Spring Core Workshop
Lab 4

1 Lab setup

Make sure you have the following items installed

- Latest version of JDK 11 (note: labs are tested with JDK 11 but should work with higher versions with no or minimal changes)
- Eclipse Enterprise Edition for Java (or a suitable alternative IDE for Enterprise Java)
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

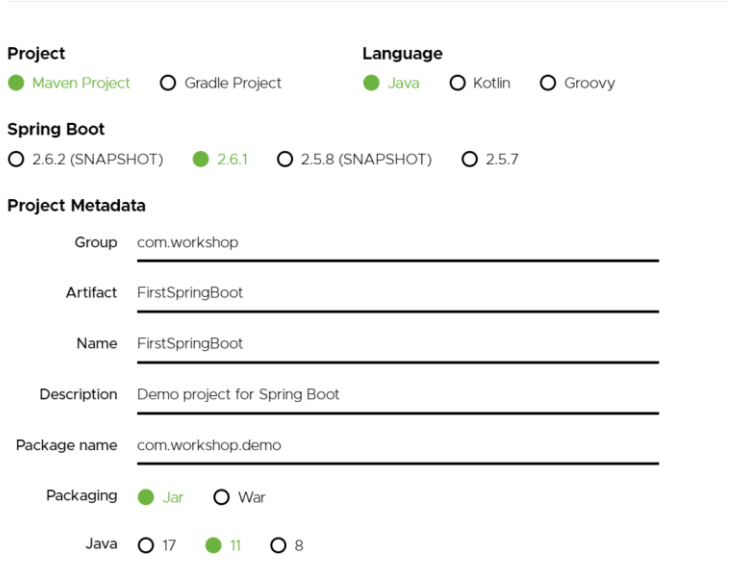
In each of the main lab folders, there are two subfolders: `changes` and `final`. The `changes` subfolder holds the source code and other related files for the lab, while the `final` subfolder holds the complete Eclipse project starting from its project root folder. We will use the code from the `changes` 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 and click Generate. There is no need add any starter dependencies at this point.



Download the Zip file and unzip it in a suitable folder anywhere (you can also place it directly 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. https://www.makeareadme.com/

`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.

In Eclipse, switch to Java EE perspective.

Go to File -> Import -> Maven -> Existing Maven Project. Select the FirstSpringBoot folder that you just unzipped and 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.

The autogenerated project 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

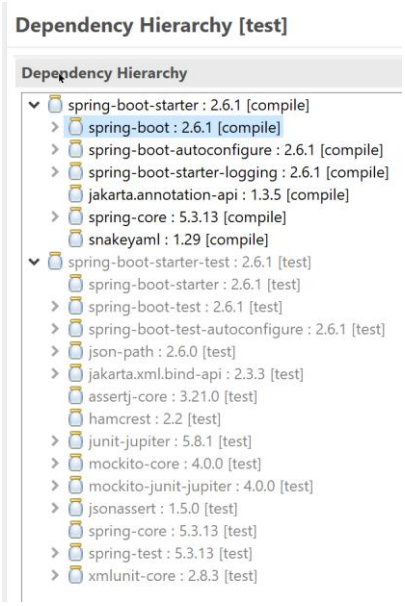
If you check out the POM for one of the latest versions of `spring-boot-starter`
https://repo1.maven.org/maven2/org/springframework/boot/spring-boot-starter/2.6.1/spring-boot-starter-2.6.1.pom
You will notice it has a <dependencies> list that includes the basic dependencies for a simple Spring Boot application:

- 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 (discussed in an earlier lab)
- 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 drill down further to check out the transitive dependencies of each of these direct dependencies by checking out their respective POMs, for e.g. for `spring-boot`:
https://repo1.maven.org/maven2/org/springframework/boot/spring-boot/2.6.1/spring-boot-2.6.1.pom
You will see it contains `spring-core` and `spring-context` as transitive dependencies.

You can obviously also 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).

Returning to the original project POM, we can see that it is child POM of a parent project `spring-boot-starter-parent`, which essentially is the parent POM for all Spring Boot projects auto-generated through Spring Initializer or through STS (the Spring Boot plugin for Eclipse)

You can find the latest versions for this here:
https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-parent

An example of one of the latest versions:
https://repo1.maven.org/maven2/org/springframework/boot/spring-boot-starter-parent/2.6.1/spring-boot-starter-parent-2.6.1.pom

Notice that the packaging type is POM. This is not a normal Maven project that has application source code which is compiled and packaged for distribution in a JAR. Instead, it exists primarily to provide configuration info to any child POMs that inherit from it. In that sense, it is conceptually similar to the super POM which is inherited by all project POMs in a standard Maven project.

In the <properties> section, there is a property <java.version> which is used in <maven.compiler.source> and <maven.compiler.target> to set the JDK version. You can override this in the <properties> section of the actual project POM.

There are two main sections here: `<resources>` and `<pluginManagement>`

The `<resources>` element is used here to specify the files in `src/main/resources` which will have variables interpolated into them (this is achieved via filtering and inclusion / exclusion).

- The first block contains `<filtering>true</filtering>`, which means that variables will be interpolated. This means environment variables / system properties can substituted into properties specified in any of the files in the <includes> section (we will see this later in action). This block will copy all `application*.(yml|yaml|properties)` files to the output folder, and will interpolate variables.

- The second doesn't contain `<filtering>true</filtering>`, which means that by default no interpolation will be done. It also contains an <excludes> section, indicating that the files specified in this filter will be excluded by this resource block. This block will copy all other files in the same folder but without variable interpolation.

From a previous lab, we demonstrated that the super POM for all Maven projects already has a separate <plugins> section that explicitly declares all the key Maven plugins that will be used in the `default` and `clean` build life cycle.

- maven-clean-plugin
- maven-resources-plugin
- maven-jar-plugin
- maven-compiler-plugin
- maven-surefire-plugin
- maven-install-plugin
- maven-deploy-plugin
- maven-site-plugin

The <pluginManagement> section is used to provide additional configuration info for some (not all) of these plugins.

Always keep in mind that for a plugin to be accessible in order for its goals to be executed, it must be declared in a <plugin> element within a separate <plugins> section OUTSIDE of the <pluginManagement> section, but still within the main <build> section.

One of the plugins in the <pluginManagement> section here is the `spring-boot-maven-plugin`, which is an important plugin for a Spring Boot specific Maven project.
https://docs.spring.io/spring-boot/docs/2.5.6/maven-plugin/reference/htmlsingle/

Some of the more important things that it will do is to:

- resolve the correct dependency versions
- package all the project dependencies (including an embedded application server if needed) in a uber / fat JAR / WAR. Thus, we no longer need to use the standard `maven-assembly-plugin` for this purpose.

You can see that the `spring-boot-maven-plugin` is explicitly defined in the <plugins> section in the <build> of the actual project POM so that it can be actually applied to the project. We need to explicitly define it here because it is not defined in the super POM of Maven.

```
    <build>
        <plugins>
```

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

Finally, to see which plugins are actually available to be used in the actual Maven project (i.e. listed in a <plugins> section outside of <pluginManagement>), you can check out the effective POM tab in the Eclipse (be aware that this is going to be monstrously large !)

The `spring-boot-starter-parent` itself is the child POM of another parent project `spring-boot-dependencies`
You can find the latest versions for this here:
https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-dependencies

An example of one of the latest versions:
https://repo1.maven.org/maven2/org/springframework/boot/spring-boot-dependencies/2.6.1/spring-boot-dependencies-2.6.1.pom

Again, we can see this dependency has a packaging type of POM. So, it also exists primarily to provide configuration info to any child POMs that inherit from it. In nearly all cases, its child POM will be `spring-boot-starter-parent`, but you can also create any standalone Maven project that inherits directly from `spring-boot-dependencies`

Its <properties> section contains the version values for an extremely (!!) large number (nearly 200 at the last count) of dependencies. Each of these dependencies are listed individually in the <dependencyManagement> section with full GAV coordinates, with their version values retrieved from the <properties> section.

This POM helps to facilitate the very important and difficult issue of dependency management. We have already previously seen that a Maven project can include a large number of direct and transitive dependencies. There may be version conflicts between common transitive dependencies shared by different direct dependencies, in addition to conflict between specific versions of direct dependencies which do not share any common transitive dependencies. This can result in hard to debug errors in the actual application code of project.

The <dependencyManagement> exists to provide configuration info for a specific list of dependencies which are subsequently applied to the current project as well as to all child projects that inherit from the current project. In this example here, the primary configuration info we are interested in is versioning (although we can also specify other configuration info as well if necessary).

The Spring development team has taken the necessary effort to test and ensure that all the dependencies listed in the <dependencyManagement> section of `spring-boot-dependencies` are guaranteed to work together **AS LONG** as the specified versions are used.

Going through this list, you will notice that there are many dependencies are actually not part of the Spring framework: they are mainly from popular Java libraries / frameworks that are likely to be used in a Spring application.

In addition, the all the main Spring Boot Starter dependencies are listed here (do a search for the term `spring-boot-starter` in the POM file):
The list of all Spring Boot Starter dependencies that are useful for implementing various aspects in an Enterprise Spring application is found at:
https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.build-systems.starters

A POM that exists primarily to provide the versions for a large group of dependencies that ensure that all these dependencies can work properly together if they are ever used in a Maven project is technically called a Bill of Materials (BOM).

All major Java libraries and frameworks have their own BOM.
For Spring framework, the BOM is in `spring-boot-dependencies`

For the Hibernate framework (the most popular framework used for implementing JPA for persistence to backend), a sample BOM is:
https://repo1.maven.org/maven2/org/hibernate/ogm/hibernate-ogm-bom/5.4.1.Final/hibernate-ogm-bom-5.4.1.Final.pom

An important point to note is that defining a dependency in the `<dependencyManagement>` section does not add it to the dependency tree of the project and make it available to the build of the project. In order for the dependency to be available for the Maven project to use to place on its compile or test path during the build, we must explicitly declare the dependencies in a separate `<dependencies>` section. A simple example illustrates here:

```
	<dependencies>

<!--		We no longer include version here since it is -->
<!--		already provided in the dependency declaration in  -->
<!--		the dependency management section -->
		<dependency>
			<groupId>org.apache.logging.log4j</groupId>
			<artifactId>log4j-core</artifactId>
		</dependency>

	</dependencies>

	<dependencyManagement>
		<dependencies>
			<dependency>
				<groupId>org.apache.logging.log4j</groupId>
				<artifactId>log4j-core</artifactId>
				<version>2.14.1</version>
			</dependency>

			<dependency>
				<groupId>org.apache.logging.log4j</groupId>
				<artifactId>log4j-slf4j-impl</artifactId>
				<version>2.14.1</version>
			</dependency>
		</dependencies>
	</dependencyManagement>
```

For the case of a standard Spring Boot project, the actual dependencies that will be used are listed in the various Spring Boot starters that we discussed earlier (`spring-boot-starter` and `spring-boot-starter-test`)

Notice that we do not specify the version number for both these dependencies in the actual project POM itself since this was already specified in the <dependencyManagement> section in `spring-boot-dependencies`

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-test</artifactId>
	<scope>test</scope>
</dependency>
```

Always bear in mind that the whole purpose of having a BOM such as `spring-boot-dependencies` is to explicitly specify the versions of all the dependencies so that we can be guaranteed that we can use any classes from any combination of these dependencies together in our application and they will work fine **AS LONG AS** the versions are adhered to.
Therefore, its never a good idea to explicitly specify a version for a dependency that you are using in your project POM unless you are very clear about what you are doing.

As an example, consider that we wish to use Kafka (a Java-based distributed event streaming platform) as part of our Spring application. There are a number of Kafka-related dependencies that we can use in our project, depending on which particular aspect of Kafka functionality we wish to access:

https://mvnrepository.com/artifact/org.apache.kafka

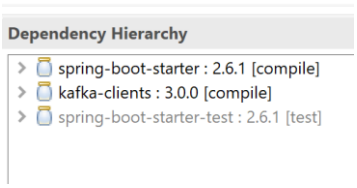
Let assume we want to create Kafka client. In that case, we will need to include one of the dependency versions listed at:
https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients

Since the version for this dependency is already specified in <dependencyManagement> in `spring-boot-dependencies`, we can directly include this dependency in our actual project POM without specifying the version and it will inherit the version specified in `spring-boot-dependencies`

Place this dependency snippet below in the <dependencies> section of the actual project POM, save and do a Maven -> Update Project:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
</dependency>
```

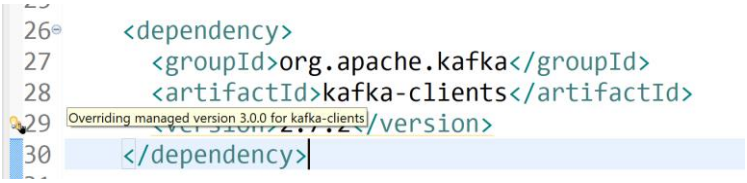
You will notice that the version number will automatically be added when you view it in the dependency hierarchy.



Let's assume you forgot that the version for this dependency is already configured in <dependencyManagement> in `spring-boot-dependencies`, and we include our own version instead, for e.g.

```
<version>2.7.2</version>
```

If you save the POM now, Eclipse will flag a warning to alert you of this issue:



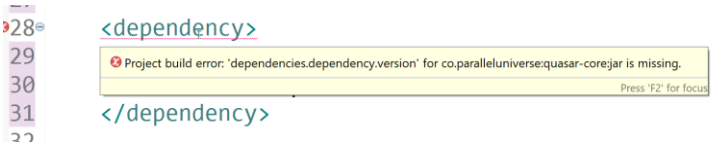
You should always exercise extreme caution if you are explicitly overriding a managed version because, as explained earlier, this can potentially result in subtle errors in your application code that are hard to detect or debug.

On the other hand, lets assume you wish to add a dependency that is not listed in <dependencyManagement> in `spring-boot-dependencies` (remember there hundreds to thousands of Java projects available as Maven dependencies on various Maven repositories listed an mvnrepository.com). Consider:
https://mvnrepository.com/artifact/co.paralleluniverse/quasar-core

If we try to include this dependency in our actual POM without specifying the version because we accidentally assumed that this dependency is already in `spring-boot-dependencies`, for e.g.

```
<dependency>
   <groupId>co.paralleluniverse</groupId>
   <artifactId>quasar-core</artifactId>
 </dependency>
```

Eclipse will immediately flag an error because version is a compulsory part of the three GAV coordinates required to uniquely identify each Maven dependency.



To summarize, when building a Spring Boot application using Initializr or STS:

a) The autogenerated project POM automatically has the parent `spring-boot-starter-parent` which in turn has the parent `spring-boot-dependencies`. Both of these are POM projects (packaging type: POM), which means they don't have any application / test code within them, just configuration info that is meant to be inherited by child POMs.

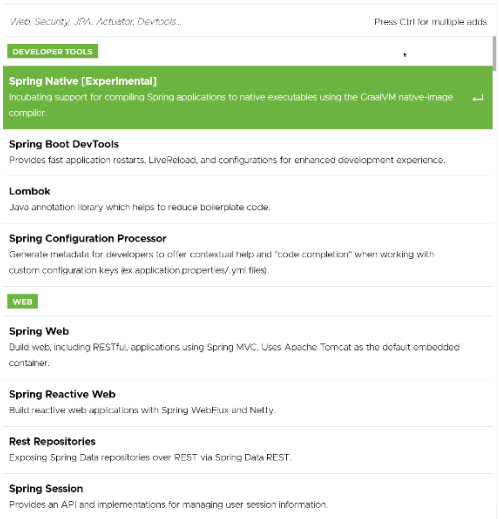
b) The `spring-boot-starter-parent` specifies the configuration info in its `<pluginManagement>` section for all the various plugins that will be used in the actual Maven project. This includes standard plugins such as `maven-compiler-plugin`, `maven-jar-plugin`, etc as well as Spring Boot specific plugins such as `spring-boot-maven-plugin`

c) The `spring-boot-dependencies` is a BOM with a `<dependencyManagement>` section that specifies the versions for all the Spring Boot Starter templates as well as any many other non-Spring dependencies that might be typically used in a Spring application.

d) When a Spring Boot application is initially created, the Spring Boot version is specified. This in turn sets the version for `spring-boot-starter-parent`, which in turn sets the version for `spring-boot-dependencies`, which in turn sets the version for every possible Spring Boot Starter template in addition to all the other non-Spring dependencies.

e) In the actual project POM, we will generally specify one or more Spring Boot Starter templates (with GA coordinate but no V), depending on the specific functionality that you will want to include in your project. The bare minimum will be `spring-boot-starter`.

f) You can also include other common non-Spring dependencies, and if they are also already listed in the `<dependencyManagement>` section of `spring-boot-dependencies`, you can leave out the version. You can get a list of these dependencies grouped into logical categories either at Initializr or STS, for e.g.



If you include any other non-Spring dependency that is not listed in the `<dependencyManagement>` section of `spring-boot-dependencies`, you MUST specify the full GAV coordinate for the dependency.

In the <plugins> section of the actual POM, you must include at least `spring-boot-maven-plugin` because this is specifically required in the build process for a Spring Boot project. You can also include the standard Maven plugins (for e.g. `maven-clean-plugin, maven-resources-plugin`, etc) if you wish to provide different configuration to override the configuration info provided in `<pluginManagement>` section of `spring-boot-starter-parent`. Generally speaking, this is not necessary except for specific use cases, so the <plugins> section is something that you can leave alone for most of your Spring Boot projects.

4 Spring Boot Application basics

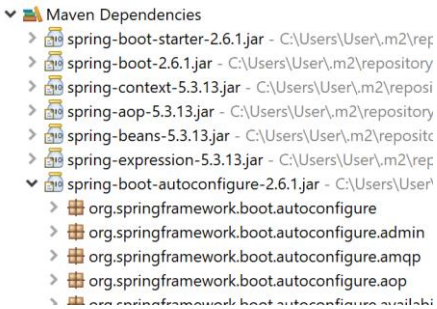
A Spring Boot project typically includes an autogenerated class annotated with @SpringBootApplication. This @SpringBootApplication annotation is actually a shorthand for the combination of 3 separate annotations:

- @Configuration
- @ComponentScan
- @EnableAutoConfiguration

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

The @EnableAutoConfiguration essentially auto configures Spring Boot by adding beans based on classpath settings, the existence of other specific beans, and various property settings. For example, if `spring-webmvc` dependency JARs is on the classpath (due to the inclusion of the `spring-boot-starter-web` starter dependency when we are creating a Spring Boot web application for e.g), this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet automatically in the background without the need to configure it (which you would need to do for a standard Maven non-Spring Boot project)

The autoconfiguration is performed by a variety of AutoConfiguration classes located in the spring-boot-autoconfigure dependencies, one of the main direct dependencies for `spring-boot-starter`. You can view all these classes in Maven dependencies entry in Eclipse



An important file inside spring-boot-autoconfigure.jar is `/META-INF/spring.factories`. This file lists all the auto configuration classes that should be enabled using @EnableAutoConfiguration.

We study a simple example here:
Consider a package holding all the relevant classes for autoconfiguration of a JDBC connection to a JDBC-compliant database.

https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/jdbc/package-summary.html

Consider one of the AutoConfiguration classes: DataSourceAutoConfiguration
https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/jdbc/DataSourceAutoConfiguration.html

The @ConditionalOnClass annotation means that the AutoConfiguration of beans within DataSourceAutoConfiguration (specified in the @Import annotation) will be considered only if the DataSource.class and EmbeddedDatabaseType.class classes are available on classpath

The class is also annotated with @EnableConfigurationProperties(DataSourceProperties.class) which enables binding the properties in application.properties to the properties of DataSourceProperties class automatically.

With this configuration, all the properties that starts with `spring.datasource.*` will be automatically bound to DataSourceProperties object.

```
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
spring.datasource.password=secret
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

The @ConditionalOnMissingBean annotation in DataSourceAutoConfiguration means that the AutoConfiguration of beans within DataSourceAutoConfiguration will be considered only if the none of the io.r2dbc.spi.ConnectionFactory based classes are present on the classpath. This is because these family of classes are used to set up reactive database drivers instead of conventional JDBC drivers. We cannot have both JDBC and reactive database drivers active at the same time.
https://spring.io/guides/gs/accessing-data-r2dbc/

You will notice as well that in the AutoConfiguration for r2dbc, its @Conditional also specifies the same class type (io.r2dbc.spi.ConnectionFactory) but it does not have a @ConditionalOnMissingBean It also specifies that its autoconfiguration will happen before DataSourceAutoConfiguration
https://docs.spring.io/spring-boot/docs/2.3.12.RELEASE/api/org/springframework/boot/autoconfigure/r2dbc/R2dbcAutoConfiguration.html

In other words, if both javax.sql.DataSource (for conventional JDBC connections) and io.r2dbc.spi.ConnectionFactory (for reactive database drivers) are on the classpath at the same time, then priority is given to the reactive database drivers

The use of the various autoconfiguration related annotations such as @ConditionalOnClass, @ConditionalOnMissingBean, @AutoConfigureBefore, @EnableConfigurationProperties, etc allows the implementation of complex setup logic for the various libraries that a Spring application will need with minimal intervention from the developer.

The primary entry point into a Spring Boot app is the SpringApplication class which is typically used to bootstrap and launch the app. By default, it will perform the following steps:

- Create an appropriate ApplicationContext container instance

- Register a CommandLinePropertySource to expose command line arguments as Spring properties
- Refresh the container instance in order to loading all singleton beans that are picked up via component scanning
- Trigger any CommandLineRunner beans for startup logic

In most circumstances, we will use the static `run(Class, String[])` method can be called directly from your main method to bootstrap your application. The first argument is the name of the @SpringBootApplication class itself, while the second is the command line arguments to pass to the app.

This returns a container of type ConfigurableApplicationContext from which we can then retrieve beans using getBean in the standard manner that we have seen before.

5 Defining and retrieving beans

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 FirstSpringBootApplication by using Run As -> Java Application.

We can 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`:

```
HighSchoolStudent.java
CollegeSchoolStudent.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.

We use @Autowired and @Qualifier on the 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 @Property Source. 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 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`.

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`:

```
FirstSpringBootApplication-v4.java
BeanConfig-v2.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 is 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: XML, annotation-based or Java based and retrieved using the getBean method of the ApplicationContext container or autowired by the container to any @Autowired dependency in any registered bean class.

The concept of bean scopes and eager vs lazy loading that we covered in a previous lab are equally applicable here as well.

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 -> 4 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`

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

Now type: `mvnw spring-boot:run`

6 Working with properties

In addition to user defined properties, you can also set Spring Boot specific properties in `application.properties` to control and configure various aspects of Spring Boot application behaviour.
The full list of properties that can be set in this way are listed at:
https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

You will see most of the core properties are related to control logging via the default Logback implementation that is provided in Spring Boot.

For a simple command line based Spring Boot application, there are not many relevant Spring Boot properties to configure at the moment. Add this property to `application.properties` and save

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

Run the app and notice that the main Spring Boot banner is no longer appearing.

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

An alternative popular format for providing properties is YAML
https://www.educative.io/blog/yaml-tutorial

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

```
application.properties
```

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

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

Run the app. This time, the required properties are injected correctly from `application.yml`

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 replace it again with `application.properties`

We can access system properties and OS environment variables in `application.properties` (and also in the properties annotated with @Value in any valid bean classes) and provide default values to be used if appropriate.

There are a variety of ways to view the complete list of environment variables in Windows 10
https://winaero.com/how-to-see-names-and-values-of-environment-variables-in-windows-10/

To view all the default system properties of the current JVM, type this in a terminal prompt:

```
java -XshowSettings:properties
```

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

```
FirstSpringBootApplication-v6.java
MathStudent-v2.java
```

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

```
application-v2.properties
```

Notice now that the MathStudent class is now marked with @Component, which mean it will be picked up by the @ComponentScan that is implicit in FirstSpringBootApplication, and will be created and registered with its default name of mathStudent. This is in addition to the bean with the name mathPerson defined from the same bean class in beansDefinition.xml which we imported earlier using @ImportResource in BeanConfig.

Run the app. Verify that the system properties and environment variables are retrieved correctly from the application, either directly or through application.properties

Now comment out this property from application.properties and save.

```
#Comment out java.vendor system property
#my.favorite.vendor=${java.vendor}
```

Run the app again. Notice now that the default value for theVendor property in MathStudent is used as the original property does not have a value set for it.

7 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.

Make sure this property is still commented out from application.properties.

```
#Comment out java.vendor system property
#my.favorite.vendor=${java.vendor}
```

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

```
CommandLineAppStartupRunner
```

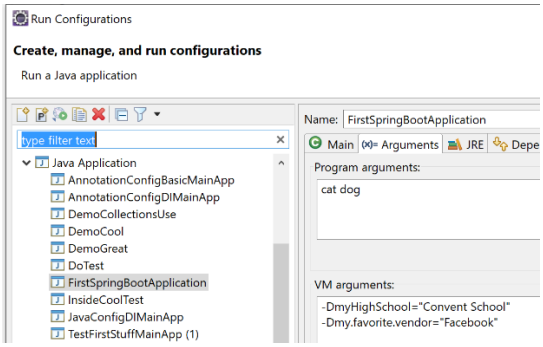
Make changes to the following files in `com.workshop.demo` in `src/main/java` from `changes`:

`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`.
The `my.favorite.vendor` property (which is not defined in `application.properties`) is accessible to the `theVendor` property in MathStudent, so the default value of Microsoft Corp is no longer used.

8 Generating an executable JAR file

To generate an executable, standalone JAR (which will be a Uber JAR) for the project, simply do a standard Eclipse Maven build with the goals:

`clean package`

The generation of the Uber JAR is actually performed by the `spring-boot:repackage` goal of the spring-boot-maven-plugin for Spring specific Maven project

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, which is conceptually similar to the older form of: JAR of JARS (or one JAR)

Navigate to this directory in a command 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:

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

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 as 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" --my.favorite.vendor="Facebook" --
spring.main.banner-mode=off
```