GemFire Developer Course

Lab Instructions

Your guide to completing the hand-on labs

Version 9.0.2a

Pivotal

Copyright Notice

This Page Intentionally Left Blank

Table of Contents

Chapter 1. Installing the GemFire labs Environment

1.1. Introduction

Welcome to the GemFire-Developer course. We have tried to make the lab setup as simple as possible. Your instructor will distribute a zip file that you can use to install all the necessary materials onto your file system.

What You will do:

- Install and configure the course materials, including a copy of GemFire, and of the Spring Tool Suite (STS)
- Start servers and verify correct configuration

Estimated completion time: 30 minutes

1.2. Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks.

1.2.1. GemFire Installation Prerequisites

1. Make sure that your system meets the system requirements listed. Refer to the product documentation at http://docs.gopivotal.com/gemfire/.
2. Check that your Java version is appropriate and that Java is on your path. To check your current Java version, type java -version at a command-line. Your output should look something like this

```
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b17, mixed mode)
```

3. To install GemFire and the lab environment, you will need Java Developer Kit 8. If this command returned an error, that means Java is either not installed, or not on your `PATH` .
   You can download Sun/Oracle Java SE here
4. You will also need a copy of apache maven. Download and install maven, set your JAVA_HOME

environment variable, and verify your installation by running "mvn --version" from the command line.

1.2.2. Install Lab Environment

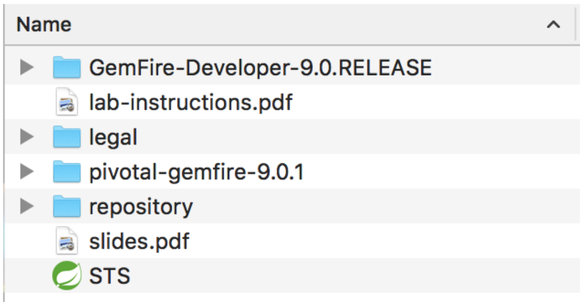
You will be using the Spring Tool Suite (STS) IDE for building and executing most of the exercises for this course. The lab installation package includes not only the STS but also GemFire and lab files. To install, perform the following steps.

1. Locate the appropriate zip file for your target platform. Several are provided for Windows (32 and 64 bit), Linux (32 and 64 bit) and Mac. Once located, unzip the file to some working directory, perhaps $HOME.

   Note

   Make sure you don't have any blank spaces in the path you choose for installation. For example, avoid choosing directories with path such as `C:\Document and Settings\gemfire`.

2. Once unzipped, cd into the course folder, which should be named GemFire-Developer-9.0.2a.RELEASE. Note the folders underneath the course directory.
   The lab files are in the GemFire-Developer-9.0.2a.RELEASE subfolder (lab folder has the same name as the course installation folder). The GemFire installation will be found at `pivotal-gemFire-9.0.1`

   Name
   GemFire-Developer-9.0.RELEASE
   lab-instructions.pdf
   legal
   pivotal-gemfire-9.0.1
   repository
   slides.pdf
   STS

   Figure 1.1. Listing of folders created by course installer

   Change directories to GemFire-Developer-9.0.2a.RELEASE/server-bootstrap. This folder contains the basic setup and initialization tooling for the server side of these labs.

1.2.3. Starting services

In this section, you will configure the environment based on your installation and start the services. To do so, perform the following steps.

1. Two files have been set up in the scripts folder to facilitate setting up the environment variables. Edit one of the configuration scripts (`gf.config` for Mac and Linux or `setEnv.bat` for Windows). The main thing you need to ensure is that you've properly set the JAVA_HOME location. In addition, verify that COURSE_HOME is correctly set. By default we assume that course home is under your home directory. Your file will possibly look like one of the following:

```
# JAVA_HOME may already be set
#export JAVA_HOME=

export COURSE_HOME=$HOME/GemFire-Developer-9.0.RELEASE

export GEMFIRE=$COURSE_HOME/pivotal-gemfire-9.0.1
export PATH=$PATH:$GEMFIRE/bin

export REPO=$COURSE_HOME/repository
```

Figure 1.2. gf.config Example

```
rem "JAVA_HOME must not have spaces in it"
set JAVA_HOME=C:\Java_64bit\jdk1.8.0

set COURSE_HOME=C:\GemFire-Developer-9.0.RELEASE

set GEMFIRE=%COURSE_HOME%\pivotal-gemfire-9.0.1
set PATH=%PATH%;%GEMFIRE%\bin;%JAVA_HOME%\bin

set REPO=%COURSE_HOME%\repository
```

Figure 1.3. setEnv.bat example

2. Open terminal window or command window and change directories to `GemFire-Developer-9.0.RELEASE/server_bootstrap/scripts` folder.
   Run the startServer.sh script or startServer.bat script to start the server processes and load the cache server with data that will be used for the client-side labs. You should see output similar to the following.

```
Got the BookMaster Region: org.apache.geode.internal.cache.LocalRegion[path='/BookMaster';scope=LOCAL'
;dataPolicy=EMPTY; concurrencyChecksEnabled]
Inserted a book: BookMaster [itemNumber=123, title=A Treatise of Treatises]
Inserted a book: BookMaster [itemNumber=456, title=Clifford the Big Red Dog]
Inserted a book: BookMaster [itemNumber=789, title=Operating Systems: An Introduction]
Got the Customer Region: org.apache.geode.internal.cache.LocalRegion[path='/Customer';
scope=LOCAL';dataPolicy=EMPTY; concurrencyChecksEnabled]
Inserted a customer: Customer [customerNumber=5598, firstName=Kari, lastName=Powell, postalCode=44444]
Inserted a customer: Customer [customerNumber=5543, firstName=Lula, lastName=Wax, postalCode=12345]
Inserted a customer: Customer [customerNumber=6024, firstName=Trenton, lastName=Garcia, postalCode=88888]
```

Figure 1.4. Output from running startServer.sh or startServer.bat

Note

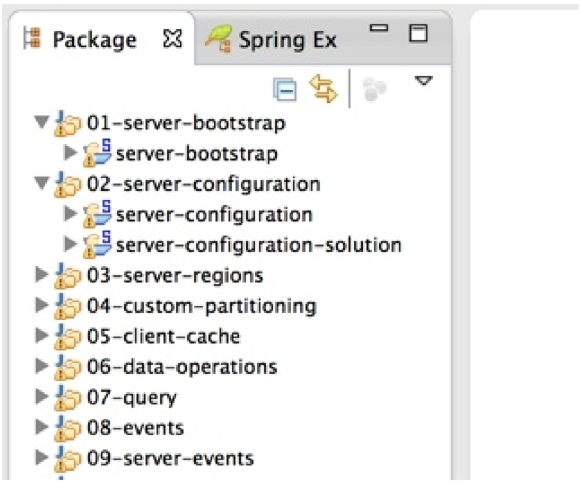
You can also stop the server processes by running stopServer.sh on Linux and Mac or stopServer.bat on Windows

1.2.4. Becoming familiar with the STS

In this section, you will start the STS, which is where most of the labs will be performed.

Begin by locating the STS icon inside the course folder.

When running the STS for the first time, the tooling will perform some initial setup that will result in a number of projects being added and organized into working sets.



```
▼ 01-server-bootstrap
    ▶ server-bootstrap
▼ 02-server-configuration
    ▶ server-configuration
    ▶ server-configuration-solution
▶ 03-server-regions
▶ 04-custom-partitioning
▶ 05-client-cache
▶ 06-data-operations
▶ 07-query
▶ 08-events
▶ 09-server-events
```

Figure 1.5. STS project structure

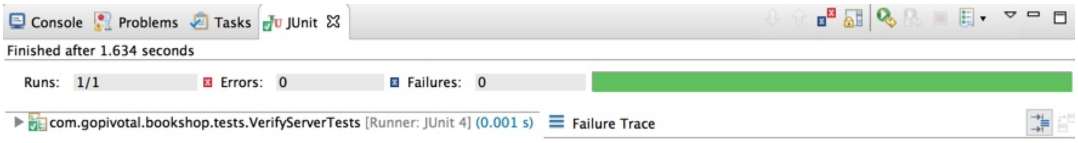
Expand the `01-server-bootstrap` working set and then the `server-bootstrap` project. Expand the `src/test/java` folder and then the `io.pivotal.training` package. Locate the `VerifyServerTests.java` class file and execute as a JUnit test.

Tip

To run this program, you can select the file, right-mouse click and select `Run As -> JUnit`

`Test.`

This test will verify that you have correctly configured the server by running a client test that accesses the distributed system, fetching an entry from the server. If you've configured everything correctly, you should see the following output as a result.



```
Finished after 1.634 seconds
Runs: 1/1   Errors: 0   Failures: 0
com.gopivotal.bookshop.tests.VerifyServerTests [Runner: JUnit 4] (0.001 s)
```

Congratulations!! You have completed this lab.

Chapter 2. Server Configuration

2.1. Introduction

In this lab, you will gain hands-on experience with the basic configuration of a GemFire server cluster.

Concepts you will gain experience with:

- Configuring and starting a locator for member discovery
- Configuring and starting GemFire cache servers
- Loading data into the GemFire distributed system

Estimated completion time: 30 minutes

2.2. Quick Instructions

This lab won't make use of Quick Instructions.

2.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks.

Important

Most of the insructions that will be provided in this lab presume that you have certain environment variables defined (for example `COURSE_HOME`, etc). Since you previously set these variables up in either the `gf.config` or `setEnv.bat` file in the `LAB_HOME/server-bootstrap/scripts` folder, the easiest thing to do is either source the `gf.config` file (**source $LAB_HOME/server-bootstrap/scripts/gf.config**) or run the `setEnv.bat` file.

2.3.1. Configuring GemFire properties file

Generally, locators and servers use the same distributed system properties file, which lists locators as the membership coordinator.

1. Browse the `server-configuration` project folder using either a terminal/command window, file explorer or from within the STS.
2. Open the `gemfire.properties` file found in this folder and specfy the following properties.
   - Set the `locators` property to have port `41111`
   - Set the `mcast-port` property to be `0`, which means Multi-cast won't be used in this cluster configuration. Set the `cache-xml-file` property to point to the server cache xml file. Note that this will be a relative path to where the servers run. As a result, you will probably want a path similar to `../xml/serverCache.xml`.
3. Save and close the file

2.3.2. Starting the Locator

You will use the **gfsh** console to start the locator (and later the servers).

1. Either open a command window or use the one you already have. Make sure you are in the `server-configuration` lab folder.
2. Start gfsh

Note

Make sure you have properly set the PATH variable as outlined at the beginning of this lab.

3. At the prompt, enter the command to start the locator. At a minimum, you should specify the name of the locator, the port, properties file and min and max memory.

Table 2.1. Locator start options

Option	Purpose
--port=<port>	The locator will listen on the port (for example, 41111) for requests from servers and clients
--name=locator	The name of the locator. If the `--dir` option is not used, this is also treated as the directory where the locator will run and log files will be written.

Option	Purpose
--properties-file=./gemfire.properties	The properties file, relative to gfsh working directory, where gemfire.properties will be found.
--initial-heap=50m	Sets the min JVM size to 50m. This should be set to the same as the max to avoid garbage collection related performance issues
--max-heap=50m	The maximum JVM size

Since we are starting multiple processes on a single machine, we are tuning memory to be much smaller than a typical locator instance to conserve memory resources.
Observe that you see the following output.

```
Starting a GemFire Locator in /Applications/GemFire-Developer-9.0.2a.RELEASE
    /GemFire-Developer-9.0.2a.RELEASE/server-configuration-solution/locator1...
.................................
Locator in /Applications/GemFire-Developer-9.0.2a.RELEASE/GemFire-Developer-9.0.2a.RELEASE/
server-configuration-solution/locator1 on 192.168.0.60[41111] as locator1 is currently online.
Process ID: 55785
Uptime: 19 seconds
GemFire Version: 8.0.0
Java Version: 1.6.0_65
Log File: /Applications/GemFire-Developer-9.0.2a.RELEASE/GemFire-Developer-9.0.2a.RELEASE/
server-configuration-solution/locator1/locator1.log
JVM Arguments: -Xserver -DgemfirePropertyFile=/Applications/GemFire-Developer-9.0.2a.RELEASE
/GemFire-Developer-9.0.2a.RELEASE/server-configuration-solution/gemfire.properties -Dgemfire
.enable-cluster-configuration=true -Dgemfire.load-cluster-configuration-from-dir=false
-Xms50m -Xmx50m -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=60 -Dgemfire
.launcher.registerSignalHandlers=true -Djava.awt.headless=true -Dsun.rmi.dgc.server
.gcInterval=9223372036854775806
Class-Path: /Applications/GemFire-Developer-9.0.2a.RELEASE/Pivotal_GemFire_800_b48319_Linux
/lib/locator-dependencies.jar

Successfully connected to: [host=192.168.0.60, port=1099]

Cluster configuration service is up and running.
```

2.3.3. Starting the Servers

In this step, you will start two cacheserver instances using the locator for discovery. Execute the following steps to complete this task.

1. Start the two servers using the same `gfsh` console using the **start server** command. Make sure you specify the configuration file for starting each server with the locators information. Use a different server name for each of the server instances started.
   These properties can either be set as arguments to the start server command or can be located in the `gemfire.properties` file. Recall that you set both the `locators` location and the `cache-xml-file` location in the `gemfire.properties` file.

2. For the purpose of these labs, you can specify a `server-port` option of `0`, which will cause the port to be auto-assigned.

3. When complete, you should see output similar to the following.

```
Starting a GemFire Server in /Applications/GemFire-Developer-9.0.2a.RELEASE
     /GemFire-Developer-9.0.2a.RELEASE/server-configuration-solution/server1...
.............
Server in /Applications/GemFire-Developer-9.0.2a.RELEASE/GemFire-Developer-9.0.2a.RELEASE/server-configuration
-solution/server1 on 192.168.0.60[59323] as server1 is currently online.
Process ID: 56108
Uptime: 2 seconds
GemFire Version: 8.0.0
Java Version: 1.6.0_65
Log File: /Applications/GemFire-Developer-9.0.2a.RELEASE/GemFire-Developer-9.0.2a.RELEASE/server-configuration-
solution/server1/server1.log
JVM Arguments: -Xserver -Dgemfire.default.locators=192.168.0.60[41111] -DgemfirePropertyFile=/Applications/
GemFire-Developer-9.0.2a.RELEASE/GemFire-Developer-9.0.2a.RELEASE/server-configuration-solution/gemfire.
properties -Dgemfire.use-cluster-configuration=true -XX:OnOutOfMemoryError="kill -9 %p" -Xms50m -Xmx50m
-XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=60 -Dgemfire.launcher.registerSignalHandlers=true
-Djava.awt.headless=true -Dsun.rmi.dgc.server.gcInterval=9223372036854775806
Class-Path: /Applications/GemFire-Developer-9.0.2a.RELEASE/Pivotal_GemFire_800_b48319_Linux/lib/server
-dependencies.jar
```

4. If you see any errors in the output, refer to the specified log file for details on the failure.

5. Verify your locator and servers are running by issuing the following command.

```
gfsh> list members
  Name   | Id
-------- | ----------------------------------------------------------
locator1 | MachineName(locator1:55785:locator)<v0>:26565
server1  | MachineName(server1:56108)<v4>:36765
server2  | MachineName(server2:56140)<v5>:52200
```

2.3.4. Running the client

In this final section, you will run a client application that will contact your distributed system via the locator and insert several `Customer` objects.

1. To begin, locate and open the `clientCache.xml` file and observe the configuration of the client cache pool. As you learned in earlier labs, this is the means by which the client application is able to contact the distributed system.
   Also, notice that there is one region configured for the Customer region and is configured to act as a proxy region.

2. Next, locate and run the `TestClient` class in the `io.pivotal.bookshop.buslogic` package. You can do this by right-mouse clicking on the file and select `Run As -> Java Application`. You should see the following output.

```
Customer Region = /Customer
Inserted a customer: Customer [customerNumber=5598, firstName=Kari, lastName=Powell, postalCode=44444]
Inserted a customer: Customer [customerNumber=5543, firstName=Lula, lastName=Wax, postalCode=12345]
Inserted a customer: Customer [customerNumber=6024, firstName=Trenton, lastName=Garcia, postalCode=88888]
cache is: GemFireCache[id = 1624207630; isClosing = false; isShutDownAll = false; ...
```

3. As a final step, shut down the servers and locator in prepratation for the next lab. You can do this from gfsh. If you previously exited from gfsh, you can re-connect and stop all services using the following command.

```
gfsh> connect --locator=localhost[41111]
Connecting to Locator at [host=localhost, port=41111] ..
Connecting to Manager at [host=192.168.0.60, port=1099] ..
Successfully connected to: [host=192.168.0.60, port=1099]

gfsh> shutdown --include-locators=true
As a lot of data in memory will be lost, including possibly events in queues, do you really want to
shutdown the entire distributed system? (Y/n): Y

No longer connected to 192.168.0.60[1099].
```

Congratulations!! You have completed this lab.

Chapter 3. Server Regions - Replicated and Partitioned

3.1. Introduction

This lab will take you through the process of creating both replicated regions and partitioned regions in GemFire. An XML file will be used for the configuration, which is generally a best practice.

Concepts you will gain experience with:

- How to create a replicated region across multiple servers
- Testing the failover and refresh of data on recovered nodes
- Configuring partitioned regions in GemFire

Estimated completion time: 45 minutes

3.2. Quick Instructions

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

3.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks.

If servers and locator are still running from the previous lab, be sure to stop them at this point. Remember, in order to shut the services down, you'll need to re-connect to the locator using the command **connect --locator=localhost[41111]**.

3.3.1. Creating Replication Regions

In this first section, you will work with the BookMaster region. The data that this region holds is a relatively small amount, is slow changing, and tends to be reference data. This is the type of data that is a good candidate for replication. We will walk through defining a region as replicated, watching shutdown of nodes, and the redundancy.

1. Browse to the server-regions project folder on the command line. This will be your starting point for this lab.
2. Use the STS to open the `serverCache.xml` file in the server-regions project folder. Add a region attribute to define the `BookMaster` region as a replicated region.
3. If the locator is not running, start it using `gfsh` and the command similar to the one used in the prior lab.
4. Start `server1` using a similar command as used in the prior lab. Do NOT start the second server yet.

Note

As of GemFire 8, the server startup command from within gfsh has changed and does not automatically include everything in the current CLASSPATH environment variable in the classpath of the servers when started. Most of this lab makes use of a special function that is registered with the server that must be on the server classpath. As a result, you will need to explicitly include the option: **--classpath=../target/classes/** to the server start commands.

5. In the STS, locate and run the `BookLoader` class to load 3 books into the `BookMaster` region.
6. Open the `ReplicationTest` class to understand how it looks for values in the `BookMaster` region.
7. Run the `ReplicationTest` to verify that the books were found.
8. Start the second server using the name `server2` so it will write log data to a different directory.
9. You can examine the region details by executing the gfsh command **show metrics --region=/BookMaster**. Note the member count and the number of entries for the cluster.
10. Stop server1 using the gfsh **stop server** command.
11. Re-run the `ReplicationTest` application to verify that the data still can be found in the remaining server (the new server 2 that was started).
12. Stop all the servers in preparation for the next section.

3.3.2. Creating Partitioned Regions

In this section, you will use the Customer region and try out different partitioning scenarios. You will be using three server instances this time so you can see the benefits of partitioning with redundancy.

1. Return to the `serverCache.xml` file and modify the `Customer` region to set the region type to partitioned.

2. If you stopped the locator along with the servers in the prior section, re-start the locator.

3. Start three servers, calling them `server1`, `server2`, and `server3`.

   Note

   If you specify the `--server-port=0` argument on each of these, the ports will be auto-assigned and registered with the locator.

4. Run the `CustomerLoader` class to load 3 customers into the distributed system.

5. You can observe the partitioning by issuing the following command.

```
gfsh> show metrics --region=/Customer --member=server1 --categories=partition
```

   Note the reported values for `bucketCount`, `primaryBucketCount` and `configuredRedundancy`. You might try this for all three servers.

6. Now, stop all the servers (but not the locator).

3.3.3. Partitioned Regions with Redundancy

In the prior partitioned region configuration, if one of the servers stops for some reason, all the data stored in that partition is lost. In this section, we'll address that by adding a redundancy factor.

1. Go back to the serverCache.xml file and modify the Customer region attributes to add a redundancy of 1 (meaning there will be one primary and one redundant copy of every entry).

   Tip

   You can either do this by modifying the region shortcut or by inserting a `partition-attributes` element and specifying this. However, in a later step, you'll add a recovery delay value so you may want to take the extra time to type in the `partition-attributes` element now.

2. Save the file and re-start the servers. Re-run the `CustomerLoader` class to re-load the customers.

3. Repeat the show metrics command to see what has changed with the updated partitioned region configuration.

4. Now, stop `server3` and repeat the **show metrics** command for the remaining two servers. You'll notice that

the `primaryBucketCount` value will have increased from 1 to 2 indicating that one of the redundant copies was promoted. Notice also that `numBucketsWithoutRedundancy` is not 0. This indicates that when the server was lost, the redundant bucket was promoted redundancy was not re-established for this or any redundant buckets that were on that server.

5. You can obtain even more detail using the special `PRBFunction` that has been registered with this project. To make use of this functionality, run the `PRBFunctionExecutor` program found under the `io.pivotal.training.prb` package in the STS. You'll see that a very extensive output is printed that displays every primary bucket and every redundant bucket for each server. You can see by which ones have a size > 0 the buckets that have entries. You should see output similar to the following for every server.

```
Member: HostMachine(server2:77234)<v2>:58224
  Primary buckets:
    Row=1, BucketId=2, Bytes=0, Size=0
    Row=2, BucketId=4, Bytes=0, Size=0
    Row=3, BucketId=9, Bytes=0, Size=0
    Row=4, BucketId=12, Bytes=0, Size=0
    Row=5, BucketId=13, Bytes=0, Size=0
             ....
    Row=20, BucketId=60, Bytes=0, Size=0
    Row=21, BucketId=61, Bytes=676, Size=1
```

6. Stop the servers once again

Congratulations!! You have completed this lab.

3.3.4. Partitioned Regions with Redundancy and Recovery Delay

This time, you will add a recovery delay so that after a period of time, redundancy will be re-established. This will address the issue identified in the prior section.

1. Go back to the `serverCache.xml` file and modify the partition-attributes element to define a recovery delay of 5 seconds.

Tip

If you used a region shortcut in the prior section, you'll need to add a partition-attributes element inside the region-attributes element for the `Customer` region

2. Save the file and re-start all the servers. Re-run the `CustomerLoader` class to re-load the customers.

3. Now, stop `server3` and repeat the **show metrics** command for the remaining two servers. If you run this command within 5 seconds of stopping `server3`, you'll likely see the `numBucketsWithoutRedundancy` is still not 0. Wait a few more seconds and repeat the command. You should see that this value will be returned to 0. This indicates that redundancy has been re-established within the remaining servers.

4. Alternatively, you can use the `PRBFunctionExecutor` class to print out more detailed bucket listing as outlined in the prior section.

5. Stop the servers for the final time. Also stop the locator.

Congratulations!! You have completed this lab.

Chapter 4. gfsh practice

4.1. Introduction

This lab is intended to increase your familiarity with `gfsh`, the gemfire shell. Unlike other labs, this lab has no code component: all tasks will be performed inside the shell.

In this lab, you will learn:

- how to launch and exit the shell
- starting and stopping locators, servers
- creating a region, inserting data, querying regions
- obtain information about cluster members, regions

Note

Full documentation for gfsh is available at: http://gemfire.docs.pivotal.io/docs-gemfire/latest/tools_modules/gfsh/chapter_overview.html

Estimated completion time: 20 minutes

4.2. Overview

This session is intended to be a brief tour of working within `gfsh`. Among other things, you will be starting a locator, servers, a region, inserting and querying data, stop and restart a server, and validating the expected behavior of the system. Let's get started.

4.3. Instructions

4.3.1. Launching and exiting gfsh

Gemfire bundles the command line tool `gfsh`. You will find it located in `$GEMFIRE/bin`, which should be in your `$PATH`, so that, launching the shell is simply a matter of invoking the command: **gfsh**.

To quit gfsh, simply type **exit**.

4.3.2. Stopping existing servers and locators

Go ahead and launch gfsh, and make sure that prior running servers and locators are stopped. To do that, perform the following:

- connect to the running locator with **connect --locator=localhost[41111]** (if you get a message saying you could not connect, it's likely you have no locator running, in which case proceed to the next section).
- invoke the command **shutdown** to stop the servers,
- stop the locator with the command **stop locator** (you'll have to supply the name of the locator you wish to stop (it should simply be 'locator')).

4.3.3. Built-in help

`gfsh` comes with built-in help at multiple levels.

In its most general form, simply invoking **help** will list all available commands.

Notice that many of the commands begin with the same verb, such as `start`, `stop`, `status`, and `list`. For example, to learn more about the start command, type **help start**.

To obtain feedback on a specific command, such as "start server," type **help start server**. Notice the output will provide information about all options that can be supplied along with this command. The list of options for starting a server is rather lengthy.

4.3.4. Start a locator

Let's begin by starting a locator. Try to type just **start locator** and see what happens. What feedback do you get?

Note

If no port number is specified, the locator will start on its default port number, 10334

4.3.5. Start a server

Start a server, make sure to give it the name "server1."

Tip

`gfsh` sports command completion. Many commands, such as **start server** come with many options. As you construct your command, use the `tab` key to obtain a list of options to append to your command.

In prior labs, we usually referenced a gemfire `properties-file`. In this case, do not supply this option as we have no need to reference, for example, the location of a server cache.xml file.

Also, rather than manage the specific port our server will listen on, let gemfire select the port number. To do this, make sure to supply the option **server-port** with a value of 0.

4.3.6. Validate running locator and server

In gfsh parlance, the locator and server we just started are known as members. To verify that we have one of each running, invoke the command **list members**. Inspect the output.

To gain more information about these two running members, explore the **describe member** command. Use the built-in help to help you figure out how to invoke the command for each your locator and server1.

Inspect the output. For the server, can you tell whether it's running, what the running port is, and the locator it's associated with?

4.3.7. Regions

Let's create a replicated region named "people." Again use the help command to assist you in figuring out the syntax. You'll need to use the **type** option to specify that the type of region you want to create is a replicated region (the value you supply here is case sensitive).

Just like we can list members, we can also list regions. Go ahead and use the **list regions** command to verify that the region named "people" was properly created.

We can also describe regions. Run the **describe region** command on the people region and inspect the output. Verify the region type.

Our region is empty for now. Let's verify this by querying it. Use the **query** command for this. If you haven't yet gone over the lab about OQL and the gemfire query language, you can use the following simple query: **select * from /people**.

4.3.8. Inserting data

Insert a simple record into the people region. Supply a key and a value and make sure you specify what region you want to "put" (hint) this record into.

Run your query one more time to verify that the record was inserted.

Insert a second record into the people region.

4.3.9. Start a second server

Start a second server named "server2." Once more, let gemfire select what port the server should listen on.

Run **list members** once more to verify that we now have two running servers.

Run the query once more against the people region to verify that it still functions properly.

Describe the people region once more. Notice that both server1 and server2 are listed as "hosting members." This is a validation that the data is being replicated across both servers.

4.3.10. Stop server1

Use the **stop server** command to stop server1. Inspect the output of **list members** to validate that server1 is no longer running. Run the query once more. Notice that it is still working because we had redundancy/replication.

Start server1 back up.

Explore the **status server** command on either of the running servers. What additional information can we glean about the server compared to the **describe member** command?

4.3.11. Exit, then return to the shell

At this point, issue the **exit** command to exit the shell. Now re-invoke gfsh and re-connect to your locator (port number should be 10334). Notice that your servers should still be running. So, exiting merely detaches you from the shell but has no impact on the running servers.

4.3.12. Shutting down

Issue the command **shutdown** to stop your servers. Using the built-in help, issue the **stop locator** command to stop your locator.

In this session, we've covered a lot of ground. Although we haven't explored the myriad of options that many of the commands sport, you should now be familiar and comfortable enough with gfsh to explore further. Congratulations!! You have completed this lab.

Chapter 5. Client Cache

5.1. Introduction

In this lab, you will gain hands-on experience working with a GemFire client cache configuration.

Concepts you will gain experience with

- Configuring a client cache XML file
- Defining regions both for local private use as well as connecting to cache server
- Configuring cache properties using gemfire.properties
- Initializing the client cache in an application

Estimated completion time: 45 minutes

5.2. Quick Instructions

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

5.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks. Before beginning this lab, make sure you have started the server side processes using the `startServer.sh` script (`startServer.bat` for Windows) in the `server-bootstrap` lab folder.

5.3.1. Configuring the Cache Client XML file

The client's cache.xml <client-cache> declaration automatically configures it as a standalone Gemfire application. At this point, the locator and server will have already been started and the listener is listening on port `41111`.

In this section, you will edit the `xml/clientCache.xml` file to perform the following tasks.

1. (`TODO-01`) Define a pool configuration with one entry configured to contact the locator at `localhost` and listening on port `41111`
2. (`TODO-02`) Define a region called `Customer` as a proxy type region using either the refid (region shortcut) or explicitly using region attributes. Similarly, create a region called `BookMaster` but this time create it as a caching proxy type region.

5.3.2. Configuring additional cache properties

In addition to providing configuration information in the cache xml file, you can also define properties using a gemfire.properties file. In this section, you will use this file to define the logging level that will be used by the client application. Some of the possible configurations are found in the table following. Consult the GemFire User Guide appendix for a full list of properties.

Table 5.1. Client Cache properties

Property	Meaning
log-level	Level of detail of the messages written to the system member's log. Setting log-level to one of the ordered levels causes all messages of that level and greater severity to be printed. Valid values from lowest to highest are fine, config, info, warning, error, severe, and none. Default = config
durable-client-id	Used only for clients in a client/server installation. If set, this indicates that the client is durable and identifies the client. The ID is used by servers to reestablish any messaging that was interrupted by client downtime. Default = none
durable-client-timeout	Used only for clients in a client/server installation. Number of seconds this client can remain disconnected from its server and have the server continue to accumulate durable events for it. Default = 300

(`TODO-03`) Open the `gemfire.properties` file and set the log level initially to `warning`.

5.3.3. Configure client application to initialize client cache

Open the `ClientCacheTests.java` file in the `io.pivotal.bookshop.tests` package (`under src/test/java`) and add code for the following methods.

- (`TODO-04`) In the `setup()` method to, add code to create a `ClientCache`. Use the appropriate property on the `ClientCacheFactory` class to define the `cache-xml-file` property to point to `xml/clientCache.xml`.
- (`TODO-05`) Also, add code to the same `setup()` method to get the `Customer` region and assign to the class private field `customers`. Add a similar statement to get the `BookMaster` region and assign to the private field `books`.

Note

You will not edit the test code itself, which performs the get operations. Details on how to make such calls to the region will be covered in the next section.

- (`TODO-06`) Finally, run the program to verify you were able to connect. If you implemented the client cache configuration and the client setup code correctly, one test should pass and one test is currently being ignored (and will be addressed in the next section of this lab).
- As an optional step, go back and edit the `gemfire.properties` file and change the log level to `fine`. Re-run the application and notice the additional detal that is output when running your client application.

5.3.4. Define a local region

In this final section, you will define a local region and run a test to load and fetch data from it.

- (`TODO-07`) Open the `xml/clientCache.xml` file and add an additional region called `LocalRegion`. Use the region shortcuts to define this as a local region.
- (`TODO-08`) Return to the `ClientCacheTests` class and comment the `@Ignore` annotation on `testLocalRegion()`.
- (`TODO-09`) Finally re-run the tests and verify that all tests pass.

5.3.5. Creating a DAO object for BookMaster

To begin, open the `BookMasterDao.java` class in the io.pivotal.bookshop.buslogic package in the IDE. The purpose of this class is to hide the basic operations involved in creating, reading, updating and deleting entries (the CRUD operations).

Note that the region is defined to store `BookMaster` objects with an `Integer` key as illustrated by the private field defined at the top of the DAO. The constructor is responsible for initializing the region using the `ClientCache` that is passed in.

You will implement the basic functionality of this class in the following steps.

- (`TODO-10`) Before implementing the functionality in `BookMasterDao`, open the corresponding JUnit test class (`DataOperationsTests.java` in the io.pivotal.bookshop.tests package in `src/test/java`). Take a moment to inspect the various test methods to see how the DAO is used.
- (`TODO-11`) Implement the `insertBook()` method. Use the key and book object to insert into the region.

Tip

Recall that there are different methods that can be used to insert an entry. Use the method that enforces that the entry can't already exist.

- (`TODO-12`) Implement the `getBook()` method. Use the supplied key to retrieve the associated entry from the region and return it.
- (`TODO-13`) Implement the `updateBook()` method. Use the supplied key and `BookMaster` object to update the existing entry in the region.
- (`TODO-14`) Implement the `removeBook()` method. Use the supplied key to delete the entry from the region. Recall that there are two methods to do this. Either one is acceptable.
- (`TODO-15`) When you have completed all the steps above, run the test suite (`DataOperationsTests`). Make sure the GemFire server process (locator and server) are already running if you haven't done so already.

Congratulations!! You have completed this lab.

Chapter 6. GemFire OQL Query

6.1. Introduction

In this lab, we will be leveraging Object Query Language (OQL) to access data within GemFire. When you are accessing data, you don't always have the key available, nor do you know what the exact structure of the data is. In this lesson, we will be implementing methods that query GemFire data.

Concepts you will gain experience with:

- Using a query connection in GemFire
- Writing an OQL style query
- Performing a query for individual attributes of an object
- Writing queries involving joins between regions

Estimated completion time: 30 minutes

6.2. Quick Instructions

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

6.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks. Before beginning this lab, make sure you have started the server side processes using the `startServer.sh` script (`startServer.bat` for Windows) in the `server-bootstrap` lab folder.

6.3.1. Implementing the general query functionality

To start, you will implement a general purpose query method, `doQuery()` that simply takes a `String` as an argument and returns a `SelectResults` result set. While not robust enough for broad usage, it will suit the purpose for this lab to have this functionality encapsulated in a method such that the remaining sections can focus primarily on writing the query string.

(`TODO-01`) Open the `OQLInquirer.java` class file (in the `io.pivotal.bookshop.buslogic` package) and locate the `doQuery()` method, which has been stubbed out for you. Implement the functionality of this method to perform the following tasks:

- Get a `QueryService` from the cache
- Create a new query using the supplied query string
- Execute the query, which will have to be done inside a `try/catch` block
- Either return the results of executing the query or throw a `QueryException` wrapping an exceptions caught in the catch block

6.3.2. Basic OQL Query with Objects

This section will do a query against the `Customer` region to access data as `Customer` Objects.

- (`TODO-02`) In the same file, locate the `doCustomerQuery()` method. In this method, you will only need to do two things.
    1. Write a proper OQL query in the query string to fetch all entries from the `Customer` region
    2. Return the results of calling the `doQuery()` method with the query string you just wrote
- (`TODO-03`) Open the `OqlInquirerTests.java` class file in the `com.gopviotal.bookshop.tests` package. Execute the tests. If you correctly implemented the `doCustomerQuery()` method in the prior step, the first test, `testBasicQuery()` should pass.

Note

Don’t worry yet that the other two tests are failing. You will fix this in the remaining sections.

6.3.3. OQL Query with Struct objects

Once you have the basic customer query working, you can move on to write a similar query to only return certain fields of the Customer object.

- (`TODO-04`) Locate the doStructQuery() method in the `OQLInquirer` class. Implement the method using the following steps.

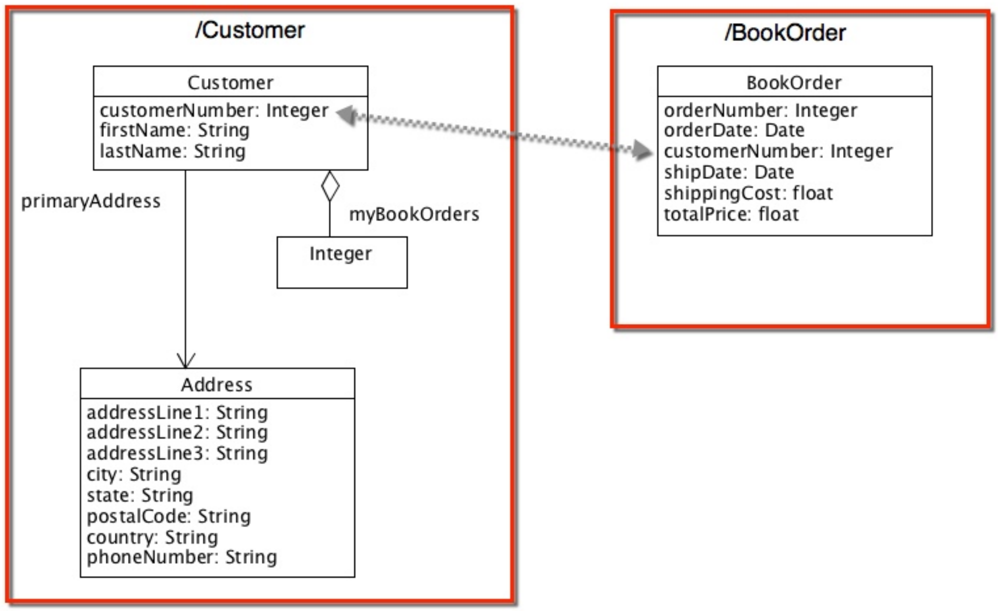
1. First, write the correct query string to return a projection list. That is, perform a query where you will return only the `customerNumber`, `firstName` and `lastName` fields of the `Customer` entries.

2. Return the results of calling `doQuery()` with the query string you just wrote

- (`TODO-05`) Return to the `OqlInquirerTests` class and re-run the tests to verify you correctly implemented this functionality. If so, then both the `testBasicQuery()` and `testStructQuery()` tests should now pass. The `testJoinQuery()` test will still fail until you complete the last section.

6.3.4. Performing Joins

In this final section, you will be creating a more complex query by performing a join operation between the Customer region and BookOrder region. The goal is to return a list of all customers who have placed an order with total price greater than $45.00. To better understand the requirements for this query, take a moment to examine the following diagram showing the class definitions found within these two regions.



You will be performing an equ-join query in which you link `Customer` entries in the `/Customer` region to the `BookOrder` entries in the `/BookOrder` region by the `customerNumber` property of each object. With this in mind, perform the following tasks.

- (`TODO-06`) Locate the `doJoinQuery()` method in the `OQLInquirer` class. Implement the method using the following steps.

  1. First write the correct query string to perform the join query. You are selecting a unique list of customers who's total order is greater than $45.00.

  2. Return the results of calling `doQuery()` with the query string - you should be returning a `SelectResults` for `Customer` objects.

- (`TODO-07`) Return to the `OqlInquirerTests` class and re-run the tests to verify you correctly implemented the join query functionality. If so, then all 3 tests should now pass.

Congratulations!! You have completed this lab.

Chapter 7. Custom Partitioned Regions

7.1. Introduction

In this lab, you will gain hands-on experience with creating a custom partition resolver.

Concepts you will gain experience with:

- Creating a custom partition scheme in GemFire
- Implementing a custom partition resolver

Estimated completion time: 30 minutes

7.2. Quick Instructions

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

7.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks.

7.3.1. Enabling Co-located regions

Recall that in order to ensure related data ends up on the same member for partitioned regions, one key step is causing the buckets to be aligned between related regions.

(`TODO-01`): Open the `serverCache.xml` file and add the appropriate configuration to the `BookOrder` region to ensure bucket assignments are aligned with the `Customer` region.

7.3.2. Custom Partitioning

To create a Custom Partitioning scheme for GemFire, we need to implement the `PartitionResolver` interface. We want to partition on `customerId`, which is a common field shared between `Customer` objects and

`BookOrder` objects. We will need to make this an attribute of the key as well as the `orderId`. We have been using `Integer` representing just the `orderId` as the key up until this point, so we will need to wrap both the `orderId` and `customerId` in a `OrderKey` class.

1. Open the OrderKey class

   a. (`TODO-02a`) Notice that it contains the key (orderNumber) and the customerNumber, which will be used for partitioning.

   b. (`TODO-02b`) Notice that the `hashCode()` and `equals()` methods are based on the `orderNumber` part of the key and not the `customerNumber`. The addition of the `customerNumber` is for the `PartitionResolver` to use, not to impact the 'equality' of entries.

2. Create the PartitionResolver

   a. (`TODO-03`) Open the class `CustomerPartitionResolver` and implement the `getRoutingObject()` method.

   b. Using the `EntryOperation`, return the part of the key that will be used for ensuring that `BookOrder` objects related to a given `Customer` land in the same bucket.

   c. Notice the `close()` and `getName()` methods. `The close()` method is a callback method required by the `CacheCallback` interface. In this case, there's nothing to do when the cache closes. Similarly, the `getName()` method is specified by the PartitionResolver interface and offers a way to attach a name to the resolver.

3. (`TODO-04`) Modify the server cache XML file to register the custom partition resolver

   a. Open `xml/serverCache.xml` file

   b. Notice that for the sake of this lab, we've reduced the number of buckets to `5`. That means any entries will land in one of these 5 buckets (0-4) depending on the key value used.

   c. Add the appropriate attributes to the `BookOrder` Region to have `io.pivotal.bookshop.domain.CustomerPartitionResolver` declared as the Partition Resolver.

   d. Start the locator

   e. Start server1 and verify the configuration. If the first server fails to start, it's likely due to incorrectly specifying the partition resolver. In addition, you'll need to make sure that the partition resolver is on the classpath.

   Tip

   As with the prior lab, you will need to explicitly include the option:

`--classpath=../target/classes/` to the server start commands. This is not only so the special function for listing partitioned region buckets is available but also because you are registering a custom PartitionResolver that must be on the server's classpath.

f. Once server 1 starts properly, start servers 2 and 3.

4. Run the System test

   a. (`TODO-05`) Open the `DataLoader` class under `io.pivotal.bookshop.buslogic` package and observe the `populateCustomers()` method and the `populateBookOrders()` method. Notice that for `cust1` with customer number `5598`, there is a corresponding book order (`17600`) that has been created as evidenced by `key1` in the `populateBookOrders()` method. Notice a similar case for `cust3`.

   b. (`TODO-06`) Run `DataLoader`. This will load `Customer` objects and `BookOrder` objects into their respective regions. If your PartitionResolver has been correctly implemented, `Customer` objects and related `BookOrder` objects should end up on the same member because they will each use the same bucket number for storage.

   c. (`TODO-07`) Verify the distribution by using the `PRBFunctionExecutor` (under the `io.pivotal.training.prb` package) to list the bucket distribution.

   d. After exectution completes, look at the console output and observe two things:

      i. The bucket numbers for the `Customer` region and `BookOrder` regions are on the same member. If they are not, go back and double check your `serverCache.xml` configuration and ensure that you've correctly specified co-location.

      ii. That the data appears be aligned as evidenced by the fact that there is one `Customer` entry in each bucket and that there appears to be one `BookOrder` entry for bucket `3` and `4`. Refer to the table below to see how the keys align to buckets. If you DON'T see the described alignment, go back and double check your `PartitionResolver` implementation.

Table 7.1. Keys aligning with bucket numbers (assuming total buckets = 5)

Key (customerNumber)	Bucket Number
5598	3
5542	2
6024	4

e. Return to gfsh (re-connect to the locator if necessary) and stop all servers and the locator using the **shutdown** command.

Congratulations!! You have completed this lab.

Chapter 8. Cache Management

8.1. Introduction

In this lab, you will learn to configure cache expiration and cache eviction.

Concepts you will gain experience with:

- Configuring cache expiration
- Configuring cache eviction

Estimated completion time: 30 minutes

8.2. Quick Instructions

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

8.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks.

8.3.1. Configuring Cache Expiration

Expiration ensures that your data is current and pertinent by removing old and unused entries from the GemFire. It can also give you some control over Region size. When you try to access an expired entry, GemFire looks to another source for a fresh update. If the data does not exist in another member, the cache loader you have installed on the data region is called.

This exercise uses idle time expiration, configured through the XML file to destroy expired entries. The program creates four entries and then waits beyond the expiration time. During the wait time, the program updates one entry and accesses another. When the expiration time elapses, the third and fourth entries (which are not accessed and not modified) expire. The cache listener installed on the region reports all changes to the entries.

8.3.1.1. Configure the XML

Configure and add the corresponding region attributes in `xml/serverCache.xml` to:

1. (`TODO-01`) Configure the `Customer` region with entry idle time expiration. Set the timeout to `10 seconds`, and `destroy` expiration action.

2. (`TODO-02`) Configure the `ServerCacheListener` cache listener to report changes to the cached data.

8.3.1.2. Configure the Application

Open the `SampleDataExpiration` class skeleton within `io.pivotal.bookshop.buslogic` package, and add the code for the following behavior:

1. (`TODO-03`) Add the code in the `getExpirationTime()` method to return the entry idle timeout setting from the region attributes (`serverCache.xml` file).

   Tip

   You can access these configured values by invoking the appropriate `get()` method on the region.

2. (`TODO-04`) Add the code in the `updateCustomer()` method to update the customer details of the key passed in. Make changes to any information, except the key value and the customer ID. A good option is to add a new address.

3. (`TODO-05`) Review the `retrieveAndPrintRegionData()` method and observe its purpose is to iterate over the entries and print out the details of each.

4. (`TODO-06`) Add the code in the `run()` method, in the following sequence:

   a. Retrieve and populate the cache with the customer's data.

   b. Show the contents of the cache (calling the `retrieveAndPrintRegionData()` method).

   c. Get the idle timeout value, and use `Thread.sleep()` method to suspend the thread execution for (timeout - 1) seconds (remember the `sleep()` method assumes time is in milliseconds).

   d. Perform a `get()` on the Customer region to get one customer, and update the details of the second customer by calling `updateCustomer()`.

   e. Use the `Thread.sleep()` method to sleep past expiration timeout.

f. Show the contents of the cache past expiration timeout.

8.3.1.3. Executing the Application

Execute the application in STS or command prompt window, and observe that after the expiration timeout, the cache contains only data that are either accessed or modified. In this case, we are running in an embedded model for simplicity. Notice that because you retrieved entry 5540 and updated entry 5541, they remain in the cache after expiration.

```
    Received afterCreate event for entry: 5540, Customer [customerNumber=5540, firstName=Lula, lastName=Wax]
    Received afterCreate event for entry: 5541, Customer [customerNumber=5541, firstName=Tom, lastName=Mcginns]
    Received afterCreate event for entry: 5542, Customer [customerNumber=5542, firstName=Peter, lastName=Fernendez]
    Received afterCreate event for entry: 5543, Customer [customerNumber=5543, firstName=Jenny, lastName=Tsai]
************ Server 1 : Loaded customers data ****************

    5540 => Customer [customerNumber=5540, firstName=Lula, lastName=Wax]
    5541 => Customer [customerNumber=5541, firstName=Tom, lastName=Mcginns]
    5542 => Customer [customerNumber=5542, firstName=Peter, lastName=Fernendez]
    5543 => Customer [customerNumber=5543, firstName=Jenny, lastName=Tsai]
Before the idle time expiration, access and update one entry each...

Accessed customer, Lula  data !!
    Received afterUpdate event for entry: 5541, Customer [customerNumber=5541, firstName=Tom, lastName=Mcginns]
************ Updated customer data ****************

    Received afterDestroy event for entry: 5543
    Received afterDestroy event for entry: 5542
After the expiration timeout, the cache contains:

    5540 => Customer [customerNumber=5540, firstName=Lula, lastName=Wax]
    5541 => Customer [customerNumber=5541, firstName=Tom, lastName=Mcginns]
Closing the cache and disconnecting.
```

8.3.2. Configuring Cache Eviction

You can use eviction to control how much heap your data regions use. Eviction controls your data region size by removing least recently used (LRU) entries to make way for new data. It kicks in when your data region reaches a specified size or entry count.

In this exercise, the data region is configured to keep the entry count at `4` or below by destroying LRU entries. A cache listener installed on the region reports the changes to the region entries.

8.3.2.1. Configuring the XML

(`TODO-07`) Configure and add the corresponding region attributes in the `serverCache.xml` to destroy entries when the region reaches entry count of `4` . You should put this just after the cache listener entry in the file.

8.3.2.2. Running the Application

Open the `SimpleDataEviction` class within the `io.pivotal.bookshop.buslogic` package. Observe that the code is written to first insert 4 Customer entries, followed by a fifth entry.

Run this program and observe the output. You should see something like the following:

```
    Received afterCreate event for entry: 5540, Customer [customerNumber=5540, firstName=Lula, lastName=Wax]
    Received afterCreate event for entry: 5541, Customer [customerNumber=5541, firstName=Tom, lastName=Mcginns]
    Received afterCreate event for entry: 5542, Customer [customerNumber=5542, firstName=Peter, lastName=Fernendez]
    Received afterCreate event for entry: 5543, Customer [customerNumber=5543, firstName=Jenny, lastName=Tsai]
************ Loaded customers data ****************
    5540 => Customer [customerNumber=5540, firstName=Lula, lastName=Wax]
    5541 => Customer [customerNumber=5541, firstName=Tom, lastName=Mcginns]
    5542 => Customer [customerNumber=5542, firstName=Peter, lastName=Fernendez]
    5543 => Customer [customerNumber=5543, firstName=Jenny, lastName=Tsai]
************ Loaded customers data ****************
    Received afterCreate event for entry: 5545, Customer [customerNumber=5545, firstName=Fifth, lastName=Customer]
    Received afterDestroy event for entry: 5540
************ Inserted one more customer data ****************
    5541 => Customer [customerNumber=5541, firstName=Tom, lastName=Mcginns]
    5542 => Customer [customerNumber=5542, firstName=Peter, lastName=Fernendez]
    5543 => Customer [customerNumber=5543, firstName=Jenny, lastName=Tsai]
    5545 => Customer [customerNumber=5545, firstName=Fifth, lastName=Customer]
```

Notice that in order to insert the fifth entry, it had to evict one of the other entries. The method it used was to locate the least recently used (the first entry) and evict it. Observe the above output and note that when entry `5545` was created that entry `5540` was destroyed.

Congratulations!! You have completed this lab.

Chapter 9. Server Side Event Handling

9.1. Introduction

In an earlier lab, you became familiar with client side even handling. In this lab, you will learn to configure events on the server side of a GemFire distributed system.

Concepts you will gain experience with:

- Creating and registering a CacheLoader
- Creating and registering a CacheWriter

Estimated completion time: 30 minutes

9.2. Quick Instructions

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

9.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks. Before beginning any of these tasks.

9.3.1. Preparation

In performing the steps of this lab, you will take a bit of a Test Driven Development approach. You will first run the series of tests, find all (or most) fail and then set about creating solutions to cause them to pass.

1. To begin, navigate to the `server-events` project in the STS. Locate the `ServerEventsTests` class in `src/test/java` under the `io.pivotal.bookshop.tests` package. Open it up and take a look at the various tests.
2. Next start the locator and one server. The easiest way to do this is to open a terminal or command window and change directories to this lab folder (`server-events`). Then issue the following command.

```
gfsh run --file=serverStart.gf
```

Note

If gfsh fails to run, it's likely because you don't have your environment variables set. Recall that these were defined in one of the first labs in the `server-bootstrap/scripts` folder. Locate that file and either run `setEnv.bat` for Windows or **source gf.config** for Mac/Linux. Once done, go back and re-try the above gfsh command.

3. Finally, run the tests (right mouse click on `ServerEventsTest -> Run As - JUnit Test`). You will see that 2 of the 3 tests currently fail.

9.3.2. Creating and Implementing a CacheLoader

The objective of the cache loader is to load data on cache misses on the GemFire server. `The load()` method is called when the `region.get()` operation can't find the value in the cache. The value returned from the cache loader is put into the cache, and returned to the `get()` operation.

1. The definition of the `BookMasterCacheLoader` class is provided in the `io.pivotal.bookshop.buslogic` package. Open this class file and add the code in the `load()` method to return a new `BookMaster` instance. It can either be explicitly generated in this method or by a helper class that you create. Ordinarily this would be loaded from an external data source such as a JDBC data source.

2. Open the `serverCache.xml` file and add the necessary configuration to register the `BookMasterCacheLoader` with the `BookMaster` region.

3. Stop and re-start the server. It should now be running with the newly registered `BookMasterCacheLoader`.

Note

To stop the servers, you can use a similar command to the above start command.

```
gfsh run --file=serverStop.gf
```

4. Re-run the `ServerEventsTests` unit tests and note that the first test (`testCacheLoader()` now passes).

9.3.3. Creating and Implementing a CacheWriter

In this section, you will implement a `CacheWriter` that performs validation of new entries. If you recall, any new entries that are created will fire the `beforeCreate()` event method. One of the benefits of having this method called before the actual insert is that we can perform validation to ensure entries meet our desired expectation.

1. To begin, refer to your `ServerEventsTests` unit test. The last two tests are designed to assert the correct behavior of your `ValidatingCacheWriter` implementation.

   Note

   In reality, the `testValidatingCacheWriterSuccess()` method would pass even if there was no `CacheWriter` registered as you've already seen. The main purpose of this method is to ensure that a correct insert does NOT generate an error.

2. Now, open the `ValidatingCacheWriter` class in the `io.pivotal.bookshop.buslogic` package. Notice that the `beforeCreate()` method is left open for you to implement the functionality. Also note that there is a method called `validateNewValue()` where the logic is performed to determine a valid entry. Take a moment to examine this code. You've had some experience already with querying from the client. This is an example of a query being performed on the server. As you can see, a prospective book is considered valid if no other entry exists having the same `itemNumber` value.

3. Now, return to the `beforeCreate()` method and implement the appropriate functionality to obtain the correct value, call the validate method and throw a `CacheWriterException` if the book is considered invalid.

4. Return to the `serverCache.xml` file and add the necessary configuration to register the `ValidatingCacheWriter` with the `BookMaster` region.

5. Save your work, stop and re-start the server.

6. Verify you've correctly implemented and registered the `ValidatingCacheWriter` by re-running the `ServerEventsTests` JUnit test. All tests should now pass.

Congratulations!! You have completed this lab.

Chapter 10. Handling Events on the Client

10.1. Introduction

In this lab, you will gain hands-on experience working with event handling in a GemFire client cache. In addition, you will gain experience writing a continuous query, which will give you a chance to blend OQL concepts with event processing concepts.

Concepts you will gain experience with:

- Writing & registering a cache listener
- Registering interest in events related to certain keys
- Writing a continuous query event handler
- Submitting a continuous query

Estimated completion time: 40 minutes

10.2. Quick Instructions

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

10.3. Instructions

This lab is broken into two key sections. The first will involve writing and registering a cache listener. The second section will focus on using a continuous query.

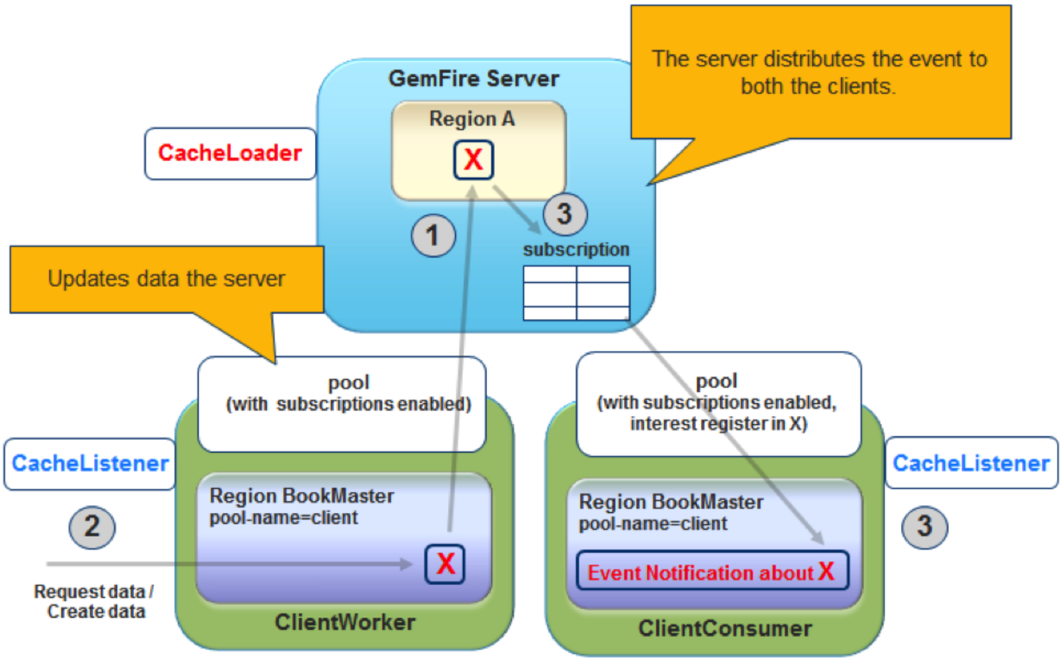
10.3.1. Using a CacheListener

In this section, you will be writing and registering a cache listener attached to your client cache. To understand this part of the lab, let's outline the components that will be involved.

This solution is made up of two components:

- One worker client that retrieves and adds data in its cache
- One consumer client, which receives events from the cacheserver

The consumer client registers interest in some or all keys in the data region. The worker client updates its cache, and the updates are automatically forwarded to the cacheserver. The server forwards to the consumer only those events in which the consumer has registered interest.


The server distributes the event to both the clients.
GemFire Server
Region A
X
CacheLoader
1
3
subscription
Updates data the server
pool (with subscriptions enabled)
pool (with subscriptions enabled, interest register in X)
CacheListener
CacheListener
Region BookMaster pool-name=client
X
Region BookMaster pool-name=client
Event Notification about X
2
3
Request data / Create data
ClientWorker
ClientConsumer

The consumer client has an asynchronous listener that reports local cache activity to standard out, so you can see what is happening. Note that while the ClientWorker could receive similar events, it hasn't registered interest or configured a CacheListener so events will only be sent to the ClientConsumer.

10.3.1.1. Implementing a CacheListener

First, you will implement a CacheListener that will log all create events. Perform the following steps.

1. (`TODO-01`) Open the `LoggingCacheListener` class in the `io.pivotal.bookshop.buslogic` package (under `src/main/java`). Notice that this class extends the `CacheListenerAdapter` class, which means that it doesn't have to implement all the methods defined by the `CacheListener` interface. This way, you only have to implement (override) the methods you are interested in for the events you want to log.

Locate the `init()` method and implement the functionality to extract a property named 'filename' to a String. Then, use this value to call `initializeLogger()` to set up the logger to point to this file.

2. (`TODO-02`) Next, locate the afterCreate() method. Implement this method to log an info type message using the logger class attribute. In the log message, include both the key and new value. Optionally, implement the other methods defined by the CacheListener to log additional events of interest (ex when an entry is updated or deleted).

Tip

Refer to the JavaDocs for the `CacheListener` interface as well as the `EntryEvent` classes to determine the correct methods to invoke.

3. (`TODO-03`) Next, open the `clientConsumerCache.xml` file and modify the pool definition to ensure that subscriptions are enabled.

4. (`TODO-04`) Finally, locate the BookMaster region definition and modify the region configuration to register this `LoggingCacheListener` class as a cache listener. Be sure to also specify the parameter to represent the 'filename' property.

10.3.1.2. Registering Interest in ClientConsumer

Registering interest is simply a matter of obtaining a reference the appropriate region and invoking `registerInterest()` with the desired key or keys.

(`TODO-05`) Open the `ClientConsumer` class (in the `io.pivotal.bookshop.buslogic` package) and locate the line marked with the `TODO` item. Make the appropriate call to the `BookMaster` region instance to register interest in entries with key `999`.

10.3.1.3. Running ClientWorker and ClientConsumer

In order to test the behavior you just implemented, it will be necessary to run both the ClientConsumer and ClientWorker in the following way.

1. First, make sure you have started the server side processes using the `startServer.sh` script (`startServer.bat` for Windows) in the `server-bootstrap/scripts` lab folder.

2. Run the ClientConsumer by either locating the class in the Package Explorer or in an editor tab. Right mouse click on the file or in the editor and select Run As -> Java Application. You will see output to the console indicating when the application has started and is ready for the other application to run.

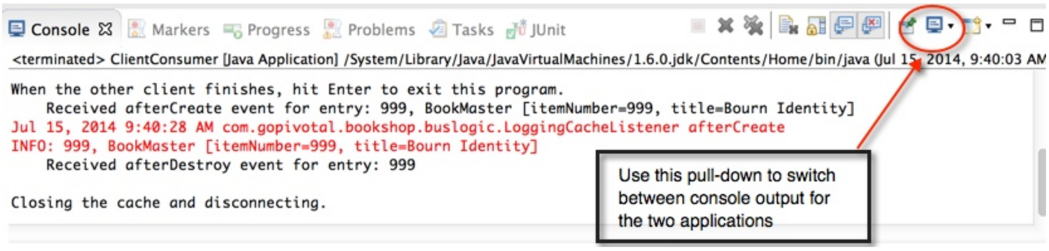
3. Next, run the ClientWorker by either locating the class in the Package Explorer or in an editor tab. Right

mouse click on the file or in the editor and select Run As -> Java Application. This application will run to a certain point. You should see the following in the console output.

```
Connecting to the distributed system and creating the cache.
Note the other client's region listener in response to these gets.
Press Enter to continue.
```

Place your cursor in the console and press Enter. At this point, the program will continue with inserting and then destroying an entry with key 999.

4. Switch back to the console for ClientConsumer. Refer to the illustration below to see how this is done.


Console
Markers
Progress
Problems
Tasks
JUnit
Use this pull-down to switch between console output for the two applications

```
<terminated> ClientConsumer [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Jul 15, 2014, 9:40:03 AM
When the other client finishes, hit Enter to exit this program.
    Received afterCreate event for entry: 999, BookMaster [itemNumber=999, title=Bourn Identity]
Jul 15, 2014 9:40:28 AM com.gopivotal.bookshop.buslogic.LoggingCacheListener afterCreate
INFO: 999, BookMaster [itemNumber=999, title=Bourn Identity]
    Received afterDestroy event for entry: 999

Closing the cache and disconnecting.
```

On the ClientConsumer console output, you should see entries that indicate that an entry was inserted per the logger. The exact output may differ from the above but should be based on how you wrote the logging message in your LoggingCacheListener.

10.3.2. Using a Continuous Query

In this next section, you will be combining your understanding of event processing with the prior experience gained with OQL style queries. This will involve implementing a CQListener to handle Continuous Query events and writing the necessary code to register the listener for a specific query.

1. (`TODO-06`) Open the `SimpleCQListener` class and implement the code of the `onEvent()` method. Write code to print out the various values of the `CqEvent` object that is passed in.

2. Next, open the CQClient class and locate the registerCq() method. Perform the following steps to set up and register the continuous query.

   a. (`TODO-07`) Use the pool instance to get a QueryService instance

   b. (`TODO-08`) Use the QueryService to create a CqAttributes instance, registering the SimpleCQListener class created in the prior step.

   c. (`TODO-09`) Write a query to trigger an event when a BookOrder is created having a totalPrice greater than $100.

   d. (`TODO-10`) Using the CqAttributes you created and the query you wrote, create a new CqQuery and then execute it. If you decide to execute with initial results, capture the results and iterate over them, printing out the orders.

3. Finally, test out your implementation

   a. Start by running the `CQClient` class. Right mouse click on the file or in the editor and select `Run As -> Java Application`. This application will run to a certain point. You should see the following in the console output.

```
Made new CQ Service
Press enter to end
```

   b. Next, locate the `DataProducer` class and run using a similar approach as the prior step. You will see the following output.

```
Press enter to populate an order over $100
```

   Place your cursor in the console and hit `Enter`. This will cause the `DataProducer` to insert an order for $100, which should trigger the `CQListener`. As a result, you should see the console switch back to the CQClient app and display whatever output you defined when implementing the `onEvent()` method of the `SimpleCQListener` class.

   c. Use the technique you used above in Running ClientWorker and ClientConsumer to switch back to the `DataProducer` program that is still running. Place your cursor in the console area again and hit `Enter`. This will insert a BookOrder for a total price less that $100. Therefore, you should see no additional output from the `SimpleCQListener`.

   d. At this point, the `DataProducer` app has terminated. Switch back to the console for the `CQClient`. First verify no additional output was generated. Then, place your cursor in the console area and hit `Enter` to cause this application to end.

Congratulations! You have completed this lab.

Chapter 11. Data Serialization

11.1. Introduction

In this lab, you will gain hands-on experience with...

Concepts you will gain experience with:

- Configuring your cache to perform PDX Auto Serialization
- Working with multiple version of domain objects in PDX
- Using PdxInstance get a single field

Estimated completion time: 40 minutes

11.2. Quick Instructions

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

11.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks. Before beginning any of these tasks.

11.3.1. Configuring the Cache for PDX Auto Serialization

In this section, you will focus on configuring PDX Auto Serialization in both the client and server configurations.

1. (`TODO-01`) To begin, open the `Customer` class definition in the `io.pivotal.bookshop.domain` package. Take a moment to examine how it's written. Note first of all that it does not implement `java.io.Serializable` at all. Also note that one of the field is an `Address` instance that represent various attributes related to a customer's address (ex. city, state, zipcode, etc). If you like, you can also open this `Address` class (in the same package) and note that it also does not implement `java.io.Serializable`. In this condition, the only

way to ensure these objects are serialized is to use one of the PDX Serialization techniques. Note also that both the `Customer` and `Address` classes have default constructors, which are a requirement for PDX Serialization.

2. (`TODO-02`) Next, open the `xml/serverCache.xml` file. Your task is to add the appropriate configuration to enable PDX Serialization in the server cache.

3. (`TODO-03`) Add an attribute to the pdx element signaling that the server should NOT de-serialize objects.

4. Use gfsh to start the locator and two servers.

Note

Unlike prior exercises, you will NOT be using the `--classpath` argument when starting servers. It is important in observing the benefit of PDX Serialization that the domain classes not be on the classpath of the servers.

5. (TODO-04) Open the `xml/clientCache.xml` file and make a similar modification to the definition to support PDX Auto Serialization. Do NOT set the attribute to force client to read as a serialized object. We actually do want the PDX de-serialization to take place on the client.

6. (`TODO-05`) On the client side, we must also specify a serializer class name. Add a `class-name` xml element and specify the fully qualified class name for the `ReflectionBasedAutoSerializer`. In addition, configure a parameter to the auto serializer called `classes` that registers the classes that should be auto serialized. You can use wild cards to be sure you get both the `Customer` class and `Address` class in the `io.pivotal.bookshop.domain` package.

7. (TODO-06) Locate the `CustomerLoader` class in the `io.pivotal.bookshop.buslogic` package and run it. This will load 3 customers into the Customer region on the server.

8. Return to gfsh and execute a query to select values from the Customer region.
   You should see something like the following as output.

```
customerNumber | firstName | lastName |                    primaryAddress                    |
myBookOrders
-------------- | --------- | -------- | ------------------------------------------------------ |
-------------------------
5598           | Kari      | Powell   | class org.apache.geode.pdx.internal.PdxInstanceImpl |
class java.util.ArrayList
6024           | Trenton   | Garcia   | class org.apache.geode.pdx.internal.PdxInstanceImpl |
class java.util.ArrayList
5543           | Lula      | Wax      | class org.apache.geode.pdx.internal.PdxInstanceImpl |
class java.util.ArrayList
```

11.3.2. Working with PDX Domain Object Versions

In this section, you'll begin to understand the power of PDX as you modify the definition of the `Customer` class and see that multiple versions of the class definition can be used within the GemFire distributed system at the same time.

1. (`TODO-07`) To begin, open the `Customer` class again and add a `telephoneNumber` field of type `String`. Also add a getter and setter method. You might also want to go to the `toString()` method and add code to ensure the values of this new field get printed.

2. (`TODO-08`) Open the `NewCustomerClient` class in the `io.pivotal.bookshop.buslogic` package and locate the `testCustomer()` method. Write the code to create a new `Customer` instance. Be sure to set the new `telephoneNumber` property. Save the entry with the key `9999`.

3. (TODO-09) Locate the `testGet()` method and add code to get the newly inserted `Customer` entry (key: `9999`) and print it out.

4. Run `NewCustomerClient` to insert this new record into GemFire.

5. Return to gfsh and re-issue the query command. This time, note that there is the additional entry for key `9999`. Notice also that there is a new field displayed for `telephoneNumber`. Note that the first three entries now show this field value as `null`, which is the expected behavior.

11.3.3. Using PdxInstance

In this last section, you will gain familiarity working with the PdxInstance object. This object will offer the ability to only de-serialize the fields that are required to perform necessary processing.

1. (`TODO-10`) Open the `PdxInstanceClient` class in the `io.pivotal.bookshop.buslogic` package and locate the `testPdxGet()` method. Add the necessary code to fetch the entry for key `9999`, processing as a `PdxInstance`. Extract and print just the `telephoneNumber` field.

Tip

It may be a good idea to add a test to ensure the instance you got back is an instance of PdxInstance and print out a notice if it isn't.

2. (`TODO-11`) Return to the `clientCache.xml` file and add the appropriate attribute to the pdx element to tell the client cache NOT to de-serialize entries into `Customer` objects. This is important if you intend to process your entries as `PdxInstance` objects.

3. Run the `PdxInstanceClient` and verify that you were able to obtain the `telephoneNumber` field and that it is the correct value. It should be the value you set in the prior section when you created the new `Customer` entry.

Congratulations!! You have completed this lab.

Chapter 12. Transaction Management

12.1. Introduction

In this lab, you will learn to configure and test GemFire transactions. You can group cache updates with GemFire transactions. Transactions allow you to create dependencies between cache modifications so they all either complete together or fail together.

Concepts you will gain experience with:

- Creating the `Customer` region, co-located with the `Order` region.
- Create a Java class to implement GemFire transaction by using the `CacheTransactionManager` API.
- Use transactions to update data on both the `Customer` region and the `Order` region.

Estimated completion time: 45 minutes

12.2. Quick Instructions

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

12.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks. Open the `transactions` project in the STS.

12.3.1. Implement the TransactionService functionality

(`TODO-01`): Before beginning, open the test harness, `TransactionalTest` under the `io.pivotal.bookshop.tests` under `src/test/java`. Observe the two tests that will be executed to validate a correct transaction commit and a transaction rollback. Note that the first test asserts that the transaction commits successfully and that the associated values have been updated.

Open the `TransactionsService` class within `io.pivotal.bookshop.buslogic` package and locate the `updateCustomerAndOrder()` method. Your responsibility is to write the necessary code to perform the fetch and update within a transaction. Perform the following steps.

1. (`TODO-02`): Define `CacheTransactionManager`, and get the transaction context.
2. Add the transactional code.
   a. (`TODO-03a`) Begin the transaction.
   b. (`TODO-03b`) Retrieve and update the `Customer` using the customerKey and updatedCustomerPhone values. Similarly, retrieve and update the Order using the orderKey and updatedOrderDate value.
   c. (`TODO-03c`): Save the `Customer` and `Order` objects in the cache with the same keys.
   d. (`TODO-03d`): Commit the transaction.
3. (`TODO-04`) Add code in the exception handler to roll back the transaction in case of any error with the transaction. This handler should catch the exception, roll back the transaction and re-throw the exception so the test harness can catch it.

12.3.2. Cache configuration and Server Start

There are several configuration changes you'll want to make to ensure transactions between two partitioned regions work properly.

1. Open the `serverCache.xml` file.
2. (`TODO-05`): Set the cache configuration attribute `copy-on-read` to `true` to avoid in place changes.
3. (`TODO-06`): Add necessary configuration so that the `Order` region is co-located with the `Customer` region.
4. (`TODO-07`): Save your work and then start the locator and two servers using the start script in the project folder.

Tip

From the command prompt, be sure to first run either **source gf.config** or **setEnv.bat**, which are both in the `server-bootstrap/scripts` project. Next, run **gfsh run --file=serverStart.gf**

12.3.3. Running the tests

Run the tests and verify both tests pass.

Also, take a look at the `listener.log` file that is created in the transactions folder. The last entries should look something like the following output.

```
[info 2015/11/18 13:27:43.635 MST server2 <ServerConnection on port 56069 Thread 0> tid=0x4a] afterUpdate:   Entry
    updated for key: 1001
                 Old value: Customer [customerNumber=C001, firstName=Lula, lastName=Wax, Phone=123-654-543,
    Address=Address [addressTag=HOME, addressLine1=123 Main St., city=Topeka, state=KS, postalCode=50505, country=US]]
                 New Value: Customer [customerNumber=C001, firstName=Lula, lastName=Wax, Phone=222-22222-0000,
    Address=Address [addressTag=HOME, addressLine1=123 Main St., city=Topeka, state=KS, postalCode=50505, country=US]]

[info 2015/11/18 13:27:43.639 MST server2 <ServerConnection on port 56069 Thread 0> tid=0x4a] afterUpdate:   Entry
    updated for key: 1001
                 Old value: Order [orderNumber=ORD001, orderDate=Tue Dec 03 00:00:00 MST 2013, customerNumber=C001,
    totalPrice=103.5, orderItems=[ProductItem:  [itemNumber=P001, Description=Toy], ProductItem:  [itemNumber=P002,
    Description=Watch], ProductItem:  [itemNumber=P003, Description=Pen]]]
                 New Value: Order [orderNumber=ORD001, orderDate=Thu Apr 25 00:00:00 MDT 2013, customerNumber=C001,
    totalPrice=103.5, orderItems=[ProductItem:  [itemNumber=P001, Description=Toy], ProductItem:  [itemNumber=P002,
    Description=Watch], ProductItem:  [itemNumber=P003, Description=Pen]]]
```

What this is actually showing is that the updates to the entries once the transaction commits. These are captured and logged using the `LoggingCacheListener` to fire on the `afterUpdate()` event.

12.3.4. Implementing a TransactionListener

In this next section, you will implement a `TransactionListener` that will allow you to also track the transactional events, namely the commit and rollback operations. This will allow you to verify that the commit and rollback events actually occur.

To begin, open the `LoggingTransactionListener` class. Note that this class extends `TransactionListenerAdapter` as well as implementing `Declarable`. You will implement two methods that override the methods on the adapter class.

1. (`TODO-08`): First, add an `afterCommit()` method that overrides the one in TransactionListenerAdapter. In the body of the method, use the logger to log an INFO message that this is an afterCommit event and then also use the `TransactionEvent` object that is passed in to obtain the list of related cache events. Iterate over these events, printing out the event details.

   Tip

   Refer to one of the methods of the `LoggingCacheListener` for an example of what you might want to print for each event.

2. (`TODO-09`): Similarly, add an `afterRollback()` method that overrides the `TransactionListenerAdapter` method and use it to log an `INFO` message that this is an `afterRollback` event. If desired, you may also be interested to obtain and print out related operation(s) that were involved in the transaction that rolled back.

3. (`TODO-10`): Open the `serverCache.xml` file and add appropriate configuration to register this LoggingTransactionListener.

4. (`TODO-11`): Use the stop script (`serverStop.gf`) to stop the locator and servers. Then, restart and re-run the TransactionTests test harness.

This time when you look at the end of the `listener.log` file that is created in the `transactions` folder, you should see additional log data that should look something like the following. In this case, you'll notice that the updates happen before the `afterCommit` event is triggered. Notice also that there are two events that are associated with this commit as shown by the event for `Customer` and `Order`.

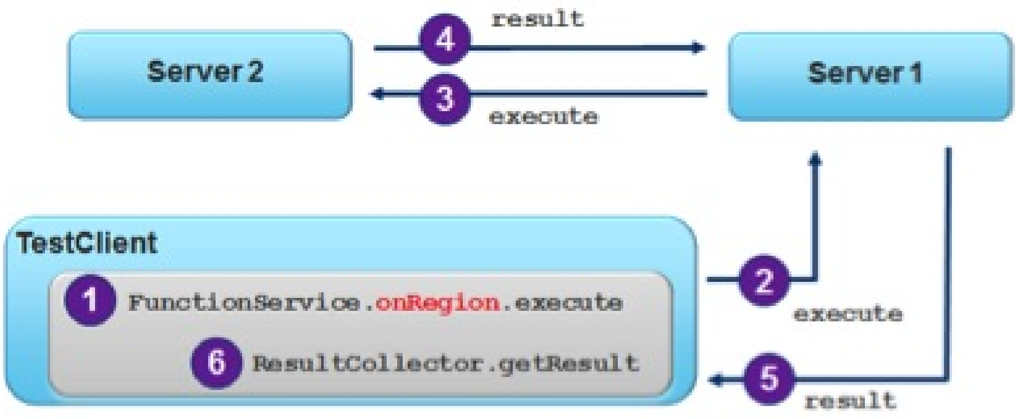
```
[info 2015/11/18 15:28:03.590 MST server2 <ServerConnection on port 57651 Thread 0> tid=0x4a] afterUpdate:   Entry
    updated for key: 1001
                 Old value: Customer [customerNumber=C001, firstName=Lula, lastName=Wax, Phone=123-654-543,
    Address=Address [addressTag=HOME, addressLine1=123 Main St., city=Topeka, state=KS, postalCode=50505, country=US]]
                 New Value: Customer [customerNumber=C001, firstName=Lula, lastName=Wax, Phone=222-22222-0000,
    Address=Address [addressTag=HOME, addressLine1=123 Main St., city=Topeka, state=KS, postalCode=50505, country=US]]

[info 2015/11/18 15:28:03.595 MST server2 <ServerConnection on port 57651 Thread 0> tid=0x4a] afterUpdate:   Entry
    updated for key: 1001
                 Old value: Order [orderNumber=ORD001, orderDate=Tue Dec 03 00:00:00 MST 2013, customerNumber=C001,
    totalPrice=103.5, orderItems=[ProductItem:  [itemNumber=P001, Description=Toy], ProductItem:  [itemNumber=P002,
    Description=Watch], ProductItem:  [itemNumber=P003, Description=Pen]]]
                 New Value: Order [orderNumber=ORD001, orderDate=Thu Apr 25 00:00:00 MDT 2013, customerNumber=C001,
    totalPrice=103.5, orderItems=[ProductItem:  [itemNumber=P001, Description=Toy], ProductItem:  [itemNumber=P002,
    Description=Watch], ProductItem:  [itemNumber=P003, Description=Pen]]]

[info 2015/11/18 15:28:03.597 MST server2 <ServerConnection on port 57651 Thread 0> tid=0x4a] afterCommit: TxId= TXId:
    192.168.0.60(DataOperations Client:31819:loner):57660:0d39b61c:DataOperations Client:1

[info 2015/11/18 15:28:03.597 MST server2 <ServerConnection on port 57651 Thread 0> tid=0x4a]    Entry updated for key:
    1001
            Old value: Customer [customerNumber=C001, firstName=Lula, lastName=Wax, Phone=123-654-543, Address=Address
    [addressTag=HOME, addressLine1=123 Main St., city=Topeka, state=KS, postalCode=50505, country=US]]
            New Value: Customer [customerNumber=C001, firstName=Lula, lastName=Wax, Phone=222-22222-0000,
    Address=Address [addressTag=HOME, addressLine1=123 Main St., city=Topeka, state=KS, postalCode=50505, country=US]]

[info 2015/11/18 15:28:03.598 MST server2 <ServerConnection on port 57651 Thread 0> tid=0x4a]    Entry updated for key:
    1001
            Old value: Order [orderNumber=ORD001, orderDate=Tue Dec 03 00:00:00 MST 2013, customerNumber=C001,
    totalPrice=103.5, orderItems=[ProductItem:  [itemNumber=P001, Description=Toy], ProductItem:  [itemNumber=P002,
    Description=Watch], ProductItem:  [itemNumber=P003, Description=Pen]]]
            New Value: Order [orderNumber=ORD001, orderDate=Thu Apr 25 00:00:00 MDT 2013, customerNumber=C001,
    totalPrice=103.5, orderItems=[ProductItem:  [itemNumber=P001, Description=Toy], ProductItem:  [itemNumber=P002,
    Description=Watch], ProductItem:  [itemNumber=P003, Description=Pen]]]

[info 2015/11/18 15:28:03.704 MST server2 <ServerConnection on port 57651 Thread 0> tid=0x4a] afterRollback: TxId= TXId:
    192.168.0.60(DataOperations Client:31819:loner):57660:0d39b61c:DataOperations Client:2

[info 2015/11/18 15:28:03.704 MST server2 <ServerConnection on port 57651 Thread 0> tid=0x4a]    Cache event received
    with operation: UPDATE
```

Congratulations!! You have completed this lab. Be sure to use the `serverStop.gf` script to stop the locator and servers.

Chapter 13. Writing Server Side Functions

13.1. Introduction

In this lab, you will gain hands-on experience with writing and registering a GemFire function. A test client will invoke the function on several GemFire servers to extract customer details for a given city.



Concepts you will gain experience with:

- Developing a GemFire function
- Register functions using XML configuration
- Execute the function on multiple servers
- Invoke the function from a client application

Estimated completion time: 30 minutes

13.2. Quick Instructions

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

13.3. Detailed Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks. Before beginning any of these tasks.

13.3.1. Develop the Function

Function execution allows you to move function behavior to the application member hosting the data. In this exercise, you develop a function to perform a sum a specified field of all entries for a given region. This will offer an opportunity to explore one of the use cases functions can be used for in GemFire. For simplicity, this 'generic' summing function, we'll assume that the type used for summing is `java.lang.Float`.

To perform this task, open the `server-functions` project in the STS.

1. Open the `GenericSumFunction` class in the `io.pivotal.bookshop.buslogic` package. Notice that this class extends `FunctionAdapter` class and also implements the `Declarable` interface.

2. You will implement the execute() function using the following steps.

   a. (`TODO-01`) Enforce that the `FunctionContext` is an instance of `RegionFunctionContext`. It's important that this be done as this function is only designed to be run via the `onRegion(..)` method calls. You might throw a `FunctionException` if this prerequisite isn't met. Assuming the `FunctionContext` is an instance of `RegionFunctionContext`, cast up to an instance of `RegionFunctionContext`.

   b. (`TODO-02`) Use the FunctionContext to get the argument passed from the client. This will represent the field on the `PdxInstance` for which the summing operation will be performed.

   c. (`TODO-03`) Use the `PartitionRegionHelper` class to get all the local region data. We'll later add additional configuraiton to ensure that the local data obtained is strictly primary region data. Also initialize an instance of `BigDecimal` as this will be used to hold the sum amount that will be returned by the function.

   d. (`TODO-04`) Iterate over the local data, which should be a collection of `PdxInstance` objects. We'll later add configuration to enable PDX Serialization and to enforce that data read on the server side remains serialized.

   e. (`TODO-05`) Use the argument provided to de-serialize (extract) the field that will be used for summing. Initially, extract an `Object`. In a successive step, make sure the field is an instance of `Float` as you will be making that assumption. If it an instance of `Float`, add this value to the current sum amount.

   f. (`TODO-06`) Once all entries have been processed, send the final sum back to the caller. Since you will be only sending one result for this execution, make sure that you signal that you are completed sending results.

3. (`TODO-07`) Add an overridden method that will enforce that only primary buckets be considered when processing local data.

Tip

If in doubt, go back and review the slides around the Function API. It will help if you annotate the method with `@Override` so that you can be sure you are correctly overriding the method.

13.3.2. Register the Function and Start Servers

You'll need to register the function you just wrote with the server cache configuration.

1. (`TODO-08`) Next, open the `serverCache.xml` file and add the necessary configuration to enable PDX Serialization using the `ReflectionBasedAutoSerializer`.
2. (`TODO-09`) Add the appropriate configuration to register the function
3. (`TODO-10`) Start the locator and 2 servers using similar approach as in the prior few labs. In particular, you will need to use the `--classpath=../target/classes` option. Next, execute `OrderLoader` class found in the `io.pivotal.bookshop.buslogic` package in STS. This will load the Order Region with data.

13.3.3. Calling the Function

Finally, you will test the function using the `SummingFunctionTest` JUnit test found in `src/test/java`.

1. In the next lab exercise, you will focus on the writing the code to execute the function and process the results. For now, you will just worry about executing the function and verifying the results.
2. (`TODO-11`) Go ahead and run the program from the STS. You see that the test passes.
3. Stop the servers and locator.

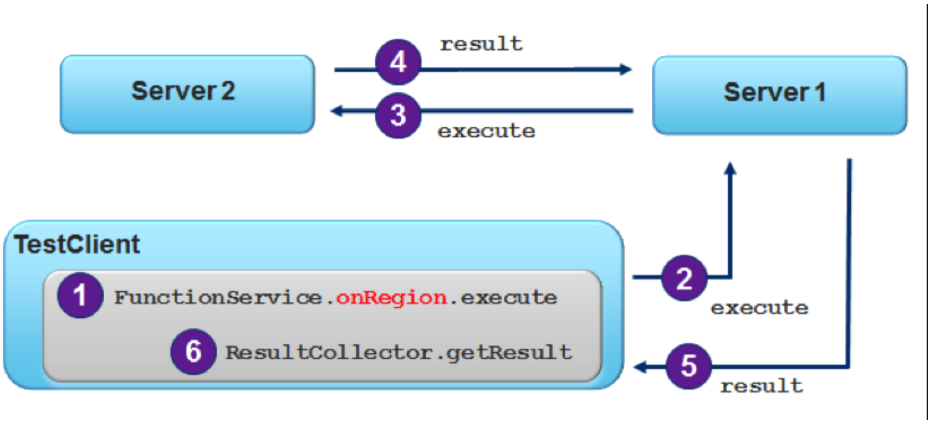
Congratulations!! You have completed this lab.

Chapter 14. Client Function Execution

14.1. Introduction

In this lab, you will gain hands-on experience working with executing server-side functions from a GemFire client cache. In the prior lab, you gained experience writing and registering and testing a function that provides generic summation capabilities. For this lab, that function has already been created and registered on the server. You task in this lab is to set up the client with the necessary configuration and programming to enable the client to call this function and process the result.

The basic execution flow is illustrated below.



Server 2
4
result
3
execute
Server 1
TestClient
1 FunctionService.onRegion.execute
6 ResultCollector.getResult
2
execute
5
result

What you will learn

- How to execute functions on the server
- How to process results from execution

Estimated completion time: 30 minutes

14.2. Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks.

14.2.1. Enabling PDX Serialization

To begin, you will be making the necessary modifications to enable client to perform PDX Serialization.

1. (`TODO-01`) To begin, open the `clientCache.xml` file and add the necessary configuration to enable PDX Serialization using the `ReflectionBasedAutoSerializer`. On the client side, you will NOT request read serialized.
2. (`TODO-02`) In the same XML file and in the PDX configuration, add additional configuration to define the appropriate classes to serialize. These will be `BookOrder` and `BookOrderItem`.

14.2.2. Implementing a Custom `Result Collector`

In this section, you will be implementing a custom `Result Collector`. The purpose of this custom result collector is to take all the sum values sent by each member executing the function and provide a final sum representing the aggregate sum of the specified field for all data in all members hosting the partitioned region.

1. (`TODO-03`) Next, open the `SummingResultCollector` class in the `io.pivotal.bookshop.buslogic` package. You will need to add a class field to hold the results. If necessary, open the `GenericSummingFunction` class, provided in this project for reference, and take note of the type used to send results. If you think about what is trying to be accomplished, the objective is to have one final sum amount to return to the caller once all results have been returned from the members executing the function. Define the variable and initialize it now.
2. (`TODO-04`) Next, go to the `addResult()` method and implement the functionality for what means to add a result to the current results.
3. (`TODO-05`) Next, locate the `clearResults()` method and implement the functionality for what it means to clear results.
4. (`TODO-06`) Finally, locate the `getResults()` method and implement this functionality. Recall that this is the method used to provide the final results back to the client caller.

14.2.3. Writing Client to Execute the Function

In this section, you will be writing the necessary client code to execute a function that has already been registered on the server and leveraging the result collector you just defined. Use the following steps to complete this section.

1. (`TODO-07`) Open the `SummingTests` test class. Locate the `shouldComputeTotalForAllOrders()` method and

add code that defines the Function execution. To do this, you will define an `onRegion()` type function call on the `BookOrder` region and you will be passing an argument in to perform the sum on the field called `totalPrice` and use the custom `ResultCollector` called `SummingResultCollector`.
Also in this step, add a call to execute the function by name (which should include the package and class). This call returns a `ResultCollector`, which you should assign to a variable so you can use it in the next step.

2. (`TODO-08`) With the result collector, call the method to get the results. This result should be a single object. Refer to the `SummingResultCollector` to see what type is returned and assign to a variable of that type. Assert that the result returned is equal to the amount `93.95`.

Note

Due to typical rounding errors that can happen with `Float` and such, you may want to create a BigDecimal to compare to using a technique like this: `new BigDecimal("93.95")`.

14.2.4. Running the Test

In this section, you will be running the test code you just finished writing to verify correct implementation of both the execution part as well as the result collector part. Before beginning this section, you will need the locator and servers running. If they are still running from the server functions lab, you can use them and ignore step 1 below. Otherwise, use the following steps to complete this lab.

1. Make sure the server and locator are running by using the `server-bootstrap` project `scripts/startServerPartitioned.sh` or `scripts/startServerPartitioned.bat` scripts.

Note

Due to the specific requirements for this lab, it is important that the GemFire servers be running using the startup configuration from the server functions lab or by running the script described above. If neither of these is true or you are uncertain, the simplest thing to do is stop all GemFire processes and use one of the start scripts in the `server-bootstrap` project.

2. (`TODO-09`) Finally, run `SummingTests` to verify all is working correctly. You may not see any output except a notification that the cache is closing.

Congratulations! You've completed this lab.

Chapter 15. Spring GemFire

15.1. Introduction

In this lab, you will gain hands-on experience working the Spring Data GemFire project to build client-side GemFire applications. These series of steps will help you appreciate simplicity of configuring GemFire clients using Spring.

What you will learn

- Basic client cache configuration using Spring Data GemFire
- Configuring and using the GemfireTemplate
- Using the GemFire Repository interface
- Registering interest using Spring Data GemFire

Estimated completion time: 45 minutes

15.2. Instructions

Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks. Before beginning this lab, make sure you have started the server side processes using the `startServer.sh` script (`startServer.bat` for Windows) in the `server-bootstrap` lab folder.

15.2.1. Basic configuration using Spring Data GemFire

In this first section, you will get a basic configuration up and running that will include setting up a client cache and defining a client region for the BookMaster region on the server.

1. (`TODO-01`) Locate and open the `spring-config.xml` file under `src/main/resources` folder. Notice that this is largely an empty file at the moment. Notice also that the Gemfire namespace has been enabled with the `gfe` prefix. Your first task is to configure the client pool that is configured to point to the locator.
2. (`TODO-02`) Create a client cache definition pointing to the pool you just defined.
3. (`TODO-03`) Define a client region and configure it as a `CACHING_PROXY`

4. (`TODO-04`) Open the `BasicSpringClientTests.java` file under `src/test/java`. The basic structure of this test harness has already been set up. Take a moment to get familiar with the basic way this test harness is configured using the Spring-aware integration test. Notice also how we injected the region into the harness using the `@Resource` annotation. Usually, we use the Spring specific `@Autowired` annotation but the nature of this object requires we inject it as a named bean.

5. (`TODO-05`) Finally, run the test by right-mouse clicking on the file in the package explorer or in the open file. Then select `Run As -> JUnit Test`. If you configured the `spring-config.xml` file properly, the tests should pass.

15.2.2. Using GemfireTemplate

In this next section, you will gain familiarity with the GemfireTemplate, one of the helper classes provided by the Spring Data GemFire project.

1. (`TODO-06`) Return to the spring-config.xml file and add a basic bean definition to instantiate an instance of the GemFire template. You can either inject the `BookMaster` region as a constructor argument or set it as a property on the bean.

   Tip

   You will use a basic bean definition as <bean id="someName" class="ClassToInstantiate"></bean> where `ClassToInstantiate` is the full class with package.

2. (`TODO-07`) Return to the `BasicSpringClientTests` class and add a definition to autowire the GemFire template you just configured into the test harness.

3. (`TODO-08`) Locate the `testGemFireTemplate()` test method and add some code to execute a simple query on the template. Use the query() method to search for books having an author of `Daisy Mae West`. Also write a couple of assert tests to assert that you got just one result back and the title of the book was `A Treatise of Treatises`.

   Tip

   Note that the results returned are `SelectResults`, which is a GemFire collection type. You can use the `asList()` method to convert the results into simple Java `List` type.

4. Re-run the test harness and ensure the tests pass.

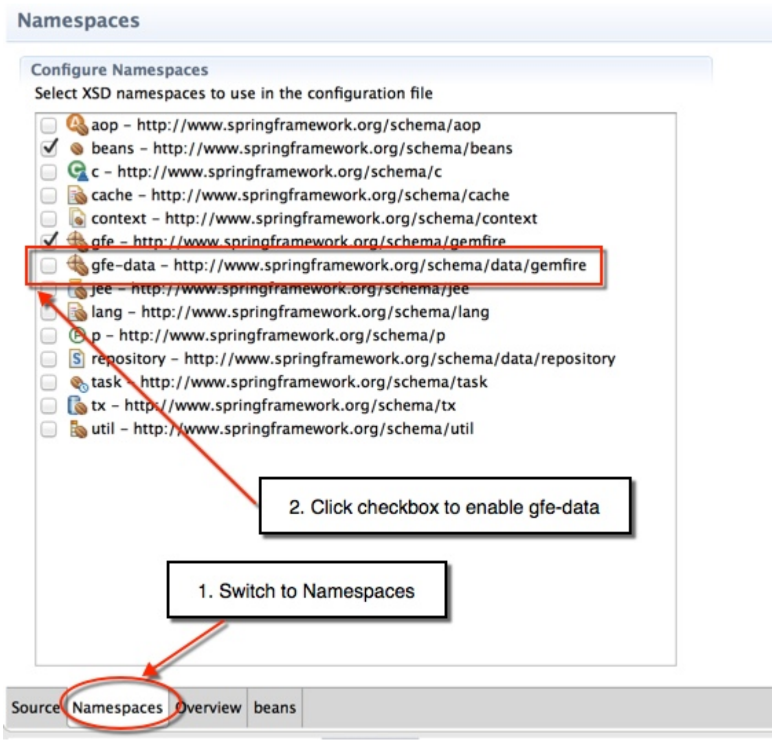
15.2.3. Using Repositories

This section will allow you to gain familiarity with the concept of Repositories. This approach allows you to create repository (or DAO) style interfaces declaratively.

1. (`TODO-09`) Open the `BookMaster` class in the `io.pivotal.bookshop.domain` package. Add appropriate annotation to the top of the class to declare that domain object is obtained from the `BookMaster` region.

2. (`TODO-10`) Create a new interface in the `io.pivotal.bookshop.buslogic` package that will be your Repository interface. Make it extend the `CrudRepository` base interface and set the entry type and key type appropriately for the generics part of the definition.
   In the same interface declaration, add a `findBy` method declaration that will support finding `BookMaster` objects by the `Author` attribute. The method have a single argument of type `String` and return a list of `BookMaster` objects.

3. (`TODO-11`) Return to the `spring-config.xml` file and add an entry to configure scanning for repositories. In order to do this, you'll first have to enable the `gfe-data` namespace as shown below.



Namespaces

Configure Namespaces

Select XSD namespaces to use in the configuration file

- ☐ aop - http://www.springframework.org/schema/aop
- ✓ beans - http://www.springframework.org/schema/beans
- ☐ c - http://www.springframework.org/schema/c
- ☐ cache - http://www.springframework.org/schema/cache
- ☐ context - http://www.springframework.org/schema/context
- ✓ gfe - http://www.springframework.org/schema/gemfire
- ☐ gfe-data - http://www.springframework.org/schema/data/gemfire
- ☐ jee - http://www.springframework.org/schema/jee
- ☐ lang - http://www.springframework.org/schema/lang
- ☐ p - http://www.springframework.org/schema/p
- ☐ repository - http://www.springframework.org/schema/data/repository
- ☐ task - http://www.springframework.org/schema/task
- ☐ tx - http://www.springframework.org/schema/tx
- ☐ util - http://www.springframework.org/schema/util

4. (TODO-12) Return to the `BasicSpringClientTests` class and add another declaration near the top of the class to autowire in your newly created `Repository` interface.

5. (`TODO-13`) Next, locate the `testGemFireRepositories()` method and add the necessary code to correctly invoke the method on the repository instance. Also write a couple of asserts to verify you get only one matching entry and the entry's title is `A Treatise of Treatises`. You can use the code from the `testGemFireTemplate()` method as a reference.

15.2.4. Configuring Listeners and registering interest

In this final section, you will explore the capabilities of Spring Data GemFire to simplify configuration of CacheListeners and to enable clients to register interest in certain keys. Since you've already performed most of the basic coding in a prior lab, all you'll do in this lab is add the appropriate Spring Data GemFire configurations and re-run the `ClientConsumer` and `ClientWorker` to test the behavior out.

1. (`TODO-14`) Open the `spring-config.xml` file again and locate the pool configuration. Add an attribute to enable client subscriptions.

2. (`TODO-15`) Locate the client-region definition you created in `TODO-03`. Add an entry inside this region definition to configure a `CacheListener` for the region. Have it point to the `LoggingCacheListener` that is found in the `io.pivotal.bookshop.buslogic` package.

3. (TODO-16) Finally, add another entry inside the client-region definition to register interest in the key `999`.

   Tip
   Note that this key is an integer type so you'll need to configure appropriately when creating the interest registration.

4. (`TODO-18`) Open the `ClientConsumer` class and take a look at the functionality implemented there. As you can see, all the class does is initialize the Spring `ApplicationContext` and then wait for the `ClientWorker` to perform some operations on the cache. What you should see when that happens is that the `SimpleCacheListener` will report that an entry was created and deleted having the key `999`.
   Go ahead and run the class now.

5. (`TODO-19`) Next, locate the `ClientWorker` class. This is basically the same class that was used in the events lab to create a new `BookMaster` entry with the key `999`. Run this class. The program will start by displaying some basic information and then pause waiting for user input to continue.
   Place your cursor in the console area and hit enter. The program will now proceed to insert an entry with key 999 and the remove it before terminating.
   Switch consoles back to the ClientConsumer and observe that the SimpleCacheListener reported that the

entry was crated and then deleted.

6. Make sure that both the ClientWorker and ClientConsumer have terminated

Congratulations! You have completed this lab.

Chapter 16. Understanding REST Support in GemFire

16.1. Introduction

In this lab, you will gain hands-on experience working the REST support in GemFire.

What you will gain experience with

- Basic configuration process for enabling REST support in GemFire
- Familiarity with the SwaggerUI as a way to understand what you can do with the REST interface
- Use Spring REST Template to interact with the REST service

Estimated completion time: 30 minutes

16.2. Instructions

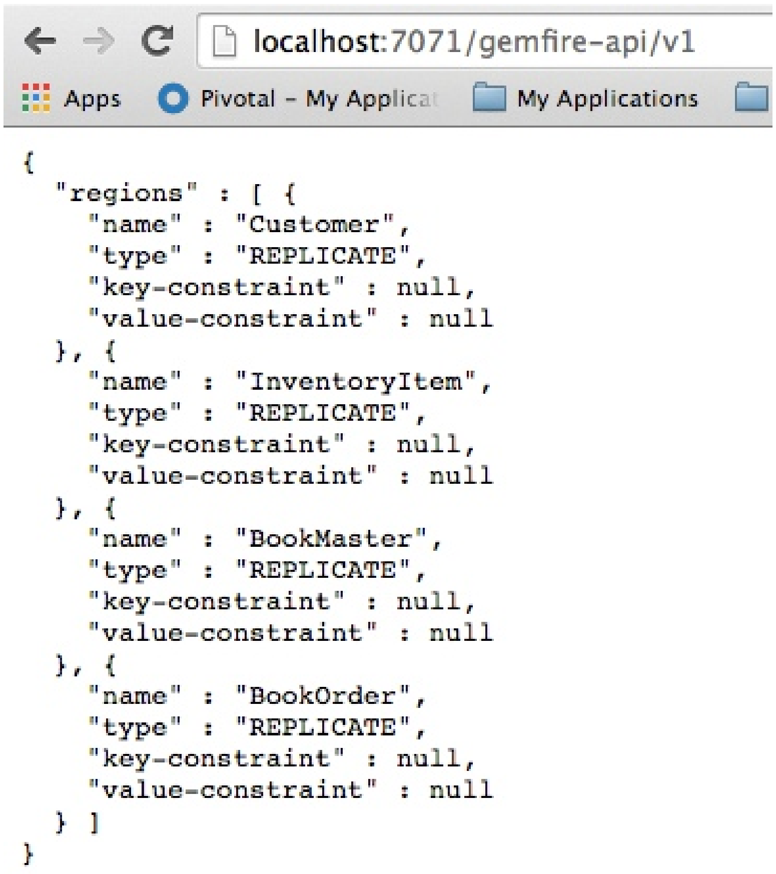
Instructions for this lab are divided into specific sections. Each section describes the steps to perform specific tasks.

16.2.1. Configuring REST Support

In this first section, you will perform the basic configuration steps necessary to fully enable REST support in GemFire.

1. (`TODO-01`) Locate and open the `serverCache.xml` file under the `xml` folder. Add the necessary configuration to enable PDX Serialization. Additionally set the configuration such that the server will NOT de-serialize objects when fetching and returning them.
2. (`TODO-02`) Start a locator process followed by a server. For the server, be sure to add the appropriate configuration items to enable REST support and to use port `7071` as the HTTP port. Once done, run the `OrderLoader` class in the `io.pivotal.bookshop.buslogic` package in `src/main/java` folder. This will populate the `BookOrder` region with a few entries.
3. (`TODO-03`) Finally, verify that you have correctly set up the REST support by opening a browser to the

following URL: `http://localhost:7071/gemfire-api/v1`. If you have correctly configured REST support, you should see something like the following.



```
{
  "regions" : [ {
    "name" : "Customer",
    "type" : "REPLICATE",
    "key-constraint" : null,
    "value-constraint" : null
  }, {
    "name" : "InventoryItem",
    "type" : "REPLICATE",
    "key-constraint" : null,
    "value-constraint" : null
  }, {
    "name" : "BookMaster",
    "type" : "REPLICATE",
    "key-constraint" : null,
    "value-constraint" : null
  }, {
    "name" : "BookOrder",
    "type" : "REPLICATE",
    "key-constraint" : null,
    "value-constraint" : null
  } ]
}
```

Figure 16.1. Using browser to verify REST service is properly configured

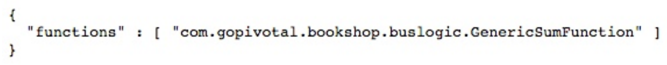
16.2.2. Using the Swagger UI

In this next section, with the Swagger UI that is provided as part of the GemFire REST support. This part of the lab won't involve any work with the IDE so we'll dispense with using TODO steps for now.

1. To begin, open your browser to the URL: `http://localhost:7071/gemfire-api/docs`. This is the starting point for the UI.

2. Click on the `functions`, `queries` and `region` links to see the REST calls that can be made for each. Notice, for example, that you can get a list of all functions by using the call to `/v1/functions`. You can also try this call out by clicking the Try it out button. Note the actual URL, which may be missing the hostname component. Also notice the pattern for how the Swagger UI displays the URL and what the actual full URL is. You will need to make use of this in future tasks.

3. Using the example provided by Swagger, see if you can formulate a URL call to obtain a list of functions deployed and type that into the browser. You should see something like the following.



```
{
  "functions" : [ "com.gopivotal.bookshop.buslogic.GenericSumFunction" ]
}
```

Figure 16.2. Sample return result from proper call to list functions

4. Take a little time to experiment with the various interfaces related to queries and regions. See if you can perform the following tasks - both from the Swagger UI and directly from the browser (or using `curl` if you have it on your machine).

   a. List all entries in the `BookOrder` region

   b. Fetch the entry from the `BookOrder` region having key `17699`

   c. Perform an ad-hoc query to select entries from the `BookOrder` region having an `orderNumber` value of `17699`

16.2.3. Using the Spring REST Template

In this final section, actually create a REST client using the Spring RestTemplate class to aid in building the request and converting the JSON object to

1. (`TODO-04`) Open the RestClientTest class in the io.pivotal.bookshop.tests package under src/test/java folder. Define the URL that will be used to make the request using the `BASE_URL` static String as a starting point. Keep in mind you want to use the URI template format to allow for parametrizing such elements as the key.

2. (`TODO-05`) Issue a call to the RestTemplate to return the object given the URL you defined and passing the key `17699` as the parameter. Don't forget to also specify what class to return for the object. Assign the result to the `order` object.

3. (TODO-06) Run the test. Did it complete successfully? If not, why not?

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

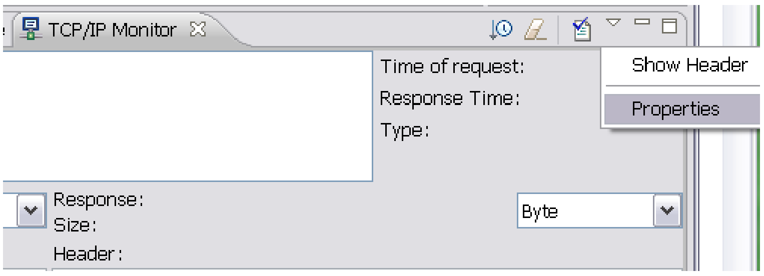


Figure 16.3. The "properties" menu entry of the TCP/IP Monitor view

   Choose "Add..." to add a new monitor. As local monitoring port, enter 7072 since this port is probably unused. As host name, enter "localhost" and as port enter 7071 since this is the port that Tomcat is running on. Press OK and then press "Start" to start the newly defined monitor.

Tip

Don't forget to start the monitor after adding it! The most common error at this point is to forget to start the monitor.

5. (TODO-07) Return to the RestClientTest class in the STS and locate the BASE_URL definition. Change the port from `7071` to `7072` so that the TCP/IP monitor you just configured can intercept the requests. Re-run the test and observe the result. You'll need to expand the TCP/IP tab (by double-clicking on it). You should be able to observe that the proper entry came back as indicated by the response body.

6. Return to the JUnit out put screen and observe in the Failure Trace the message being reported. Notice that the expected form of a Data object doesn't match the form being supplied by the JSON object being returned by the server.

7. (TODO-08) Add the appropriate JSON formatting annotation to the `orderDate` field in order to instruct the converter to properly interpret the date being provided by the JSON object. Provide the same formatting

annotation to the `shipDate` field.

8. (TODO-09) Re-run the test. It should now pass. Take a moment to consider what has just happened. You submitted a REST request to the server via the `RestTemplate` but you did it via a very typical method call. The data returned from the server was a JSON object (as verified by the TCP/IP monitor). However, the `RestTemplate` performed a conversion with the assistance of some well placed formatting annotations on the domain object. What you received was a fully populated `BookOrder` object that we were able to assert had the proper order number.

Congratulations! You have completed this lab.

Appendix A. Eclipse Tips

A.1. Introduction

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

A.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.

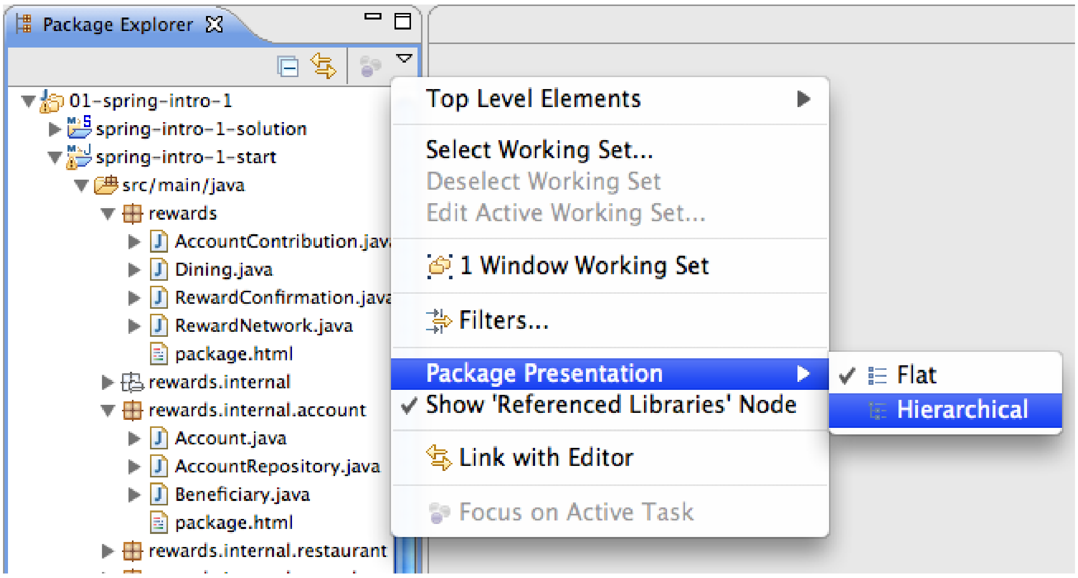
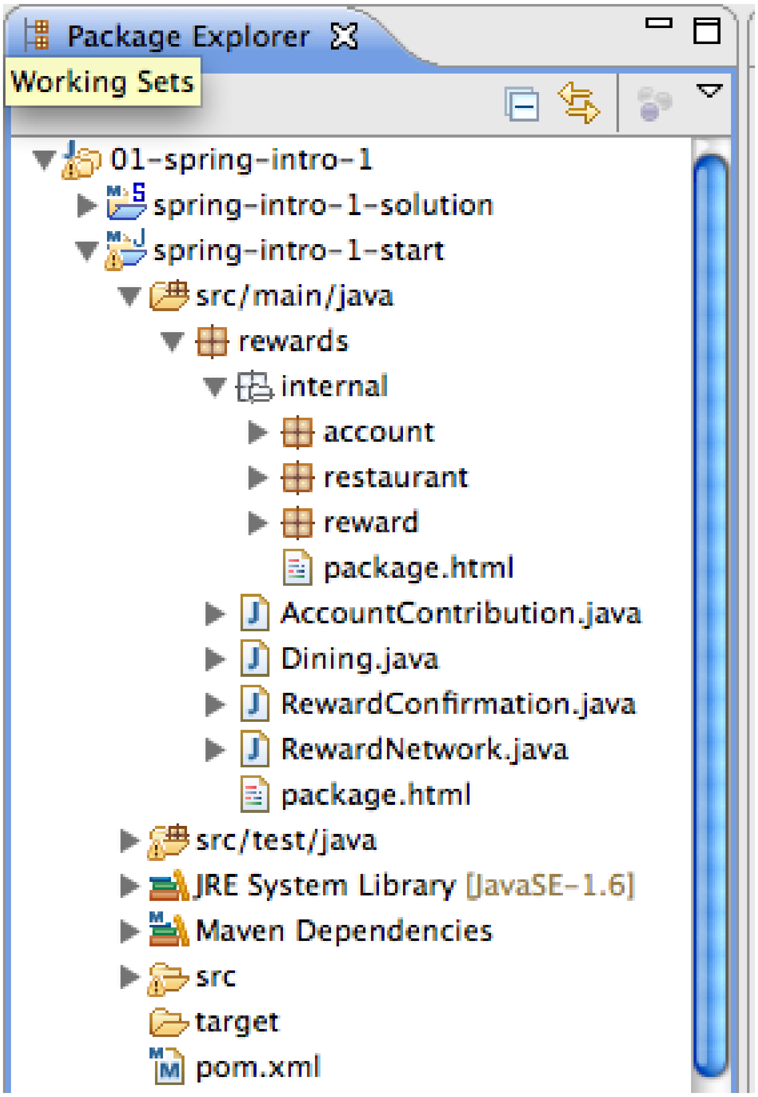


Figure A.1. Swtiching Views

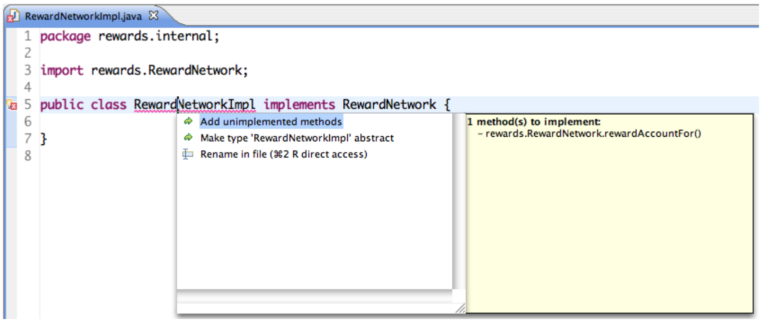
Switch between hierarchical and flat views by selecting the menu inside the package view (represented as a triangle), selecting either Flat or Hierarchical from the Package Presentation submenu



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

Figure A.2. The hierarchical view shows nested packages in a tree view

A.3. Add Unimplemented Methods



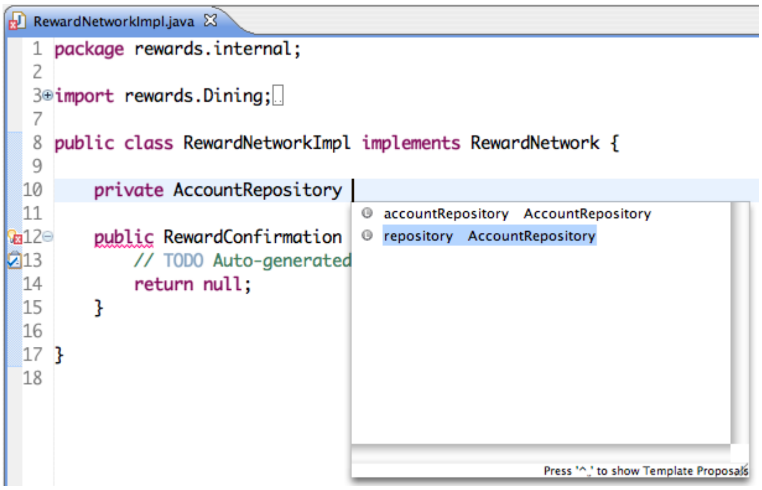
```
package rewards.internal;

import rewards.RewardNetwork;

public class RewardNetworkImpl implements RewardNetwork {

}

```

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

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

Figure A.3. Add unimplemented methods" quick fix

A.4. Field Auto-Completion



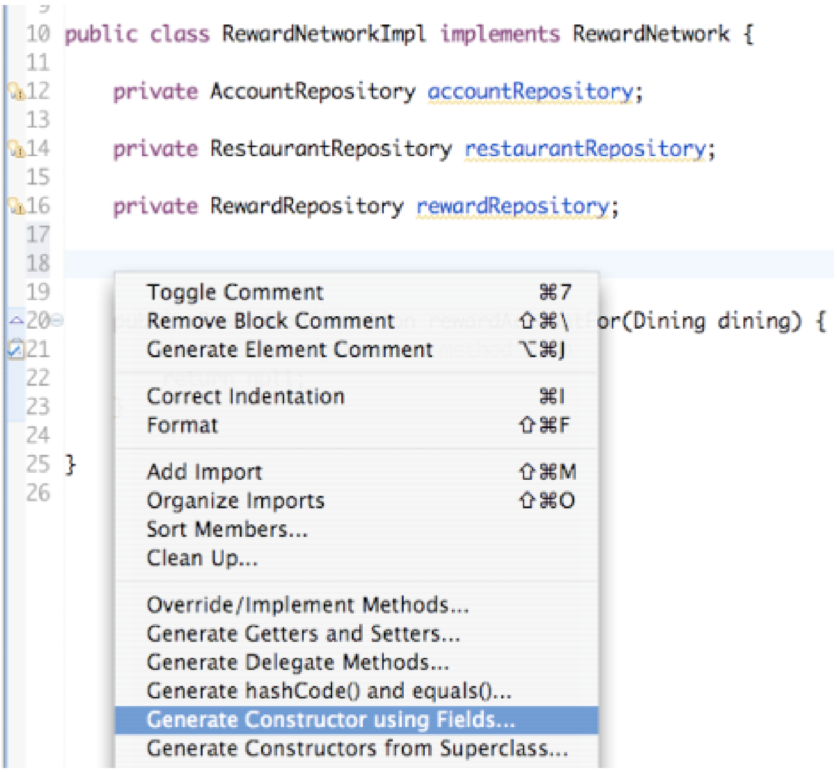
```
package rewards.internal;

import rewards.Dining;

public class RewardNetworkImpl implements RewardNetwork {

    private AccountRepository 

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

}

```

accountRepository   AccountRepository
repository   AccountRepository
Press '^,' to show Template Proposals

Figure A.4. Field name auto-completion

A.5. Generating Constructors From Fields

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



```
public class RewardNetworkImpl implements RewardNetwork {

    private AccountRepository accountRepository;

    private RestaurantRepository restaurantRepository;

    private RewardRepository rewardRepository;


                                                     or(Dining dining) {




}
```

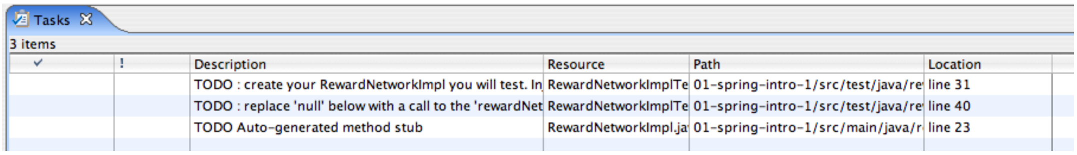
Toggle Comment ⌘7
Remove Block Comment ⇧⌘\
Generate Element Comment ⌥⌘J
Correct Indentation ⌘I
Format ⇧⌘F
Add Import ⇧⌘M
Organize Imports ⇧⌘O
Sort Members...
Clean Up...
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate hashCode() and equals()...
Generate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

Figure A.5. Generating Constructors

A.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.

A.7. Tasks View



Tasks ☒

3 items

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

Figure A.6. 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.

A.8. Rename a File

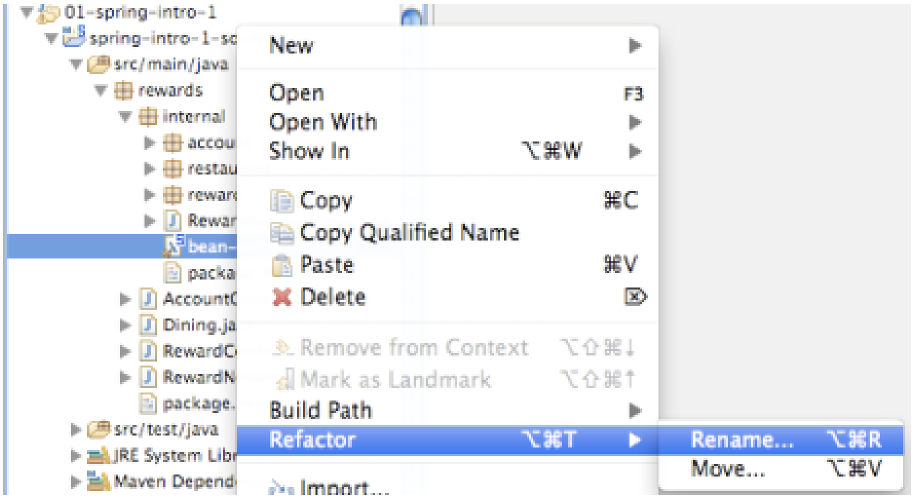
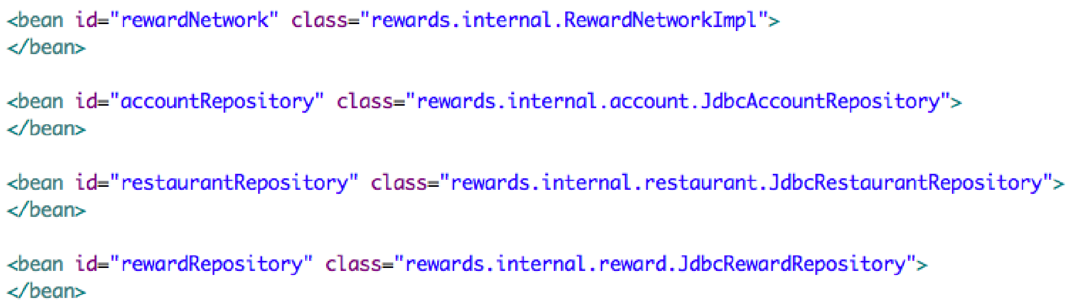


Figure A.7. Renaming a Spring configuration file using the Refactor command

Appendix B. Spring XML Configuration Tips

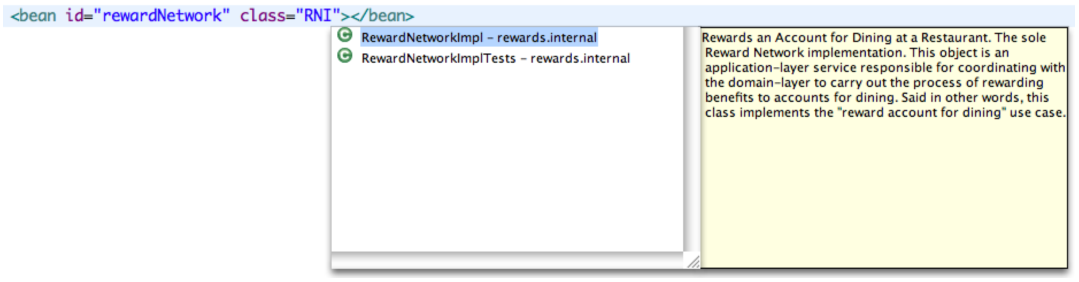
B.1. Bare-bones Bean Definitions



```
<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>
```

Figure B.1. Bare-bones bean definitions

B.2. Bean Class Auto-Completion



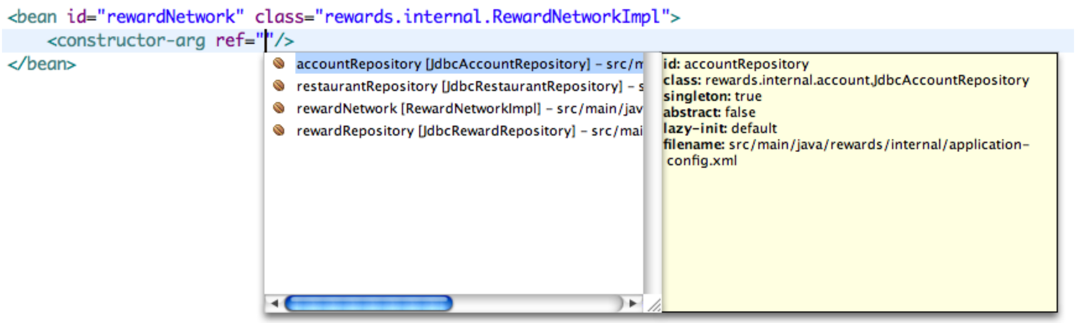
```
<bean id="rewardNetwork" class="RNI"></bean>
```

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.

Figure B.2. Bean class auto-completion

B.3. Constructor Arguments Auto-Completion



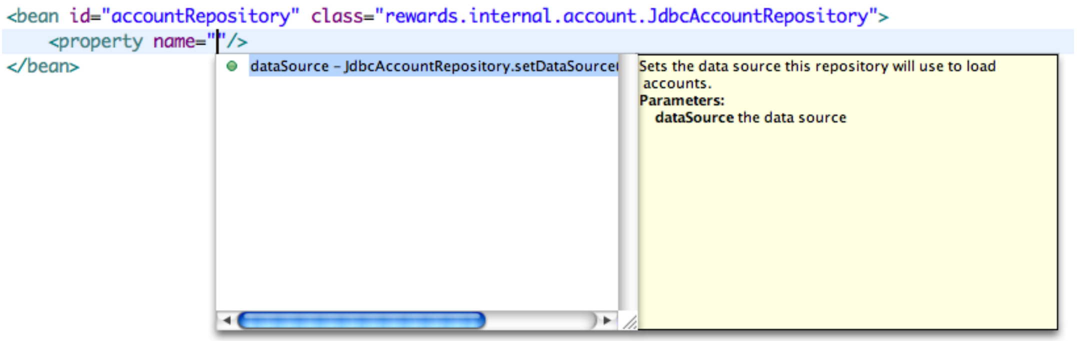
```
<bean id="rewardNetwork" class="rewards.internal.RewardNetworkImpl">
    <constructor-arg ref=""/>
</bean>
```

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

Figure B.3. Constructor argument auto-completion

B.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

Figure B.4. Bean property name completion

Appendix C. JUnit

C.1. JUnit Testing Framework

JUnit is a regression testing framework, useful for both unit testing and integration/system testing. See the JUnit website for more information.