INSTRUCTIONS

Dependency Injection with Java Configuration

Purpose

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

Learning Outcomes

What you will learn:

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

Specific subjects you will gain experience with:

1. Spring Java configuration syntax
2. Spring embedded database support

You will be using the *12-javaconfig-dependency-injection* project.

Estimated time to complete: 45 minutes

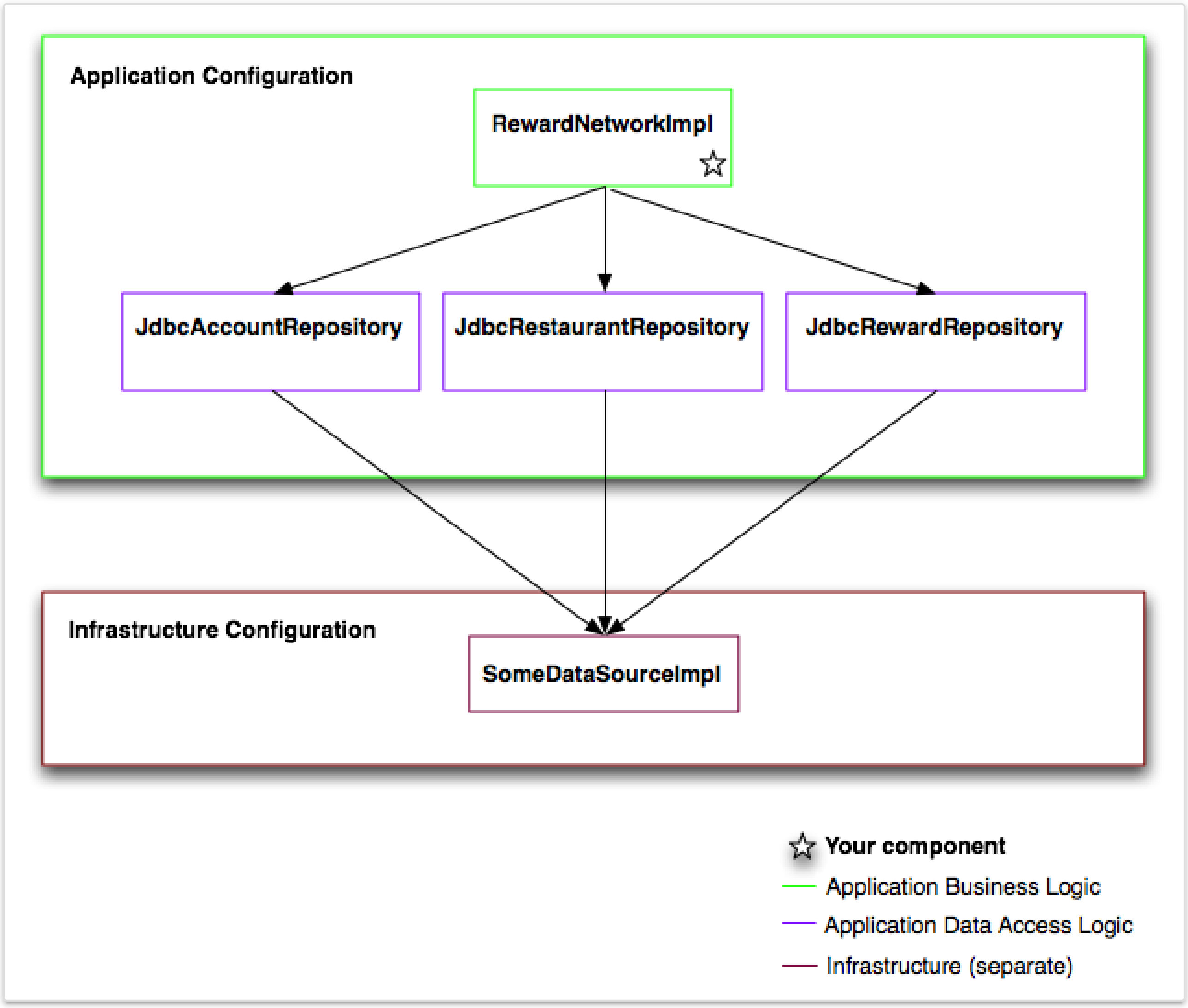
Use Case

In the previous exercise you have coded your `RewardNetworkImpl`, the central piece of this reward application. You have unit tested it and verified it works in isolation with dummy (stub) repositories.

Now it is time to tie all the *real* pieces of the application together, integrating your code with supporting services that have been provided for you. In the following steps you will use Spring to configure the complete rewards application from its parts. This includes plugging in repository implementations that use a JDBC data source to access a relational database.

The following diagram illustrates the parts of the rewards application you will configure and how they should be wired together:

Figure 1 Rewards Application Configuration


Application Configuration
RewardNetworkImpl
JdbcAccountRepository
JdbcRestaurantRepository
JdbcRewardRepository
Infrastructure Configuration
SomeDataSourceImpl
Your component
Application Business Logic
Application Data Access Logic
Infrastructure (separate)

It is split into two categories: **Application Configuration** and **Infrastructure Configuration**.

The components in the **Application Configuration** box are written by you and makeup the application logic.

The components in the **Infrastructure Configuration** box are not written by you and are lower-level services used by the application.

In the next few steps you will focus on the application configuration piece. You will define the infrastructure piece later.

In your project, you will find your familiar `RewardNetworkImpl` in the `rewards.internal` package.

You will find each of the JDBC-based repository implementations it needs, located with their domain objects, inside the `rewards.internal` package.

Each repository uses the JDBC API to execute SQL statements against a DataSource that is part of the application infrastructure. The DataSource implementation you will use is not important for this exercise but will become important later.

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 here.

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

Implement Application Configuration

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

TODO-01: Enable the Spring Java configuration

- Under the `src/main/java` folder, find the `config` package and open the class `RewardsConfig`.

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

- You will however add some code and Spring annotations to it to create the result illustrated in the `RewardsConfig` section of the Rewards application configuration.

- Place a `@Configuration` annotation on the `RewardsConfig` class.
- Why did you do this step?

  The `@Configuration` tells Spring to treat this class as a set of configuration instructions to be used when the application is starting up.

TODO-02: Create factory methods

- Within the `RewardsConfig` class, define four methods annotated with the `@Bean` annotation.
- Each method should instantiate and return one of the beans in the illustration, `accountRepository`, `restaurantRepository`, `rewardRepository`, and `rewardNetwork`. For example, you should create an `accountRepository()` method that instantiates `JdbcAccountRepository` and returns it.
- Looking back at the illustration, you can see that each of the three repositories has a dependency on a `DataSource` that will be defined elsewhere. This means in each repository method you must make a call to the repository's `setDataSource()`, passing in a reference to the `dataSource`.
- Why did we do this step?

  You need to build bean factories for each Spring bean you will use in your application.

  While you will likely use *Annotation-based Configuration* (that you will see in the *Component Scanning* module), *Java Configuration* is the most flexible option that carries the spirit of Factory or Builder type patterns. You will see this recurring in Spring Boot Auto-configuration projects, as well as projects where you may want to create beans that you cannot annotate. Such examples include 3rd party code, or legacy code you are not allowed to modify.
- But where will you get the `DataSource` from when it is defined in another file (in our case in `TestInfrastructureConfig`)? You will answer in next step.

TODO-03: Wire the `DataSource`

- The class already has a `DataSource` instance variable defined for you, but you need to give it a value.
- Define a constructor for `RewardsConfig ` that accepts a `DataSource` and sets member `dataSource`.
- Spring will automatically provide a reference to the DataSource defined in the `TestInfrastructureConfig` class, assuming both configuration files are specified at startup.

- Why did you do this step?
  - You want to abstract the data source, to make it easy to test.
  - Because this is the only constructor on the class, `@Autowired` is not needed.

    Spring will call it automatically when asked to create an instance of `RewardsConfig`.
- **Constructor Injection**: You just saw constructor injection in play. You will see it used heavily in modern Spring code. Constructor injection is favored over field injection for several reasons:
  - Easier to test.
  - Safer than field injection, particularly when forcing immutability of injected members through `final`.
  - Ability to decouple domain POJOs from Spring.

TODO-04: Implement factory methods

- Implement the factory methods you created in TODO-02: Create factory methods.
- You should be aware that Spring will assign each bean an ID based on the `@Bean` method name.

  The instructions below will assume that the ID for the `RewardNetwork` bean is `rewardNetwork`.

  Therefore, for consistency with the rest of the lab, give your `RewardNetworkImpl` `@Bean` method the name `rewardNetwork`.
- As you define each bean, follow bean naming conventions.
- The arrows in the Rewards application configuration representing bean references follow the recommended naming convention.
- For best practices, a bean's name should describe the *service* it provides.
  - It should not describe implementation details.
  - For this reason, a bean's name often corresponds to its *interface*.

    For example, the class `JdbcAccountRepository` implements the `AccountRepository` interface. This interface is what callers work with.

    By convention, then, the bean name should be `accountRepository`. Similarly each bean method should return an interface not a concrete class.

So the `accountRepository()` method should return `AccountRepository` *not* `JdbcAccountRepository`.

Implement and Run Unit Test

TODO-05: Test the configuration class

- Open `RewardsConfigTests` - in the `config` package under `src/test/java`.
- Note that a mock `dataSource` has been defined already.
- Uncomment the code between `/\*` and `*/`.
- Ensure `RewardsConfigTests` compiles correctly.

  If not, fix `RewardsConfig` as necessary.

  Make sure your `@Bean` methods have the right names, return the right implementations and that their signatures define *interfaces* as return types.
- Run the JUnit test from your IDE. If you are not familiar with how to run JUnit test from your IDE, see these appendix instructions.
- Once the test passes, move on to the next step!
- Why did we do this step?
  - To unit test the `RewardsConfig` class independently of Spring.

Implement and Run Integration Tests

TODO-06: Test infrastructure configuration

- In the previous step you visualized bean definitions for your application components. In this step you will investigate the infrastructure configuration necessary to test your application.
- To test your application, each JDBC-based repository needs a `DataSource` to work.

  For example, the `JdbcRestaurantRepository` needs a DataSource to load `Restaurant` objects by their merchant numbers from rows in the `T_RESTAURANT` table.

  So far, though, you have not defined any `DataSource` implementation. In this step you will see how to setup a `DataSource` in a separate configuration file in your test tree. It’s in the test area, because it is only for testing - it is not the one you would use in production.
- In the `src/test/java` source folder, navigate to the root `rewards` package. There you will find the `TestInfrastructureConfig` class. Open it.

- You will see that a `DataSource` is already configured for you. You don't need to make any changes to this bean, but you do need to understand what is defined for you here.

TODO-07: Import `RewardsConfig` to `TestInfrastructureConfig` class

- This `TestInfrastructureConfig` class will also serve as the master configuration class for our upcoming test.
- To have it serve in the master configuration class role, add an `@Import` to the class to reference the `RewardsConfig.class`.
- Why did we do this step?
  - To import `RewardsConfig`, Spring will create it as a Spring bean and automatically call its constructor passing in the `DataSource` created by `TestInfrastructureConfig`.
- Spring ships with support for creating a DataSource based on in-memory databases such as H2, HSQLDB and Derby. The code you see is a quick way to create such a database.
- The `EmbeddedDatabaseBuilder` references external files that contain SQL statements. These SQL scripts will be executed when the application starts. Both scripts are on the classpath, so you can use Spring's resource loading mechanism and prefix both of the paths with `classpath:`.

  Note that the scripts will be run in the order specified (top to bottom) so the order matters in this case.
- The `EmbeddedDatabaseBuilder` is also an example of a Fluent API builder pattern, which is common in modern Spring applications.

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

TODO-08: Create the system test class

- Start by creating a new JUnit Test Case called `RewardNetworkTests` in the `rewards` package inside the `src/test/java` source folder. If you are not familiar with how to generate a JUnit test from your IDE, see these appendix instructions.
- Use your IDE Test Case generation feature to set up a new Test Case for `RewardNetwork`.
- Also note that you might need to change the version of JUnit that will be used to JUnit Jupiter

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

TODO-09: Implement the test setup logic

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

- First, notice that you have a `public void setUp()` method annotated with `@org.junit.jupiter.api.BeforeEach` - this may have been done for you if you used the JUnit test case wizard.

  `@BeforeEach` is the JUnit 5 equivalent of JUnit 4's `@Before`.

- Within `setUp()` call `SpringApplication.run`, providing it the `TestInfrastructureConfig.class` that you want to load.

  Doing this will bootstrap your application by having Spring create, configure, and assemble all beans defined in the two configuration files (since one imports the other).

- Ask the context to get the `rewardNetwork` bean for you, which represents the entry-point into the rewards application.

- Assign the bean to a private field of type `RewardNetwork` you can reference from your test methods.

  Be sure to assign the reference to the `rewardNetwork` bean to a field of type `RewardNetwork` and not `RewardNetworkImpl`. A Spring `ApplicationContext` encapsulates the knowledge about which component implementations have been selected for a given environment. By working with a bean through its interface you decouple yourself from implementation complexity and volatility.

- Don't ask the context for beans "internal" to the application.

- The `RewardNetwork` is the application's entry-point, setting the boundary for the application defined by an easy-to-use public interface. Asking the context for an internal bean such as a repository or data source is questionable.

TODO-10: Implement the test logic

- Verify that Spring can successfully create your application on test setup. To do this, modify the `test()` method:

  - Rename the method to `testRewardForDining`
  - Leave the method body blank for now.

- Run your test class from your IDE.

- After your test runs, you should see sucessful test result indicating `setUp()` ran without throwing any exceptions.
- If you see red, inspect the failure trace in the JUnit view to see what went wrong in the setup logic. Carefully inspect the stack trace - Spring error messages are usually very detailed in describing what went wrong. The most useful information is usually at the *bottom* of the stack trace, so you my need to scroll down to see it.
- Once you have successful test, move on to the last step!

TODO-11: Run the tests

Now the test setup logic works you can modify our test to invoke the `RewardNetwork.rewardAccountFor(Dining)` method and verify that all the pieces of your application work together to carry out a successful reward operation.

- You will not have to write the Unit Test yourself.
- Have a look at `RewardNetworkImplTest.testRewardForDining()`.
- You can just copy and paste its content into `RewardNetworkTests.testRewardForDining()`.
- Notice that in a real life application you would not have the same content for both tests. You are making things fast so you can focus on Spring configuration rather than spending time on writing the test itself.
- Rerun your test. If it fails, check that you didn't forget the `@Import` in `TestInfrastructureConfig` (this was **TODO-07**)
- When you have successful test, congratulations! You have completed this lab.

Summary

You have just used Spring to configure the components of a realistic Java application and have exercised application behavior successfully in a test environment inside your IDE.