Enterprise Java with Spring
Spring Core
Lab 4

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 Creating a Spring Boot project via Spring Initializr

Navigate to the Spring Initialzr at:

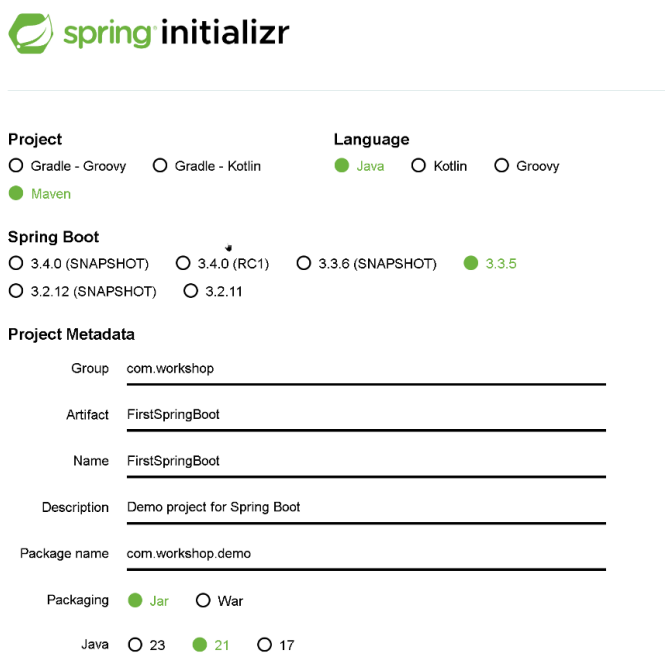
https://start.spring.io/

Key in the values for the fields as shown below.

Group: `com.workshop`

Artifact: `FirstSpringBoot`
Name: `FirstSpringBoot`
Description: `Demo project for Spring Boot`
Package name: `com.workshop.demo`

The latest stable version of Spring Initializr will be automatically selected (which may be different from the one shown below). Use this instead of any of the newer snapshot versions which may still be unstable.



There is no need add any dependencies at this point for this basic project. We will do this later.

When done, click Generate.

Download the Zip file and extract its contents with 7-zip. Place the extracted folder in your Eclipse workspace directory.

Check the contents of the root folder.
In addition to the standard `pom.xml`, it contains 3 other visible files:

HELP.md is a markdown file typically used in projects hosted on a GitHub repo to provide an overall general description of the project. It is standard with pretty much all open source projects these days.

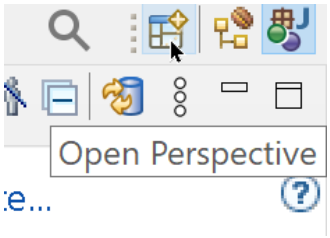
`mvnw` and `mvnw.cmd` are Linux and Windows script files used to execute Maven Wrapper.
Maven Wrapper is an extension of standard Maven that allows the source code and Maven build system. This allows other developers to build the project code without worrying about having the correct matching version of Maven installed. Instead of the usual `mvn` command, we will now use `mvnw` instead followed by the standard phases to be executed (e.g. clean package).

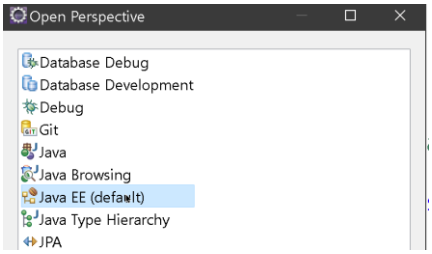
There is hidden folder `.mvn` in the root project folder. The `.mvn/wrapper` directory has a jar file maven-wrapper.jar that downloads the required version of Maven if it's not already present. It installs

it in the `./m2/wrapper/dists` directory under the user's home directory. The location for downloading maven is provided in `maven-wrapper.properties` file:

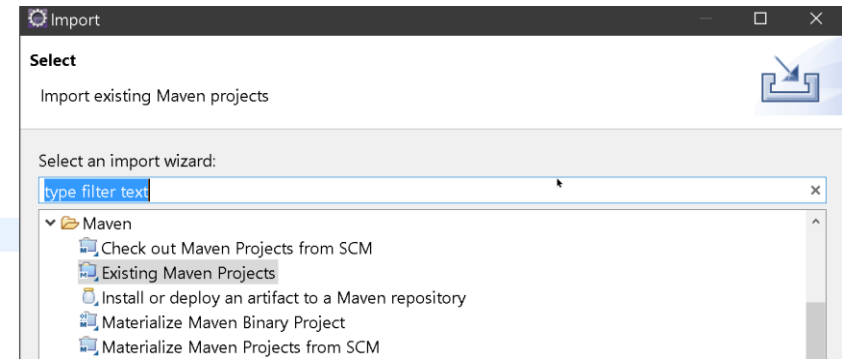
There is also a single system file `.gitignore`, which is typically placed in the root folder of a Git project. This facilitates this Maven Spring Boot project to be directly checked into a Git versioning system.

Switch to Java EE perspective if you are not already in there, either using the icons at the upper right hand corner or via the menu: Window -> Perspective -> Open Perspective

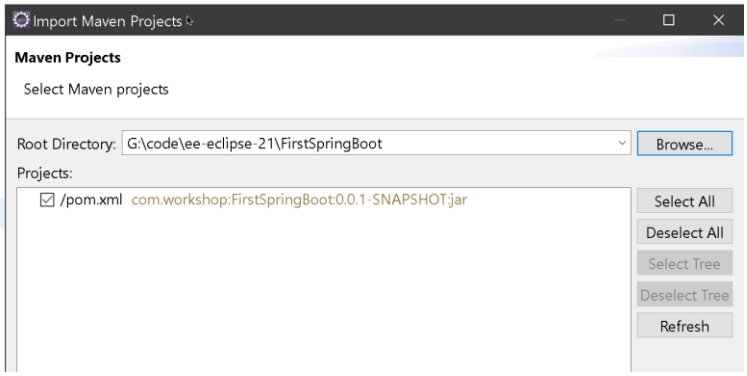




Go to File -> Import -> Maven -> Existing Maven Project.



Click Next, select Browse, and select the FirstSpringBoot folder, which should also automatically highlight the pom.xml in that folder. Click Finish.



3 Spring Boot Starter templates and POM details

Dependency management is an important and complex task in large Maven projects which can potentially include dozens of dependencies from different libraries and frameworks. Spring Boot starter templates provide a easy way to help aggregate together related dependencies with the correct versions in order to implement a variety of useful functionalities, such as database connections for relational and NoSQL databases, web services, social network integration, performance and monitoring, logging, template rendering, and so on. Spring Boot comes with over 50+ different starter modules, and the list keeps growing with each new release of Spring.

Open the project POM in Eclipse and check the dependencies tab. Notice that this autogenerated POM currently has 2 dependencies here (`spring-boot-starter` and `spring-boot-starter-test`), both of which are Spring Boot starters. Notice that these two starter dependencies do not have their version number specified: this is because that version is specified in the `spring-boot-dependencies` BOM which we will discuss later.

You can check the latest version for both these dependencies at:
https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter
https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-test

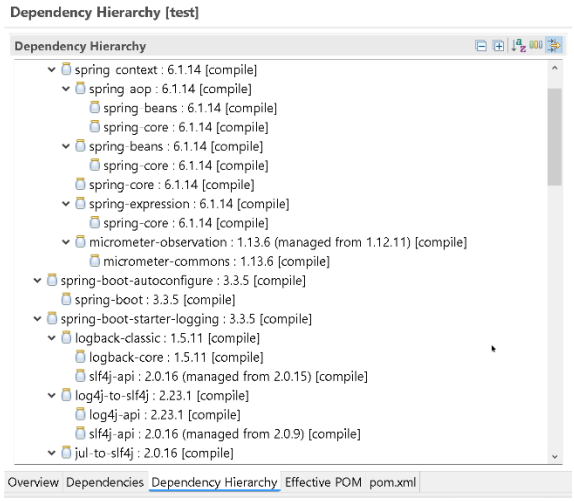
If you check out the POM for one of the latest versions of `spring-boot-starter`

You will notice it has a <dependencies> list that includes the basic dependencies for a simple Spring Boot application:

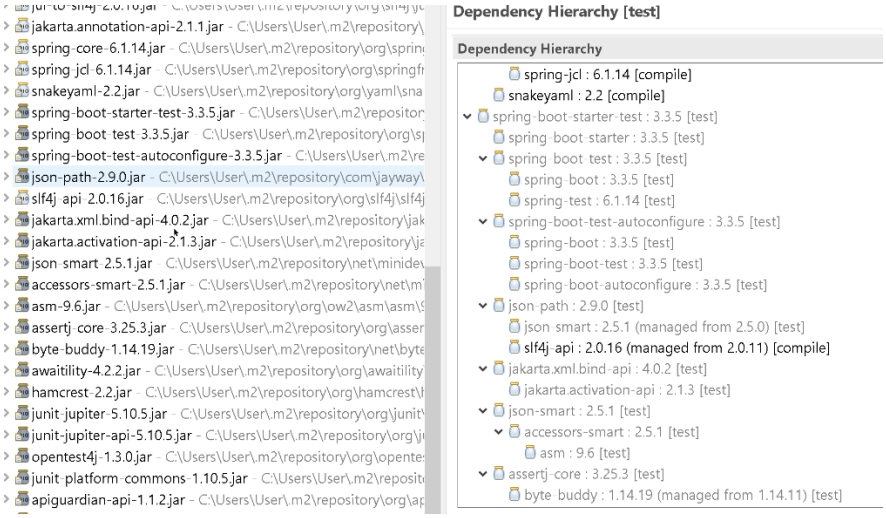
- spring-boot - the core of Spring Boot itself
- spring-boot-autoconfigure - this handles auto configuration of Spring Boot
- spring-boot-starter-logging - this provides logging infra for Spring Boot, which by default is based on Logback
- spring-core - the core of Spring DI / IOC framework (which we included as a dependency in the POM of our Maven projects in earlier lab sessions)
- etc, etc

The Spring development team has taken the necessary testing effort to ensure that all these direct dependencies (as well as their transitive dependencies) will work together for their specified versions. This prevents the possible versions conflicts in transitive dependencies that we have seen in an earlier lab.

You can view the entire dependency tree in the Dependency Hierarchy Tab.



Notice that the JAR dependencies grouped in `spring-boot-starter-test` are all shaded in the Maven dependencies view in the Project explorer because this dependency has the `test` scope (as discussed in an earlier lab). This means they will only be available for compiling unit test code placed in `src/test/java` folder.



Every Spring Boot project comes with an autogenerated class `xxxApplication` located in the specified package `src/main/java`. Here, the name of the file is `FirstSpringBootApplication.java`

This class is annotated with @SpringBootApplication. This @SpringBootApplication annotation is actually a shorthand for the combination of 3 separate annotations:

- @Configuration
- @ComponentScan
- @EnableAutoConfiguration

We have already seen the meaning of @Configuration and @Component scan in the context of Java-based configuration of the IoC container in previous lab sessions.

The @EnableAutoConfiguration in addition auto configures Spring Boot by adding beans based on classpath settings, the existence of other specific beans, and various property settings.

For example, if we are creating a web application (which we will be doing in a later lab session), we will include the `spring-boot-starter-web` starter dependency. This will result in specific `spring-webmvc` dependency JARs being added to the application build classpath. When this happens, the auto configuration functionality will set up a DispatcherServlet automatically in the background as this is a core feature in a Spring web app. If auto configuration was not present (such as in the case of a standard Maven non-Spring Boot project), then the developer would need to manually create it themselves. We will see an example of this in a later lab session.

Run the app by right clicking in `FirstSpringBootApplication` and select Run as -> Java application.

The output in the Console should show the Spring Boot logo along with some basic logging messages.

4 Defining and retrieving beans with annotation configuration

The source code for this lab is found in `Basic-Spring-Boot/changes` folder.

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

```
CyclingExercise
Exercise
FirstSpringBootApplication
JoggingExercise
SwimmingExercise
```

Run the app by right clicking in `FirstSpringBootApplication` and select Run as -> Java application

In output listing the registered beans, we should be able to see the 3 beans whose classes we had annotated with @Component and which were picked up via component scanning enabled through the @ComponentScan annotation that is implicit within @SpringBootApplication.

In addition to these 3 beans, there are also many other beans registered with the container. These beans were loaded and registered via Spring's autoconfiguration process described earlier and remain available in the container to perform a variety of tasks specific to a Spring Boot application.

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

```
CollegeStudent.java
HighSchoolStudent.java
Student.java
```

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

```
FirstSpringBootApplication-v2.java
```

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

```
application.properties
```

Here, we are using the same annotations we have used before in the Java-based and annotation-based configuration approach. All the classes are marked with @Component and are therefore automatically picked up and registered as beans in the IoC container.

We use @Autowired and @Qualifier on the `myExercise` dependencies for HighSchoolStudent and CollegeStudent in order to select between 3 possible candidate bean classes (CyclingExercise, JoggingExercise and SwimmingExercise).

All the user defined properties that we previously placed in `highschool.properties` and `collegestudent.properties` are now centralized in the `application.properties`.

We also do not need to specify this file name as the location of our user-defined properties using @PropertySource as we did in a previous lab session. The autoconfiguration of @SpringBootApplication makes `application.properties` the default properties file for all Spring Boot applications, and the application will look in here to inject values for dependencies marked with @Value (such as the dependencies in both HighSchoolStudent and CollegeStudent).

We can also define a variety of other Spring related properties in this file, as we will see later.

Run FirstSpringBootApplication in the normal fashion. Verify the output is as expected.

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

`BeanConfig.java`

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

`FirstSpringBootApplication-v3.java`

Here, we define an explicit @Configuration class with a @Bean method in it to produce a bean that encapsulates an existing Java library class (Random), which we subsequently retrieve in `FirstSpringBootApplication`.
This is a repetition of an exercise that we performed in a previous lab to distinguish between the need to use @Bean vs @Component to define and produce a bean.

Run FirstSpringBootApplication in the normal fashion. Verify the output is as expected.

Remember that @SpringBootApplication also implicitly incorporates the @Configuration annotation as well. This means we can shift the @Bean method definition from `BeanConfig` into FirstSpringBootApplication and it would still run correctly.

Try this out to verify that this is indeed the case.

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

`MathStudent.java`

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

`beansDefinition.xml`

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

```
BeanConfig-v2.java
FirstSpringBootApplication-v4.java
```

We can also use the @ImportResource annotation to specify an XML configuration file with bean definitions to be loaded into the container as well. This annotation can be specified in any class with the @Configuration annotation (so here this will be either BeanConfig or FirstSpringBootApplication)

Run FirstSpringBootApplication in the normal fashion. Verify the output is as expected.

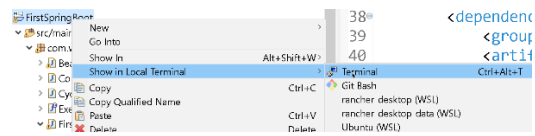
We can see now that a bean can be defined using any one of the 3 configuration approaches that we have studied in previous lab sessions: XML, annotation-based or Java based.
These beans are registered with the IoC container and can then be retrieved using the getBean method of the ApplicationContext container or injected by the IoC container to any @Autowired dependency in any registered bean class.

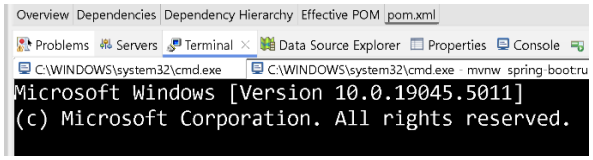
So far, we have been running the application directly from within Eclipse. We can also run it as a Maven build.

Right click on the project entry and select Run As -> 3 Maven Build -> and use `spring-boot:run` as the goal in the Run Configuration.

This uses the `run` goal of the `spring-boot-maven-plugin` that was configured in the project POM.

We can alternatively run the app using Maven Wrapper from the command line.
Right click on the project, select Show in Local Terminal -> Terminal which should open a Terminal view in the project root folder that is visible in one of the bottom panes.





Alternatively, you can just open a command terminal or shell in Windows in the normal way and navigate directly to this folder.

Then type this command to run the Maven goal: `mvnw spring-boot:run`

5 Configuring application properties

In addition to user defined properties, you can also set Spring framework specific properties in `application.properties` to control and configure various aspects of Spring Boot application behaviour.

The `application.properties` file is the central place to define various settings and properties that control the application's behavior. It allows you to define application-wide properties in one place, making it easier to manage configurations without scattering settings across multiple files or hardcoding values.

The full list of properties that can be set in the file:

For a simple command line based Spring Boot application, there are not many relevant Spring Boot properties to configure at the moment. Later, as we build more larger apps with more complex functionality, we will configure more properties in here.

Add this property to `application.properties` and save

```
spring.main.banner-mode=off
```

Run the app again.
Notice that the main Spring Boot banner is no longer appearing as part of the initial log output messages.

The format shown in `application.properties` uses the dot to separate terms in a hierarchical structuring of names.

An alternative popular format for specifying properties in a Spring Boot application is YAML

Rename the following files in `src/main/resources`

```
application.properties
```

to

```
application.tmp
```

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

```
CollegeStudent-v2.java
FirstSpringBootApplication-v5.java
```

Run the app. As expected, there are errors because we are missing the `application.properties` which provides the necessary properties to inject into the @Value dependencies in both CollegeStudent and HighSchoolStudent.

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

```
application.yml
```

This essentially specifies the properties from `application.properties` in YAML format.

Run the app. The Spring framework automatically picks up this YAML properties file and obtains the required properties from it to perform DI correctly.

Note that we have removed the Map dependency from CollegeStudent as it is not possible to specify this in YAML format. For that purpose we would need to specify properties in a class using @ConfigurationProperties

You can delete `application.yml` and rename `application.tmp` to `application.properties`

6 Implementing start up logic and passing command line arguments

As mentioned earlier, the primary entry point into a Spring Boot app is the SpringApplication class which will create beans from any classes that implement the CommandLineRunner interface and launch them via their callback `run` method. These allow the implementation of start up logic in an app. For e.g. if you are running a web app that interacts with a backend database, any related initialization of domain objects from a database table can be executed here.

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

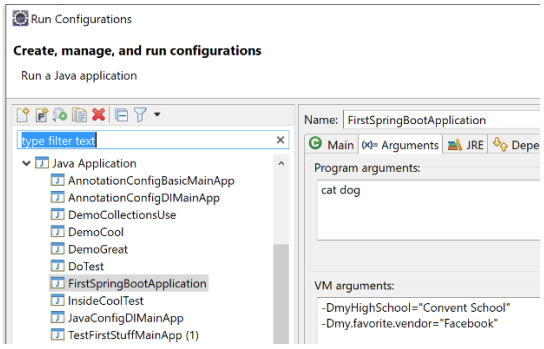
`CommandLineAppStartupRunner`

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

`FirstSpringBootApplication-v7.java`

Notice that we have now restored FirstSpringBootApplication back to its original autogenerated form. The classes that we wish to use in our app are now placed as @Autowired dependencies in CommandLineAppStartupRunner.

To pass command line arguments to this app when running it, right click on the project, select Run As -> Run Configurations.
Look for FirstSpringBootApplication in the Java Application section and click on the Arguments tab. Enter these values



Program Arguments tab: `cat dog`
VM Arguments tab:

```
-DmyHighSchool="Convent School"
-Dmy.favorite.vendor="Facebook"
```

When done, click on Run.

Verify that the two arguments passed in (`cat dog`) are output correctly.
The VM arguments preceded by -D in the form of key-value pairs will be incorporated as properties into Spring's Environment abstraction and will be accessible for injection into all @Value annotated properties in any bean class.

If there are already existing properties specified in the application which are identical to the command line arguments passed in, then these are overridden by the command line arguments.
For e.g. the `myHighSchool` property specified here overrides the one specified in `application.properties`.

7 Generating an executable JAR file

To generate an executable, standalone JAR (which will be an Uber JAR) for the project, right click on the project entry and select Run As -> 3 Maven Build -> and use `clean package` as the goal in the Run Configuration.
When the build is complete, right click on the project entry and perform a refresh.

The generation of the Uber JAR is actually performed by the `spring-boot:repackage` goal of the `spring-boot-maven-plugin` that was configured in the project POM.

This generates two files in the `target` subfolder: `xxx.jar` (which contains the classes for your application code as well as the Spring-related JAR dependencies) and `xxx.jar.original`

In `xxx.jar`

- The `BOOT-INF/classes` contains all the classes for your application code as well as any properties files placed in `src/main/resources`
- The `BOOT-INF/lib` contains all the Spring-related JARs that your application requires as dependencies
- The root of the JAR contains a package `org.springframework.boot.loader` which helps perform custom class loading that allows the reading of JARs within this JAR

Right click on the target folder, - Show in Local Terminal -> Terminal

There are two ways to execute this JAR file and pass it command line arguments as well as Spring properties.

The proper form (where the Spring properties are specified as system properties with -D preceding the `-jar` option:
NOTE: You need to copy the command which is split across two lines and join them into a single line before pasting into the command prompt.

```
java -DmyHighSchool="Convent School"
-jar FirstSpringBoot-0.0.1-SNAPSHOT.jar cat dog
```

A shorter form approach of the proper form is to use -- as a shortcut for the -D prefix:

```
java -jar FirstSpringBoot-0.0.1-SNAPSHOT.jar cat dog
--myHighSchool="Convent School"
```

Here, we are actually passing the properties as command line arguments to the application, but the Spring framework correctly interprets them as user-defined Spring properties and uses them as such.

We can also pass along any relevant Spring Boot specific properties, for e.g. to run the app without displaying the obligatory Spring Boot banner at the start:

```
java -jar FirstSpringBoot-0.0.1-SNAPSHOT.jar cat dog
--myHighSchool="Convent School" --spring.main.banner-mode=off
```