## Spring Boot

7 of 45 lessons completed

16%



# Spring Boot - A Closer Look Lab

# # Purpose

In this lab you will go a level deeper into core Spring Boot concepts while "Bootifying" a previous version of the Rewards application.

# # Learning Outcomes

What you will learn about Spring Boot:

1. Dependency Management
2. Auto-configuration
3. Packaging & Runtime
4. Spring Boot Testing

Specific subjects you will gain experience with:

1. Spring Boot Starter Bill-of-Materials (BOMs)
2. Datasource Auto-configuration
3. Application deployment artifact packaging
4. Simple application runtime with `CommandLineRunner`
5. Simple `SpringBootTest`

You will be using the *32-jdbc-autoconfig* 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

Take a Spring application using JDBC and convert it to a Spring Boot application.

The application builds the data access layer using a `Datasource` and a `JdbcTemplate`. JDBC implementations of the `AccountRepository`, `CustomerRepository` and `RewardRepository` have been implemented for you.

You will alter that application in three ways:

1. *Bootify* the application, meaning you will wrap the application with Spring Boot, and demonstrate how Spring Boot simplifies your development expe through starters and auto-configuration.
2. *Bootify* the application's integration test.
3. Add a `CommandLineRunner` to demonstrate how Spring Boot can package and run an application with no additional runtime dependencies.
4. Demonstrate how to exclude an auto-configuration.
5. Demonstrate external configuration.

# # Getting Started

You will start with the `32-jdbc-autoconfig` project.

You will also need a Terminal or Command window to run Maven or Gradle manually - IntelliJ/Eclipse/STS support a terminal window within the IDE.

Navigate to the parent directory containing the course projects (this is the `lab` directory of the code you cloned/unzipped from github).

# # 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

## Review of a Spring App – Dependencies

- Navigate to the project, and run the full suite of tests to verify the build is successful and tests pass.
- Examine the `pom.xml` or `build.gradle` file of the project. What do you see for the project dependencies?
  - You should see a set of libraries covering Spring Framework, JDBC and HSQLDB database
  - You should also see a set of libraries for Testing

## Bootify Your Spring App

In this section we will wrap our app with Spring Boot.

### Refactor to Starters

Part of the value point of Spring Boot is *simplified dependency management*. In this section, you will refactor from discrete Spring Framework and 3rd pa dependencies to the Spring Boot Starters.

- In your Terminal/Command window, run Maven goal or Gradle task for displaying dependencies. (The example commands below assume you are r them at the `lab` directory.)

```
Windows: mvnw -pl *-common -pl *jdbc-autoconfig dependency:tree -Dincludes=org.springframework
Linux/MacOS: ./mvnw -pl *-common -pl *jdbc-autoconfig dependency:tree -Dincludes=org.springframework
```

```
Windows: gradlew :32-jdbc-autoconfig:dependencies --configuration compileClasspath -Dincludes=org.springframework
Linux:MacOS: ./gradlew :32-jdbc-autoconfig:dependencies --configuration compileClasspath -Dincludes=org.springframew
```

- What dependencies do you see for the `jdbc-autoconfig` project?
  - You should see dependencies for Spring JDBC and Spring Test and all their transitive dependencies.
  - You will also see the `spring-boot-starter` as this is required for `SpringApplication.run()` whether you use the rest of Spring or not.

**TODO-01** : Add Spring Boot Plugin:

- Add the Spring Boot plugin to the project's `pom.xml` or `build.gradle` file

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

```
apply plugin: "org.springframework.boot"
```

What does the Spring Boot plugin do for you?

You will see, as we build the project, Spring Boot plugin will generate the runtime deployment artifact for you through the `repackage` goal for Maven the `bootJar` task for Gradle. The `repackage` goal is run as part of the Maven `package` goal and the `bootJar` task is run as part of Gradle `assemble`

You will see this in action later in this lab.

**TODO-02** : Refactor to Spring Boot Starters:

1. Remove the Spring JDBC and Spring Test dependencies from the project's `pom.xml` or `build.gradle` file.
2. Add the Spring Boot Starters for:
    - JDBC
    - Testing (only for Maven)

    Hint: If you need help to understand the starter dependencies, take a look at the *Spring Boot Intro* lab `pom.xml` or `build.gradle` that was generate the Spring Initializr.

- Rerun the maven `dependency:tree` Maven goal or `dependencies` Gradle task for the project:
- What dependencies do you see now?
    - You should now see the Boot starter dependencies instead.

Typically your Spring Boot `pom.xml` or `build.gradle` file is simpler than the original but this example is sufficiently simple that there is little difference.

## Create a Spring Boot Application

Start with creation of the Spring Boot application:

**TODO-03** : Turn the `RewardsApplication` into a Spring Boot application:

1. You already have a `RewardsApplication` class provided for you. This is the shell for the Spring Boot application.
2. Annotate the `RewardsApplication` class accordingly.

    What exactly does the annotation do?

    **Hint:** In your IDE, look at a decompiled version of the `@SpringBootApplication` annotation. (Or look at the Javadoc of the annotation.)
3. Notice that a `main()` method has already been defined and the Spring boot classes initialized using `SpringApplication.run`

**TODO-04** : Let Spring Boot execute database scripts

1. Move the `schema.sql` and `data.sql` files in the `src/test/resources/rewards.testdb` directory to the `src/main/resources/` directory.

    Why are you doing this?

    - You are refactoring the directory structure to the default that Spring Boot expects for its life-cycle initialization - specifically for automatic databa initialization.

In the non-Spring Boot version, the `EmbeddedDatabaseBuilder` in `SystemTestConfig` can specify where the SQL initialization files are found.

In Spring Boot applications, the default files are `schema.sql` and `data.sql` and they must be in the classpath root.

> **Note:** You may choose to specify these files using properties (as described in the slides).

- The starting point of the project does not have a runner, so the files originally were provided to set up a test data fixture.

  You are moving the files from `test/resources` to `main/resources` so you can use the same files in the application runtime to demonstrate th `CommandLineRunner`.

**TODO-05** : Setup a command line runner to print the Reward account count:

1. Implement a `CommandLineRunner` in your `RewardsApplication` class. Remember it must be configured as a Spring bean.
2. Add code to use a `JdbcTemplate` to query the number of accounts using `SQL` query string already declared in the class. Request the result as a `Lor` assign it to `numberOfAccounts`. Remember Spring Boot will auto-configure a `JdbcTemplate` bean for you automatically.
3. Use the provided `logger` to log the returned number of accounts at `info` level- something like following:

```
logger.info("Number of accounts: {}" , numberOfAccounts);
```

The Spring Boot `CommandLineRunner` and `ApplicationRunner` abstractions are guaranteed to run *at most once* before `SpringApplication.run()` me returns. Multiple runners may be configured, and can be ordered with the `@Order` annotation.

## Capture Properties Into a Class

**TODO-06** : Use `@ConfigurationProperties` to capture properties

Spring Boot `@ConfigurationProperties` allows developer to map properties, especially properties with hierarchical structure, into a class.

1. Note that `application.properties` file already contains the following properties

```
rewards.recipient.name=John Doe
rewards.recipient.age=10
rewards.recipient.gender=Male
rewards.recipient.hobby=Tennis
```

2. Annotate `RewardsRecipientProperties` class with `@ConfigurationProperties` with `prefix` attribute set to `rewards.recipient`
3. Create fields (along with needed getters/setters) that reflect the properties above in the `RewardsRecipientProperties` class

```
@ConfigurationProperties(prefix = "rewards.recipient")
public class RewardsRecipientProperties {

    private String name;
    private int age;
    private String gender;
    private String hobby;

    // getter and setter methods

}
```

4. Now use one of the 3 schemes below to enable `@ConfigurationProperties` (Feel free to try all of them.)
   1. Add `@EnableConfigurationProperties(RewardsRecipientProperties.class)` to `RewardsApplication` class
   2. Add `@ConfigurationPropertiesScan` to `RewardsApplication` class (This is supported from Spring Boot 2.2.1)
   3. Annotate the `RewardsRecipientProperties` class with `@Component`
5. Implement a new command line runner that displays the name of the rewards recipient

```
@Bean
CommandLineRunner commandLineRunner2(RewardsRecipientProperties rewardsRecipientProperties) {
    return args -> System.out.println("Recipient: " + rewardsRecipientProperties.getName());
}
```

## Run Your Spring Boot Application

**TODO-07 (Optional)** : In your IDE, run the application:

1. Add `debug=true` property to your `application.properties`. This causes Spring Boot to log everything it does and what auto-configuration choice does and does not make.
2. Run the application. What do you see on the console?
3. You should see your log output from the command line runner:

```
INFO : rewards.RewardsApplication - Number of accounts:21
```

You should also see an *Auto-Configuration Report*, that is prefixed as follows:

```
============================
CONDITIONS EVALUATION REPORT
============================

Positive matches:
-----------------
```

Do you see `DataSourceAutoConfiguration` under `Positive matches`? Now you know where the `DataSource` came from.

Do you also see `JdbcTemplateAutoConfiguration` under `Positive matches`?

## Bootify Your Integration Test

**TODO-08** : Disable explicit DataSource creation in `SystemTestConfig`:

1. Open `SystemTestConfig.java`.
2. Stop Spring invoking the `dataSource()` bean factory method by commenting out the `@Bean` annotation.

**TODO-09** : Return to your IDE and refactor `RewardNetworkTests` into a Spring Boot Integration test:

1. Run this test without making any change, it will fail. It fails because the Spring Boot auto-configuration is not enabled when the test is run.
2. Remove the `@ContextConfiguration` and `@ContextConfiguration(classes = {SystemTestConfig.class})` annotations.
3. Add `@SpringBootTest` annotation to run as a Spring Boot Test
    - There is no need to specify the configuration class to use as the annotation will *automatically* component scan for any `@Component` (or `@Configuration`) classes in the current package or below. Since the `SystemTestConfig` class is in the same package, it will be discovered a processed. This includes processing the `@Import` annotation that references the `RewardsConfig` class containing all the other bean definitions
    - This is considered an *End-To-End* integration test, including the wired components.

      Note that in a real production application you are most likely to configure an external database. Spring Boot offers properties to do this.
4. Run the `RewardNetworkTests` and verify it succeeds.
5. Do an experimentation
    - Specify the configuration class with `@SpringBootTest` like following:

```
@SpringBootTest(classes={SystemTestConfig.class})
```

- Run the test and observe that it fails. Think about why it fails.

  The failure occurs because Spring Boot autoconfiguration is disabled: when you specify the configuration class, `@SpringBootTest` stops searc for configuration class annotated with `@SpringBootConfiguration`, which contains `@EnableAutoConfiguration`.

- Revert the change and verify the test succeeds again

You will dig deeper into Spring Boot Testing in a later unit.

## Override Auto-Configuration

You have seen to this point that Spring Boot will detect and automatically configure a `DataSource` on your behalf. But how would this work if you needed configure *multiple* databases from your application? In this case auto-configuration cannot really help you.

You have two options to handle this situation:

1. Use the default `DataSource` auto-configuration with supporting default configuration, and also explicitly set additional `DataSource` beans with differe names, as specified by the `@Qualifier` annotation. You can also set the order of precedence using the `@Order` annotation.
2. Disable auto-configuration for `DataSource`, and explicitly declare multiple DataSource beans using Java Configuration.

We will use option 2 and disable the `DataSource` auto-configuration both programmatically and using configuration properties.

### Disable DataSource Auto-Configuration Programmatically

Do the following steps if you have extra time.

**TODO-10 (Optional)** : Switch to explicit `DataSource` configuration:

1. Add `DataSource` bean explicitly in the `RewardsConfig` class by by uncommenting the code.
   - Note the debug log message, so we can tell if this method is being used.
2. Remove the code that injects `DataSource` bean since we no longer need it.
3. Fix any compile errors in the `RewardsConfig` class.
4. Notice this reverts to the standard Spring way of building a `DataSource`.

**TODO-11 (Optional)** : Disable `DataSource` auto-configuration:

1. Annotate the `@SpringBootApplication` to *exclude* the `DataSource` auto-configuration - `@SpringBootApplication` has an `exclude` attribute.
   - The bean to exclude is `DataSourceAutoConfiguration` class you were told to remember earlier
2. Import the `RewardsConfig` configuration
   - This is required since the `RewardsConfig` configuration now provides DataSource bean and will not be auto-detected through component scanning

---

*Note*: Technically you don't have to disable data-source auto-configuration given that Spring Boot will use application defined `DataSource` bean over auto configured one.

**TODO-12 (Optional)** : Turn on debug level logging and rerun:

1. In your `src/main/resources/application.properties` configuration modify the `logging.level.config` property to `DEBUG`.
2. Run the 'RewardNetworkTests' test.
   - Do you see your `dataSource` creation debug log output?

     If so you just proved your `dataSource` was generated in your Java config.
   - Do you see the `DataSourceAutoConfiguration` is no longer matched?

     Look for the following console output:

```
Exclusions:
-----------

org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```

*Troubleshooting*: If you experience `BeanCurrentlyInCreationException`, think about why that is the case and how to solve it.

*Notes for Thought*: When you *Bootified* your application, did you auto-wire a `JdbcTemplate`? If you did, what might be some implications for a multi-datal solution? Would you need to remove use of an auto-wired JdbcTemplate? How would you do this?

## Build and Run Using Command Line Tools – Optional

Do this part of the lab only if you have extra time.

**TODO-13 (Optional)** : Let's see what the Spring Boot Maven/Gradle plugin is doing:

1. From either your IDE or your Terminal/Command window, execute the Maven `package` goal or Gradle `assemble` task.

   The following leverages the parent project multi-module build with the maven/gradle wrapper, and is executed from the project root `lab` directory.

   Notice that, when using Maven, it is necessary to skip the tests using the `-Dmaven.test.skip=true` flag due to the fact that you have not yet Booti testing portion of the project.

   ```
   Windows: mvnw clean package -pl *common -pl *jdbc-autoconfig -Dmaven.test.skip=true
   Linux/MacOS: ./mvnw clean package -pl *common -pl *jdbc-autoconfig -Dmaven.test.skip=true
   ```

   ```
   Windows: gradlew :32-jdbc-autoconfig:clean :32-jdbc-autoconfig:assemble
   Linux/MacOS: ./gradlew :32-jdbc-autoconfig:clean :32-jdbc-autoconfig:assemble
   ```

2. In `32-jdbc-autoconfig`, a `target` directory (for Maven), or a `build/libs` directory (for Gradle) should now exist. Review its contents, what do you

   - You should see two generated JAR files:
     - `32-jdbc-autoconfig-5.3.23.jar` and
     - `32-jdbc-autoconfig-5.3.23.jar.original` (for Maven)
     - `32-jdbc-autoconfig-5.3.23-original.jar` (for Gradle)

     Notice that the "original" is *much* smaller. The other JAR is executable and contains all the necessary dependencies (hence it is called a "fat" J

3. Extract the jar file to a temporary directory, and view the contents using `jar`.

   Windows:

   ```
   mkdir temp
   copy *jdbc-autoconfig\target\*.jar temp (for Maven)
   copy *jdbc-autoconfig\build\libs\*.jar temp (for Gradle)
   cd temp
   jar xvf *.jar
   ```

   Linux or MacOS:

```
mkdir temp
cp *jdbc-autoconfig/target/*.jar temp (for Maven)
cp *jdbc-autoconfig/build/libs/*.jar temp (for Gradle)
cd temp
jar xvf *.jar
```

4. What do you see?

   You will see the classpath resources, manifest file and supporting compile scope package classes are included.

   - Look carefully at the `BOOT-INF` directory.
     - What do you see?
     - What do its subdirectories contain?
   - Look in the `META-INF` directory and display the contents of `MANIFEST.MF` using `more`.
     - What do you see?

   You should see the jar is generated to be run as a standalone application on your behalf:

   - Contains all the necessary runtime dependencies - `BOOT-INF` holds all your compiled classes and all the dependency jars.
   - The manifest declares a main entry point (the `Main-Class:` property)

5. There are many ways to run this application, either directly using the JAR, using `spring-boot:run` goal from Maven or in your IDE as you did earlie

   Look over the *Running and Testing a Spring Boot Project* article lesson discussing various options for running a Spring Boot application.

   For now run `java -jar 32-jdbc-autoconfig-5.3.23.jar`, you should get the same output as before.

### Reviewing What We Did

We overrode Spring Boot's default behavior and defined a `DataSource` for ourselves. But which approach more appropriate?

Think about imperative declaration of `DataSource` auto- configuration disablement, versus disabling via configuration.

- If your use case is similar to wiring multiple data sources, it makes more sense to programmatically disable the auto- configuration given this is an fi aspect of your design.
- If your design style favors non-functional concerns in configuration then Spring Boot external configuration is an available option.

## # Summary

You should surmise by the end of the lab that Spring Boot can save you work:

- Simplified dependency management
- Simplified configuration
- Packaged Runtime
- Better test coverage with simple integration tests

Congratulations, you have completed the lab!

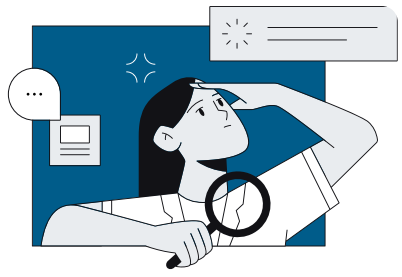
**Summary** **Instructors**

In this lab you took the concepts you learned in this module and apply them to your Spring Application.

## Give Feedback

Help us improve by sharing your thoughts.

Give Feedback





Unlock your full potential with Spring courses designed by experts.