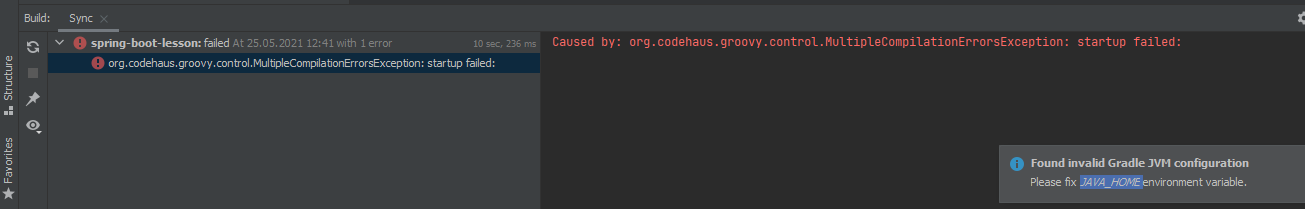
Contents

[Lesson1. First application 2](#_Toc72845345)

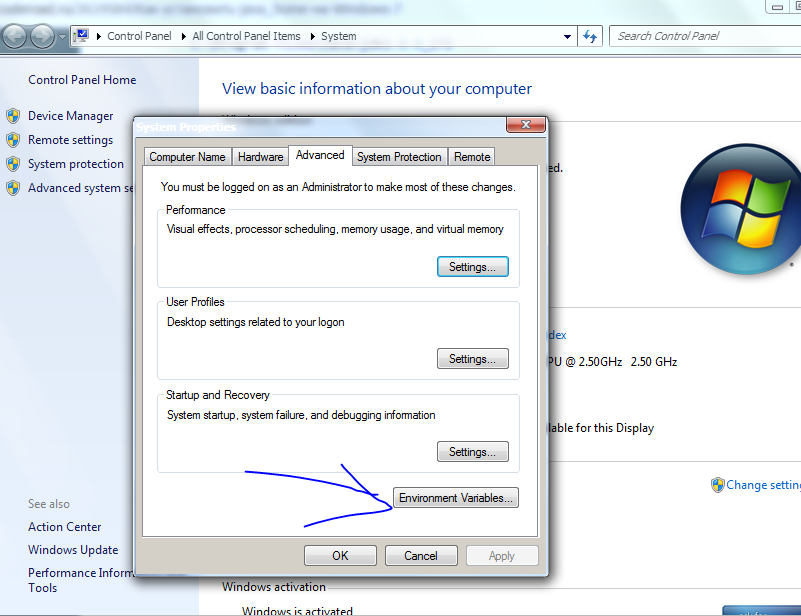
[Lesson2. Conditional and autoconfigure 5](#_Toc72845346)

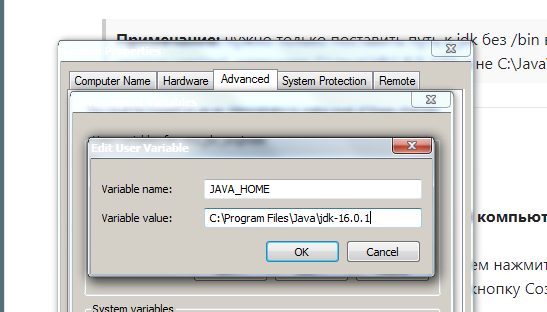
## Lesson1. First application

To create our application we don’t use spring initializer, because it does a lot of things automatically. Instead of this we create empty Gradle project. And here we have the first problem that we didn’t have with maven:



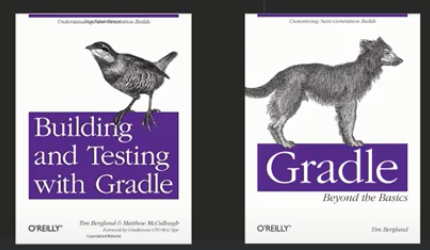
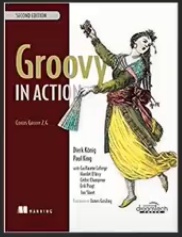
Solution:





To solve compilationError I changed JAVA\_HOME to 15th SDK and created new project with it. 16th SDK throws errors with both Maven and Gradle. Still don’t know why.

Recommended books for Gradle:

Convenient view:

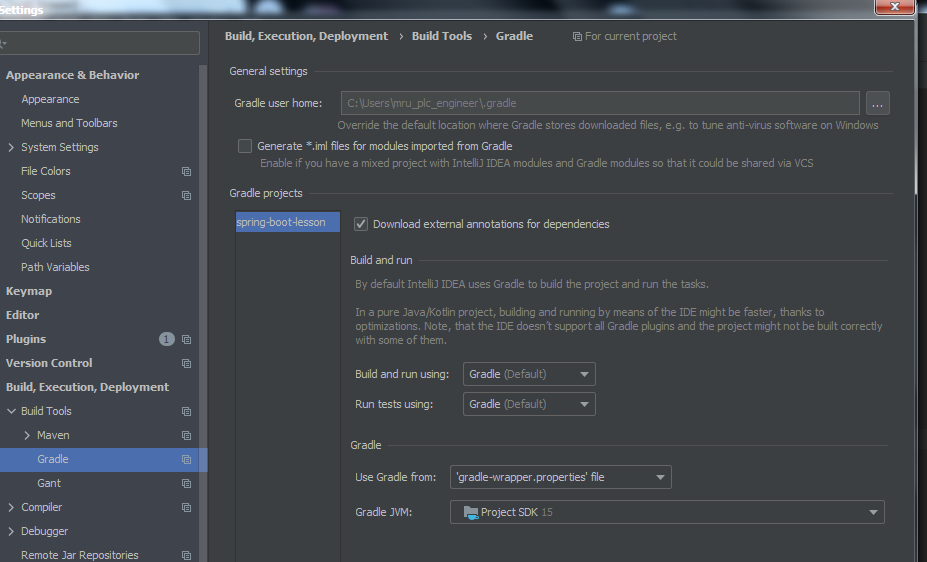
Shift-shift->enter Presentation Mode . And then mouse up View->…->Project. We have nothing but code))

To exit : View->Exit Presentation mode.

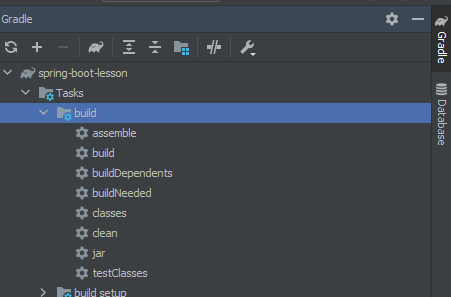
Gradle or Maven wrappers allow us to move our application from one pc to another without bounding to build tool version. We can change wrapper version in property file and it will be pulled up.



We check Gradle JVM version and change if needed.

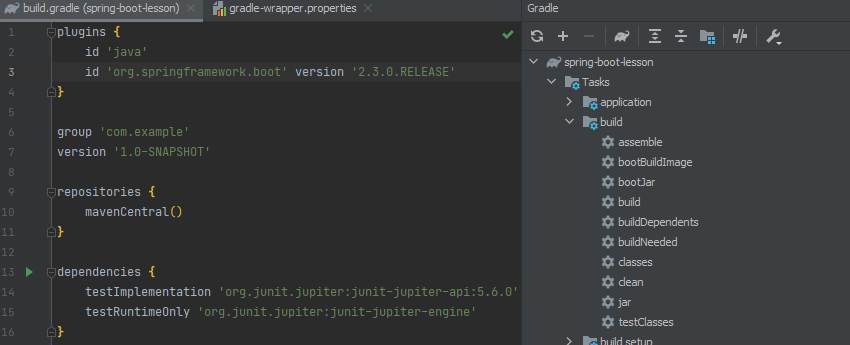


So far we have following application lifecycle:



But it’s an ordinary lifecycle (without using spring boot). Spring boot has different lifecycle. So we add two plugins.

Lifecycle change after adding the first plugin:

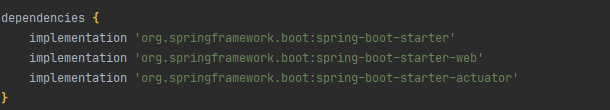


Second plugins helps to avoid collisions of dependency versions:

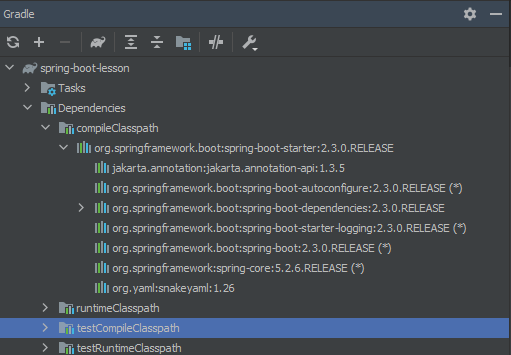


With that plugin we don’t even need to specify versions for most of our libraries.

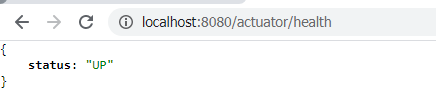
For beginning we add following dependencies:



IDEA with Gradle shows all used dependencies grouped by scope:



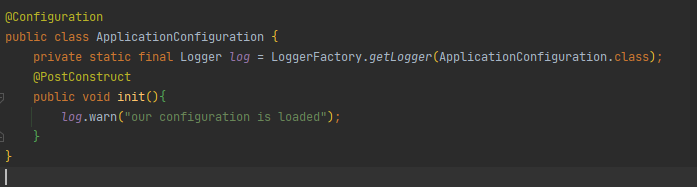
We add default springboot class with main method and run the app:



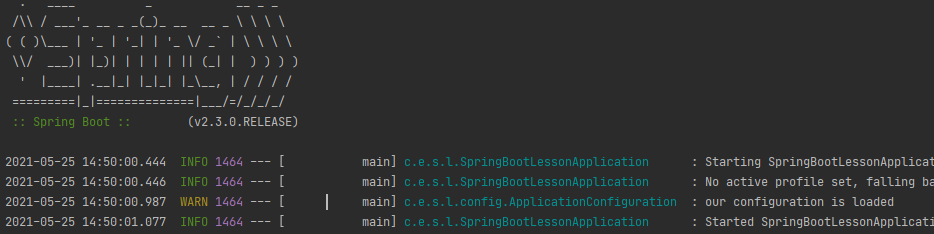
## Lesson2. Conditional and autoconfigure

Spring Boot brought simplicity to configuration of spring modules (MVC, Data…) . In Spring Framework we had to create a lot of configurational beans. And very often they just were taken from another project or Google. Spring Boot supplies default configuration for each module, which can be overridden if required.

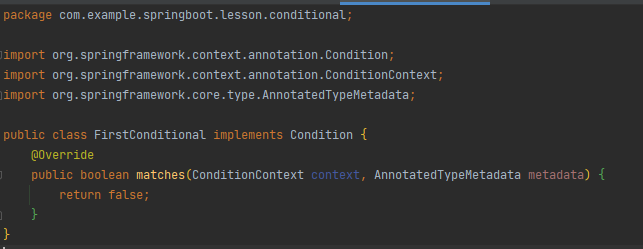
We make our first configuration class and delete all dependencies except spring-boot-starter:

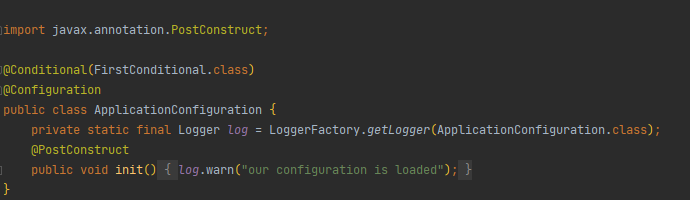


If we start our app it will be loaded:

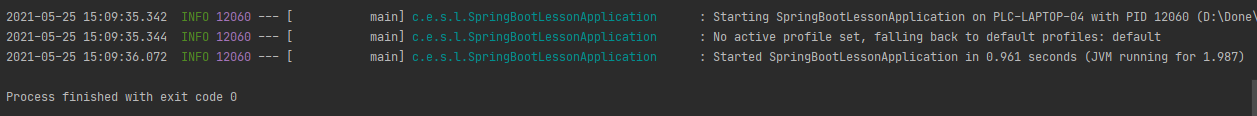


Now let’s use @Conditional. @Conditional is annotation on which autoconfiguration is based. Conditional can get custom or standard conditions. Let’s create a custom:

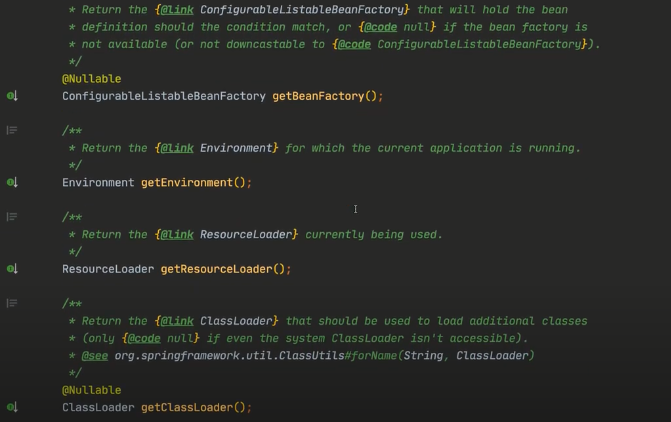


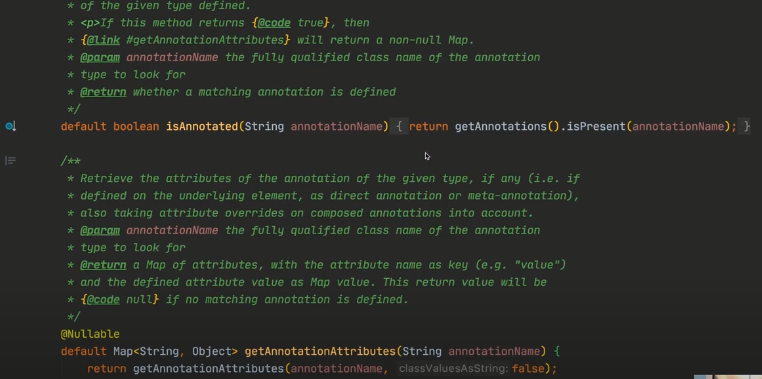


“return false” is ok for us so far. With it we can demonstrate that our configuration will not be loaded anymore.

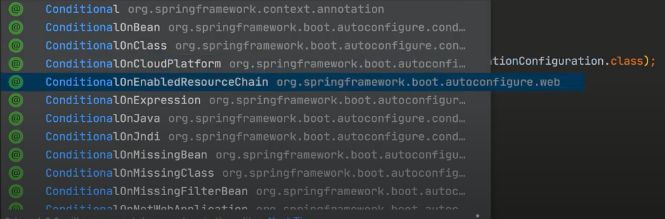


But we have to note that method matches of Condition takes context and metadata. From that we can get access to everything we need to decide if our configuration should be loaded.



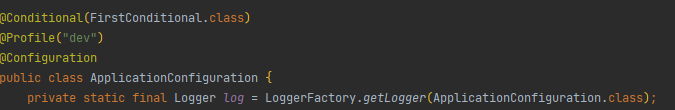


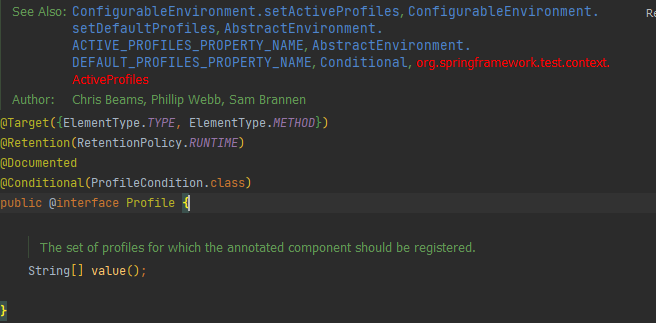
And we have a lot of standard conditionals:



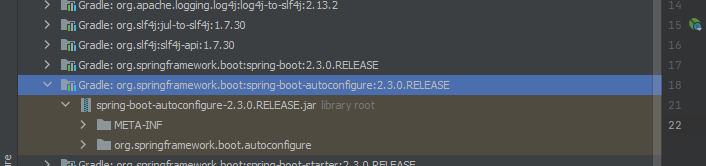
And they are enough. In real practice it probably will not be needed to write custom one.

Also it’s need to know annotation profile (actually it’s also conditional) which takes active profile from metadata:

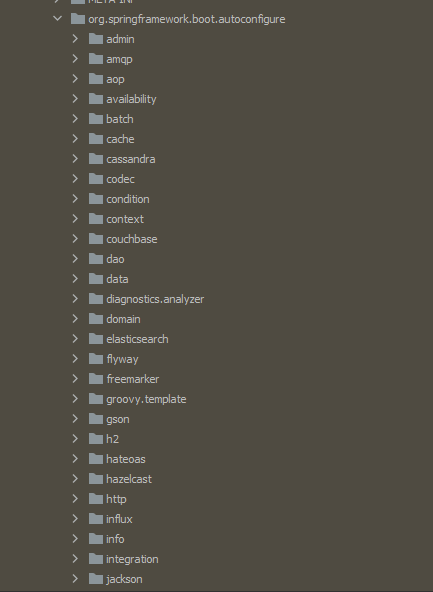




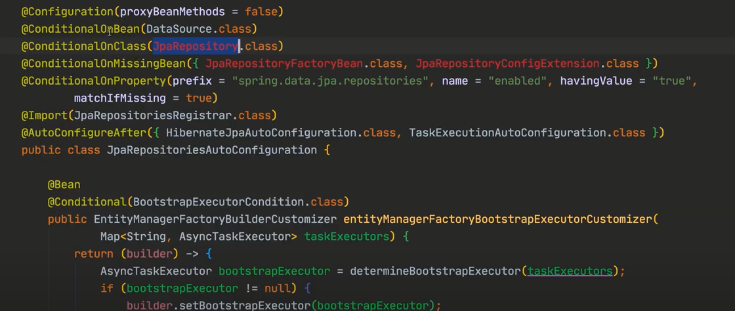
Here is autoconfiguration package:



And we can see that Spring Boot automatically configure a lot of modules:



For instance it’s how JpaRepositoryConfiguration looks:



It will be loaded if all conditions are satisfied.