Spring Framework Fundamentals

Second Edition

Practical Guide

Table of Contents

Practical Session 1 (Chapter 5, Wiring Applications)

For the rest of the course, you will be creating a **Customer Management System.**

For details on the system, watch the “practical requirements” video in the Chapter 5 folder - Richard will describe the requirements for the system and he’ll also show you the code that has already been written for you.

Alternatively, see page 12 of this document for full details on the requirements for the system.

Your task is:

1. Create Mock Implementations of the two service interfaces we have provided.
2. Configure the mock implementations in the Spring container
3. Write a simple console based test harness that gets the services from the container and tests their functionality.

For this chapter, make the two services independent of each other. In the next chapter, we’ll combine the two services to provide full call handling functionality.

Note: in the CustomerManagementService interface, there are two very similar methods:

public Customer findCustomerById(String customerId);

public Customer getFullCustomerDetail(String customerId);

The difference between them is that the first should return just the customer, with a null for the associated calls. The second should return the customer and their calls. This difference is needed when we use a database, we want to avoid loading in too much data.

However, for this in-memory session, it doesn't matter and you can implement both methods as returning the full customer object.

If you have any problems with this practical, a full walkthrough solution will follow in chapter 5.

Practical Session 2 (Chapter 8, More Wiring)

There is a third service interface that we have supplied for you. This is in the “calls” package, and is called the CallHandlingService:

```
public interface CallHandlingService
{
   public void recordCall(String customerId, Call newCall)
          throws CustomerNotFoundException;
}
```

Recording a call involves two steps:

1. Update the customer service with the new call
2. Update the diary service with any actions from the call

The idea is that this service integrates the CustomerManagementService and DiaryManagementService into a single service.

Your task is to:

- Implement this service, injecting the two services in as dependencies.
- Decide on whether to use setter or constructor injection

If you haven't already, install the SpringIDE and use it to check your configuration.

Practical Session 3 (Chapter 12, JDBCTemplate)

In this chapter, you'll upgrade your Mock Implementation of the CustomerManagementService to use a production quality implementation instead.

We have already provided you with a DAO interface for Customers – this is provided in the "dataaccess" package.

The steps to follow are:

- Create a production implementation of the CustomerManagementService, this time calling the dao.
- Implement the DAO using the JDBCTemplate
- Inject the DAO into the Service class, using the XML
- Configure a connection pool using Apache-DBCP

To get you started, the SQL for creating the table would be:

```
.1   CREATE TABLE CUSTOMER(CUSTOMER_ID VARCHAR(20),
COMPANY_NAME VARCHAR(50), EMAIL VARCHAR(50), TELEPHONE
VARCHAR(20), NOTES VARCHAR(255))

.2   CREATE TABLE TBL_CALL(NOTES VARCHAR(255),
TIME_AND_DATE DATE, CUSTOMER_ID VARCHAR(20))
```

Note: it is important you don't call the second table "CALL" - this is a reserved word on the HSQLDB database!

We've already supplied you with the implementation of the ActionDao – you just have to implement the customerDAo

We've provided the HSQLDB jar file for you.

- Database URL is : jdbc:hsqldb:file:database.dat;shutdown=true
- The Driver Class name is org.hsqldb.jdbcDriver
- Username = sa
- Password = (blank)

Clue: the getFullCustomerDetail() method is probably the hardest - you need to find the customer record and it's associated calls. You could do this using an SQL join, but it's easier to do one query to find the customer, then do a second select on the call table. Finally, just connect the calls to the customer (customer.setCalls).

Practical Session 4 (Chapter 16, AOP Timing Advice)

Note: This exercise is optional .You do NOT have to complete this exercise in order to continue with the rest of the course.

AOP has a wide variety of uses, but to establish the principles, we're going to repeat the implementation of the performance timing advice in this practical session.

The steps to follow are:

- Write an "Around Advice" class that is capable of timing a method invocation. This advice will be the same as in the theory video.
- If you have time, modify your advice so that it also triggers on DAOs as well as service classes.

Use either the aop:namespace (XML) or the annotations to configure the pointcuts. The solution video will use XML.

You will need some extra syntax to complete this practical - please watch the chapter 16 video, where Richard will take you through logical operators and how to include subpackages.

Note: you will need to use the XML schema in this chapter as shown below. You will be able to find this in the Spring reference manual but we've reproduced it here for convenience – ***don't try to type it for yourself – copy and paste!***

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

Practical Session 5 (Chapter 21, Transactions)

Make the methods in your three service classes transactional.

Decide on what the transaction attributes for each method should be (hint: REQUIRED is probably needed for all of them, but think about which could be tagged with READ_ONLY).

You can either of the approaches (tx: namespace or annotations). In the solution video, we're going to use the annotation approach.

To check that your transaction management is working, you'll need to set logging on. We have provided a file called log4j.properties.disabled. Rename this file to remove the ".disabled", and then you can modify this file to set logging on for your particular transaction manager.

As before, here is a copy of the namespace declaration you'll need at the top of your XML:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"

    xsi:schemaLocation="
     http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx.xsd
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop.xsd">
```

Practical Session 6 (Chapter 23, Autowiring)

Note: This exercise is optional .You do NOT have to complete this exercise in order to continue with the rest of the course.

If you want to practice with annotations, use autowiring and either set of annotations (@Service, @Component, @Repository OR @Named) to replace as much of the XML wiring as you can.

Do this in a second copy of your XML file - call it application-annotations.xml - we don't want to lose the work you've done in previous exercises!

Practical Session 7 (Chapter 26, JPA Data Access)

Note: This exercise is optional .You do NOT have to complete this exercise in order to continue with the rest of the course.

Create a new implementation of the CustomerDAO and ActionDAO using an ORM. We will be using JPA in the walkthrough video.

If you haven't written any JPQL (The Query Language) before, here are some hints:

For getting all the customers:

- `select customer from Customer as customer`

For getting a customer by ID:

- `select customer from Customer as customer where customer.customerId=:customerId`

To get a customer by companyName:

- `select customer from Customer where customer.companyName=:companyName`

Note that by default, when retrieving instances of a particular class, JPA does **not** retrieve instances of associated classes. So in this case, no Call objects will be retrieved. This is done to save memory.

We do have the getFullCustomerDetail method, which must also retrieve the linked calls. To do this, you can use the query:

- `select customer from Customer as customer left join fetch customer.calls where customer.customerId=:customerId`

The “left join fetch” command tells Hibernate to include the calls as part of the query.

For your ActionDao, you’ll need to find all incomplete actions:

- `select action from Action as action where action.owningUser=:user and action.complete=false`

Notes and Hints:

- If you are using XML wiring instead of annotations, you must include the following directive:

  ```
  <context:annotation-config/>
  ```

- For addCall, you can just find the customer and then add a call to it - JPA will detect the object has changed and will issue SQL to update the calls table.

- For the update method, we need some JPA not covered on the video. You will need to call:

  ```
  em.merge(customerToUpdate);
  ```

  The merge method "makes the object persistent again"; this means that if the object's state is different to the state in the database, then an update statement will be issued - exactly what we need!

  For the delete method, you need to make the object persistent again, before you delete it. To do this,

  ```
  oldCustomer = em.merge(oldCustomer);
  em.remove(oldCustomer);
  ```

  (On the walkthrough video, Richard initially makes a mistake and calls em.merge(oldCustomer), without assigning the result. This is an obscure part of the JPA API, you must assign the result if you then go on to call remove. We cover this in the full JPA/Hibernate course; don't worry, Richard corrects the error).

Practical Session 8 (Chapter 30, Integration Tests)

To complete the practical work, write a set of integration tests to test the functionality of your three services.

This will replace your client console application that you have used through the course.

Don't worry about writing extensive tests; the important thing is to try out the Spring support - two or three test methods should be enough.

CRM System Requirements

You are writing a system that enables companies to manage their customer's details. In other words, a CRM system (Customer Relationship Management).

The requirements for the system are:

- The CRM system needs to store details of customers.
- Every time a customer calls the company, we need to be able to record the details of the customer's call. For example : "at 4:30pm on the 2nd December, Larry from Acme Corp called asking if we are interested in buying their Rubber Bands"
- When a call is received from a customer and the call is logged, we may need to record one or more *Actions* against the call. For example: "need to call back Martha by 9th December to order 144 boxes" and "raise purchase order with accounts department by 3 December".
- The system must be able to manage a diary for all of the users of the system, and provide email alerts when an action is due.

In the "starting code" folder, we've provided you with a simple domain model for the system.

The Customer class represents a single customer – eg Acme Corp

The Call class represents a single call made by a customer – eg "Larry Wall called from Acme on 2 December".

The Action class represents a job that we have to perform as a result of a call – eg "Check with marketing that the sales brochure will be ready by November 6 2016".

The classes are related as follows:

- Each customer can make many calls.
- Each call is owned by a single Customer

If you are familiar with UML, we've expressed the domain model graphically in the following diagram:

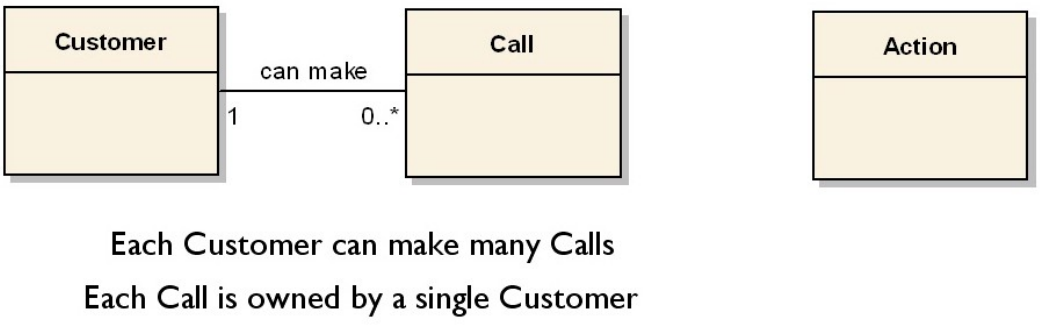

Figure 1 – The simple domain model

Note that we are not relating the call and action class together. We decided to keep them separate for two reasons:

1. The actions might be stored in some kind of third party calendar or diary system, so it is necessary to keep them separate.
2. We want to keep the amount of SQL you need to write to a minimum, in order to keep the course focussed on Spring.

Naturally, if you wish, you can add in the Call/Action relationship into your own implementation.

The Use Cases

Create Customer

When a new customer calls, we need to capture their information (in particular their email address and telephone number).

We'd also need a matching Delete Customer and Update Customer use case.

Add Call Report

When a call is made or received, the details of the call are entered. As a minimum, the time and date is recorded. Optionally (and recommended), notes are added.

Any actions required are also recorded (for example: "Call Claire back next Monday with Ross' availability"). Actions are recorded in the user's diary.

View Customer Details / View Call Log

We'd like to able to list all of the customers in the system, or to be able to search by various criteria.

Additionally, the user will be able to select a particular customer and then be able to see the full call log for that customer.

View Diary

The user can view their diary. Eventually we'd like a full calendar on an interactive web page, in the style of Outlook/Google Calendar. For this course, a simple list of all actions for a particular user will be fine.