# Introduction to Spring

# Maven Multi Projects

# Definition

- A multi-module project is built from an aggregator POM that manages a group of submodules. In most cases, the aggregator is located in the project's root directory and must have packaging of type pom.
- Now, the submodules are regular Maven projects, and they can be built separately or through the aggregator POM.
- By building the project through the aggregator POM, each project that has packaging type different than pom will result in a built archive file.

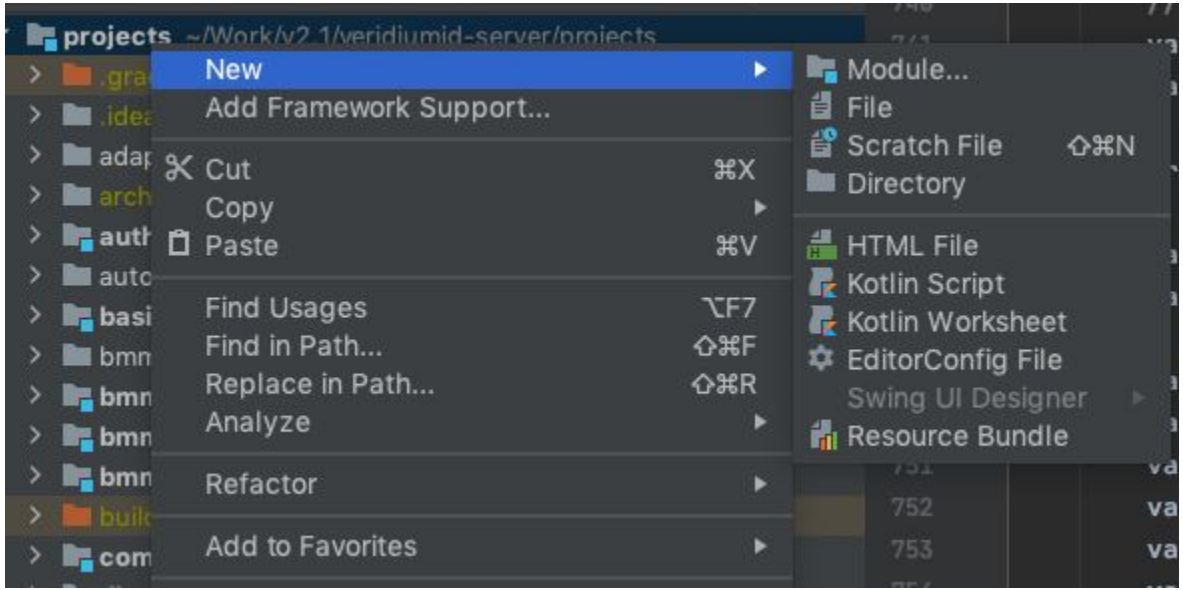
# Parent

- Maven supports inheritance in a way that each pom.xml file has the implicit parent POM, it's called Super POM and can be located in the Maven binaries. These two files are merged by Maven and form the Effective POM.
- Hence, we can create our own pom.xml file which will serve us as the parent project. Then, we can include there all configuration with dependencies and set this as the parent of our child modules, so they'll inherit from it.

# Module

- Submodules or subprojects are regular Maven projects that inherit from parent POM.
- As we already know, inheritance let us share the configuration and dependencies with submodules.
- However, if we'd like to build or release our project in one shot, we have to declare our submodules explicitly in parent POM.
- Ultimately, our parent POM will be the parent as well as the aggregate POM.

# Create a module



# Create modules

- Create 2 module in the project
- In the first module move the content from the main current src
- In the second create an empty hello world Spring App

# Spring Scanning

# How Spring Scanning works

- All the classes and sub-packages will be scanned automatically under Spring Boot main class package.
    - EnableAutoConfiguration

- During the scan, it will detect @Component, @Configurations, @Bean annotated classes, and methods.

# Scan External projects

- By using @ComponentScan annotation, we can autowire the classes from another project or another jar.

# Circular Dependency

- It happens when a bean A depends on another bean B, and the bean B depends on the bean A as well:
  - Bean A → Bean B → Bean A

- Of course, we could have more beans implied:
  - Bean A → Bean B → Bean C → Bean D → Bean E → Bean A

# Circular Dependency Workaround @Lazy

- A simple way to break the cycle is saying Spring to initialize one of the beans lazily.
- That is: instead of fully initializing the bean, it will create a proxy to inject it into the other bean.
- The injected bean will only be fully created when it’s first needed.

# @Lazy example

```
@Component
public class CircularDependencyA {

    private CircularDependencyB circB;

    @Autowired
    public CircularDependencyA(@Lazy CircularDependencyB circB) {
        this.circB = circB;
    }
}
```

# Fat Jar

# Fat Jar

- An uber-JAR—also known as a fat JAR or JAR with dependencies—is a JAR file that contains not only a Java program, but embeds its dependencies as well.
- This means that the JAR functions as an “all-in-one” distribution of the software, without needing any other Java code.
- You still need a Java runtime, and an underlying operating system, of course

# Add build plugin

```
</plugin>
<plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.2-beta-5</version>
    <configuration>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <mainClass>com.ctiliescu.shopping.ShoppingApplication</mainClass>
            </manifest>
        </archive>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
    </configuration>
    <executions>
        <execution>
            <id>assemble-all</id>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

# Run a fat jar

- Java -jar jar_name.jar

- Pro:
  - A single JAR file is simpler to deploy. There is no chance of mismatched versions of multiple JAR files. It is easier to construct a Java classpath, since only a single JAR needs to be included.
- Cons:
  - Every time you need to update the version of the software, you must redeploy the entire uber-JAR

# Application properties on runtime

# Spring Boot Application Properties

- Spring Boot Framework comes with a built-in mechanism for application configuration using a file called application.properties.
- It is located inside the src/main/resources folder, as shown in the following figure.
- Spring Boot provides various properties that can be configured in the application.properties file.
- The properties have default values. We can set a property(s) for the Spring Boot application. Spring Boot also allows us to define our own property if required.

# @Value

- Annotation used to load a application properties defined in configuration file

```
@Value("${value.from.file}")
private String stringValue;
```

# @Profile

- We use the @Profile annotation — we are mapping the bean to that particular profile; the annotation simply takes the names of one (or multiple) profiles.

- -Dspring.profiles.active=dev

- However, the most important profiles-related feature that Spring Boot brings is profile-specific properties files. These have to be named in the format application-{profile}.properties

# VM options

- Talking about JVM options, then there are three types of options that you can include to your JVM, standard, non-standard, and advanced options.
- If you try an advanced option, you always use the option with -XX .
- Similarly, if you’re applying a non-standard option, you use -X .
- Standard options don’t prepend anything to the option.

# Environment variable

- An environment variable is a dynamic-named value that can affect the way running processes will behave on a computer.
- They are part of the environment in which a process runs.
- For example, a running process can query the value of the TEMP environment variable to discover a suitable location to store temporary files, or the HOME or USERPROFILE variable to find the directory structure owned by the user running the process.

# @Import - 1

- We also learned that @Import is very similar to @ComponentScan, except for the fact that
  - @Import has an explicit approach
  - @ComponentScan uses an implicit one.

- Typically, we start our applications using @ComponentScan in a root package so it can find all components for us. If we're using Spring Boot, then @SpringBootApplication already includes @ComponentScan, and we're good to go. This shows the power of convention.

# @Import - 2

- Making an analogy with our annotations, @ComponentScan is more like convention, while @Import looks like configuration.

- Adding everything into the context risks starting conflicts about which bean to use. Besides that, we may get a slow start-up time

# @Import - 3

```java
@Configuration
public class FishConfig {

  @Bean
  public GoldFish goldFish() {
    return new GoldFish();
  }

  @Bean
  public Guppy guppy() {
    return new Guppy();
  }

  @Bean
  public Salmon salmon() {
    return new Salmon();
  }

}
```

```java
@Configuration
public class BirdConfig {

  @Bean
  public Eagle eagle() {
    return new Eagle();
  }

  @Bean
  public Ostrich ostrich() {
    return new Ostrich();
  }

  @Bean
  public Peacock peacock() {
    return new Peacock();
  }
}
```

# @Import - 4

```
@Configuration
@Import({FishConfig.class, BirdConfig.class})
public class ImportBeansConfig {

  @Bean
  public ExampleBean exampleBean() {
    return new ExampleBean();
  }

  @Bean
  public SampleBean sampleBean() {
    return new SampleBean();
  }
}
```

# @Import - 5

```
public class ExampleBean {

  @Autowired
  private Salmon salmon;

  @Autowired
  private Guppy guppy;

  @Autowired
  private Peacock peacock;

  public void printObjects() {
    System.out.println("---------- Print ExampleBean Objects ----------");
    System.out.println(salmon);
    System.out.println(guppy);
    System.out.println(peacock);
  }
}
```

# / Q&A


OBILE/ACADEMY