

# Enterprise Java with Spring

## Spring Core

### Lab 2

<b>1</b>	<b>LAB SETUP .....</b>	<b>1</b>
<b>2</b>	<b>ANNOTATION-BASED CONFIGURATION BASICS.....</b>	<b>2</b>
2.1	USING EXPLICIT COMPONENT NAME .....	2
2.2	USING DEFAULT COMPONENT NAME .....	3
2.3	SCANNING PACKAGE HIERARCHY FOR @COMPONENT CLASSES WITH <CONTEXT:COMPONENT-SCAN>.....	3
2.4	USING @SERVICE, @REPOSITORY AND @CONTROLLER .....	5
<b>3</b>	<b>ANNOTATION-BASED DI CONFIGURATION .....</b>	<b>5</b>
3.1	CONSTRUCTOR INJECTION .....	6
3.2	SETTER / METHOD INJECTION.....	7
3.3	FIELD INJECTION.....	7
3.4	PROBLEMS WITH MULTIPLE CANDIDATE BEANS .....	7
3.5	USING AUTOWIRING BY NAME TO SELECT BETWEEN MULTIPLE CANDIDATE BEANS.....	7
3.6	USING @QUALIFIER TO SELECT BETWEEN MULTIPLE CANDIDATE BEANS .....	8
3.7	USING @PRIMARY TO GIVE HIGHER PREFERENCE TO A BEAN .....	8
3.8	INJECTING LITERAL VALUES FROM A PROPERTIES FILE WITH @VALUE .....	9
3.9	MARKING OPTIONAL DEPENDENCIES WITH @AUTOWIRED(REQUIRED = FALSE) .....	9

## 1 Lab setup

Make sure you have the following items installed

- Latest LTS JDK version (at this point: JDK 21)
- A suitable IDE (Eclipse Enterprise Edition for Java) or IntelliJ IDEA
- Latest version of Maven (at this point: Maven 3.9.9)
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

In each of the main lab folders, there are two subfolders: `changes` and `final`. The `changes` subfolder holds the source code and other related files for the lab, while the `final` subfolder holds the complete Eclipse project starting from its project root folder. We will use the code from the `changes` subfolder to build up our applications from scratch and you can always fall back on the complete Eclipse project if you encounter any errors while building up the application.

## 2 Annotation-based configuration basics

The source code for this lab is found in `Annotation-Config-Basics/changes` folder.

We can create a Maven project from scratch, or we can make a copy from any of the existing Maven projects.

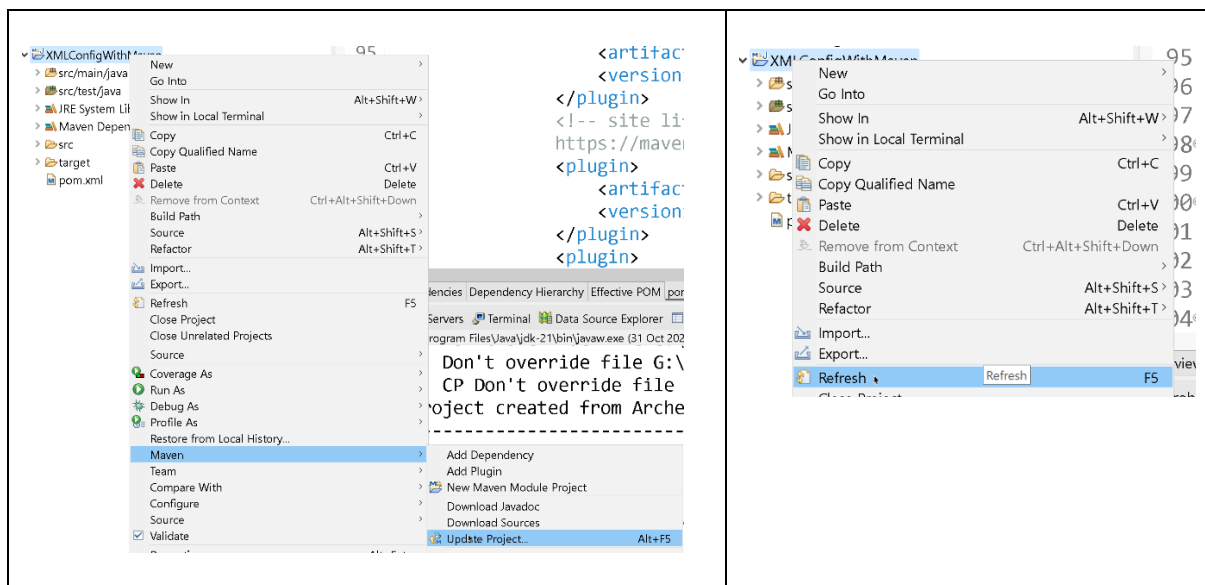
Choose any previous Maven lab project to make a copy from, for e.g.: `XMLConfigWithMaven`

In the Project Explorer, right click on `XMLConfigWithMaven`, select `Copy` and then right click in any empty space in the Explorer and select `Paste`.

For the new project name, type: `AnnotationConfigBasics`

Replace the contents of the `pom.xml` in the project with `pom.xml` from `changes`.

Right click on the project, select `Maven -> Update Project`, and then click on the project name and then refresh.



Delete all the packages and files in `src/main/java` and `src/main/resources`. We will start populating the project from scratch.

### 2.1 Using explicit component name

Copy the following files from `changes` into `src/main/resources`

`beansDefinition.xml`

Create a new package `com.workshop.annotation` in `src/main/java`:

Copy the following files from changes into `com.workshop.annotation` in `src/main/java`:

```
AnnotationConfigBasicMainApp.java
Exercise.java
SwimmingExercise.java
```

Important points to note:

- The container will scan the package (and subpackages) specified in `context:component-scan` to locate classes for purposes of initializing and registering beans. These classes will be annotated with `@Component`
- If an explicit name is specified for the `@Component`, this name is used to retrieve the bean from the container via `getBean`

The `@Component` annotation in Spring Framework is a class-level annotation that marks a Java class as a bean, registering it in the Spring `ApplicationContext` container and making it automatically detectable via component scanning

Open and right click on `AnnotationConfigBasicMainApp` and select `Run As -> Java Application`.

Verify that the correct bean has been created and its output logged to the console correctly.

Comment out the `@Component` in `SwimmingExercise` and run again

Notice now that we obtain a `NoSuchBeanDefinitionException` as the container is not able to locate any class in the specified package to instantiate as a bean.

Uncomment the `@Component` in `SwimmingExercise` and run again

## 2.2 Using default component name

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from changes:

```
SwimmingExercise-v2.java
AnnotationConfigBasicMainApp-v2.java
```

The default name of a component (if no explicit name is specified) is the name of the bean class with the first letter in lowercase.

Open and right click on `AnnotationConfigBasicMainApp` and select `Run As -> Java Application`.

Verify that the correct bean has been created and its output logged to the console correctly.

## 2.3 Scanning package hierarchy for `@Component` classes with `<context:component-scan>`

Create a new package `com.workshop.annotation.basics` in `src/main/java`

Copy the following files from `changes` into this new package:

`CyclingExercise.java`

Create a new package `com.workshop.second` in `src/main/java`

Copy the following files from `changes` into this new package:

`JoggingExercise.java`

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from `changes`:

`AnnotationConfigBasicMainApp-v3.java`

Open and right click on `AnnotationConfigBasicMainApp` and select `Run As -> Java Application`.

Notice that both `swimmingExercise` and `cyclingExercise` have been registered as beans, because both these classes are in the package hierarchy specified in `<context:component-scan>` and are also annotated with `@Component`

However `joggingExercise` is not registered even though it is annotated with `@Component` as it is in a completely different package hierarchy from the one specified in `<context:component-scan>`

You will notice a list of additional classes displayed: these are Spring framework's internal beans that are used to perform configuration of the initialized beans and also autowiring using the `@Autowired` annotation that we will examine later:

```
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.event.internalEventListenerProcessor
org.springframework.context.event.internalEventListenerFactory
```

In `beansDefinition.xml`, make the following change:

```
<context:component-scan base-package="com.workshop.annotation.basics" />
```

Run `AnnotationConfigBasicMainApp` again. This time, notice that only the `cyclingExercise` bean is picked up.

In `beansDefinition.xml`, make the following change:

```
<context:component-scan base-package="com.workshop.annotation,
com.workshop.second"/>
```

Notice now that all 3 classes (`swimmingExercise` and `cyclingExercise` and `joggingExercise`) are registered as beans in the container.

## 2.4 Using `@Service`, `@Repository` and `@Controller`

All these 3 annotations are specialized forms of `@Component` and the container will scan for classes annotated with them and initialize and register them in the same manner that it does for classes annotated with `@Component`.

`@Repository` is typically used with the Spring Data framework for persisting data to a backend database. Its goal is to catch persistence-specific exceptions and re-throw them as one of Spring's unchecked exceptions.

`@Service` is typically used to mark classes whose methods implement the core business logic of the application, typically in the service layer. They are used in Spring MVC and REST API apps.

`@Controller` is used to mark classes that are used as controllers in Spring MVC apps which will handle incoming HTTP requests.

Replace the `@Component` on all 3 classes below with these annotations instead. You can randomly choose any class to apply these annotations to, for e.g.

- Apply `@Service` to `SwimmingExercise`
- Apply `@Repository` to `CyclingExercise`
- Apply `@Controller` to `JoggingExercise`

Run `AnnotationConfigBasicMainApp` again. Notice that all 3 classes are picked up and registered as beans in the container as usual.

Close all the open editor tabs to prepare for the next project creation.

Close this project to prevent confusion with the next project you are going to create (right click on project entry and select Close Project)

## 3 Annotation-based DI configuration

The source code for this lab is found in `Annotation-Config-DI/changes` folder.

We can create a Maven project from scratch, or we can make a copy from any of the existing Maven projects.

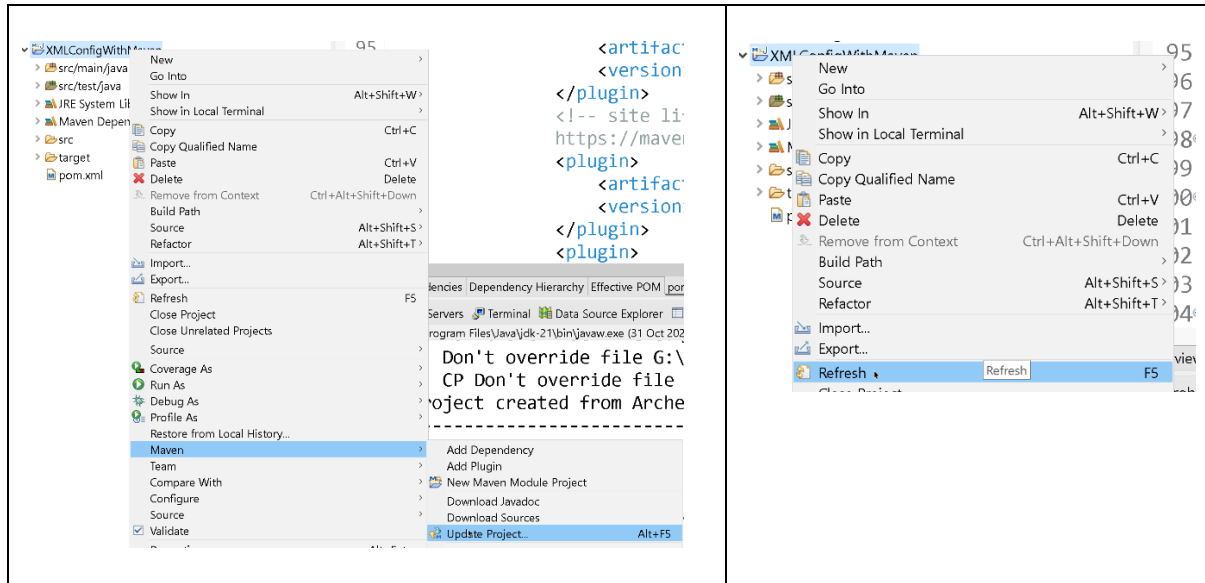
Choose any previous Maven lab project to make a copy from, for e.g.: `AnnotationConfigBasics`

In the Project Explorer, right click on `AnnotationConfigBasics`, select Copy and then right click in any empty space in the Explorer and select Paste.

For the new project name, type: `AnnotationConfigDI`

Replace the contents of the `pom.xml` in the project with `pom.xml` from changes.

Right click on the project, select Maven -> Update Project, and then click on the project name and then refresh.



Delete all the packages and files in `src/main/java` and `src/main/resources`. We will start populating the project from scratch.

### 3.1 Constructor injection

Copy the following files from `changes` into `src/main/resources`

`beansDefinition.xml`

Create a new package `com.workshop.annotation` in `src/main/java`:

Copy the following files from `changes` into `com.workshop.annotation` in `src/main/java`:

`AnnotationConfigDIMainApp.java`  
`CollegeStudent.java`  
`Exercise.java`  
`Student.java`  
`SwimmingExercise.java`

The `@Autowired` annotation is a core part of Spring's dependency injection (DI) mechanism. When Spring detects `@Autowired` on a field, constructor, or method, it searches the container for a bean of a compatible type, injects that bean into the target component automatically based on type matching.

In this case, the dependency `myExercise` in `CollegeStudent` is of type `Exercise`, and currently there is only one bean, `swimmingExercise`, that is of this type: so it is autowired for DI here.

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`.

Verify that the correct bean has been created and its output logged to the console correctly.

### 3.2 Setter / method injection

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from changes:

`CollegeStudent-v2.java`

Here, `@Autowired` is applied to the setter method.

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`.

Verify that the correct bean has been created and its output logged to the console correctly.

### 3.3 Field injection

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from changes:

`CollegeStudent-v4.java`

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`.  
Verify that the correct bean has been created and its output logged to the console correctly.

### 3.4 Problems with multiple candidate beans

Copy the following files from changes into `com.workshop.annotation` in `src/main/java`:

`CyclingExercise.java`  
`JoggingExercise.java`

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`.

Now we have a `NoUniqueBeanDefinitionException` thrown because the container is unable to decide between these three candidate beans of the same type (`CyclingExercise`, `JoggingExercise`, `SwimmingExercise`) to initialize the dependency. In addition, a `UnsatisfiedDependencyException` is also thrown, since the container is now no longer able to inject the dependency for the `collegeStudent` bean.

### 3.5 Using autowiring by name to select between multiple candidate beans

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from changes:

`CollegeStudent-v5.java`

Here we rename the dependency in `collegeStudent` (the single private field) to the name of the registered bean that we wish to be autowired for DI. Remember that a bean is automatically registered with the IoC container using its class name in camel case.

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`. Verify that the correct bean has been created and its output logged to the console correctly.

### 3.6 Using `@Qualifier` to select between multiple candidate beans

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from changes:

`CollegeStudent-v6.java`

Here, we use the `@Qualifier` annotation to explicitly state the name of the bean that we wish to be autowired for DI based on field injection. The field name can be any value now (`myExercise`) and does not need to match any particular bean name as this is specified in the `@Qualifier` annotation

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`.

When working with constructor injection (as opposed to field injection), `@Qualifier` must be applied to the specific arguments in the constructor method signature.

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from changes:

`CollegeStudent-v7.java`

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`.

Notice that the bean is now instantiated with the correct dependency due to the use of the `@Qualifier` annotation in the constructor method signature

### 3.7 Using `@Primary` to give higher preference to a bean

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from changes:

`CollegeStudent-v8.java`  
`CyclingExercise-v2.java`

Here we apply the `@Primary` annotation to the actual bean class that we wish to be used as the candidate bean to clarify ambiguity between multiple potential candidate beans in autowiring for DI.

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`.



### 3.8 Injecting literal values from a properties file with @Value

Make changes to the following files in `src/main/resources` from `changes`

```
highSchool.properties  
beansDefinition-v2.xml
```

Copy the following files from `changes` into `com.workshop.annotation` in `src/main/java`:

```
HighSchoolStudent.java
```

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from `changes`:

```
AnnotationConfigDIMainApp-v2.java
```

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`. Notice that the correct values are read for `HighSchoolStudent`'s fields.

### 3.9 Marking optional dependencies with @Autowired(required = false)

Copy the following files from `changes` into `com.workshop.annotation` in `src/main/java`:

```
Study.java
```

Make changes to the following files in `com.workshop.annotation` in `src/main/java` from `changes`:

```
HighSchoolStudent-v2.java
```

Open and right click on `AnnotationConfigDIMainApp` and select `Run As -> Java Application`. The application runs fine even if there is no implementation available for the `myStudy` dependency in `HighSchoolStudent`, as this is marked as optional. In `HighSchoolStudent`, change

```
@Autowired(required = false)  
private Study myStudy;
```

to

```
@Autowired  
private Study myStudy;
```

and this time running the application results in a `NoSuchBeanDefinitionException` thrown because the container is unable to locate any class of type `Study` to instantiate as a bean. In addition, a `UnsatisfiedDependencyException` is also thrown, since the container is now no longer able to inject the dependency for the `highSchoolStudent` bean.

If we wish to make a dependency optional and access it within our code, we must make sure to include a null check to safeguard against autowiring failures.

Copy the following files from changes into com.workshop.annotation in src/main/java:

StudyAtNight.java

Make changes to the following files in com.workshop.annotation in src/main/java from changes:

HighSchoolStudent-v3.java

Open and right click on AnnotationConfigDIMainApp and select Run As -> Java Application.

The dependency is initialized correctly and the output is as expected.

Now comment out the @Component from StudyAtNight, so it will no longer be picked up and registered as a bean

Run the app again. This time the use of Optional as well as the null check ensures that there is no NoSuchBeanDefinitionException exception or a NullPointerException thrown.