48 of 48 lessons completed

100%

Annotations and Component Scanning Lab

Purpose

In this lab you will gain experience using the annotation support from Spring to configure the `Rewards` application.

You will use an existing setup and transform that to use annotations such as `@Autowired`, `@Repository` and `@Service` to configure the components of th application. You will then run a top-down system test that uses JUnit.

Learning Outcomes

What you will learn:

1. How to use component scanning with annotations
2. The advantages and drawbacks of those annotations
3. How to implement your own bean lifecycle behaviors

Specific subjects you will gain experience with:

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

You will be using the *16-annotations* project.

Estimated time to complete: 45 minutes.

Prerequisites

The prerequisites are included as part of the *Introduction to the Spring Professional Learning Path* course *Lab Setup* lesson.

If you already completed it, you should be ready to do this lab. If not, assuming you already have JDK 11 or 17 installed and Java IDE, you will need to do following:

1. Download the **lab codebase zip file**.
2. Once you have downloaded the file, unzip it under a directory of your choice. The unzipped directory `core-spring-labfiles/lab` contains the lab projects.
3. Run `./mvnw clean install` (if you plan on using Maven as your build tool).

Use Case

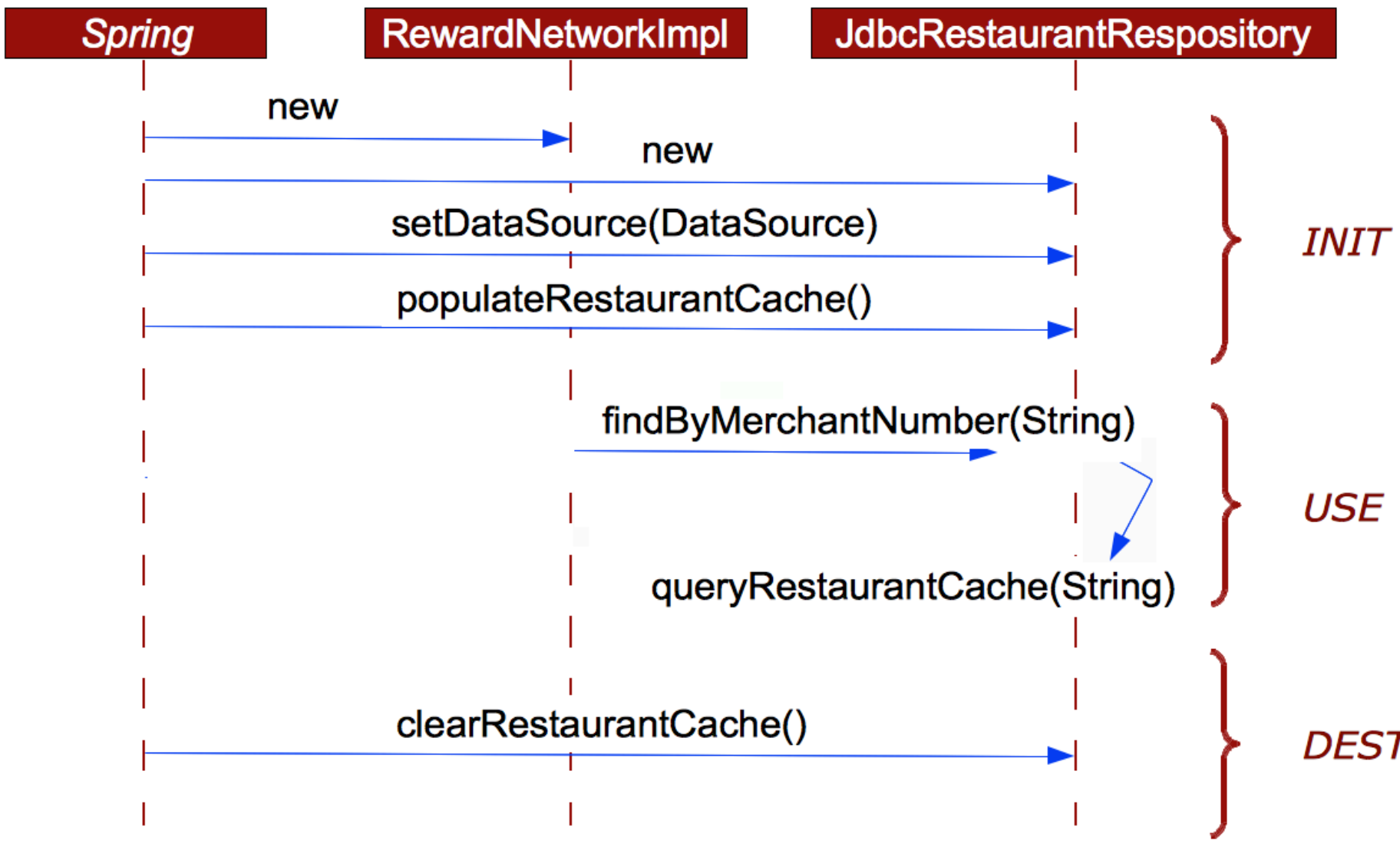
In this lab, we are using a version of the `Rewards` application that is already fully functional. It is the completed version of the previous lab.

It has repository implementations that are backed by JDBC, which connect to an in-memory embedded HSQLDB database. You will then rewrite some of application code to make use of annotations.

You will also leverage Spring lifecycle methods to sequence the Rewards Restaurant repository cache creation and clearing.

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

Figure 1 JdbcRestaurantRepository life cycle



Quick Instructions

If you are already knowledgeable with the lesson concepts, you may consider jumping right to the code, and execute the lab in form of embedded TODO comments. Instructions on how to view them are at the *Using TODO Tasks* article.

If you aren't sure, try the TODO instructions first and refer to the lab instructions by TODO number if you need more help.

Instructions

Verify the Integration Test

TODO-01: Verify the integration test

- The project features an integration test that verifies the system's behavior. It's called `RewardNetworkTests` and lives in the `rewards` package.
- Run this test and verify the test runs successfully.
- Open the application configuration class `RewardsConfig`. Review the `@Beans` that wires up all the dependencies. As you can see, we're using const arguments.

  Remember that the infrastructure components (the `DataSource` for example) are located in a separate configuration class.
- Navigate back to `RewardNetworkTests` and review the `setUp()` method. It specifies the `TestInfrastructureConfig.java` infrastructure configur file.

Dependency Injection Using Annotations and `@ComponentScan`

You will refactor the application to use annotation based configuration.

TODO-02: Remove bean factory configuration

In the `RewardsConfig` class:

- Remove the `@Bean` methods for all beans.
- Remove the `@Autowired DataSource`.

The resulting class should contain no methods and no variables.

- Re-run the test.
- It should fail now.
- What just happened?

  Spring has no idea how to inject the dependencies anymore, since you have removed the configuration directive. Next, we'll start adding configurati metadata using stereotype annotations and the `@Autowired` annotation.

TODO-03: Annotate `RewardNetworkImpl` and wire its dependencies

- Open the `RewardNetworkImpl` class and annotate it with the `@Service` stereotype.
- Why?

  The `RewardNetworkImpl` class is not a repository or controller, and is reflective of domain logic that could be exposed as a service. You could have well annotated with `@Component`. All stereotype annotations derive from `@Component`. The `@Component` annotation is what Spring component scanni looks for to create Spring beans during application startup.
- Annotate the constructor with `@Autowired` (constructor injection) or you can annotate the individual private fields with `@Autowired` (field injection)
  - Contructor injection is highly preferred over field injection
  - In constructor injection, if there ia a single constructor, the usage of `@Autowired` is optional

TODO-04: Annotate `JdbcRewardRepository` and wire its dependencies

- Open the `JdbcRewardRepository` class and annotate it with the `@Repository` stereotype annotation.
- Mark the `setDataSource()` method with that same `@Autowired` annotation. This will tell Spring to inject the setter with a instance of a bean matchi `DataSource` type.

TODO-05: Annotate `JdbcAccountRepository` and wire its dependencies

- Open the `JdbcAccountRepository` class
- Annotate it as a `@Repository`
- Annotate the `setDataSource()` method with `@Autowired`.

TODO-06: Annotate `JdbcRestaurantRepository` and wire its dependencies

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

TODO-07: Set up component scanning

Although our classes are now properly annotated, we still have to tell Spring to search through our Java classes to find the annotated classes and carry c configuration.

- Open the `RewardsConfig` class and add the `@ComponentScan("rewards.internal")` annotation.

  This annotation turns on a feature called component scanning which looks for all classes annotated with annotations such as `@Component`, `@Reposi` or `@Service` and creates Spring beans from those classes. It also enables detection of the dependency injection annotations. The "rewards.internal argument is the base package that we want Spring to look from, this will keep Spring from unnecessarily scanning all `org.*` and `com.*` packages c classpath.
- Re-run the test. It should pass. If it does not, check your work.

Implement Init and Destroy Callbacks

If you recall the design described in *Rewards Application* from the *Introduction to the Spring Professional Learning Path* course *Rewards Reference Dom* lesson, restaurant data is read often but rarely changes. You can browse `JdbcRestaurantRepository` and see that it has been implemented using a sin cache. `Restaurant` objects are cached to improve performance (see methods `populateRestaurantCache` and `clearRestaurantCache` for more detail

Review the The JdbcRestaurantRepository life-cycle.

The cache works as follows:

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

Initialize

- Open `JdbcRestaurantRepository` in the `rewards.internal.restaurant` package.
- Notice that we are using the constructor to inject the dependency.
- Run the test `RewardNetworkTests` and verify it passes.
- What if you had decided to use *setter injection* instead of *constructor injection*? It is interesting to understand what happens then.

TODO-08: Use Setter Injection

- Change the dependency injection style from constructor injection to setter injection: Move the `@Autowired` from the constructor to the `setDataSour` method.
- Execute `RewardNetworkTests` to verify.
- It should fail and you should see a `NullPointerException`. Why did the test fail? Investigate the stack-trace to see if you can determine the root ca

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

TODO-09: Handle `populateRestaurantCache` on `@PostConstruct`

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

It is arguable that `populateRestaurantCache` should never have been in the constructor, since it goes beyond constructing the object to running applica code. Using `@PostConstruct` is a better approach.

Destroy

Your test seems to run fine, let us now have a closer look.

TODO-10: Add print statement

- Open `JdbcRestaurantRepository` and add a simple print statement in `clearRestaurantCache` so we can see when it is being run:

```
System.out.println("clearRestaurantCache invoked");
```

TODO-11: Handle `clearRestaurantCache` on `@PreDestroy`

- Re-run `RewardNetworkTests` - check the console output.

- Notice that your `clearRestaurantCache invoked` message was not generated so `clearRestaurantCache` is not called, which means that your ca never cleared.
- Add an annotation to mark this method to be called on shutdown.
- Save your work and run `RewardNetworkTests` one more time.
- You should now see `clearRestaurantCache invoked` output to the console.

Later in this course, you will learn that there is a more elegant way to work with JUnit. By using Spring's Testing support, an `ApplicationContext` can ac be created automatically so you do not have to do it by hand.

Summary

Your repository is being successfully integrated into your application, and Spring is correctly issuing the lifecycle callbacks to populate and clear your cac Good job!

Summary **Instructors**

In this lab you gained experience using Spring Component Scanning to configure the completed reference domain.

You used Spring to configure the pieces of the application, then run a top-down system test to verify application behavior.