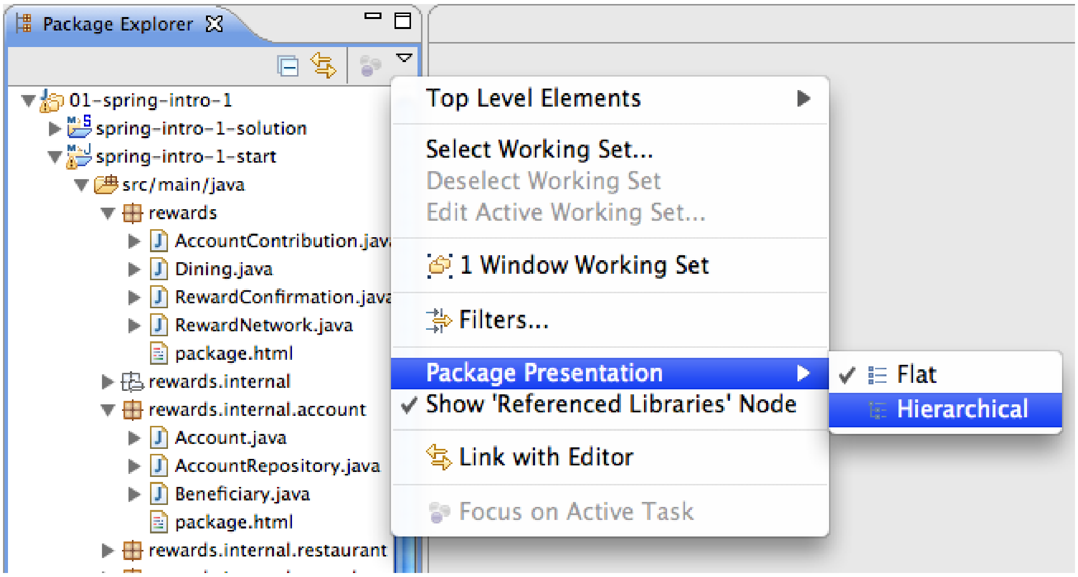
Appendix B. Eclipse Tips

B.1. Introduction

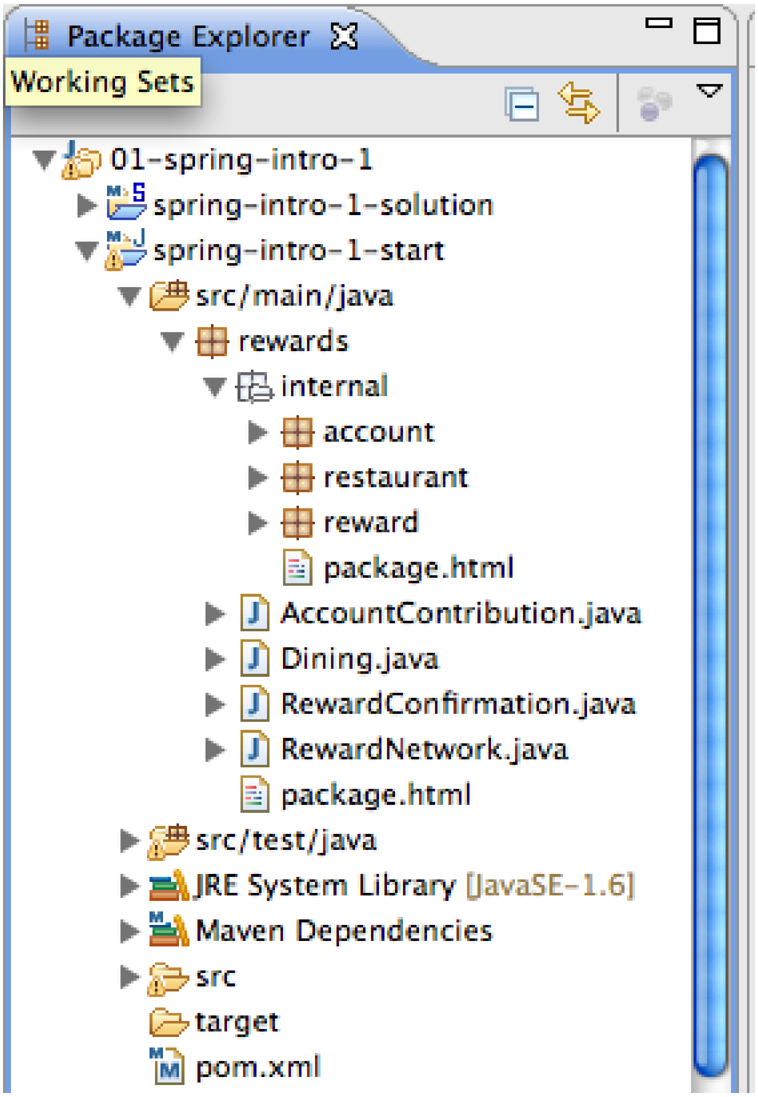
This section will give you some useful hints for using Eclipse.

B.2. Package Explorer View

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



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



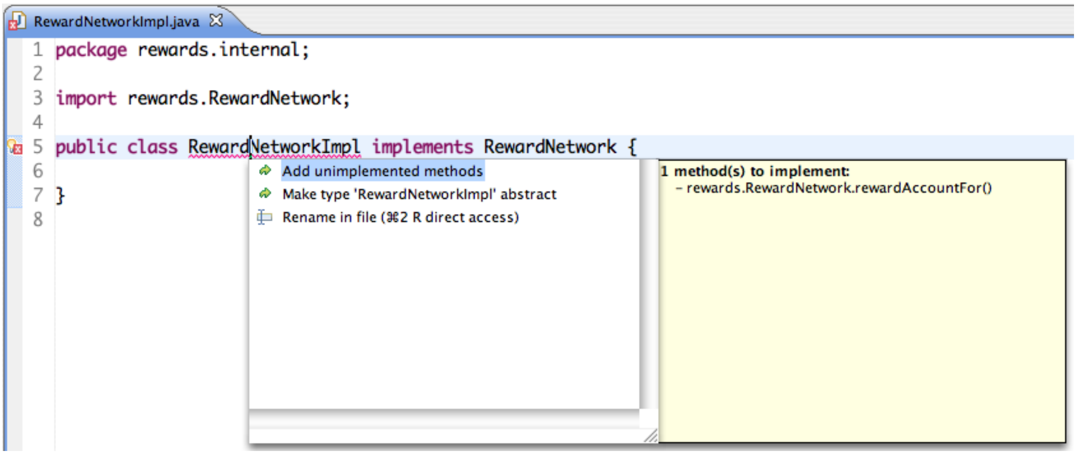
Package Explorer

Working Sets

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

The hierarchical view shows nested packages in a tree view

B.3. Add Unimplemented Methods


RewardNetworkImpl.java

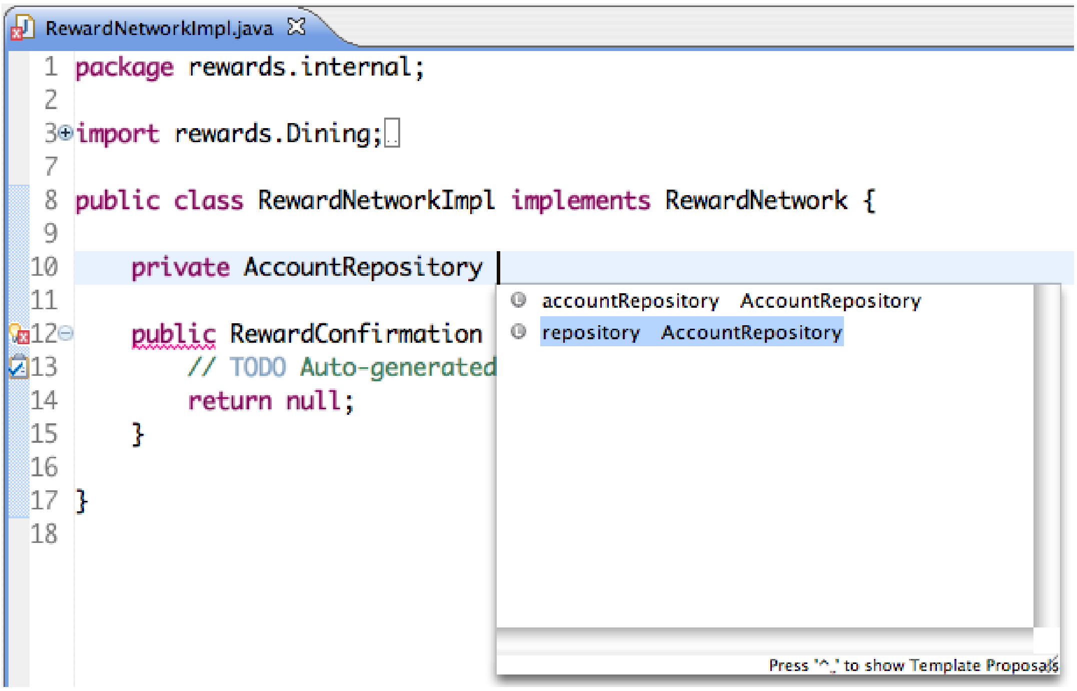
```
package rewards.internal;

import rewards.RewardNetwork;

public class RewardNetworkImpl implements RewardNetwork {

}

```

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

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

"Add unimplemented methods" quick fix

B.4. Field Auto-Completion


RewardNetworkImpl.java

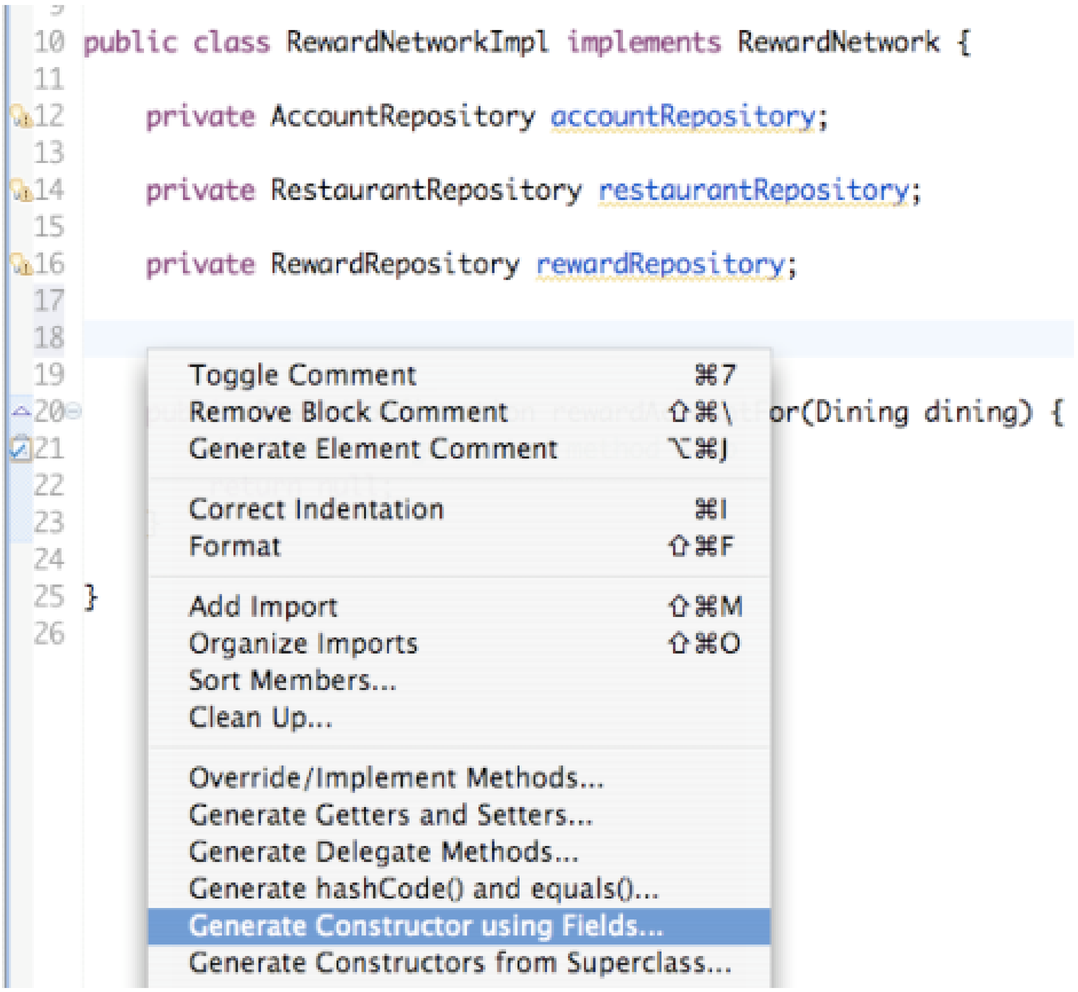
```
package rewards.internal;

import rewards.Dining;

public class RewardNetworkImpl implements RewardNetwork {

    private AccountRepository 

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

}

```

accountRepository   AccountRepository
repository   AccountRepository

Press '^,' to show Template Proposals

Field name auto-completion

B.5. Generating Constructors From Fields



```
public class RewardNetworkImpl implements RewardNetwork {

    private AccountRepository accountRepository;

    private RestaurantRepository restaurantRepository;

    private RewardRepository rewardRepository;


                                                       or(Dining dining) {




}
```

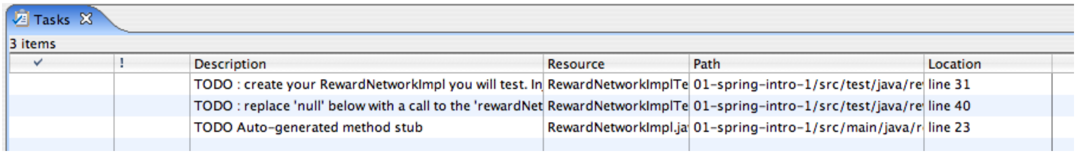
Toggle Comment ⌘7
Remove Block Comment ⇧⌘\
Generate Element Comment ⌥⌘J
Correct Indentation ⌘I
Format ⇧⌘F
Add Import ⇧⌘M
Organize Imports ⇧⌘O
Sort Members...
Clean Up...
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate hashCode() and equals()...
Generate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

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

B.6. Field Naming Conventions

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

B.7. Tasks View

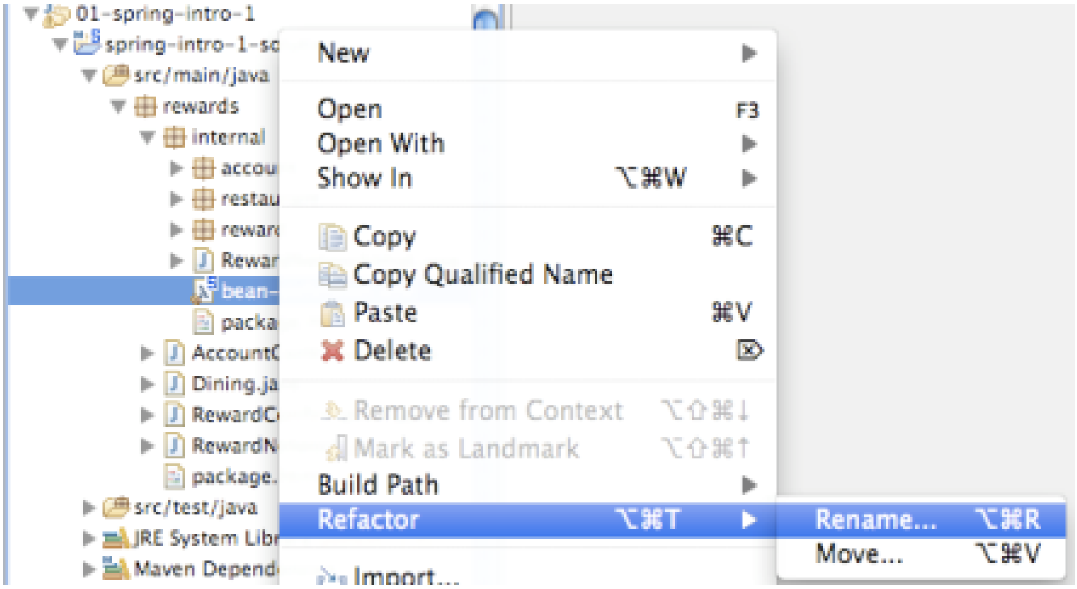


✓	!	Description	Resource	Path	Location
		TODO : create your RewardNetworkImpl you will test. In	RewardNetworkImplTe	01-spring-intro-1/src/test/java/re	line 31
		TODO : replace 'null' below with a call to the 'rewardNet	RewardNetworkImplTe	01-spring-intro-1/src/test/java/re	line 40
		TODO Auto-generated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

The tasks view in the bottom right page area

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

B.8. Rename a File



Renaming a Spring configuration file using the Refactor command

Appendix C. Using Web Tools Platform (WTP)

C.1. Introduction

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

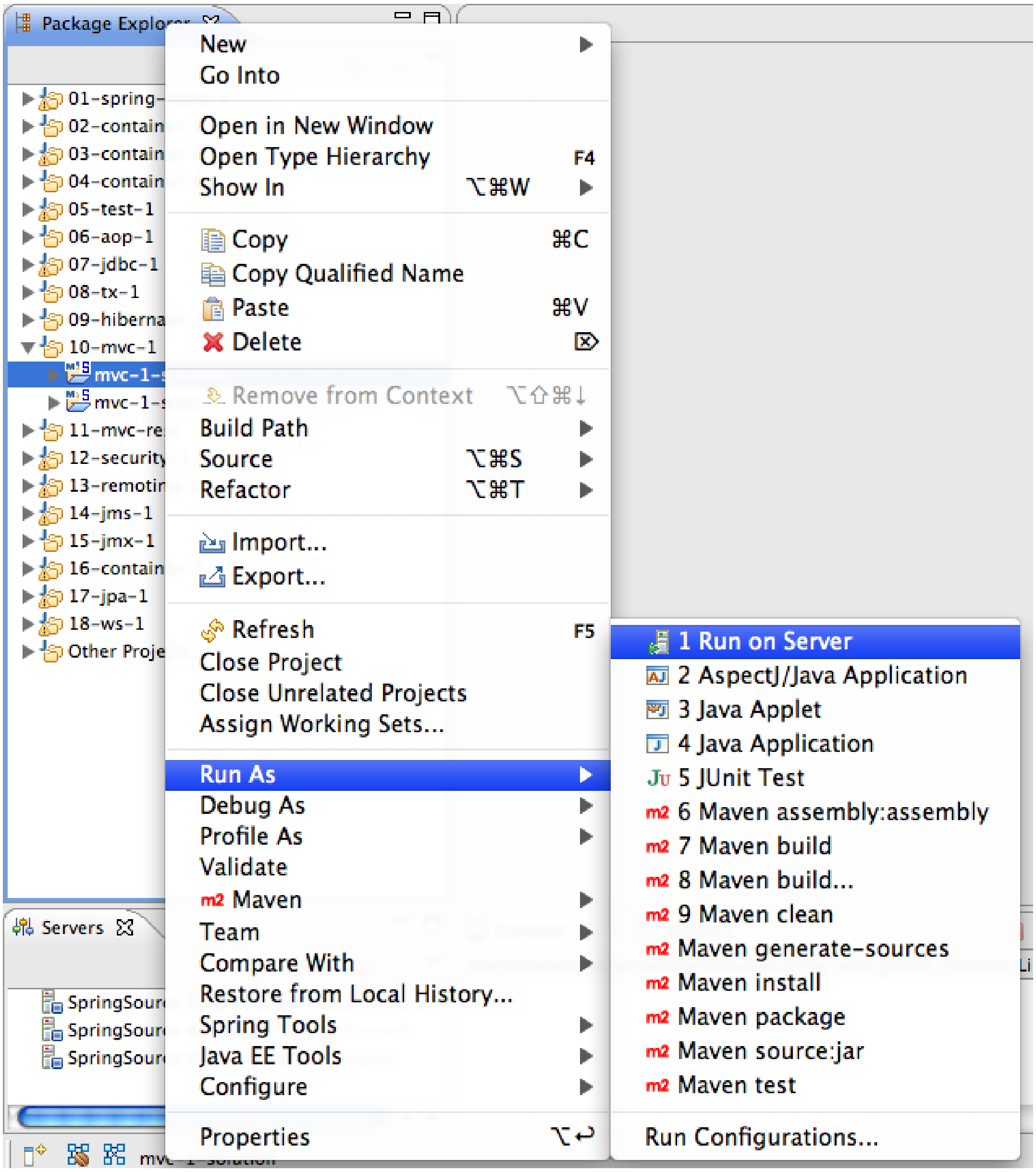
C.2. Verify and/or Install the Tomcat Server

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

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

C.3. Starting & Deploying to the Server

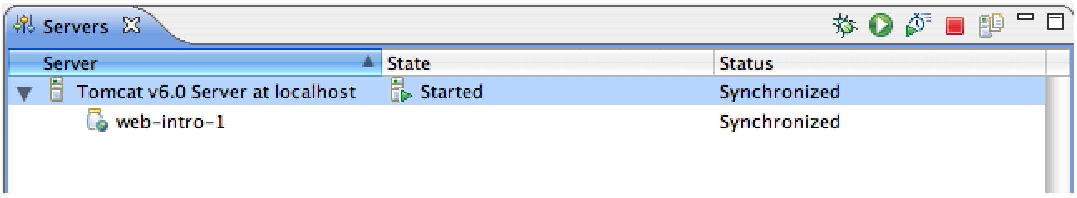
The easiest way to deploy and run an application using WTP is to right-click on the project and select '*Run As*' then '*Run On Server*'.



Run On Server

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

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



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

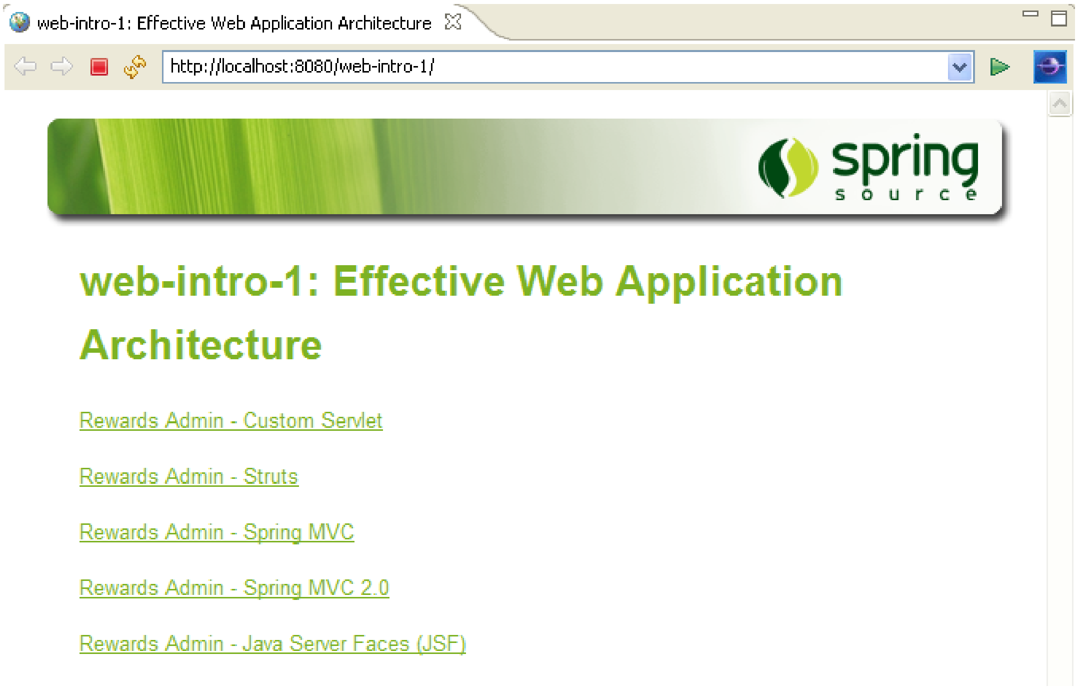
Running on Server

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

Tip

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

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


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

web-intro-1: Effective Web Application Architecture

Rewards Admin - Custom Servlet

Rewards Admin - Struts

Rewards Admin - Spring MVC

Rewards Admin - Spring MVC 2.0

Rewards Admin - Java Server Faces (JSF)

Start Browser

Tip

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