WA1850 Lab Guide

Spring Fundamentals

ascendient learning

Part of **Accenture**

Revision 2.0.0 published on 2025-07-25.

Ascendient, LLC
1 California Street Suite 2900
San Francisco, CA 94111
https://www.ascendientlearning.com/

USA: 1-877-517-6540, email: getinfousa@ascendientlearning.com
Canada: 1-877-812-8887 toll free, email: getinfo@ascendientlearning.com

Table of Contents

Spring Project Basics

MODULE 1

In this lab, you will set up a new Spring project in your IDE with help from Gradle.

Solution code is provided for you in a separate folder, so you can compare your work if needed. For maximum benefit, try to complete the lab without looking at the solution.

1.1. IDE Reference Labs

The following Labs provide instructions for using the IDE you are working with. They are located at the end of this LabGuide. If you need IDE related help, refer to the appropriate document.

- IntelliJ IDEA IDE - Quick Reference
- Eclipse IDE - Quick Reference

Before moving on, locate the reference for the IDE you are using and become familiar with its contents.

Feel free to open the reference document in a separate tab or window so you can refer to it as needed.

1.2. Check the development environment

1. Open a terminal.
2. Navigate to the `C:\LabWork` folder. If you do not have this folder, create it now.
3. Check to see if Gradle is installed:

   ```
   gradle --version
   ```

   You should see output similar to the following:

```
------------------------------------------------------------
Gradle 8.7
------------------------------------------------------------

Build time:   2024-03-22 15:52:46 UTC
Revision:     650af14d7653aa949fce5e886e685efc9cf97c10

Kotlin:       1.9.22
Groovy:       3.0.17
Ant:          Apache Ant(TM) version 1.10.13 compiled on January 4 2023
JVM:          21.0.6 (Microsoft 21.0.6+7-LTS)
OS:           Windows 11 10.0 amd64
```

Gradle is pre-installed on the lab machines. If you don't see the output above, check with your instructor.

1.3. Copy the Starter Project.

1. Copy the starter project from LabFiles to your working LabWork directory as directed below:

   Copy:

   - Copy the folder: `LabFiles/spring-project-basics/starter/spring-project-basics`
   - Paste it into your working `LabWork` directory.
   - When you are done, you should have the folder:

     `LabWork/spring-project-basics`

2. Navigate into the `spring-project-basics` directory.

1.4. Run the Application from the Command Line

With the project created, you can now run its default application. The generated project includes a simple Java application with a main method.

The application can be run from the command line or imported into an IDE and run from there. First, you will run it from the command line.

1. Use the following command to run the application:

```
gradle run
```

   You should see output similar to this:

```
> Task :app:run
Hello World!
```

1.5. Import and Run the Application from your IDE

Run the application from your IDE allows you to debug and modify the code more easily.

1. Open the `C:\LabWork\spring-project-basics` project in your IDE.

   If you need help, refer to the `Import-or-Open-a-Project` instructions in the IDE reference for the IDE you are using.

2. Wait for the project to be fully imported before moving on to the next step.

3. Run the application. Its main class `App.java` is located in the `src/main/java/com/example` folder.

   If you need help, refer to the `Run the Application's Main Class` instructions in the IDE reference for the IDE you are using.

   The following should appear in the command's output:

```
Hello World!
```


1.6. Explore the Project

The project you are working with is structured and configured to provide a quick starting point for a typical Java application using the Gradle build process.

1. Take a moment to explore the structure and artifacts that are set up. The main folders in a Gradle/Maven based project are:
   - `src/main/java`: This is where the main application code goes.
   - `src/test/java`: This is where the test code goes.
   - `src/main/resources`: This is where resources like properties files, etc. go.
   - `src/test/resources`: This is where test resources go.

     > **Tip**
     >
     > Each of the above locations qualifies as being "a root of the Java classpath". This means that when you reference a file in your code, it is assumed to be relative to one of these locations.

2. Open the project's `build.gradle` file.

   This is where the project's code dependencies are defined.

   Notice the following:

   - The `dependencies` section includes a few common dependencies. This is where you'll place Spring framework dependencies.
   - The `repositories` section describes where dependencies can be downloaded from. The default repository for Java projects is: `mavenCentral()`. Many companies like to override this for security reasons, to use their own privately hosted repository.

3. Open `App.java` (in src/main/java/com/example)

   The file has a method with the signature: `public static void main(String[] args)`. In Java, such methods can be invoked from the command line. The `main` method serves as the entry point to the application.

4. Open `AppTest.java` (in src/test/java/com/example).
    - This is a JUnit test class. Build tools like Gradle understand that classes under src/test/java are test classes to be used when testing code.
5. Run `AppTest.java` as a JUnit test.

    If you need help, refer to the `Run a JUnit Test Class` instructions in the IDE reference for the IDE you are using.
6. Wait for the test to complete, then check the results. The test should pass.

1.7. Examine Project Dependencies

In this section of the lab you will identify essential dependencies in the `build.gradle` file. Some of these dependencies are required to use Spring and run JUnit tests.

> **Tip**
>
> When working directly with the Spring Framework like we are in this course several dependencies are required. Spring Boot which is based on the Spring Framework simplifies things by bundling multiple dependencies.

1. Open `build.gradle` and review its list of dependencies.
2. Add comments as shown below to identify the Spring framework and Testing dependencies:

```
dependencies {
        // Spring Framework Dependencies
    implementation "org.springframework:spring-core:6.2.8"
    implementation 'org.springframework:spring-context:6.2.8'

    // Testing related dependencies
    testImplementation 'org.springframework:spring-test:6.2.8'
    testImplementation 'org.junit.jupiter:junit-jupiter:5.10.2'
    testImplementation "org.assertj:assertj-core:3.11.1"
    testImplementation "org.mockito:mockito-core:5.7.0"
```

```
    testImplementation "org.mockito:mockito-junit-jupiter:5.7.0"
}
```

This set of dependencies will allow you to use the Spring Framework and to run basic JUnit tests. We will add more dependencies later.

3. Save the `build.gradle` file.
    - Your IDE should automatically detect the changes and download the new dependencies. If not, then follow the `Synchronize Gradle Dependencies` instructions in the IDE reference for the IDE you are using.

1.8. Create a Java Class named "Greeting"

1. Create new Java class named `Greeting.java` in the `com.example` package.

   If you need help, refer to the `Create a New Java Class` instructions in the IDE reference for the IDE you are using.

2. Create a new public method in the `Greeting` class called **getGreeting()** which returns a simple String message:

```
public String getGreeting() {
    return "Hello World from the Greeting class!";
}
```

3. Save the `Greeting.java` file.

4. Open `App.java` in the IDE.

5. Modify the `System.out.println` statement in the `main()` method of `App.java` to call the `getGreeting()` method on an instance of `Greeting` class to get the message that is printed out.

   The updated main method should look something like this:

```
public static void main(String[] args){
    System.out.println(new Greeting().getGreeting());
}
```

6. Organize your imports.
7. Save all files.
8. Run the application (`App.java`) again.

   It should display the following message:

```
Hello World from the Greeting class!
```

That's the way to do it in plain Java! You create a class, instantiate it, and call its methods. In the next sections, you will learn how to use Spring to get an instance of the `Greeting` class wherever its needed.

1.9. Make `Greeting` a Spring Bean

In this section, you will create a Spring configuration class that defines the `Greeting` class as a Spring bean. This allows Spring to manage the lifecycle of the `Greeting` instance and inject it wherever needed.

1. Create new Java class in the `com.example` package named `Config.java`.
2. Add Spring's `@Configuration` annotation to the class definition.

   Make sure that the `org.springframework.context.annotation.Configuration` annotation is imported as well.:

   If you need help adding the import, refer to the `Add Required Imports` instructions in the IDE reference for the IDE you are using.

```java
@Configuration
public class Config {
    ...
}
```

   - The annotation designates this class as a configuration class containing Spring bean definitions.

     > **Tip**

> In most IDEs while you type the name of a class, the IDE will often offer to complete the text for you. This "code completion" or "Intellisense" not only avoids typos, but automatically imports the correct type for you. Be aware that sometimes there are multiple types with the same name, so you may need to select the correct one.

3. Add a method inside the `Config` class annotated with `@Bean` that defines a Spring bean of type `Greeting` having the name `greetingBean`:

```
@Bean
public Greeting greetingBean() {
    return new Greeting();
}
```

   - The name of the method (`greetingBean`) becomes the ID of the bean.
   - The return Type of the method is the TYPE of the bean.
   - The method body contains a statement that instantiates and returns the bean.

4. Organize your imports, so that the `@Bean` annotation is imported from `org.springframework.context.annotation.Bean`.

5. Save all files

   - You should now have a `Greeting` class, a `Config` class, and an updated `App` class.

1.10. Modify App.java

The `App.java` class needs to be modified to use the Spring context to retrieve the `Greeting` bean instead of creating it directly. This allows Spring to manage the bean's lifecycle and dependencies.

1. Open `App.java`.

2. Replace the `main()` method with the following:

```
public static void main(String[] args) {
    ApplicationContext spring = new
AnnotationConfigApplicationContext(Config.class);
    Greeting greetingBean = spring.getBean(Greeting.class);
    System.out.println(greetingBean.getGreeting());
}
```

Code Notes:

- `AnnotationConfigApplicationContext` instantiates a Spring `ApplicationContext` using definitions found in `Config.class`.
- `getBean()` retrieves an instance of the bean from the context based on its type.
- `System.out.println()` prints the greeting message to the console.

3. Organize imports.

   If you need help organizing imports, refer to the `Add Required Imports` instructions in the IDE reference for the IDE you are using.

4. Save your work.

5. Run the application.

   It should display the following message:

```
Hello World from the Greeting class!
```

1.11. Create an Integration Test

Anything that's coded should be tested. You will revise the `AppTest.java` file to test that our Spring configuration correctly builds the beans as directed.

1. Open `src/test/java/com.example.AppTest.java`.

2. Remove (or comment out) the existing contents of the class definition (everything between the { and }).

   - The result should look like this:


```
class AppTest {
    ...
}
```

3. In the class, define a static variable named 'greetingBean' with the type type Greeting:

```
private static Greeting greetingBean;
```

   - This will hold a reference to the bean you are testing.

4. Add a method named `setup` to the `AppTest` class.

   i. Annotate it with @BeforeAll

   ii. Make it `public` and `static`

   iii. Have it get an instance of `ApplicationContext` using the `Config` class.

   iv. Retrieve the `greetingBean` from the context and assign it to the `greetingBean` class variable.`

   Your code should look something like this:

```
@BeforeAll
public static void setup() {
    ApplicationContext spring = new
AnnotationConfigApplicationContext(Config.class);
    greetingBean = spring.getBean(Greeting.class);
}
```

5. Add a method named `testGreeting` to the class:

   i. Annotate it with @Test.

   ii. Add an Assertion to verify that the greeting message is as expected.

   Your implementation should look something like this:

```
@Test
void testGreeting() {
    Assertions.assertThat(greetingBean.getGreeting()).isEqualTo("Hello World
```

```
from the Greeting class!");
}
```

- This method uses `AssertJ` to test that the greeting message is as expected.

6. Organize your imports.
   - Be sure to use the `AssertJ` import for Assertions rather than the JUnit one by the same name, or your code will not compile. The resulting import should say `import org.assertj.core.api.Assertions;`.
7. Save all modified files.
8. Run the `AppTest.java` class as a Junit test

   If you need help, refer to the `Run a JUnit Test Class` instructions in the IDE reference for the IDE you are using.
9. Wait for the test to complete, then check the results.

   The test should pass.
10. If the test fails, do the following and re-try.
    - Check your code for typos or missing imports.
    - Check that you have the required dependencies in your `build.gradle` file.
11. If the dependencies are in `build.gradle` but the class is still not found, you may need to synchronize Gradle dependencies. See the section in the IDE reference document titled: `Synchronize Gradle Dependencies`.

1.12. Review

In this lab you learned:

- How to create a new Spring project using Gradle.
- How to add Spring dependencies to the build file.

- How to create a `@Configuration` class for our bean definitions.
- How to create a simple `@Bean` definition.
- How to instantiate the Spring context and retrieve a bean.
- How to create a simple integration test.

Spring is a powerful framework that can do much more than this. In the next lab, you will explore a more complex example.

Spring Fundamentals, Spring Configuration

MODULE 2

In this lab, you will add code to complete an existing project, **spring-configuration**. You will add code to define and test the configuration of some Spring beans.

Within the codebase you will find numbered **TODO** comments that describe what actions to take for each step. By finding all files containing **TODO**, and working on the steps in numeric order, it is possible to complete this exercise without looking at the instructions. If you do need guidance, you can always look at these instructions for full information. Just be sure to perform the **TODO** steps in order.

Solution code is provided for you in a separate folder, so you can compare your work if needed. For maximum benefit, try to complete the lab without looking at the solution.

If you have time, there is an optional challenge to work with `@Profile`, `@PropertySource`, and `@Value`.

2.1. Copy the Starter Project.

1. Copy the starter project from LabFiles to your working LabWork directory as directed below:

   Copy:

   - Copy the folder: `LabFiles/spring-configuration/starter/spring-configuration`
   - Paste it into your working `LabWork` directory.
   - When you are done, you should have the folder:

     `LabWork/spring-configuration`

2.2. Run the application from the command prompt:

1. Open a command prompt and change to the `LabWork/spring-configuration` directory.
2. Run the following command to build and run the application:

   ```
   gradle run
   ```

   The command output should include a message like this:

   ```
   spring-configuration app is starting...
   ```

2.3. Import and run the application into `your IDE`:

1. Import the `spring-configuration` folder into your IDE as a Gradle project.

   If you need help, refer to the `Import-or-Open-a-Project` instructions in the IDE reference for the IDE you are using.
2. Wait until the project is finished importing, before you move on to the next step.
3. Try running the application again from your IDE. The main class to run is `com.example.App`.

   It should run without errors, and you should see the same message as before:

   ```
   spring-configuration app is starting...
   ```

2.4. Find the TODO instructions

Each TODO has a number next to it like this: 'TODO-01'.

1. Search for 'TODO' comments using your IDE's 'Find-In-Files' feature.

If you need help, refer to the `Find in Workspace Files` section of the IDE reference for the IDE you are using.

You should find several TODO comments in the code. These comments will guide you through the steps to complete this lab.

Each TODO has a number next to it like this: 'TODO-01'.

There are 19 TODOs in total, and they are spread across these files.

- App.java
- Config.java
- AlternateConfig.java
- AlternateRegisterTest.java
- RegisterTest.java

  Make sure to work through the TODO instructions in numeric order (starting from 'TODO-01')!

  At this point you can follow along in these instructions, or you can just work through the TODOs in order. If you get stuck, you can always come back to these instructions for help.

2.5. @Configuration class

1. Open src/main/java **com.example.Config.java**.
2. TODO-01: Add the `@Configuration` annotation to the class definition.
3. This tells Spring that this class contains bean definitions.
4. TODO-02: Within the Config class, write a `@Bean` method to define a Spring Bean.
5. The method should return a new instance of `SalesTaxCalculator`.
6. The bean's name should be "taxCalculator".

7. TODO-03: Write another `@Bean` method.
8. The bean's type should be `Register`.
9. The bean's name should be "register".
10. The method should take one parameter of type `TaxCalculator`.
11. The method should instantiate and return a `Register` instance. Dependency inject the instance with the `TaxCalculator`.
12. Organize your imports and save your work.

2.6. Main Application Class

1. Open src/main/java **com.example.App.java**.
2. TODO-05: Within the `public static void main()` method, instantiate the Spring `ApplicationContext` using `AnnotationConfigApplicationContext`.
3. Pass the `Config.class` to the constructor.
4. Use the previous lab as a guide if needed.
5. TODO-06: Retrieve the `Register` bean from the context using the `getBean()` method.
6. Lookup the bean by its name or type - your choice
7. TODO-07: Call the `calculateTotal()` method on the `Register` instance.
8. Pass in a double value of 100.0.
9. Print the result to the console.
10. TODO-08: Organize your imports and save your work. Run the application.
    - Expect to see the total cost printed to the console.

2.7. Integration Test

Whatever we code, we should test. We will write a test to verify that our Spring configuration correctly builds the beans as directed.

1. Open src/test/java **com.example.service.RegisterTest.java**.
2. TODO-09: Define a static variable of type `Register`.
    - Name the variable whatever you like.
3. TODO-10: Add a setup method annotated with `@BeforeAll`.
4. The method should be static and void. It can have any name you like.
5. Annotate the method with `@BeforeAll`.
6. Instantiate the `ApplicationContext` as you did earlier.
7. Retrieve the `Register` bean from the context and assign it to the static variable.
8. TODO-11: Add a test method.
9. Name the method whatever you like. Return type should be void.
10. The test logic should call the `computeTotal()` method on the `Register` bean.
11. Pass in a double value of 100.00.
12. Use the `assertThat` method to verify that the result is 105.00.

> ⓘ **Note**
>
> The `assertThat()` method is part of the AssertJ framework. See the static import at the top of the class.

13. TODO-12: Organize your imports and save your work. Run the test, it should pass.
    - If the test fails, review your code and try again.

2.8. OPTIONAL: Use @Profile, @PropertySource, and @Value Annotations

If you have time, there are some improvements we can make to this application.

At present, the tax rate is hard-coded in the `SalesTaxCalculator` class. We can use the `@Value` annotation to inject the tax rate from a properties file. We will also use the `@Profile` annotation to define different configurations for different environments.

1. Open src/main/java **com.example.AlternateConfig.java**.
2. TODO-13: Add `@Profile` and `@PropertySource` annotations to this `@Configuration` class.
3. Set the `@Profile` attribute to **alternate**. This configuration class will be activated only when the **alternate** profile is active.
4. Add a `@PropertySource` annotation to read 'app.properties' from the classpath root.
5. i.e. `@PropertySource("classpath:app.properties")`
6. TODO-14: Define a SalesTaxCalculator `@Bean`.
7. Name the bean whatever you like.
8. Define a method parameter of type double named "taxRate".
9. Use the `@Value` annotation to inject the value of the property "${tax.rate}".
10. Instantiate `SalesTaxCalculator` using the constructor which takes a tax rate.
11. Return the `SalesTaxCalculator` instance.
12. TODO-15: Organize your imports and save your work.
13. Open src/main/resources **app.properties**.
    - This properties file externalizes the tax rate from our code.

- We could also set the tax rate via an environment variable or system property.

2.9. OPTIONAL @Test the Alternate Configuration

Whatever we code, we should test. Our next test will verify that our alternative configuration builds our beans and injects the tax rate as expected.

1. Open src/test/java **com.example.service.AlternateRegisterTest.java**.
2. TODO-16: Within the `setup()` method, set a Java system property *spring.profiles.active* to the value *alternate*.
   - Use `System.setProperty()` to set the property.
   - When the Spring context is created, it will activate the **alternate** profile.
3. TODO-17: Still within the `setup()` method, after the system property has been set, instantiate the `ApplicationContext` using `AnnotationConfigApplicationContext`.
4. You can look at the code you just finished in App.java or RegisterTest.java for guidance.
5. Retrieve the `Register` bean from the ApplicationContext. Notice that a static variable has already been provided to hold this:

```
ApplicationContext spring =
    new AnnotationConfigApplicationContext(AlternateConfig.class);
register = spring.getBean(Register.class);
```

6. TODO-18: Write a @Test method to verify the computeTotal method.
7. Use your previous test method as a guide.
8. Adjust the logic to assert the total based on the configured tax.rate.
9. TODO-19: Organize your imports.

10. Save your work.
11. Run the test.

    The test should pass.

2.10. Summary

You have just gained practice creating Spring configuration classes and beans. You have applied dependency injection and tested all your work.

If you did the optional section, you used `@Profile` to selectively instantiate a set of beans, and you dynamically read and injected a parameter using `@PropertySource` and `@Value`.

Spring Fundamentals, Spring Configuration with Annotations

MODULE 3

In this lab, you will add code to complete an existing project, **spring-configuration-annotations**. In the previous lab you used `@Configuration` classes with `@Bean` methods, in this lab you will achieve the same result using `@Component` and `@ComponentScan` techniques. As before, you will create test cases covering everything implemented.

Within the codebase you will find ordered **TODO** comments that describe what actions to take for each step. By finding all files containing **TODO**, and working on the steps in numeric order, it is possible to complete this exercise without looking at the instructions. If you do need guidance, you can always look at these instructions for full information. Just be sure to perform the **TODO** steps in order.

Solution code is provided for you in a separate folder, so you can compare your work if needed. For maximum benefit, try to complete the lab without looking at the solution.

If you have time, there is an optional challenge working with `@Profile` and `@Value`.

3.1. Copy the Starter Project.

1. Copy the starter project from LabFiles to your working LabWork directory as directed below:

   Copy:

   - Copy the folder: `LabFiles/spring-configuration-annotations/starter/spring-configuration-annotations`
   - Paste it into your working `LabWork` directory.
   - When you are done, you should have the folder:

     `LabWork/spring-configuration-annotations`

3.2. Run the application from the command prompt:

1. Open a command prompt and change to the `LabWork/spring-configuration-annotations` directory.
2. Run the following command to build and run the application:

   ```
   gradle run
   ```

   The command output should include a message like this:

   ```
   spring-configuration-annotations app is starting...
   ```

3.3. Import and run the application into `your IDE`:

1. Import the `LabWork\spring-configuration-annotations` project into your IDE.

   If you need help, refer to the `Import or Open a Project` section of the IDE reference for the IDE you are using.
2. Wait until the project is finished importing, before you move on to the next step.
3. Try running the application again from your IDE. The main class to run is `com.example.App`.

   It should run without errors, and you should see the same message as before:

   ```
   spring-configuration-annotation app is starting...
   ```

3.4. Find the TODO instructions

Each TODO has a number next to it like this: 'TODO-01'.

1. Search for 'TODO' comments using your IDE's 'Find-In-Files' feature. If you need help, refer to the `Find in Workspace Files` section of the IDE reference for the IDE you are using.

   You should find 13 TODO comments spread across these code files:

   - RegisterTest.java
   - Config.java
   - Register.java
   - SalesTaxCalculator.java
   - CompoundSalesTaxCalculator.java
   - RegisterCompoundTest.java

     Make sure to work through the TODO instructions in numeric order (starting from 'TODO-01')!

   - To complete this lab, follow along with the instructions in this document.

3.5. Run the existing test

The current configuration of this lab matches the final state of the previous lab; `@Configuration` classes with `@Bean` methods define all the application components. Begin by running a test to verify this state.

1. TODO-01: Open `src/test/java/com.example.service.RegisterTest.java`. Run the test, it should pass.

3.6. Alter the Existing Configuration

1. Open `src/main/java/com.example.Config.java`.
2. TODO-02: Comment out the `@Bean` definitions in this class.

> **Tip**
>
> To quickly comment out multiple lines in most IDE's, highlight the lines and press Ctrl-/.

3. Re-run the previous test. It should *fail* - this is expected at this point. Do you understand why?
   - The exception encountered should be `NoSuchBeanDefinitionException`. Within the test class, in the `@BeforeAll` method, you should see code that attempts to retrieve a bean from the `ApplicationContext`, but we have just removed the `@Bean` method which creates it.
   - In the coming steps we will create this bean using annotation-driven configuration.
4. TODO-03: Add a `@ComponentScan` annotation to the top of the `Config` class. Set the basePackages attribute to reference the service package.

```
@Configuration
@PropertySource("classpath:app.properties")
@ComponentScan(basePackages = "com.example.service")
public class Config {
    ...
}
```

   - The `@ComponentScan` annotation tells Spring to scan the specified package for classes annotated with `@Component`, `@Service`, `@Repository`, `@Controller`, or other stereotypes. Spring will create beans for these classes automatically.
   - "basePackages" tells Spring where to look for components. In this case, we are telling Spring to look in the `com.example.service` package and sub-packages.
5. Organize your imports and, save your work.

6. TODO-04: Open `src/main/java/com.example.service.Register.java`. Add an annotation to this class to define it as a Spring bean.
    - There are several annotations you can choose from, `@Component`, `@Service`, `@Repository`, `@Controller` etc. Your choice should reflect the purpose of the bean. Since this is not a web controller or database repository, `@Component` or `@Service` would be appropriate.
    ```
    @Component
    public class Register {
        ...
    }
    ```
7. Organize your imports and save your work.
8. TODO-05: Open `src/main/java/com.example.service.SalesTaxCalculator.java`. Add an annotation to this class to define it as a Spring bean.
    - Use `@Component` or `@Service` as before.
    ```
    @Component
    public class SalesTaxCalculator implements TaxCalculator {
        ...
    }
    ```
9. TODO-06: Add code to have Spring call the constructor with the tax rate parameter. Have Spring set the tax rate using an external parameter.
    - When a class has multiple constructors, Spring will use the zero-parameter version if one exists. In our case, we would like it to call the other constructor by annotating it with `@Autowired`.
    - Spring's `Environment` abstraction already reads properties from the `app.properties` file due to the `@PropertySource` annotation in the `Config` class. We can use `@Value` to inject the property into the constructor parameter.
    - The `@Value` annotation takes a single attribute. To direct it to obtain a property from the `Environment`, use **${...}** syntax. The property in `app.properties` is named `tax.rate`, so the full syntax is **@Value("${tax.rate}")** :
    ```
    @Autowired
    public SalesTaxCalculator(@Value("${tax.rate}") double taxRate) {
    ```

```
        this.taxRate = taxRate;
    }
```

10. Organize your imports, save your work.

11. TODO-07: Once you have completed alterations to `Config`, `Register`, and `SalesTaxCalculator`, return to the `RegisterTest` test you ran previously. Run it again. It should pass.

    - The `@Bean` methods have been replaced with `@Component` classes, and the `@ComponentScan` annotation tells Spring to scan the service package for these classes.

3.7. OPTIONAL: Use @Profile to Define an Alternate Tax Calculator Configuration

If you have time, there are some improvements we can make to this application.

Some jurisdictions may require us to use a more complex tax calculator, a compound calculator which adds local taxes to the standard tax. We would like to allow our application to be configured to use either one tax calculator or the other.

1. TODO-08: Open `src/main/java/com.example.service.CompoundSalesTaxCalculator.java`. Add an annotation to this class to define it as a Spring bean.

    - Use `@Component` or `@Service` as before.

    ```
    @Component
    public class CompoundSalesTaxCalculator implements TaxCalculator {
        ...
    }
    ```

    - Now we have two tax calculators, `SalesTaxCalculator` and `CompoundSalesTaxCalculator`. Spring has no issue with multiple beans of the same type. However, if we attempt to inject a `TaxCalculator` somewhere, how will Spring behave?
    - Keep going to find the answer.

2. TODO-09: Add `@Value` annotations to populate the two constructor parameters based on values in the `app.properties` file.
    - This tax calculator uses two separate tax rates, `tax.rate` and `local.tax.rate`. Use `@Value` to inject these values into the constructor parameters.
    - Since there is only one constructor, `@Autowired` is not needed.

```
public CompoundSalesTaxCalculator(
  @Value("${tax.rate}") double taxRate,
  @Value("${local.tax.rate}") double localTaxRate) {

  this.taxRate = taxRate;
  this.localTaxRate = localTaxRate;
}
```

3. TODO-10: Organize imports, save your work.

4. Run the test `RegisterTest` again. It should FAIL.
    - Find the root cause of this exception: when Spring attempts to inject a `TaxCalculator` into the `Register` class, it finds two candidates. Spring does not know which one to use. We will fix this in a moment.

5. TODO-11: Open `src/test/java/com.example.service.RegisterCompoundTest.java`. This test is designed to test the `Register` class using only one tax calculator, the `CompoundSalesTaxCalculator`. To do this, we will use `@Profile`
    - Within the `setup()` method, before instantiating the `ApplicationContext`, add a line of code to set a JVM system property to activate the `compound` profile. To set JVM system properties within code, use `System.setProperty()`. The property to set is **spring.profiles.active**, and we will define a new profile called **compound** to represent the use of the compound sales tax calculator.
    - Add this line of code before the line which instantiates the `AnnotationConfigApplicationContext`.

```
System.setProperty("spring.profiles.active", "compound");
```

6. Return to the `CompoundSalesTaxCalculator` class and add a `@Profile` annotation to the class.
    - Use **compound** as the profile name.
    - Organize imports and save your work.

```
@Component
@Profile("compound")
public class CompoundSalesTaxCalculator implements TaxCalculator {
    ...
}
```

7. TODO-12: Back in the `RegisterCompoundTest` remove the `@Disabled` annotation from the test method.
    - `@Disabled` is a JUnit annotation used to temporarily disable a test. We use it to simplify the lab by avoiding distracting test failures until we reach a specific point in the lab steps.
8. Run the test. We still encounter a FAILURE.
    - We still encounter two `TaxCalculator` beans. How can this be? We explicitly assigned the `CompoundSalesTaxCalculator` to the **compound** profile, and we explicitly activated the **compound** profile. Why is the `SalesTaxCalculator` still being considered?
    - Try to determine the cause of this issue before moving to the next step.
9. TODO-13: Return to the `SalesTaxCalculator` class. Add a `@Profile` annotation to assign it to the "!compound" profile.
    - Although we have assigned `CompoundSalesTaxCalculator` to the **compound** profile, the `SalesTaxCalculator` still belongs to the **default** profile. **Default** profile beans are always instantiated.
    - We need to use `@Profile` to activate `SalesTaxCalculator` **only when compound is NOT active** so the beans are mutually exclusive.
    - Add `@Profile("!compound")` to the `SalesTaxCalculator` class.

```
@Component
@Profile("!compound")
public class SalesTaxCalculator implements TaxCalculator {
    ...
}
```

10. Organize all imports and save all work. Re-run the `RegisterCompoundTest`. It should now pass.

11. Re-run the `RegisterTest` test. It should also pass.

3.8. Review

In this lab we learned:

- Spring Beans can be defined either with `@Bean` methods in a `@Configuration` class, or with `@Component`, `@Service`, `@Repository`, or `@Controller` annotations.
- The `@ComponentScan` annotation tells Spring where to look for components.
- `@Autowired` can be used to select a specific constructor when a class has multiple constructors.
- The `@Value` annotation can be used to inject properties into a bean.
- `@Profile` can be used to define alternate configurations for different environments.

Spring Fundamentals, Testing

MODULE 4

A central mission of the Spring framework is to encourage and simplify automated testing. When working with Spring, tests are first-class citizens of the development process, not afterthoughts.

In previous labs we used JUnit to test our Spring-based applications, but we did not take advantage of Spring's testing framework. In this lab you will modify an existing project, **spring-testing**, to make full use of its testing capabilities.

Within the codebase you will find ordered **TODO** comments that describe what actions to take for each step. By finding all files containing **TODO**, and working on the steps in numeric order, it is possible to complete this exercise without looking at the instructions. If you do need guidance, you can always look at these instructions for full information. Just be sure to perform the **TODO** steps in order.

Solution code is provided for you in a separate folder, so you can compare your work if needed. For maximum benefit, try to complete the lab without looking at the solution.

If you have time, there is an optional challenge working with Mockito, one of the most widely used testing frameworks in the Java space today, and a good skill for any Java developer to have.

4.1. Copy the Starter Project.

1. Copy the starter project from LabFiles to your working LabWork directory as directed below:

   Copy:

   - Copy the folder: `LabFiles/spring-testing/starter/spring-testing`
   - Paste it into your working `LabWork` directory.
   - When you are done, you should have the folder:

     `LabWork/spring-testing`

4.2. Import and run the application into `your IDE`:

1. Import the `spring-testing` folder into your IDE as a Gradle project.

   If you need help, refer to the `Import-or-Open-a-Project` instructions in the IDE reference for the IDE you are using.

2. Wait until the project is finished importing, before you move on to the next step.

4.3. Adding the Spring Test Framework Dependency

To add the Spring Test framework dependency to the project you will need to edit the `build.gradle` file. This will enable the use of Spring's testing capabilities for unit testing.

1. Find `TODO 01` in the codebase. It should be located in the `build.gradle` file.

2. Add a *testImplementation* for **spring-test** in the `dependencies` block. It follows the same naming convention / version as the other Spring dependencies, so use them as a guide.

```
dependencies {
    // other dependencies
    testImplementation 'org.springframework:spring-test:6.2.8'
    //  other dependencies
}
```

3. Save the `build.gradle` file.

   If you are using an IDE, it should automatically recognize the change and download the dependency. If you are using the command line, you can run `./gradlew build` or `gradle build` in the project root directory to download the dependency.

```
dependencies {
    // other dependencies
    testImplementation 'org.springframework:spring-test:6.2.8'
```

```
    //  other dependencies
}
```

4.4. Refactoring the RegisterTest Class

In this section you will refactor the `RegisterTest` class to use Spring's testing framework. This will allow you to take advantage of Spring's dependency injection and other features in your tests.

1. Find `TODO-02` in the codebase. It should be located in the `src/test/java/com.example.service.RegisterTest.java` file.
2. Add the `@SpringJUnitConfig(Config.class)` annotation to the class to enable Spring testing.
    - This annotation is a combination of `@ExtendWith(SpringExtension.class)` and `@ContextConfiguration`.
    - `@ContextConfiguration` specifies the configuration class to load the Spring application context, `Config.class` in this case.
3. Remove the `@BeforeAll` method. Spring will load the `ApplicationContext` automatically when the test starts.
4. Remove the `static` modifier from the `register` field. (Spring performs dependency injection on objects, not classes). Optionally remove the `private` modifier as well; it is not highly relevant for test classes.
5. Add an `@Autowired` annotation to the `register` field to inject the `Register` bean.

```
@SpringJUnitConfig(Config.class)
public class RegisterTest {
    @Autowired Register register;
    ...
}
```

6. Organize imports,

7. Save the RegisterTest.java file
8. Run this test,

   The test should pass.

   You may see a warning in the console about a Java agent being loaded dynamically. This is normal and can be ignored for this lab. It is related to the Spring Test framework's use of ByteBuddy for dynamic class generation.

```
WARNING: A Java agent has been loaded dynamically (C:
\Users\markp\.gradle\caches\modules-2\files-2.1\net.bytebuddy\byte-buddy-
agent\1.14.9\dfb8707031008535048bad2b69735f46d0b6c5e5\byte-buddy-agent-1.14.9.jar)
WARNING: If a serviceability tool is in use, please run with -XX:
+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -
Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future
release
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader
classes because bootstrap classpath has been appended
```

4.5. Refactoring the RegisterCompoundTest Class

1. Find `TODO-03` in the codebase. It should be located in the `src/test/java/com.example.service.RegisterCompoundTest.java` file.
2. Add the `@SpringJUnitConfig(Config.class)` annotation to the class to enable Spring testing.
3. Add the `@ActiveProfiles("compound")` annotation to activate the "compound" profile.
4. Remove the `@BeforeAll` method. Spring will load the `ApplicationContext` automatically when the test starts.
5. Remove the `static` modifier from the `register` field. (Spring dependency injects objects, not classes). Optionally remove the `private` modifier as well; it is not highly relevant for test classes.

6. Add an `@Autowired` annotation to the `register` field to inject the `Register` bean.

```
@SpringJUnitConfig(Config.class)
@ActiveProfiles("compound")
public class RegisterCompoundTest {
    @Autowired Register register;
    ...
}
```

7. Organize imports,
8. Save the RegisterCompoundTest.java file
9. Run this test,

   The test should pass.

4.6. OPTIONAL: Override Properties Used in Tests

Often during testing we wish to override properties obtained from configuration files, environment variables, or other sources. Spring provides a way to do this using the `@TestPropertySource` annotation.

1. Find `TODO-04` in the codebase. It should be located near the bottom of the `src/test/java/com.example.service.RegisterCompoundTest.java` file
2. Use an annotation to override the tax rate properties used in the test.
3. Add the `@TestPropertySource` annotation to the class.
4. Use the `properties` attribute to specify the properties to override. Curly braces (i.e. `{}`) are used to define the key/value pairs.
5. The properties to override are `tax.rate` and `local.tax.rate`. Set them to `0.03` and `0.04` respectively.
6. Adjust the test method to expect a result of `100.00 + 7` (i.e. `107.00`). This is the expected result when the tax rate is `0.03` and the local tax rate is `0.04`.

> **Tip**
>
> Any properties defined using `@TestPropertySource` will override those defined in (almost) any other property sources, including `@PropertySource`, environment variables, and JVM system properties.

```
@SpringJUnitConfig(Config.class)
@ActiveProfiles("compound")
@TestPropertySource(properties = {"tax.rate=0.03", "local.tax.rate=0.04"})
public class RegisterCompoundTest {
    ...
}
```

7. Organize imports,
8. Save the RegisterCompoundTest.java file
9. Run this test,

   The test should pass.

4.7. OPTIONAL: Using Mockito

Mockito is a popular mocking framework that can be used in conjunction with JUnit to test Java code. It is widely used in the Java community and a good skill for any Java developer to have.

It takes a bit of practice to get comfortable with mocking, so if you have time, try this challenge.

1. TODO-05: Open `src/test/java/com.example.service.mockito.MockitoRegisterTest.java`. Configure it to use Mockito to test the `Register`.
2. Use the `@ExtendWith` JUnit annotation with `MockitoExtension` to link JUnit and Mockito together.

3. Note that this test will test our object without using the Spring framework at all. True POJOs should be usable without reliance upon a specific framework.

```
@ExtendWith(MockitoExtension.class)
public class MockitoRegisterTest {
    ...
}
```

4. TODO-06: Define a variable called `taxCalculator` of type `TaxCalculator`. Annotate this variable with `@Mock`.

5. In Mockito, a *mock* is a dynamic implementation of an interface which we program to support our test.

```
@Mock TaxCalculator taxCalculator;
```

6. TODO-07: Define a variable called `register` of type `Register`. Annotate this variable with `@InjectMocks`.

7. `Register` is the class we are testing.

8. Ordinarily, Spring would instantiate `Register` and inject it with `TaxCalculator`. Instead, we use Mockito to instantiate `Register` and inject it with the mock `TaxCalculator`.

```
@InjectMocks Register register;
```

9. TODO-08: Within the `@Test` method, use Mockito's `when()` method to program the mock.

10. The `when()` method is statically imported. It is used to program the mock to return a specific value when a method is called.

11. For our test, we want the mock to return `7.0` when the `taxCalculator.calculateTax()` method is called. Any input value to `calculateTax()` is acceptable.

```
//  When the taxCalculator.calculateTax() method is called
//  with any double argument, then return 7.0.
when(taxCalculator.calculateTax(anyDouble())).thenReturn(7.0);
```

12. TODO-09: Within the `@Test` method, call the method under test (`computeTotal`) on the `register` object.
13. The `register` object is the class we are testing. We want to test its `computeTotal` method. Call this method passing a value such as `100`.
14. Internally, the `computeTotal` method will call `taxCalculator.calculateTax()`. This mock method should return `7.0` as programmed above.
15. We are testing if the `computeTotal` method correctly adds the tax to the input value. We expect the result to be `107.0`. Store the result in a variable.

```
double result = register.computeTotal(100);
```

16. TODO-10: Use AssertJ to verify the result.
17. AssertJ is a popular assertion library that provides a fluent API for writing assertions. It is a good alternative to JUnit's built-in assertions.
18. Use AssertJ to assert that the result is equal to `100.00 + 7.0`.

```
assertThat(result).isEqualTo(100.00 + 7.0);
```

19. TODO-11: Verify that the mock had its `calculateTax()` method called.
20. Mockito's `verify()` method is used to verify that a mock was called with specific arguments.
21. Sometimes verifying the mock behavior is the only way to prove that a test is successful. In this case it is trivial since we can perform assertions on the return value. We just want you to see an example.

```
verify(taxCalculator).calculateTax(anyDouble());
```

22. Organize imports,
23. Save the MockitoRegisterTest.java file
24. Run the test,

    The test should pass.

4.8. Review

In this lab you learned:

- How to use Spring's testing framework to test Spring-based applications.
- How to override properties used in tests.
- How to use Mockito to test Java code without using the Spring framework.

Spring Fundamentals, Aspect Oriented Programming

MODULE 5

One of the most fascinating capabilities of the Spring framework is the ability to easily implement Aspect Oriented Programming (AOP). In this lab, we will learn how to use AOP in Spring.

Within the codebase you will find ordered **TODO** comments that describe what actions to take for each step. By finding all files containing **TODO**, and working on the steps in numeric order, it is possible to complete this exercise without looking at the instructions. If you do need guidance, you can always look at these instructions for full information. Just be sure to perform the **TODO** steps in order.

Solution code is provided for you in a separate folder, so you can compare your work if needed. For maximum benefit, try to complete the lab without looking at the solution.

5.1. Copy the Starter Project.

1. Copy the starter project from LabFiles to your working LabWork directory as directed below:

   Copy:

   - Copy the folder: `LabFiles/spring-aop/starter/spring-aop`
   - Paste it into your `LabWork` directory.
   - When you are done, you should have the folder:

     `LabWork/spring-aop`

5.2. Open the project.

1. Use your IDE to open `C:\LabWork\spring-aop`.

2. TODO 01: Open `build.gradle`. Notice there are three new dependencies added to support AOP:
    - org.aspectj:aspectjweaver
    - org.springframework:spring-aop
    - org.springframework:spring-aspects

Spring uses *AspectJ* internally for some of its features, such as pointcut syntax.

5.3. Enable Aspect Oriented Programming

1. TODO-02: Open `src/main/java/com.example.Config.java` Add an annotation to this configuration class to enable AspectJ proxying:

```
@EnableAspectJAutoProxy
@Configuration
@PropertySource("classpath:app.properties")
@ComponentScan("com.example")
public class Config {
    ...
}
```

The `@EnableAspectJAutoProxy` annotation serves as a master switch to enable AOP capability within a Spring application. It tells Spring to look for beans annotated with `@Aspect`, process `@Pointcut` annotations, and create proxies for all beans requiring them.

5.4. Build the LoggingAspect

1. TODO-03: Open `src/main/java/com.example.aspect.LoggingAspect.java`. Annotate this class as an aspect and a component.
2. Add the `@Aspect` annotation to the class.
3. Add the `@Component` annotation to the class.

```
@Aspect
@Component
public class LoggingAspect {
    ...
}
```

The `@Aspect` annotation tells Spring that this class contains advice and pointcut definitions. The `@Component` annotation registers this class as a Spring bean.

4. TODO-04: Find the `logActivity()` method. Annotate this method to execute before other methods are called. The pointcut should select any method starting with "get*" in any class in the `com.example.dao` package.

5. Place the `@Before` annotation on the method. It indicates that the advice should be executed before the target method is called.

6. The pointcut expression should be `execution(* com.example.dao.*.get(..))`.

    - **execution** indicates we wish to select a method execution.
    - The first wildcard ***** matches any return type.
    - **com.example.dao.*** matches any class in the `com.example.dao` package.
    - **get*** matches any method starting with "get".
    - **(..)** matches any number of parameters; zero or more.

7. The "Examples" section in [Spring Framework Reference](https://docs.spring.io/spring-framework/reference/core/aop/ataspectj/pointcuts.html) has similar examples you may find useful.

```
@Before("execution(* com.example.dao.*.get*(..))")
public void logActivity(JoinPoint joinPoint) {
    ...
}
```

8. TODO-05: Alter the line of code defining "methodName". Use the joinPoint parameter to get the class name and method name being called. The name will be used to log the name of the method to the console.

    - The `joinPoint` parameter provides information about the target method on the target object. It contains information about the method's signature.

```
String methodName =
    joinPoint.getSignature().getDeclaringTypeName() +
    "." +
    joinPoint.getSignature().getName();
System.out.println("The " + methodName + " method was called.");
```

9. TODO-06: Organize your imports, save your work. Move on to the next step.

5.5. Test the LoggingAspect

To effectively test the LoggingAspect, we will need to execute methods on beans which should be affected by the advice.

1. TODO-07: Open `src/test/java/com.example.dao.PurchaseDaoImplTests.java`. Annotate this class to make it a Spring test class. Include the configuration class you wish to load.
    - The `@SpringJUnitConfig` annotation tells Spring to load the `Config` configuration class specified in the annotation.
    ```
    @SpringJUnitConfig(Config.class)
    public class PurchaseDaoImplTests {
        ...
    }
    ```
2. TODO-08: Inject a `PurchaseDao` object.

    ```
    @Autowired PurchaseDao dao;
    ```
3. TODO-09: Organize imports, save your work.
    - A dialog may appear asking which imports to add for the 'Config' class. If so then make sure to choose the one in your project - `com.example.Config`.
    - Notice that there are two existing test methods in this class. Observe what they do, each calls methods on the `PurchaseService` object which begin with "get". This should trigger the advice in the `LoggingAspect` class.
    - Run the tests. You should see the output from the `LoggingAspect` class in the console. If you do not see the expected output, review your annotations and pointcut expressions.

```
The com.example.dao.PurchaseDao.getPurchase method was called.
The com.example.dao.PurchaseDao.getAllPurchases method was called.
```

- You may see warnings in the console like the ones below, they will not affect the outcome of the test.

```
WARNING: A Java agent has been loaded dynamically (C:
\Users\markp\.gradle\caches\modules-2\files-2.1\net.bytebuddy\byte-buddy-
agent\1.14.9\dfb870731008535048bad2b69735f46d0b6c5e5\byte-buddy-agent-1.14.
9.jar)
WARNING: If a serviceability tool is in use, please run with -XX:
+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -
Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future
release
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader
classes because bootstrap classpath has been appended
```

5.6. OPTIONAL: Build the StopWatchAspect

If you have time (and if you can't get enough of AOP), you can build another aspect to measure the time it takes to execute methods.

Consider that such a timer would need to 1) capture the current time before a method is called 2) capture the current time after the method is called 3) calculate the difference between the two times. We will need an advice type that can execute both before and after a method invocation.

1. TODO-10: Open `src/main/java/com.example.aspect.StopwatchAspect.java`. Annotate this class as an aspect and a component.
2. Add the `@Aspect` annotation to the class.
3. Add the `@Component` annotation to the class.

```
@Aspect
@Component
public class StopwatchAspect {
    ...
}
```

4. TODO-11: Find the `recordTime()` method. Annotate this method with an advice type capable of capturing activity both before and after a method is called. The pointcut should select any method starting with "save*" in the `com.example.dao` package.

5. Place the `@Around` annotation on the method. It indicates that the advice should be executed before and after the target method is called.

6. The pointcut expression should be `execution(* com.example.dao..save(..))`.

    - **execution** indicates we wish to select a method execution.
    - The first wildcard ***** matches any return type.
    - **com.example.dao.*** matches any class in the `com.example.dao` package.
    - **save*** matches any method starting with "save".
    - **(..)** matches any number of parameters; zero or more.

7. The "Examples" section in [Spring Framework Reference](https://docs.spring.io/spring-framework/reference/core/aop/ataspectj/pointcuts.html) has similar examples you may find useful.

```
@Around("execution(* com.example.dao.*.save*(..))")
public Object recordTime(ProceedingJoinPoint joinPoint) throws Throwable {
    ...
}
```

8. TODO-12: Within the advice method, instantiate a new `Stopwatch` object and call its `start()` method.

    - The `Stopwatch` class is an inner class provided for you.

```
Stopwatch stopwatch = new Stopwatch();
stopwatch.start();
```

9. TODO-13: Within the try / catch block, call the target method on the target object, assign the result to the `result` variable.

    - With `@Around` advice, the target method will not be called unless coded explicitly.
    - Use the `joinPoint.proceed()` method to call the target method on the target object.

- Critical: make sure to capture the result of the target method in a variable. This value must be returned to the caller.
```
result = joinPoint.proceed();
```

10. TODO-14: Within the finally block, stop the stopwatch and display the elapsed time.
    - Call the `stop()` method on the `stopwatch` object.
    - Call the `displayElapsedTime()` method on the `stopwatch` object, passing the `methodName`. Notice how the `methodName` is derived.
```
stopwatch.stop();
stopwatch.displayElapsedTime(methodName);
```

11. TODO-15: Organize imports, save your work. Move on to the next step.

5.7. Test the StopwatchAspect

1. TODO-16: Return to `src/test/java/com.example.dao.PurchaseDaoImplTests.java`. Find the `savePurchase()` method. Remove the `@Disabled` annotation. Run the test, it should pass AND you should see the elapsed time in the console.
    - The `@Disabled` annotation is used to temporarily disable a test.
    - The expected output should look similar to the following:
```
The com.example.dao.PurchaseDao.getPurchase method was called.
The com.example.dao.PurchaseDao.getAllPurchases method was called.
Elapsed time for com.example.dao.PurchaseDao.savePurchase is 2.165 seconds
```
    - Warnings may appear in the console, they will not affect the outcome of the test.
    - If you do not see the expected output, review your annotations and pointcut expressions.

5.8. Review

In this lab we learned:

- How to enable Aspect Oriented Programming in Spring.
- How to create an aspect.
- How to create pointcut expressions.
- How to create advice methods.
- How to test aspects.

Spring Fundamentals, JDBC Support

MODULE 6

Java Database Connectivity (JDBC) is one of the essential parts of the JDK, used widely when working with relational databases. However, it is a very low-level API, forcing developers to write redundant, error-prone code with checked Exceptions. The Spring framework elevates your level of productivity with the `JdbcClient`.

Within the codebase you will find ordered **TODO** comments that describe what actions to take for each step. By finding all files containing **TODO**, and working on the steps in numeric order, it is possible to complete this exercise without looking at the instructions. If you do need guidance, you can always look at these instructions for full information. Just be sure to perform the **TODO** steps in order.

Solution code is provided for you in a separate folder, so you can compare your work if needed. For maximum benefit, try to complete the lab without looking at the solution.

If you have time, there is an optional challenge working with Mockito to test the service layer.

6.1. Copy the Starter Project.

1. Copy the starter project from LabFiles to your working LabWork directory as directed below:

   Copy:

   - Copy the folder: `LabFiles/spring-jdbc/starter/spring-jdbc`
   - Paste it into your `LabWork` directory.
   - When you are done, you should have the folder:

     `LabWork/spring-jdbc`

6.2. Open the project.

1. Use your IDE to open `C:\LabFiles\spring-jdbc`.
2. TODO 01: Open `build.gradle`. Notice there are two new dependencies added to working with databases:
3. org.hsqldb:hsqldb
4. org.springframework:spring-jdbc
    - HSQLDB, or HyperSQL, is an in-memory database ideal for local testing. Spring JDBC is the Spring module that provides support for JDBC.
    - "runtimeOnly" refers to dependencies that are needed at runtime, but not at compile time. We don't want our code to have direct references to the library's code.

6.3. Define Database and JdbcClient

1. TODO 02: Open `src/main/java/com/example/Config.java`. Define a DataSource bean.
2. Name the bean `dataSource`.
3. The type of the bean shoule be `DataSource`.
4. Use the `EmbeddedDatabaseBuilder` class to create an in-memory database.
5. The type of database should be `HSQLDB`.
6. On startup, the `schema.sql` and `data.sql` scripts should be automatically run. These files are located in `src/main/resources/`, which are part of the classpath root.

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
```



```
        .addScript("classpath:schema.sql")
        .addScript("classpath:data.sql")
        .build();
}
```

7. TODO 03: Define a JdbcClient bean.
8. Name the bean `jdbcClient`.
9. The type of the bean should be `JdbcClient`.
10. The bean method should take a `DataSource` parameter.
11. Use the `JdbcClient.create()` factory method to create a new instance. Inject it with the `DataSource` parameter.

```
@Bean
public JdbcClient jdbcClient(DataSource dataSource) {
    return JdbcClient.create(dataSource);
}
```

6.4. Implement the `PurchaseDaoImpl`.

1. TODO 04: Open the `src/main/java/com/example/dao/PurchaseDaoImpl.java` file. Annotate it as a Spring bean.
    - Use the stereotype annotation you feel is most descriptive. This object reads and writes to a database.

```
@Repository
public class PurchaseDaoImpl implements PurchaseDao {
    ...
}
```

2. TODO 05: Have Spring inject the `JdbcClient` into this class.
3. Use whatever injection technique you like (constructor, setter, field).
    - To use constructor injection, you will need to define a constructor that takes a `JdbcClient` parameter.

- To use setter injection, you will need to define a setter method that takes a `JdbcClient` parameter.
- To use field injection, you will only need to annotate the existing field with `@Autowired`.

```
@Autowired JdbcClient jdbcClient;
```

4. TODO 06: Implement the `getAllPurchases()` method.
5. Use the `jdbcClient` to retrieve all purchases from the database.
6. The SQL statement is provided in the method.
7. Use the `sql` method to specify the SQL statement.
8. Use the `query` method to indicate what class to map the returned ResultSet to. Pass a `Purchase.class` parameter to easily map columns to `Purchase` properties.
    - The goal is to return a `List` of `Purchase` objects.

```
public List<Purchase> getAllPurchases() {
    String sql = "SELECT * FROM PURCHASE";
    return jdbcClient
        .sql(sql)
        .query(Purchase.class)
        .list();
}
```

9. TODO 07: Implement the `getPurchase()` method.
10. Use the `jdbcClient` to retrieve a single purchase from the database.
11. The SQL statement is provided in the method.
12. Use the `sql` method to specify the SQL statement.
13. Use the `param` method to specify the `id` parameter.
14. Use the `query` method to indicate what class to map the returned ResultSet to. Pass a `Purchase.class` parameter to easily map columns to `Purchase` properties.
    - The goal is to return a single `Purchase` object.

```
public Purchase getPurchase(int id) {
    String sql = "SELECT * FROM PURCHASE WHERE ID = ?";
    return jdbcClient
        .sql(sql)
        .param(id)
        .query(Purchase.class)
        .single();
}
```

15. TODO 08: Implement the `savePurchase()` method.
16. Use the `jdbcClient` to insert a purchase into the database.
17. The SQL statement is provided in the method.
18. Use the `sql` method to specify the SQL statement.
19. Use the `param` method to specify the name, product, and date parameters, in order.
20. Use the `update` method to execute the insert.
    - This is correct, all update, insert, and delete operations are done with the `update` method.

```
public void savePurchase(Purchase purchase) {
    String sql = "insert into PURCHASE (CUSTOMERNAME, PRODUCT, PURCHASEDATE)
values(?,?,?)";
    jdbcClient
        .sql(sql)
        .param(purchase.getCustomerName())
        .param(purchase.getProduct())
        .param(purchase.getPurchaseDate())
        .update();
}
```

21. Organize your imports, save your work, move on to the next step.

6.5. Implement the `PurchaseServiceImpl`.

1. TODO 09: Open the `src/main/java/com/example/service/PurchaseServiceImpl.java` file. Annotate it as a Spring bean.
    - Use the stereotype annotation you feel is most descriptive. This object is a service layer class.
    ```
    @Service
    public class PurchaseServiceImpl implements PurchaseService {
        ...
    }
    ```
2. TODO 10: Have Spring inject the `PurchaseDao` into this class.
3. Use whatever injection technique you like (constructor, setter, field).
    ```
    @Autowired PurchaseDao purchaseDAO;
    ```
4. Organize your imports, save your work, move on to the next step.

6.6. Implement the `PurchaseDaoImplTests`.

1. TODO 11: Open `src/test/java/com/example/dao/PurchaseDaoImplTests.java`. Annotate it as a Spring test class.
    - Include the configuration class you wish to load.
    ```
    @SpringJUnitConfig(Config.class)
    public class PurchaseDaoImplTests {
        ...
    }
    ```
2. TODO 12: Have Spring inject a `PurchaseDao` into this class.
    ```
    @Autowired PurchaseDao dao;
    ```
3. TODO 13: Examine the `findAllPurchases()` test method.
    - This method calls the dao's `getAllPurchases()` method, then checks the results.

- The returned List is checked to ensure it is not null and has multiple elements.
- The first element is checked to ensure its properties are mapped correctly.

4. Remove the `@Disabled` annotation from the `findAllPurchases()` test method.
5. Organize your imports. Choose 'com.example.Config' for the Config class if prompted.
6. Save the file
7. Run the test,
8. Verify that the test passes.
    - For the purposes of this lab, any warnings that appear in the console can be disregarded as they do not affect the test results.
9. TODO 14: Implement the `getPurchase()` test method.
    - This method should call the dao's `getPurchase()` method with an ID of 1, 2, or 3.
    - Use `assertThat` to check the returned `Purchase` object is not null and has its properties mapped correctly.
    - Use the previous test method for guidance.

```java
@Test
public void getPurchase() {
    Purchase p = dao.getPurchase(2);

    // Make sure the purchase has its properties mapped:
    assertThat(p).isNotNull();
    assertThat(p.getId()).isNotNull();
    assertThat(p.getCustomerName()).isNotNull();
    assertThat(p.getCustomerName()).isEqualTo("Paul");
    assertThat(p.getProduct()).isEqualTo("Football");
}
```

10. TODO 15: Implement the `savePurchase()` test method.
    - Alter the `Purchase` test data as you like, and call the `savePurchase()` method.
    - A `getPurchase(String customerName, Date date)` method is available on the dao to retrieve the purchase you just saved. Call it to retrieve the new purchase.
    - Add assertions to make sure the purchase retrieved matches the one that was saved.

```
@Test
public void savePurchase() {
    Purchase p = new Purchase();
    p.setCustomerName("Sample");
    p.setProduct("Sample Product");
    p.setPurchaseDate( new Date());

    dao.savePurchase(p);
    Purchase newPurchase =
dao.getPurchase(p.getCustomerName(),p.getPurchaseDate());

    // Make sure the purchase was saved properly:
    assertThat(newPurchase).isNotNull();
    assertThat(newPurchase.getId()).isNotNull();
    assertThat(newPurchase.getCustomerName()).isNotNull();
    assertThat(newPurchase.getCustomerName()).isEqualTo(p.getCustomerName());
    assertThat(newPurchase.getProduct()).isEqualTo(p.getProduct());
}
```

11. TODO-16: Organize your imports, save your work. Run this test class. All tests should pass.

6.7. OPTIONAL: Test PurchaseServiceImpl using Mockito

If you have time, you can test the `PurchaseServiceImpl` using Mockito. This is an optional challenge.

1. TODO-17: Open `src/test/java/com/example/service/PurchaseServiceImplTests.java`. Annotate it as a Mockito test class.
   - Use the JUnit `@ExtendWith` annotation combined with the `MockitoExtension` class.
     ```
     @ExtendWith(MockitoExtension.class)
     public class PurchaseServiceImplTests {
         ...
     }
     ```
2. TODO-18: Define a Mock object called `purchaseDao` of type `PurchaseDao`.
   - Annotate this variable with `@Mock`.
     ```
     @Mock PurchaseDao purchaseDao;
     ```
3. TODO-19: Define a variable called `purchaseService` of type `PurchaseServiceImpl`.
   - Annotate this variable with `@InjectMocks`. This tells Mockito to inject it with the mock `purchaseDao`.
     ```
     @InjectMocks PurchaseServiceImpl purchaseService;
     ```
4. TODO-20: Within the `testFindAllPurchases()`, use Mockito's `when()` method to program the mock.
   - When the mock's `getAllPurchases()` method is called, return the `expectedPurchases` list.
     ```
     when(purchaseDao.getAllPurchases()).thenReturn(expectedPurchases);
     ```
   - Note that the `expectedPurchases` list is populated with test data for you.
5. TODO-21: Call the method under test.
   - Capture the results of the `findAllPurchases()` method in a variable.
     ```
     List<Purchase> result = purchaseService.findAllPurchases();
     ```

6. TODO-22: Test the result.
    - Assert that the returned result contains the same elements in the `expectedPurchase` list.
    - This kind of assertion is tricky. Fortunately AssertJ has a method for this.
    ```
    assertThat(result).containsExactlyElementsOf(expectedPurchases);
    ```
7. TODO-23: Verify that the mock had its `getAllPurchases()` method called.
    - Use the `verify()` method. Your mock object is the parameter to this method.
    - For some tests, `verify()` is critical; for others, it's trivial.
    ```
    verify(purchaseDao).getAllPurchases();
    ```
8. TODO-24: Organize your imports, save your work, run this test class. All tests should pass.

6.8. Review

In this lab we learned:

- How to configure a `DataSource` bean using an in-memory database.
- How to define a `JdbcClient` bean.
- How to use the `JdbcClient` to perform SELECT and INSERT operations on a database.
- How to test the DAO.
- How to use Mockito to test the service layer.

Spring Fundamentals, Transaction Management

MODULE 7

Spring provides a declarative approach to transaction management, separating implementation from demarcation. This allows your code to enjoy the benefits of transaction management without being coupled to a specific implementation.

Within the codebase you will find ordered **TODO** comments that describe what actions to take for each step. By finding all files containing **TODO**, and working on the steps in numeric order, it is possible to complete this exercise without looking at the instructions. If you do need guidance, you can always look at these instructions for full information. Just be sure to perform the **TODO** steps in order.

Solution code is provided for you in a separate folder, so you can compare your work if needed. For maximum benefit, try to complete the lab without looking at the solution.

7.1. Copy the Starter Project.

1. Copy the starter project from LabFiles to your working LabWork directory as directed below:

   Copy:

   - Copy the folder: `LabFiles/spring-transactions/starter/spring-transactions`
   - Paste it into your `LabWork` directory.
   - When you are done, you should have the folder:

     `LabWork/spring-transactions`

2. Use your IDE to open `C:\LabWork\spring-transactions`.

7.2. Configure the PlatformTransactionManager

1. Search for the **TODO** comments in the codebase.

2. The first **TODO** is in `src/main/java/com.example.Config.java`. Open this file.

   `Config.java` contains the configuration for the Spring ApplicationContext.

3. Add an annotation to this class to enable Spring transaction management.

4. Spring's transaction management is not enabled by default, but it is easy to switch on.

5. Use `@EnableTransactionManagement`.

```
@EnableTransactionManagement
@Configuration
@PropertySource("classpath:app.properties")
@ComponentScan("com.example")
public class Config {
    ...
}
```

   - This annotation tells Spring to look for beans with `@Transactional` annotations and create proxies to add transactional behavior to these beans.

6. TODO-02: Define a `PlatformTransactionManager` bean.

7. Name the bean `transactionManager`.

8. The type of the bean should be `PlatformTransactionManager`.

9. The bean method should accept a `DataSource` parameter.

10. Instantiate and return a `DataSourceTransactionManager`, injecting it with the `DataSource` parameter.

```
@Bean
public PlatformTransactionManager transactionManager(DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}
```

7.3. Make the `PurchaseServiceImpl` Transactional

1. TODO-03: Open `src/main/java/com.example.service.PurchaseServiceImpl.java`. Make the methods in this bean transactional.

2. Add an annotation to this bean to make all methods transactional.

3. Be sure to import the annotation from `org.springframework…`.

```
@Service
@Transactional
public class PurchaseServiceImpl implements PurchaseService {
    ...
}
```

When the Spring ApplicationContext starts, it will recognize the `@Transactional` annotation and create a proxy around the `PurchaseServiceImpl` bean. This proxy will intercept method calls, adding transactional behavior (begin, commit, rollback) to each method.

7.4. Test the Transactional Behavior

1. TODO-04: Open `src/test/java/com.example.service.PurchaseServiceImplTests.java`. Annotate this class to make it a Spring test class.

   - Include the configuration class you wish to load.

```
@SpringJUnitConfig(Config.class)
public class PurchaseServiceImplTests {
    ...
}
```

2. TODO-05: Have Spring inject a `PurchaseService` into this class.

   - Use the `@Autowired` annotation.

```
@Autowired PurchaseService purchaseService;
```

3. TODO-06: Have Spring inject the `PlatformTransactionManager` into this class.
    - Use the `@Autowired` annotation.
    - Ordinarily, we do not use the transactionManager directly in our code, but here we will use it to manually control transactions to explore how the the `PurchaseServiceImpl` transactional behavior works.

```
@Autowired PlatformTransactionManager transactionManager;
```

4. TODO-07: Remove the `@Disabled` annotation from the `testSavePurchase` method. Observe the code that starts a transaction.
    - The `@Disabled` annotation is used to temporarily bypass a test.
    - The `getTransaction` method of the `PlatformTransactionManager` is used to manually start a transaction.
    - Our goal is to verify that the transaction defined in `PurchaseServiceImpl`is able to work independently of any other transactions occurring simultaneously.
    - You do not need to make any changes to this code, just understand what it is doing.

```
//  Manually start a transaction.
TransactionStatus status =
    transactionManager.getTransaction(
        new DefaultTransactionDefinition());
```

5. TODO-08: Observe the test logic.
    - We are creating a test `Purchase` object.
    - The `Purchase` object will be saved using the `PurchaseService`.
    - No changes are needed here, move on to the next step.

```
Purchase p = new Purchase("Praveen", new Date(), "lava lamp");
```

6. TODO-09: Call the method on the `PurchaseService` to save the `Purchase` object. Pass in the `Purchase` object defined above.

```
purchaseService.savePurchase(p);
```

7. TODO-10: Observe the rollback code.
    - The `transactionManager` is used to rollback the transaction.

- Consider: will this rollback the affect the previous save operation or not?

  The `@Transactional` annotation on `PurchaseServiceImpl` adds transactional behavior to each method, including `savePurchase()`.

  However, our test logic has already started a transaction before calling `savePurchase()`.

  Will this result in one single transaction that is rolled back, or will `savePurchase()` have a separate transaction?

  There is nothing you need to code here, move on to the next step.

```
transactionManager.rollback(status);
```

8. TODO-11: Within the try / catch block, call the `findPurchase()` method on the `PurchaseService` to retrieve what was just saved.

   - The `findPurchase` method takes two parameters: the customer name and the purchase date. Use the values from the test data created above.
   - Assign the result to the `retrievedPurchase` variable.

```
retrievedPurchase =
    purchaseService.findPurchase(
        p.getCustomerName(),
        p.getPurchaseDate());
```

9. TODO-12: Observe the assertions. They verify the purchase was saved properly. Organize imports and save your work. Run this test. Initially it will **FAIL**. Do you understand why?

   - The rollback logic has rolled back the `Purchase` object that was saved. Did you expect this?
   - Recall that the default propagation behavior of `@Transactional` methods is **required**. This means if a transaction already exists, the method will join it. The single transaction was rolled back by the test logic.
   - This is expected. In the next step, we will change the transaction propagation behavior to **requires_new** to see how this affects the test.

7.5. Change the Transaction Propagation Behavior

1. TODO-13: Return to `src/main/java/com.example.service.PurchaseServiceImpl.java`. Override the transaction propagation behavior on the `savePurchase()` method.
    - Add a `@Transactional` annotation to the `savePurchase` method. Method-level annotations override class-level annotations.
    - Set the `propagation` attribute to `Propagation.REQUIRES_NEW`.
    - This will require a new transaction to be started when the `savePurchase` method is called.

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void savePurchase(Purchase purchase) {
    purchaseDAO.save(purchase);
}
```

    - Now a new transaction will begin when this method is called, regardless of whether one is already in progress. The inner transaction will be committed or rolled back independently of any outer transaction.
2. TODO-14: Organize imports, save your work. Return to the previous test. Run it again. It should pass.
    - The `savePurchase` method is now in its own transaction. The rollback logic in the test will not affect it.

7.6. Review

In this lab we learned:

- How to enable Spring transaction management.
- How to define a `PlatformTransactionManager` bean.
- How to make methods in a bean transactional.
- How to manually control transactions in a test.
- How to change the transaction propagation behavior of a method.

Spring Fundamentals, JPA Support

MODULE 8

Java Persistence API (JPA) provides a higher-level abstraction to JDBC when working with relational databases. Spring works well with JPA, handling the `EntityManagerFactory` for you, and providing the same declarative transaction management support we saw earlier.

Within the codebase you will find ordered **TODO** comments that describe what actions to take for each step. By finding all files containing **TODO**, and working on the steps in numeric order, it is possible to complete this exercise without looking at the instructions. If you do need guidance, you can always look at these instructions for full information. Just be sure to perform the **TODO** steps in order.

Solution code is provided for you in a separate folder, so you can compare your work if needed. For maximum benefit, try to complete the lab without looking at the solution.

If you have time, there is an optional challenge working with Mockito to test the service layer.

8.1. Copy the Starter Project.

1. Copy the starter project from LabFiles to your working LabWork directory as directed below:

   Copy:

   - Copy the folder: `LabFiles/spring-jpa/starter/spring-jpa`
   - Paste it into your `LabWork` directory.
   - When you are done, you should have the folder:

     `LabWork/spring-jpa`

2. Use your IDE to open `C:\LabWork\spring-jpa`.

8.2. Check out the new dependencies

1. Search for the **TODO** comments in the codebase.

   The first todo, **TODO 01** is in the `build.gradle` file.

2. Open the `build.gradle` file.

3. Notice the new dependencies added for working with JPA:

   `org.springframework:spring-orm`
   `jakarta.persistence:jakarta.persistence-api`
   `org.hibernate.orm:hibernate-core`

   - Jakarta Persistence API is the *API* that we reference in our code. Hibernate is a JPA *implementation*.
   - Notice that the Hibernate is marked "runtimeOnly". It will not be available when compiling, but will be on the classpath when running. This prevents us from inadvertently coupling our code to the Hibernate implementation.
   - Notice that HyperSQL and Spring JDBC are still present. We still need an in-memory database, and Spring's ORM support is based on Spring's JDBC support.

8.3. Define EntityManagerFactory bean and PlatformTransactionManager

1. TODO 02: Open `src/main/java/com.example.Config`. Examine the existing `localContainerEntityManagerFactoryBean`.

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource ds)
{
    var emfb = new        LocalContainerEntityManagerFactoryBean();            //
(1)

emfb.setDataSource(ds);
(2)
    emfb.setPackagesToScan("com.example.domain");                   // (3)
```

```
    var adapter = new
HibernateJpaVendorAdapter();                                    // (4)
    emfb.setJpaVendorAdapter(adapter);

    var props = new
Properties();                                                              //
(5)
    props.setProperty("hibernate.show_sql", "true");
    props.setProperty("hibernate.format_sql", "true");
    props.setProperty("hibernate.use_sql_comments", "true");

    emfb.setJpaProperties(props);
    return emfb;
}
```

2. The `LocalContainerEntityManagerFactoryBean` is part of the Spring framework. It is a specialized form of Spring FactoryBean that creates JPA `EntityManagerFactory` instances.

3. It requires a `DataSource` to be injected. This provides a database connection and also makes the `EntityManagerFactory` aware of the *type* of database being used (e.g. Oracle, MySQL, etc.) This is important as different databases have different SQL dialects and capabilities.

4. The `setPackagesToScan` tells where to look for JPA `@Entity` classes. This is a comma-separated list of packages. Similar to Spring's `@ComponentScan`, but we are looking for entities.

5. The `HibernateJpaVendorAdapter` is a Spring class that configures Hibernate-specific settings. It plugs JPA into the Hibernate implementation.

6. The `Properties` object is used to set JPA properties, some of which may be Hibernate specific. In this case, we are setting properties to display well-formatted SQL, which helps greatly when debugging.

   No changes are needed to this bean definition. It is already correct.

7. TODO 03: Define a `PlatformTransactionManager` for JPA.

8. The name of the bean should be `transactionManager`.

9. The type of the bean should be `PlatformTransactionManager`.
10. The method should accept an `EntityManagerFactory` as a parameter.
11. Instantiate and return a `JpaTransactionManager` injected with the `EntityManagerFactory`.

```
@Bean
public PlatformTransactionManager transactionManager(EntityManagerFactory emf) {
    return new JpaTransactionManager(emf);
}
```

> ⓘ Note
>
> There is no bean definition for `EntityManagerFactory`. Spring's `LocalContainerEntityManagerFactoryBean` is a `FactoryBean` that automatically creates an `EntityManagerFactory` when needed. Spring uses this `FactoryBean` pattern to create beans that are more complex to instantiate.

8.4. Define the Customer Entity

1. TODO 04: Open `com.example.domain.Customer`. Annotate the class and fields as follows:
2. Annotate the class with `@Entity` to mark it as a JPA entity.
3. Annotate the class with `@Table(name="CUSTOMERS")` to override the table name.
4. Annotate the `id` field with `@Id` to mark it as the primary key.

5. Annotate the `id` field with `@GeneratedValue(strategy=GenerationType.IDENTITY)` to describe the primary key generation strategy.
    - This annotation tells JPA to use the database's identity column when generating a primary key value during INSERT. This is a common strategy for primary key generation.
6. Annotate the `name` field with `@Column(name="CUSTOMER_NAME")` to override the column name.
    - By default, JPA will use the field name as the column name. In this case, the column is named `CUSTOMER_NAME`.
    - Notice that we do not need to annotate any other fields like `email`. By default, JPA considers all fields persistent and assumes the field name matches the column name.

```
@Entity
@Table(name="CUSTOMERS")
public class Customer {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        long id;

        @Column(name="CUSTOMER_NAME")
        String name;

    // Other fields...
}
```

8.5. Define the `Purchase` entity.

1. TODO 05: Open `com.example.domain.Purchase`. Annotate the class and fields as follows:
2. Annotate the class with `@Entity` to mark it as a JPA entity.
3. Annotate the class with `@Table(name="PURCHASES")` to override the table name.

4. Annotate the `id` field with `@Id` to mark it as the primary key.
5. Annotate the `id` field with `@GeneratedValue(strategy=GenerationType.IDENTITY)` to describe the primary key generation strategy.
6. Annotate the `customer` field with `@ManyToOne` to mark it as a many-to-one relationship.
    - The database will have a foreign key relationship between `PURCHASES` and `CUSTOMERS`.
    - When reading a `Purchase`, the `Customer` will be populated automatically.
    - When saving a new `Purchase`, the `Customer` will need to be provided to establish the relationship.
7. Annotate the `purchaseDate` field with `@Column(name="PURCHASE_DATE")` to override the column name.

```
@Entity
@Table(name="PURCHASES")
public class Purchase {
        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        private int id;

        @ManyToOne
        private Customer customer;

        @Column(name="PURCHASE_DATE")
        private Date purchaseDate;

    // Other fields...
}
```

8.6. Implement JPA code within PurchaseDaoImpl

1. TODO 06: Open `com.example.dao.PurchaseDaoImpl`
2. Annotate the class as a Spring bean.

3. Use the most descriptive stereotype annotation. This class is *not* a service layer object or a web controller.

```
@Repository
public class PurchaseDaoImpl implements PurchaseDao {
    ...
}
```

4. TODO 07: Inject the `EntityManager` into the class.

5. Use the special JPA annotation to inject the `EntityManager` into this class.

6. Do NOT use `@Autowired`, `@Resource`, or `@Inject`. The `EntityManager` is a special case.

```
@PersistenceContext
EntityManager em;
```

   IMPORTANT:

   - `@PersistenceContext` causes Spring to inject a special *proxy* `EntityManager`. This proxy contains lookup logic to find the actual `EntityManager` associated with the current thread and transaction. It delegates all calls to this object. This allows our code to simultaneously handle hundreds of threads and concurrent transactions.
   - Injecting the `EntityManager` directly with `@Autowired` would be a mistake: there is no such bean in the ApplicationContext, and it is not thread-safe.
   - Injecting the `EntityManagerFactory` directly would also be a mistake: it would not be aware of the current transaction used by any given thread.

7. TODO 08: Implement the `getAllPurchases` method.

8. Remove the existing `return null;` statement.

9. Use the `EntityManager` to create a query.

10. The JPQL query string is provided. It will select all purchases.

11. The `createQuery` method needs to know the type of entity being returned. This will be `Purchase.class`.

12. The `getResultList` method will return a `List<Purchase>`.

```
public List<Purchase> getAllPurchases() {
    String jpql = "SELECT p FROM Purchase p";
    return em
        .createQuery(
            jpql,
            Purchase.class)
        .getResultList();
}
```

13. TODO 09: Implement the `getPurchase` method.
14. Remove the existing `return null;` statement.
15. Use the `EntityManager` to find a single `Purchase`.
    - Use the `find` method, not `createQuery`. We are looking for a single object by its primary key.
    - The `find` method requires the type of entity and the primary key value. The type must be a class annotated with `@Entity`.
    - `find` returns a single result.

```
public Purchase getPurchase(int id) {
    return em.find(Purchase.class, id);
}
```

16. TODO 10: Implement the second `getPurchase` method.
17. Remove the existing `return null;` statement.
18. Use the `EntityManager` to create a query.
19. The JPQL query string is provided. It will select a single purchase by customer name and date. Notice the placeholders `:name` and `:date`.
20. The `createQuery` method needs to know the type of entity being returned. This will be `Purchase.class`.
21. Parameters to the query need to be set using the `setParameter` method.
22. The `getSingleResult` method will return a single `Purchase`.

```
public Purchase getPurchase(String name, Date date) {
    String jpql = "SELECT p FROM Purchase p WHERE p.customerName = :name AND
p.purchaseDate = :date";
    return em
        .createQuery(jpql, Purchase.class)
        .setParameter("name", name)
        .setParameter("date", date)
        .getSingleResult();
}
```

23. TODO 11: Implement the `savePurchase` method.
24. Check the incoming `Purchase` object's ID to determine if this entity already exists in the persistence context.
    - A positive ID indicates an existing entity. Use the `merge` method to update the entity in the persistence context. (This may result in the entity being SELECTed from the database if it is not already in memory. It will be UPDATED when the `EntityManager` is flushed.)
    - A zero ID indicates a new entity. Use the `persist` method to add the entity to the persistence context. (This results in an INSERT to the database.)

    ```
    public void savePurchase(Purchase purchase) {
        if(purchase.getId() < 1) {
            em.persist(purchase);
        } else {
            em.merge(purchase);
        }
    }
    ```
25. Organize your imports, save your work.

8.7. Implement the PurchaseServiceImpl

1. TODO 12: Open `com.example.service.PurchaseServiceImpl`. Annotate the class as a Spring bean.
    - Use the most descriptive stereotype annotation. This class is a service layer object.

```
@Service
public class PurchaseServiceImpl implements PurchaseService {
    ...
}
```

2. TODO 13: Inject the `PurchaseDao` into the class.
    - Use whatever dependency injection technique you prefer.
    ```
    @Autowired PurchaseDao purchaseDao;
    ```
3. Organize your imports, save your work.

8.8. Implement PurchaseDaoImplTests

JPA offers to take care of much of the low level work associated with database CRUD operations, but we still need to have integration tests to ensure that it works correctly.

1. TODO 14: Open `src/test/java/com.example.dao.PurchaseDaoImplTests`. Annotate the class as a Spring test class.
    - Make the test methods transactional. This will rollback any changes after each test is complete, making the tests repeatable.
    ```
    @SpringJUnitConfig(Config.class)
    @Transactional
    public class PurchaseDaoImplTests {
        ...
    }
    ```
2. TODO 15: Inject the `EntityManager` into the class.
3. Use the special JPA annotation to inject the `EntityManager` into this class.
4. Do NOT use `@Autowired`, `@Resource`, or `@Inject`. The `EntityManager` is a special case.

    ```
    @PersistenceContext EntityManager em;
    ```

5. TODO 16: Inject the `PurchaseDao` into the class.
    - This is the main object under test.
    ```
    @Autowired PurchaseDao repo;
    ```
6. TODO 17: Implement the `findAllPurchases` test method.
7. Remove the `@Disabled` annotation.
8. Use the `PurchaseDao` to retrieve all purchases.
9. Use AssertJ `assertThat()` to check that the list is not null and has a size greater than zero.
10. Assert that the first purchase has its properties mapped.
    - Notice that the customer name is accessed on the `Customer` referenced through the `Purchase` object.
    ```
    @Test
    public void        findAllPurchases() {
        List<Purchase> purchases = repo.getAllPurchases();

        //  Make sure there are multiple purchases, and
        //  that purchases have their properties mapped:
        assertThat(purchases).isNotNull();
        assertThat(purchases.size()).isGreaterThan(0);
        Purchase p = purchases.get(0);
        assertThat(p).isNotNull();
        assertThat(p.getId()).isNotNull();
        assertThat(p.getCustomer().getName()).isNotNull();
        assertThat(p.getCustomer().getName()).isEqualTo("Bruce");
    }
    ```
11. TODO 18: Implement the `getPurchase` test method.
12. Use the `PurchaseDao` to retrieve a single purchase by ID.
    - Use an ID value of 1, 2, or 3 to match the data in `data.sql`. The ID that you use will determine the values to use in your assertions below.
13. Make sure the purchase has its properties mapped.
    - Use the method you just completed for guidance.

```
@Test
public void        getPurchase() {
    Purchase p = repo.getPurchase(2);

    //  Make sure the purchase has its properties mapped:
    assertThat(p).isNotNull();
    assertThat(p.getId()).isNotNull();
    assertThat(p.getCustomer().getName()).isNotNull();
    assertThat(p.getCustomer().getName()).isEqualTo("Paul");
    assertThat(p.getProduct()).isEqualTo("Football");
}
```

14. TODO 19: Implement the `savePurchase` test method.
15. Remove the `@Disabled` annotation.
16. Note the existing logic using `EntityManager` `find` to get an existing `Customer`. (We want to try adding a new `Purchase` to an existing `Customer`.)
17. Note the existing `Purchase` object created for test data. Change these values if you like. Important: notice how the `Customer` property is set to the `Customer` object just retrieved; this establishes the foreign key relationship.
18. Use the `PurchaseDao` to save the `Purchase`.
19. Clear the persistence context by calling the `EntityManager` `clear()` method.
    - Important: We want to make sure the `Purchase` we are about to retrieve is coming from the database, not from the persistence context in memory.
20. Use the `PurchaseDao` to retrieve the `Purchase` by ID. The ID can be found on the `Purchase` object that was just saved.
    - Interesting: where did this ID come from? It was generated by the database when the `Purchase` was saved, and JPA updated the `Purchase` object in the persistence context with this new value.
21. Use AssertJ to check that the saved `Purchase` properties match the original `Purchase` properties.

```
@Test
// @Disabled
public void testSaveAndFind() {
```

```
    //  Get an existing customer:
    Customer c = em.find(Customer.class, 1);

    //  Create a new purchase:
    Purchase p = new Purchase();
    p.setCustomer(c);
    p.setProduct("Incan ceremonial headmask");
    p.setPurchaseDate(new Date());

    //  Save...
    repo.savePurchase(p);

    //  Clear...
    em.clear();

    //  Find...
    Purchase purchase = repo.getPurchase(p.getId());

    //  Assert...
    assertEquals(p.getProduct(), purchase.getProduct());
    assertEquals(p.getPurchaseDate(), purchase.getPurchaseDate());
}
```

22. Organize your imports. If you are prompted to choose a type for the Config class choose "com.example.Config".

23. Save your work.

24. Run this test

    The test should pass.

8.9. OPTIONAL: Implement `PurchaseServiceImplTests` using Mockito.

If you have extra time and would like to get more practice working with Mockito, run through the following steps to test the service layer.

1. TODO 21: Open `src/test/java/com.example.service.PurchaseServiceImplTests`. Annotate the class as a Mockito test class.
    - Use JUnit's `@ExtendWith` annotation combined with Mockito's `MockitoExtension` class.
    ```
    @ExtendWith(MockitoExtension.class)
    public class PurchaseServiceImplTests {
        ...
    }
    ```
2. TODO 22: Define a variable called `purchaseDao` of type `PurchaseDao`.
    - Have Mockito setup this variable as a mock object.
    ```
    @Mock PurchaseDao purchaseDao;
    ```
3. TODO 23: Define a variable called `purchaseService` of type `PurchaseServiceImpl`.
    - Have Mockito inject this variable with the mock `purchaseDao`.
    ```
    @InjectMocks PurchaseServiceImpl purchaseService;
    ```
4. TODO 24: Within the `@Test` method, use Mockito's `when()` method to program the mock.
    - When the mock's `getAllPurchases()` method is called, have it return the expected purchases.
    ```
    when(purchaseDao.getAllPurchases()).thenReturn(expectedPurchases);
    ```
5. TODO 25: Call the method under test.
    - Capture the results of the `findAllPurchases()` method in a variable.
    ```
    List<Purchase> result = purchaseService.findAllPurchases();
    ```

6. TODO 26: Test the result. It should contain the same elements in the expected purchase list.
    - Use AssertJ to verify this. It has a time-saving method for comparing collection contents: `containsExactlyElementsOf()`.
      ```
      assertThat(result)
          .containsExactlyElementsOf(expectedPurchases);
      ```
7. TODO 27: Verify that the mock had its `getAllPurchases()` method called.
    - Use Mockito's `verify()` method to check this.
      ```
      verify(purchaseDao).getAllPurchases();
      ```
    - Note that `verify()` is sometimes the only way we can test that a method behaved as expected. In this case it is not necessary, but doesn't hurt.
8. Organize your imports, save your work. Run this test, it should pass.

8.10. Review

In this lab we learned:

- How to configure JPA with Spring.
- How to implement the JpaTransactionManager.
- How to define JPA entities.
- How to implement a DAO layer using JPA.
- How to test the JPA-based DAO layer.

IntelliJ IDEA IDE - Quick Reference

MODULE 9

This reference guide provides the specific steps for common tasks in the IntelliJ IDE.

9.1. Common Operations

The IntelliJ IDE is installed on your development machine and is available to use when working on the hands-on labs in this course.

IntelliJ is a powerful IDE with many features, some of which you may already be familiar with if you have used it before. This list of common operations is for those new to IntelliJ or who may not have used IntelliJ in a while.

The following operations covered in this quick reference are ones you will likely need to perform while working on the labs in this course.:

9.1.1. Import or Open a Project

1. Start IntelliJ IDEA.
2. From the `Welcome Screen`, click **Open**.
3. Navigate to the root folder of the project you want to open.
4. Select the `build.gradle` file from the project's root folder.
5. Click **OK**.
6. In the next dialog, choose **Open as Project**.
7. Wait for the project to load.

9.1.2. Run the Application's Main Class

1. Open the project in IntelliJ IDEA.
2. Wait for it to load completely.
3. Navigate to the main class (e.g., `com.example.App`) in the Project view.

4. Right-click on the file or the class name in the editor.
5. Select `Run 'App.main()'`.
    - *Shortcut*: With the main class open in IntelliJ:
        a. Look for the green 'play' icon ▷ in the gutter next to the `main`
        b. Click the `play` icon to run the class.

9.1.3. Run a JUnit Test Class

1. Open your project in IntelliJ IDEA.
2. Open the test class you want to run.
3. Right-click on the class name in the editor or in the Project view.
4. Select `Run 'TestClassName'`.
    - *Shortcut*: Click the green 'play' icon ▷ in the gutter next to the class declaration.

9.1.4. Create a New Java Class

1. In the Project view, right-click on the package where you want to create the new class.
2. Select `New` → `Java Class`.
3. In the "New Java Class" dialog, enter the name of the class.
4. Click `OK`.
5. The new class will be created in the specified package and opened in the editor.

9.1.5. Configure IntelliJ Auto Import Features

IntelliJ does not automatically add imports as you type or paste code unless you enable its `Auto Import` feature.

1. Go to `File` → `Settings` (or `IntelliJ IDEA` → `Preferences` on macOS).
2. Navigate to `Editor` → `General` → `Auto Import`.
3. Check the box for `Add unambiguous imports on the fly`.
4. Optionally, check `Optimize imports on the fly` to automatically remove unused imports.
5. Click on the 'Apply' button
6. Click on the 'OK'. button.

9.1.6. Add Required Imports

Have you configured IntelliJ to add imports on the fly? If not then do so now. See the `Configure IntelliJ Auto Import Features` section above for instructions. Once configured, IntelliJ will add imports automatically as you type or paste code.

- To add imports manually do this:
    i. Paste or add code new code
    ii. Unresolved class names will be highlighted in red.
    iii. Place your cursor on the unresolved class name and:
        a. Enter [Alt+Enter] (Windows/Linux)
        b. Enter [Option+Enter] (macOS).
    iv. Then Select **Import class** from the context menu.
- To Optimize all Imports in a file at once, press:
    i. [Ctrl+Alt+O] (Windows/Linux)

ii. [Cmd+Option+O] (macOS)

9.1.7. Find in Workspace Files

1. Open the IntelliJ `find in files` dialog using one of these methods:
    i. `[Ctrl+Shift+F]` (Windows/Linux)
    ii. `[Cmd+Shift+F]` (macOS)
    iii. `Menu → Edit → Find → Find in Files`
2. Alternatively, go to the main menu: `Edit` → `Find` → `Find in Files`.
3. In the search box, type the text you wish to look for. For example `TODO`.
4. Press [Enter].

> **Tip**
>
> IntelliJ also has a dedicated **TODO** tool window (`View` → `Tool Windows` → `TODO`) that automatically lists all `TODO` comments.

9.1.8. Synchronize Gradle Dependencies

1. After saving edits to `build.gradle`, a notification bar should appear at the top of the editor.
    i. Click the **Load Gradle Changes** icon (a small elephant with a refresh arrow) in the notification.
2. If the notification does not appear;
    i. Open the **Gradle** tool window on the right-hand side of the IDE.
    ii. In the Gradle tool window, right-click on the root project and select `Refresh Gradle Project`.

3. If you have made changes to the `build.gradle` file and need to refresh dependencies manually, you can do so from the command line:

```
gradle --refresh-dependencies
```

9.1.9. Handling "cannot be resolved to a type" Errors

1. The "cannot be resolved to a type" error typically indicates that the class is not found in the classpath.
2. Ensure that the required dependency is included in the `build.gradle` file.
3. If the dependency is present, but the class is still not found, you may need to synchronize Gradle dependencies (see the section in this document titled: `Synchronize Gradle Dependencies`).
4. After synchronizing, retry the operation that was failing due to the error.

Eclipse IDE - Quick Reference

MODULE 10

This reference guide provides the specific steps for common tasks in the Eclipse IDE.

10.1. Configure Eclipse Settings

Using a Local Gradle Installation

Importing Gradle based projects can take some time. You can speed up the process by optimizing the Gradle project imports.

1. Use the Windows File Explorer to determine the location of the Gradle installation on your lab machine. Make sure you've got the right directory. Start by checking this directory, `C:\software\gradle-8.7`.
2. Open Eclipse.
3. Choose `Preferences → Gradle` from the `Window` menu.
4. In the `Gradle` settings area choose `local installation directory`.
5. Click on the `browse` button and enter the location you found earlier where Gradle has been installed on your lab machine.
6. Click on the `Select Folder` button to save your choice.
7. Back in the Gradle settings area, click on the `Apply and Close` button.

Eclipse will now use the locally installed version of Gradle instead of Gradle Wrapper by default when importing Gradle based projects.

10.2. Common Operations

The Eclipse IDE is installed on your development machine and is available to use when working on the hands-on labs in this course.

Eclipse is a powerful IDE with many features, some of which you may already be familiar with if you have used it before. This list of common operations is for those new to Eclipse or who may not have used Eclipse in a while.

The following operations covered in this quick reference are ones you will likely need to perform while working on the labs in this course.:

10.2.1. Import or Open a Project

1. Open Eclipse IDE.
2. Choose `Import…` from the `File` menu. The Import dialog will open.
3. In the Import dialog, expand the `Gradle` folder and select `Existing Gradle Project`.
4. Click the `Next` button. The `Import Gradle Project` dialog will open.
5. For `Project Root Directory`, click `Browse…` and navigate to and select the root directory of the project you wish to import.
6. Click the dialog's 'select folder' button to confirm your selection.
7. Click the `Finish` button on the `Import Gradle Project` dialog.
8. Wait for the project to be imported. This may take a few moments depending on the size of the project and the number of dependencies.

10.2.2. Run the Application's Main Class

1. Navigate to the application's main class (e.g., `com.example.App`) in the Eclipse Package Explorer.
2. Right-click on the main class file.
3. Select `Run As` → `Java Application`.
4. Output from tha application will be displayed in the console tab in Eclipse.

10.2.3. Run a JUnit Test Class

1. Open the test class you want to run.
2. Right-click on the Java file in the Eclipse Package Explorer or in the editor window.
3. Select `Run As` → `JUnit Test`.
4. Wait for the test to complete.

   The results will be displayed in the JUnit view, which shows the status of each test case (passed, failed, etc.).

10.2.4. Create a New Java Class

1. In the Eclipse Package Explorer, right-click on the package where you want to create the new class.
2. Select `New` → `Class`.
3. In the "New Java Class" dialog, enter the name of the class and select any additional options (like adding a main method if needed).
4. Click `Finish`.

10.2.5. Add Required Imports

After pasting code with unresolved class names:

1. Use the keyboard shortcut [Ctrl+Shift+O] (Windows/Linux) or [Cmd+Shift+O] (macOS).
2. Alternatively, go to the main menu: `Source` → `Organize Imports`.

   The above command should add all necessary imports as well as remove unused ones for the current file.

   If the class is not found, you should check that the required dependency included in the `build.gradle` file?

If you have the required dependency in your `build.gradle` file, but the class is still not found, you may need to synchronize Gradle dependencies (see the section in this document titled: `Synchronize Gradle Dependencies`)

10.2.6. Find in Workspace Files

1. Go to the main menu: `Search` → `File…`.
2. In the "Containing text" field, type the text you wish to look for. For example `TODO`.
3. Ensure the "Scope" is set to **Workspace** or **Enclosing Projects**.
4. Click **Search**.

> **Tip**
>
> Eclipse has a dedicated **Tasks** view that automatically finds all `TODO` comments. Open it via `Window` → `Show View` → `Tasks`.

10.2.7. Synchronize Gradle Dependencies

After making changes to `build.gradle` and saving the file, you may need to synchronize the Gradle dependencies to ensure that Eclipse recognizes any new or updated dependencies.:

1. Open a command prompt and navigate to the project's root directory (`C:\LabWork\spring-project-basics`).
2. Run the following command to refresh the Gradle project:

```
gradle --refresh-dependencies
```

3. Go back to the Eclipse IDE.
4. Right-click on the current project in the Eclipse Package Explorer.
5. Select `Gradle` → `Refresh Gradle Project`.

6. Wait for the synchronization process to complete.
7. If you were missing any imports, you can now use the `Organize Imports` command ([Ctrl+Shift+O] or [Cmd+Shift+O]) to add them.

10.2.8. Handling "cannot be resolved to a type" Errors

1. The "cannot be resolved to a type" error typically indicates that the class is not found in the classpath.
2. Ensure that the required dependency is included in the `build.gradle` file.
3. If the dependency is present, but the class is still not found, you may need to synchronize Gradle dependencies (see the section in this document titled: `Synchronize Gradle Dependencies`).
4. After synchronizing, retry the operation that was failing due to the error.
