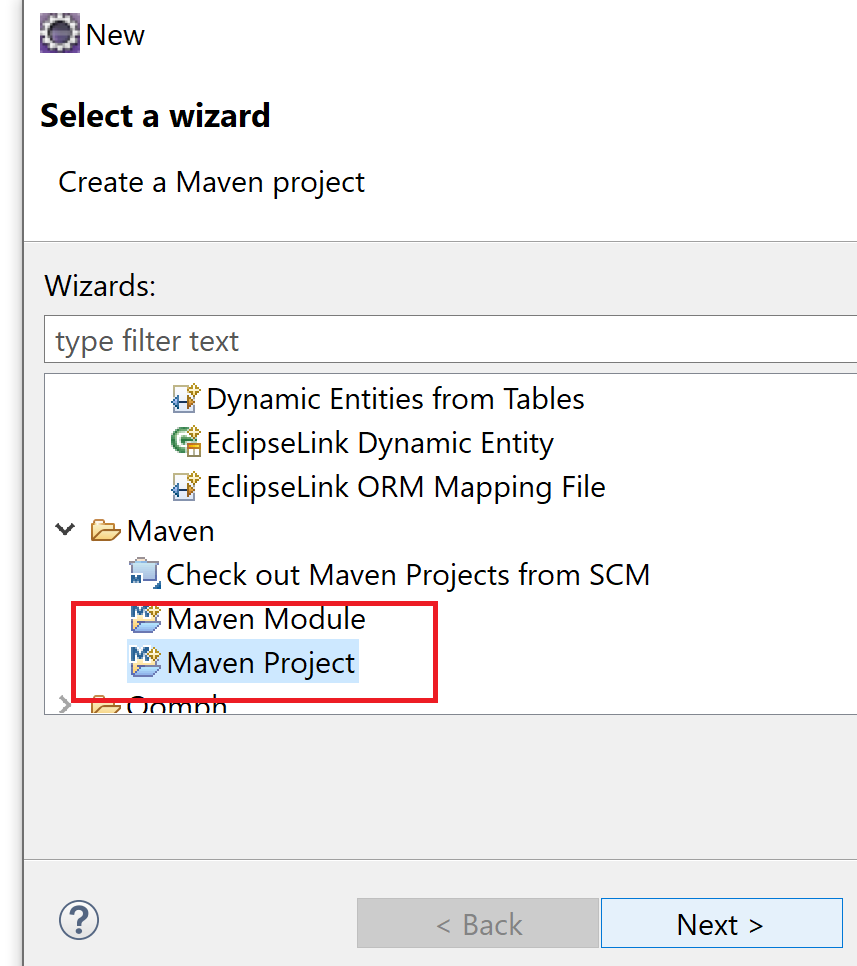
**Spring Boot:**

At the heart of Spring Boot lies the Spring Framework; Spring Boot extends this to make auto-configuration. Spring Boot makes it easy to create stand-alone, production-grade, Spring-based Applications that you can “just run.” We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration. Spring Boot has auto-configuration for infrastructure like JMS, JDBC, JPA, RabbitMQ, and lots more. Spring Boot also offers auto-configuration for different frameworks like Spring Integration, Spring Batch, Spring Security, and many others. When these frameworks or capabilities are detected, Spring Boot will configure them with opinionated but sensible defaults.

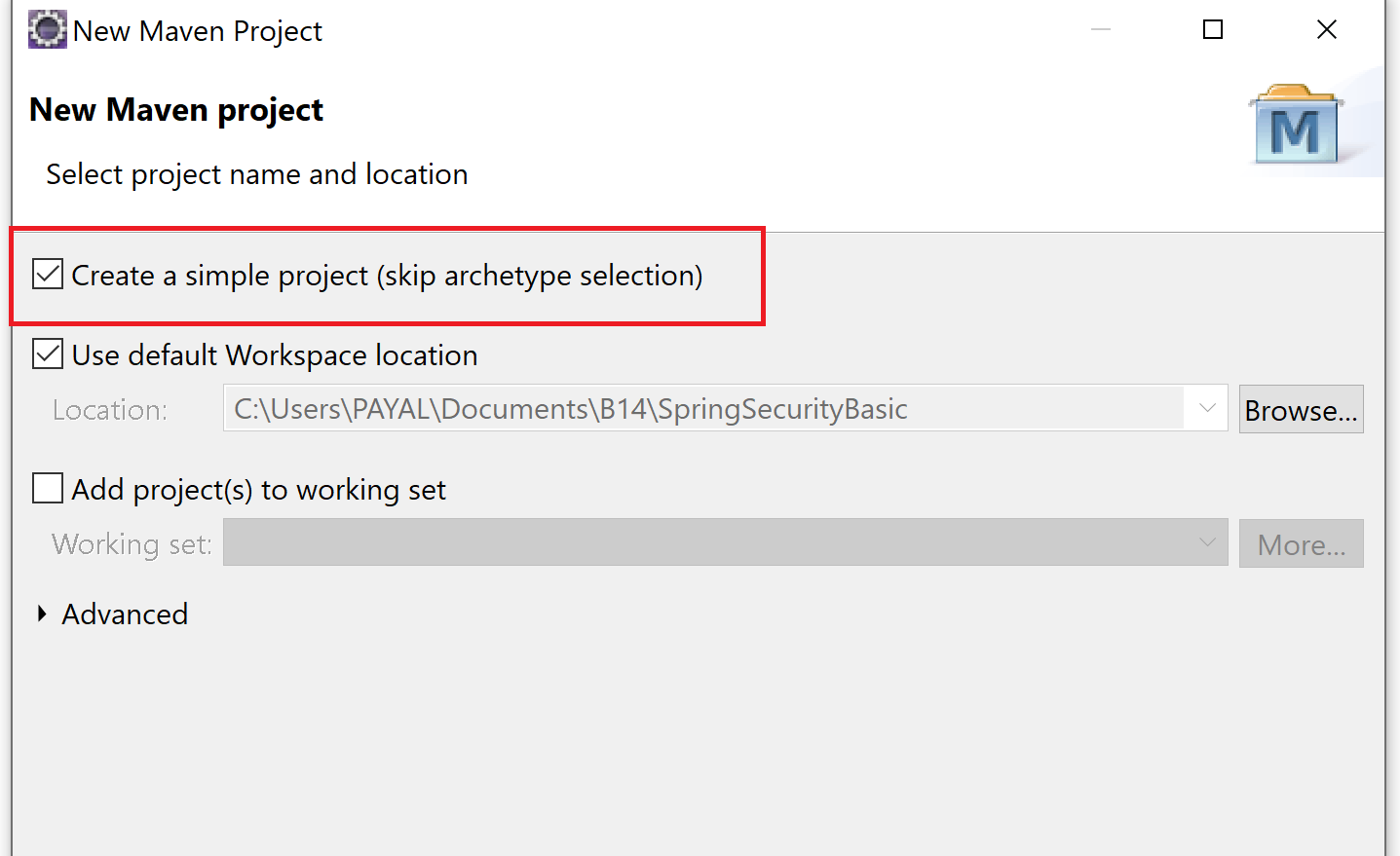
1. Some boilerplate code can be reduced.
2. Dependencies better. Spring boot has given you a parent pom..
3. Minimal or no configuration: spring-webmvc: Automatically create dispatcherservlet :/
4. Provides a plugin to create a executable jar file
5. Spring boot provides u embedded servers: tomcat, jetty packaged as part of ur application. Even for a web application, it gives you a jar file.

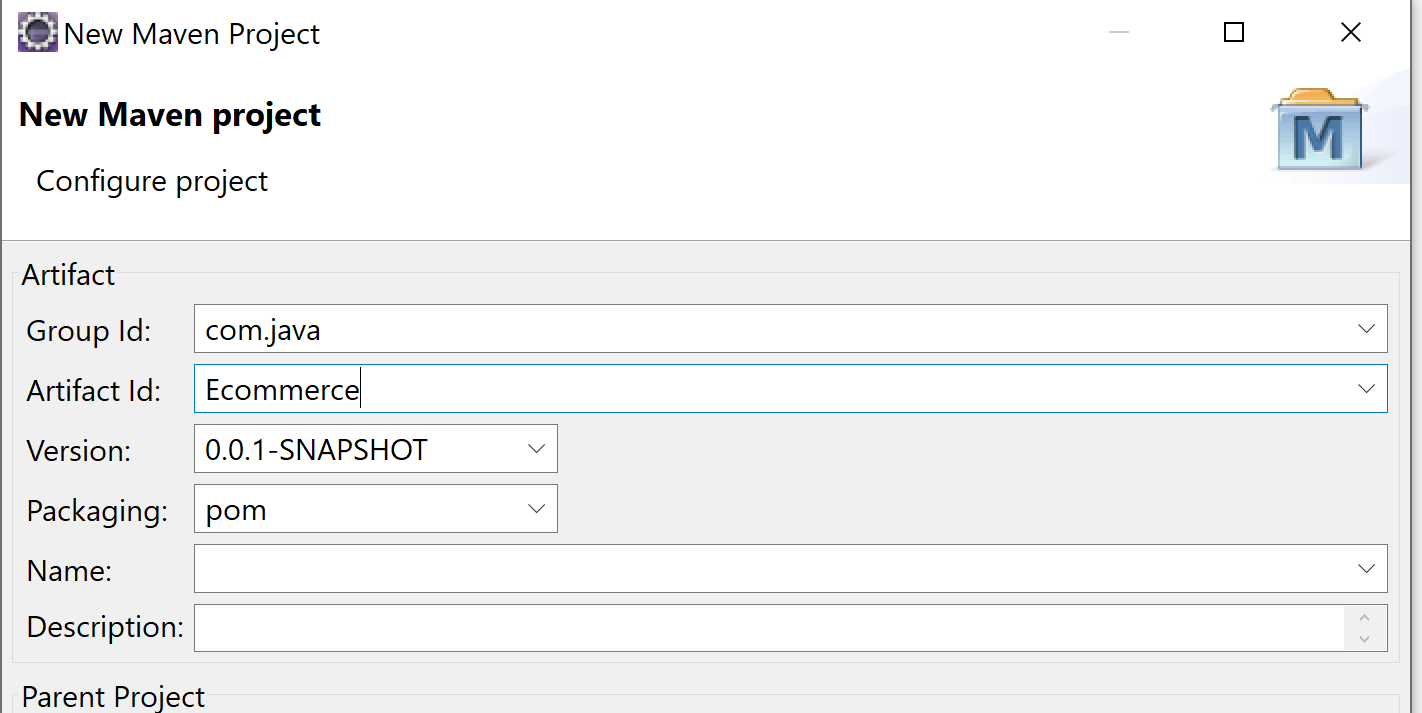
Parent pom: spring-boot-starter-parent

Maven: inheritance



Within a project you can have many modules





1. Setting java version to be 1.8
   1. Properties: maven.compiler.source & maven..compiler.target: 1.8
   2. Plugin -> maven-compiler-plugin: <source>1.8 target:1.8

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

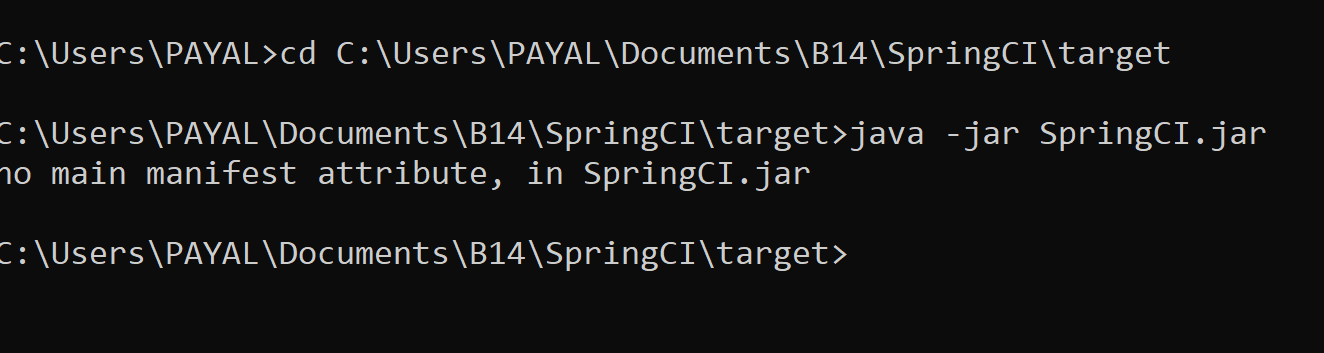
<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

1. Jar file -> Is it an executable jar file?
   1. Maven package: It creates a jar file but it does not add ur libraries by default.
      1. To use java -jar, we should specify which is ur main class.



**Fat jar: having all dependencies also**

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>3.2.3</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<transformers>

<transformer implementation=*"org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"*>

<manifestEntries>

<Main-Class>com.java.Main</Main-Class>

<Build-Number>1</Build-Number>

</manifestEntries>

</transformer>

</transformers>

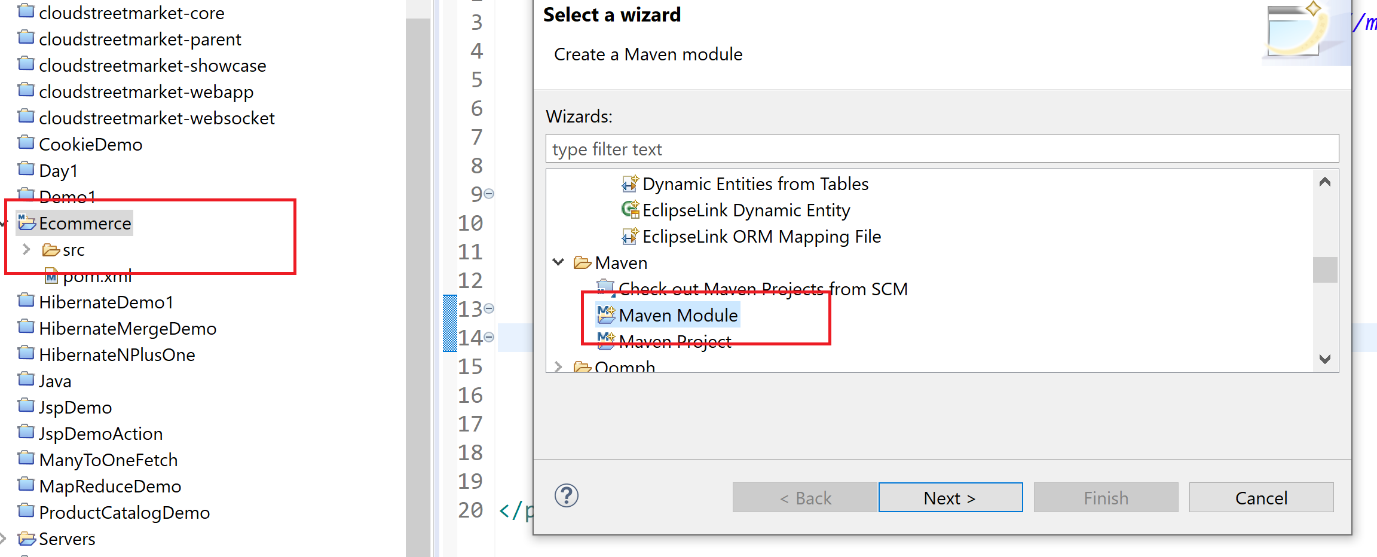
</configuration>

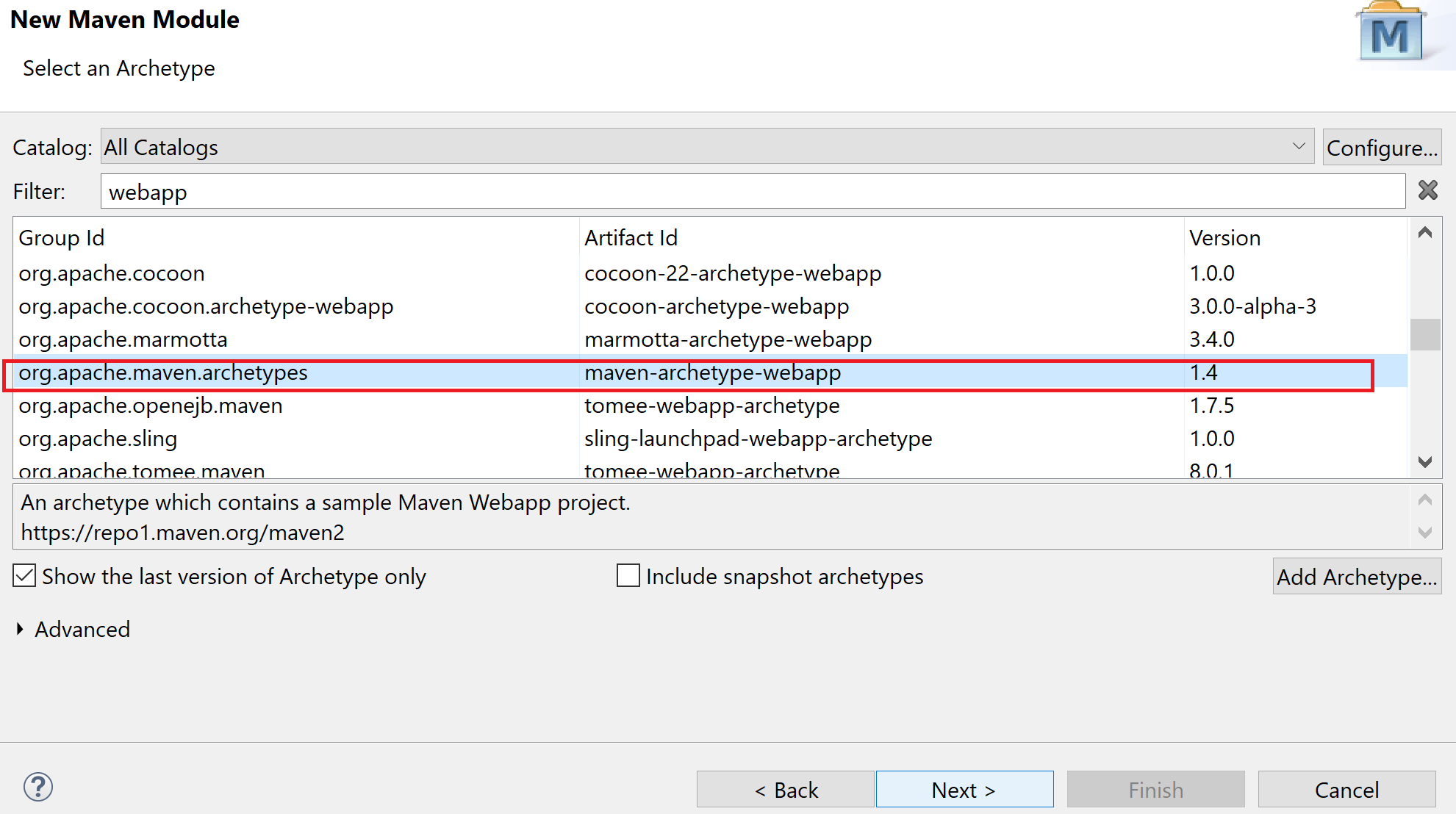
</execution>

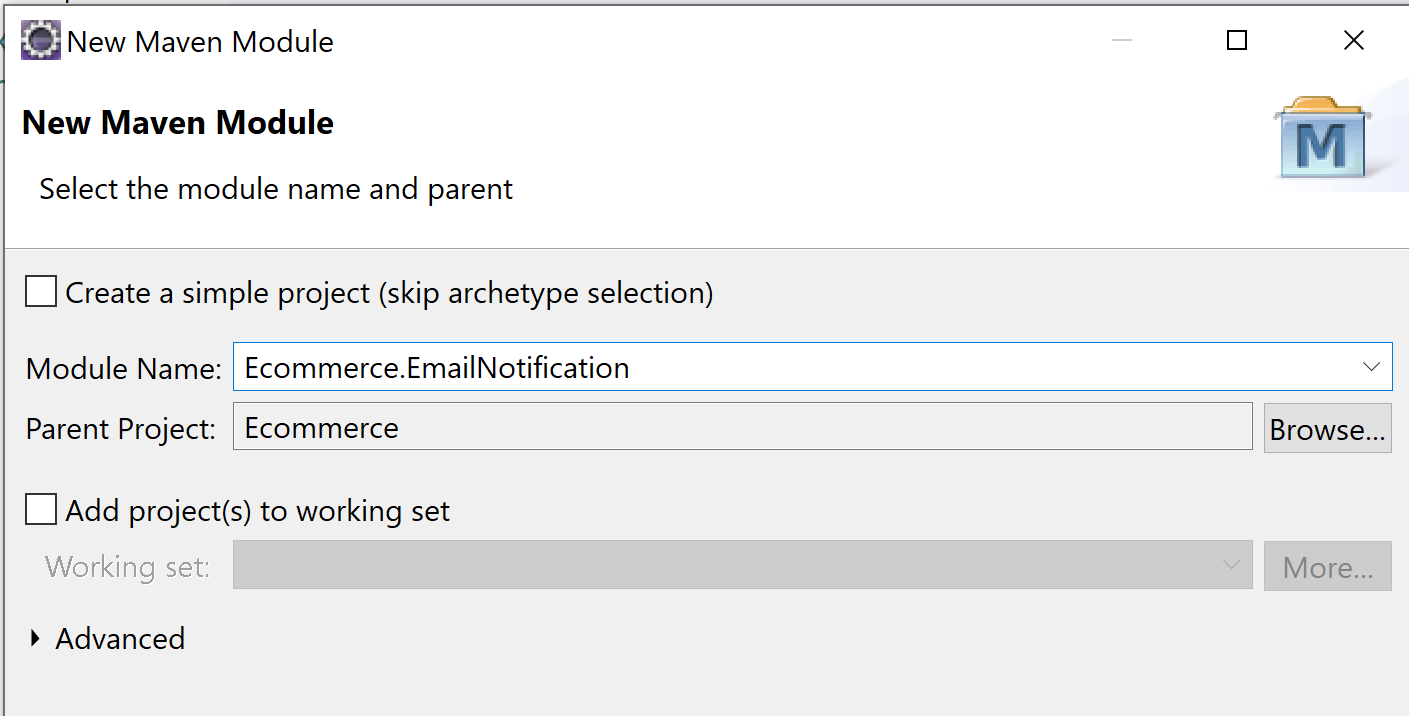
</executions>

</plugin>

Creates Uber jar/ fat jar : which has libraries also packaged

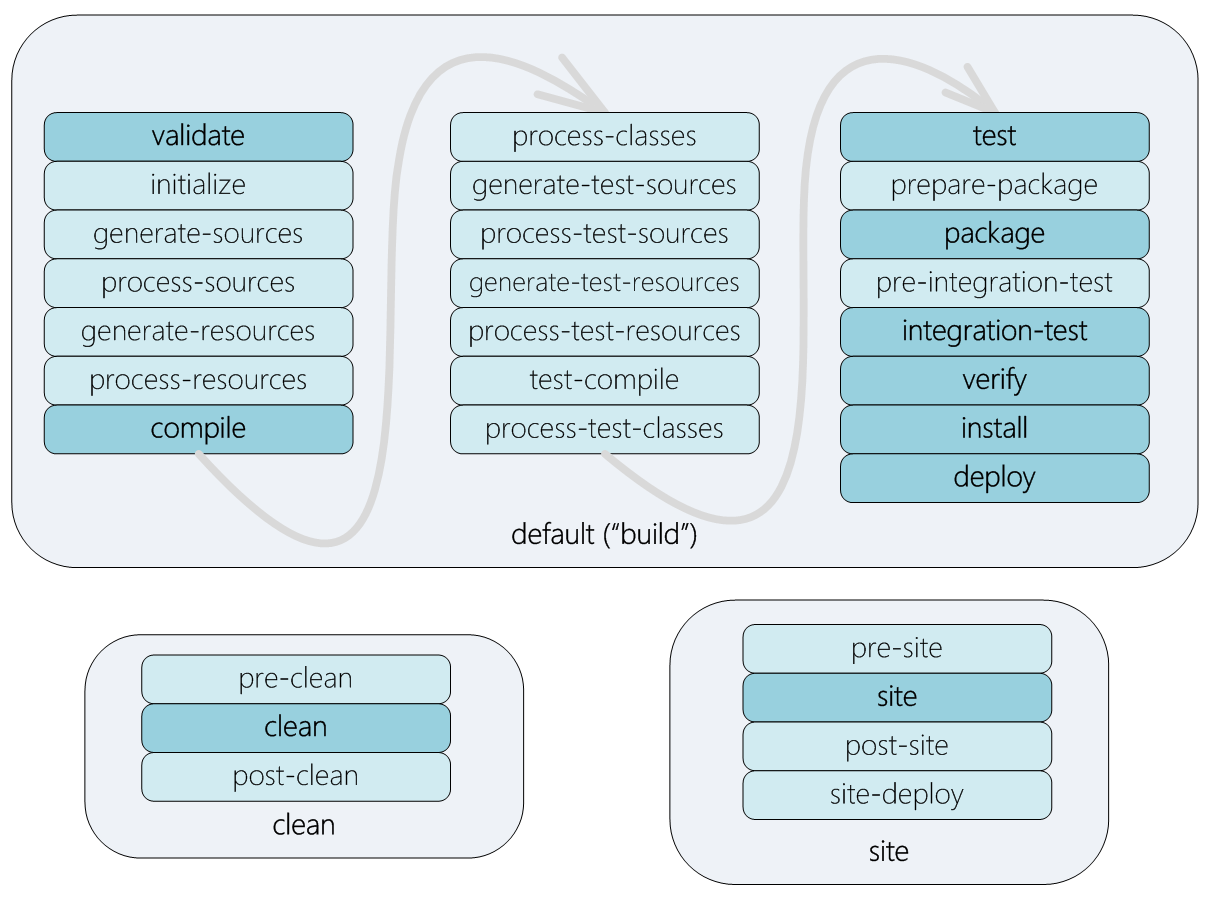


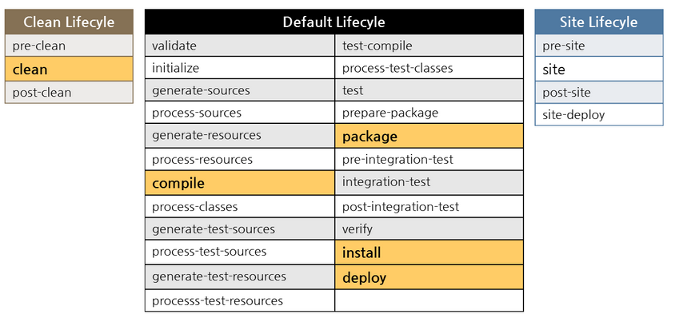




Spring boot provides you : spring-boot-maven-plugin: creates an executable jar file

1 phase may have multiple goals





War file: jar file+ webcontent: You need a server to run your war file.

What if you embed your server into your application??

Maven provides you even server plugins:

Tomcat7-maven-plugin

**Spring boot project**

1. Maven -> dependencies
2. Spring Tool suite plugin
3. <https://start.spring.io/>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.2.6.RELEASE</version>

</parent>

Manage ur versions for all libraries

For desktop application, we add  spring-boot-starter . The spring-boot-starter will pull in all the core dependencies needed to start a very basic Spring Boot application, things like the Spring Framework, Logback for logging, and Spring Boot itself.

 The main method calls SpringApplication.run with the Main.class and arguments from the main method. The run method returns an ApplicationContext, which is used to retrieve the bean names from ApplicationContext.

Q) To print name of beans created by spring:

var ctx = SpringApplication.run(DemoApplication.class, args);

        System.out.println("# Beans: " + ctx.getBeanDefinitionCount());

        var names = ctx.getBeanDefinitionNames();

        Arrays.sort(names);

        Arrays.asList(names).forEach(System.out::println);

@SpringBootApplication is an annotation, annotated by these 3 annotations:

@Configuration

@EnableAutoConfiguration

@ComponentScan

Hence called composed annotation

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan

public @interface SpringBootApplication { ... }

Here the @SpringBootConfiguration annotation is used instead of the @Configuration annotation. The @SpringBootConfiguration is a specialized @Configuration annotation. It indicates that this is a Spring Boot-based application. When using @SpringBootConfiguration in your application, there can only be one class annotated with this annotation!

**Spring-boot-maven-plugin:**

**<build>**

**<plugins>**

**<plugin>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-maven-plugin**</artifactId>**

**</plugin>**

**</plugins>**

**</build>**

This plug-in takes care of creating a fat JAR. It takes the original JAR and repackages it with all the dependencies inside it. The operations team needs to do java -jar <your-application>.jar to launch the application.

Q1) Create a simple java desktop application using spring with

class A{

B b;

void print(){

Sysout(b.x)}

}

class B{int x;

};

objects created by spring and call print method

Q2) Modify this to create a factory method in class B which returns object of B. Use that method to get object of B in class A. Use spring injection

**ApplicationRunner / CommandLineRunner:**

Spring Boot has an interface ApplicationRunner , which can be used to run some code after startup of the application. When Spring Boot detects a bean of type ApplicationRunner, it will invoke its run method as soon as the application has started.

**Spring Batch** relies on the runners to trigger the execution of the jobs.

*It is important to note that if any exception is thrown in the run(String... args) method, this will cause the context to close and an application to shut down. Wrapping the risky code blocks with try/catch is recommended to prevent this from happening.*

**Externalize Properties**

By default, Spring Boot supports getting properties from numerous locations. By default, it will load a file named application.properties, and use the environment variables and Java System properties. When running from the command line, it will also take command line arguments into consideration.

For our application the following resources are taken into considering in given order.

1. Command line arguments
2. application.properties outside of the packaged application:

 in the same location as the artifact add an application.properties and put values in there for the different properties.

1. application.properties packaged inside the application

Also we can specify diff property file using @PropertySource

@PropertySource("classpath:your-external.properties")

@SpringBootApplication

**public class** MyApplication { **...** }

. The profiles to activate can be passed through the spring.profiles.active property. The profile-specific application-{profile}.properties takes precedence over the non–profile-specific one. Each will get loaded and with that you can override properties, which makes the list a bit longer.

1. 1.Command line arguments

1. 2.application-{profile}.properties outside the packaged application

1. 3.application.properties outside the packaged application

1. 4.application-{profile}.properties packaged inside the application

1. 5.application.properties packaged inside the application

  @Bean

**public** ApplicationRunner calculationRunner(Calculator calculator,

                         @Value("${lhs}") int lhs,

                         @Value("${rhs}") int rhs,

                         @Value("${op}") char op) {

**return** args -> calculator.calculate(lhs, rhs, op);

  }

If there is no default value specified, an IllegalArgumentException will be thrown for missing properties.

 you can also instruct Spring Boot to load additional property files using the command line parameters from Table [2-1](https://learning.oreilly.com/library/view/spring-boot-2/9781484239636/html/464036_1_En_2_Chapter.xhtml#Tab1).

***Table 2-1***

Configuration Parameters

| **Parameter** | **Description** |
| --- | --- |
| spring.config.name | Comma separated string of file names to load,  default application |
| spring.config.location | Comma seperated string of resource locations (or files)  to consider for loading property files from, default classpath:/,classpath:/config/,  file:./,file:./config/ |
| spring.config.additional-location | Comma seperated string of additional resource  locations (or files) to consider for loading property  files from, default empty |

To load the your-external.properties using --spring.config.name=your-external would be sufficient; however, this would break loading the application.properties. It is better to use --spring.config.name=application,your-external; now all the location will be searched for both application.properties and your-external.properties and the profile-specific versions will be taken into consideration.

**Logging:**

Spring Boot ships with a default configuration for the supported log providers (Logback,[5](https://learning.oreilly.com/library/view/spring-boot-2/9781484239636/html/464036_1_En_2_Chapter.xhtml#Fn5) Log4j 2,[6](https://learning.oreilly.com/library/view/spring-boot-2/9781484239636/html/464036_1_En_2_Chapter.xhtml#Fn6) and Java Util Logging). Next to the default configuration, it also adds support for configuring the logging levels through the regular application.properties

Spring Boot uses SLF4J as the logging API, and when writing components you should use those interfaces to write your logging.

logging.level.org.springframework.web=DEBUG

logging.level.root=<level>

By default, Spring Boot will only log to the console. If you want to write to a file as well, you need to specify either logging.file or logging.path

The first takes the name of the file; the second, the path. The default filename used is spring.log and the default directory used is the Java temp directory.

logging.file=application.log

logging.path=/var/log

 You can specify how many files to retain with the logging.file.max-history (default is 0 meaning unlimited) and logging.file.max-size to specify the file size (default is 10MB).

#### Using Your Preferred Logging Provider

Spring Boot by default uses Logback as the provider for the logging. It does, however, support Java Util Logging as well as Log4j 2. To use another logging framework, you will have to first exclude the default framework and include your own. Spring Boot has a spring-boot-starter-log4j2 to include all necessary dependencies for Log4j 2. To exclude the default Logback logging, you need to add an exclusion rule to the spring-boot-starter dependency; this is the main dependency that brings in the logging.

**<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter**</artifactId>**

**<exclusions>**

**<exclusion>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-logging**</artifactId>**

**</exclusion>**

**</exclusions>**

**</dependency>**

**<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-log4j2**</artifactId>**

**</dependency>**

 Spring Boot starters are bootstrap libraries that contain a collection of all the relevant transitive dependencies that are needed to start a particular functionality. Each starter has a special file, which contains the list of all the provided dependencies Spring provides. Let's take a look at the following link for a spring-boot-starter-test definition as an example:

<https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-starters/spring-boot-starter-test/src/main/resources/META-INF/spring.provides>

**@EnableAutoConfiguration**: This annotation is a part of the Spring Boot annotation, which is a meta-annotation on its own (you will find that Spring libraries rely very heavily on the meta-annotations so they can group and compose configurations together). It imports the EnableAutoConfigurationImportSelector and AutoConfigurationPackages.Registrar classes that effectively instruct Spring to automatically configure the conditional beans depending on the classes available in the classpath

**Autoconfiguration** is a feature that allows library developers to automatically configure beans in the Spring context based on different conditions of the application, such as the presence of certain classes in the classpath, the existence of a bean or the activation of some property. It’s the Spring Boot approach to [Convention over configuration](https://en.wikipedia.org/wiki/Convention_over_configuration) paradigm, that is, to minimize the number of decisions a programmer has to make when using a framework, while providing sensible defaults but not losing flexibility

## How to define an Autoconfiguration class

The first thing to keep in mind is that an **AutoConfiguration** class looks like any regular Spring @Configuration class, but is enriched with @Conditional annotations that will activate or not a bean, or a set of them, only under certain circumstances. Let’s take a closer look at an extract of the KafkaAutoConfiguration class provided by Spring itself.

@Configuration  
@ConditionalOnClass(KafkaTemplate.class)  
public class KafkaAutoConfiguration {  
   
 @Bean  
 @ConditionalOnMissingBean(KafkaTemplate.class)  
 public KafkaTemplate<?, ?> kafkaTemplate (  
 ProducerFactory<Object, Object> kafkaProducerFactory,  
 ProducerListener<Object, Object> kafkaProducerListener) {  
 KafkaTemplate<Object, Object> kafkaTemplate =  
 new KafkaTemplate<>(kafkaProducerFactory);  
 if (this.messageConverter != null) {  
 kafkaTemplate.setMessageConverter(this.messageConverter);  
 }  
 kafkaTemplate.setProducerListener(kafkaProducerListener);  
 kafkaTemplate.setDefaultTopic(  
 this.properties.getTemplate().getDefaultTopic());  
 return kafkaTemplate;  
 }  
   
 @Bean  
 @ConditionalOnProperty(  
 name = "spring.kafka.producer.transaction-id-prefix")  
 @ConditionalOnMissingBean  
 public KafkaTransactionManager<?, ?> kafkaTransactionManager (  
 ProducerFactory<?, ?> producerFactory) {  
 return new KafkaTransactionManager<>(producerFactory);  
 } // More bean definitions here}

We can see this configuration class would only be activated if the KafkaTemplate class is present in the application classpath (@ConditionalOnClass annotation), so if the developer added the specific Apache Kafka dependency, Spring will autoconfigure itself with the appropriate beans.

Conditional annotations can also be used at the bean definition level, we can see a couple of them in the above example: @ConditionalOnMissingBean only registers the new component in absence of the required bean type or name and @ConditionalOnProperty can be leveraged to create a bean based on the presence of some property definition.

Spring provides plenty of conditional annotations out-of-the-box, all of them well documented within its [reference documentation](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-developing-auto-configuration.html) and with mostly self-explanatory names:

* @ConditionalOnClass
* @ConditionalOnMissingBean
* @ConditionalOnBean
* @ConditionalOnJava
* @ConditionalOnJndi
* @ConditionalOnMissingClass
* @ConditionalOnExpression
* @ConditionalOnNotWebApplication
* @ConditionalOnWebApplication
* @ConditionalOnProperty
* @ConditionalOnResource
* @ConditionalOnSingleCandidate

If these conditions are not enough for our use case, developers have the ability to create their own custom conditions, extending the SpringBootConditionclass and using it with the @Conditional annotation.

## Register your own Autoconfiguration

In order to use an **Autoconfiguration** class, Spring needs to know where to look for it. This step is done using the standard META-INF/spring.factories file, adding the full name of the configuration class under the entry org.springframework.boot.autoconfigure.EnableAutoConfiguration.

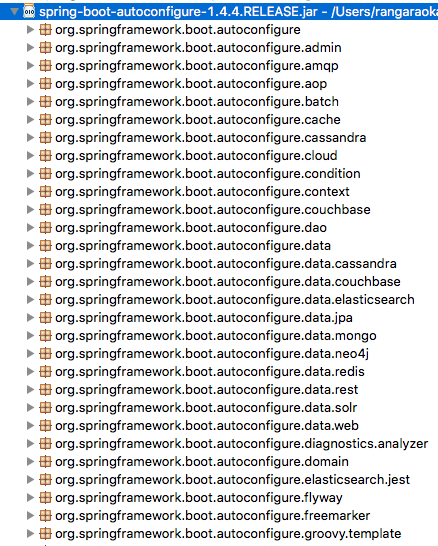
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
 com.empathybroker.samples.autoconfigure.SampleAutoConfiguration

And that’s all. Package your library, deliver it to your preferred artifact manager and developers can reassure themselves of the dependency and that everything is autoconfigured with the provided defaults.

## But… couldn’t all this be done with the old and well-known @Import annotation?

Well, not really. The Autoconfiguration mechanism is guaranteed to occur **after** all the user-defined beans have been registered. This ensures the @Conditional checking is done when the whole user configuration is available and so, makes sure user defined beans takes precedence over the autoconfiguration ones. Actually, the use of conditional annotations is only recommended in **Autoconfiguration** classes to avoid these ordering pitfalls.

All auto configuration logic is implemented in spring-boot-autoconfigure.jar. All auto configuration logic for mvc, data, jms and other frameworks is present in a single jar.



Other important file inside spring-boot-autoconfigure.jar is /META-INF/spring.factories. This file lists all the auto configuration classes that should be enabled under the EnableAutoConfiguration key. A few of the important auto configurations are listed below.

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\

org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\

org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\

org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\

org.springframework.boot.autoconfigure.security.SecurityFilterAutoConfiguration,\

org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\

org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\

org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration,\

Try to debug application by doing logging.level.org.springframework: DEBUG

When you restart the application, you would see an auto configuration report printed in the log. Similar to what you see below, a report is produced including all the auto configuration classes. The report separates the positive matches from negative matches. It will show why a specific bean is auto configured and also why something is not auto configured.

=========================

AUTO-CONFIGURATION REPORT

=========================

Positive matches:

-----------------

DispatcherServletAutoConfiguration matched

- @ConditionalOnClass classes found: org.springframework.web.servlet.DispatcherServlet (OnClassCondition)

- found web application StandardServletEnvironment (OnWebApplicationCondition)

Negative matches:

-----------------

ActiveMQAutoConfiguration did not match

- required @ConditionalOnClass classes not found: javax.jms.ConnectionFactory,org.apache.activemq.ActiveMQConnectionFactory (OnClassCondition)

AopAutoConfiguration.CglibAutoProxyConfiguration did not match

- @ConditionalOnProperty missing required properties spring.aop.proxy-target-class (OnPropertyCondition)

Condition interface

public interface Condition {

boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);

}

 It has a method matches and our conditional logic should go inside this method Then we can use the class we have defined in the @Conditional annotation to check for a condition.

public class DevDataSourceCondition implements Condition {

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

String dbname = context.getEnvironment().getProperty("database.name");

return dbname.equalsIgnoreCase("dev");

}

}

@Configuration

public class EmployeeDataSourceConfig {

@Bean(name="dataSource")

@Conditional(value=DevDataSourceCondition.class)

public DataSource getDevDataSource() {

return new DevDatabaseUtil();

}

@Bean(name="dataSource")

@Conditional(ProdDataSourceCondition.class)

public DataSource getProdDataSource() {

return new ProductionDatabaseUtil();

}

}

**Create a bean if some bean is present**

public class BeanPresennceCondition implements Condition {

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

EmployeeBeanConfig employeeBeanConfig = null;

try {

employeeBeanConfig = (EmployeeBeanConfig)context.getBeanFactory().getBean("employeeBeanConfig");

}catch(NoSuchBeanDefinitionException ex) {

}

return employeeBeanConfig != null;

}

}

@Autowired

private JdbcTemplate jdbcTemplate;

Hibernate properties

**spring.jpa.hibernate.ddl-auto=create-drop**

**@EnableScheduling:**

@SpringBootApplication

@EnableScheduling

public class BookPubApplication {...}

1. As a @Scheduled annotation can be placed only on methods without arguments, let's add a new run() method to the StartupRunner class and annotate it with the @Scheduled annotation, as shown in the following line:

@Scheduled(initialDelay = 1000, fixedRate = 10000)

public void run() {

logger.info("Number of books: " +

bookRepository.count());

}

### **Spring 4.0 @Profile annotation declaration:**

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.TYPE, ElementType.METHOD})

@Documented

@Conditional(ProfileCondition.class)

public @interface Profile {

String[] value();

}

If we observe both declaration, we can find the following changes to @Profile definition.

* In Spring 3.x, we can use the @Profile annotation only at the class level. We cannot use at method level.
* From Spring 4.0 onwards, we can use @Profile annotation at the class level and the method level.
* @Profile annotation is refactored to use @Conditional annotation.

The **@ConditionalOnProperty**annotation is, in my experience, the most commonly used conditional annotation in Spring Boot projects. It allows to load beans conditionally depending on a certain environment property:

**@Configuration**

**@ConditionalOnProperty(**

value**=**"module.enabled"**,**

havingValue **=** "true"**,**

matchIfMissing **=** **true)**

**class** **CrossCuttingConcernModule** **{**

**...**

**}**

 If the property is not set at all, it will still be loaded, because we have defined matchIfMissing as true.

### @ConditionalOnExpression

If we have a more complex condition based on multiple properties, we can use @ConditionalOnExpression:

**@Configuration**

**@ConditionalOnExpression(**

"${module.enabled:true} and ${module.submodule.enabled:true}"

**)**

**class** **SubModule** **{**

**...**

**}**

The SubModule is only loaded if both properties module.enabled and module.submodule.enabled have the value true. By appending :true to the properties we tell Spring to use true as a default value in the case the properties have not been set.

### @ConditionalOnBean

Sometimes, we might want to load a bean only if a certain other bean is available in the application context:

**@Configuration**

**@ConditionalOnBean(OtherModule.**class**)**

**class** **DependantModule** **{**

**...**

**}**

### @ConditionalOnResource

If we want to load a bean depending on the fact that a certain resource is available on the class path, we can use @ConditionalOnResource:

**@Configuration**

**@ConditionalOnResource(**resources **=** "/logback.xml"**)**

**class** **LogbackModule** **{**

**...**

**}**

**@ConditionalOnJava**

Load a bean only if running a certain version of Java:

**@Configuration**

**@ConditionalOnJava(JavaVersion.**EIGHT**)**

**class** **OnJavaModule** **{**

**...**

**}**

**@ConditionalOnWebApplication**

Load a bean only if we’re running inside a web application:

**@Configuration**

**@ConditionalOnWebApplication**

**class** **OnWebApplicationModule** **{**

**...**

**}**

**@ConditionalOnCloudPlatform**

Load a bean only if we’re running on a certain cloud platform:

**@Configuration**

**@ConditionalOnCloudPlatform(CloudPlatform.**CLOUD\_FOUNDRY**)**

**class** **OnCloudPlatformModule** **{**

**...**

**}**

### Combining Conditions with OR

If we want to combine multiple conditions into a single condition with the logical “OR” operator, we can extend AnyNestedCondition:

**class** **OnWindowsOrUnixCondition** **extends** **AnyNestedCondition** **{**

**OnWindowsOrUnixCondition()** **{**

**super(ConfigurationPhase.**REGISTER\_BEAN**);**

**}**

**@Conditional(OnWindowsCondition.**class**)**

**static** **class** **OnWindows** **{}**

**@Conditional(OnUnixCondition.**class**)**

**static** **class** **OnUnix** **{}**

**}**

Here, we have created a condition that is satisfied if the application runs on windows or unix.

The AnyNestedCondition parent class will evaluate the @Conditional annotations on the methods and combine them using the OR operator.

If we want to combine conditions with “AND” logic, **we can simply use multiple @Conditional... annotations** on a single bean. They will automatically be combined with the logical “AND” operator so that if at least one condition fails, the bean will not be loaded:

**@Bean**

**@ConditionalOnUnix**

**@Conditional(OnWindowsCondition.**class**)**

**WindowsAndUnixBean** **windowsAndUnixBean()** **{**

**return** **new** **WindowsAndUnixBean();**

**}**

### Combining Conditions with NOT

Similar to AnyNestedCondition and AllNestedConditions, we can extend NoneNestedCondition to only load beans if NONE of the combined conditions match.

We can create a custom annotation for any condition. We simply need to meta-annotate this annotation with @Conditional:

**@Target({** **ElementType.**TYPE**,** **ElementType.**METHOD **})**

**@Retention(RetentionPolicy.**RUNTIME**)**

**@Documented**

**@Conditional(OnLinuxCondition.**class**)**

**public** **@interface** **ConditionalOnUnix** **{}**

Spring will evaluate this meta annotation when we annotate a bean with our new annotation:

**@Bean**

**@ConditionalOnUnix**

**LinuxBean** **linuxBean(){**

**return** **new** **LinuxBean();**

**}**