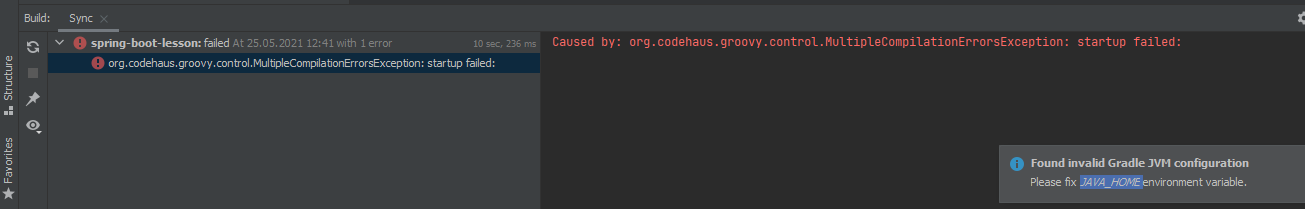
Contents

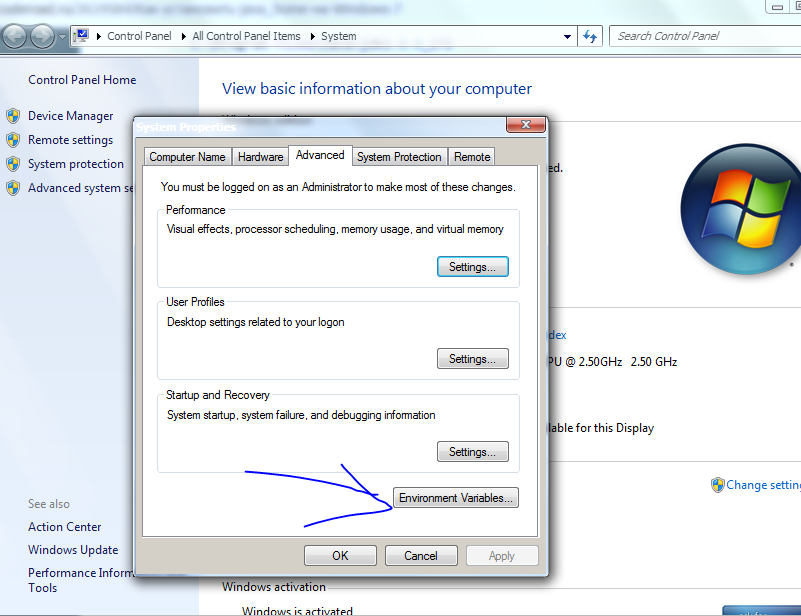
1. [Lesson1. First application 2](#_Toc73011692)
2. [Lesson2. Conditional and autoconfigure 5](#_Toc73011693)
3. [Lesson3. Lombok 9](#_Toc73011694)
4. [Lesson4. Test starter 9](#_Toc73011695)
5. [Lesson5. Data JPA starter: preparations 11](#_Toc73011696)
6. [Lesson6. Data JPA starter: adding entities and repositories 12](#_Toc73011697)
7. [Lesson7. Data JPA starter: testing repositories 12](#_Toc73011698)
8. [Lesson8. Data JPA starter: custom querries 14](#_Toc73011699)

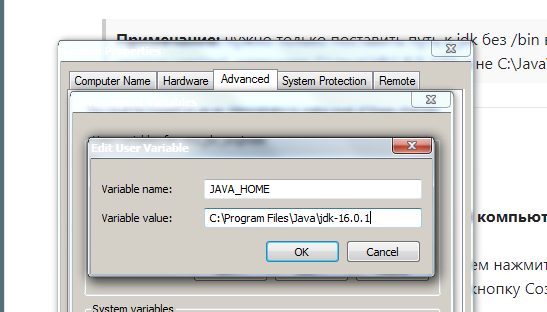
## Lesson1. First application

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:



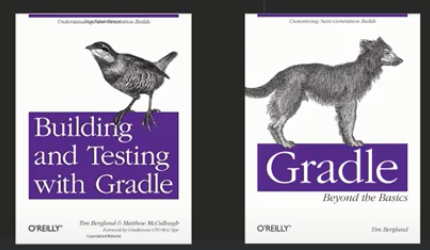
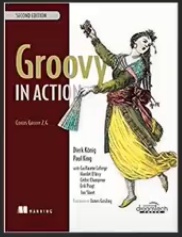
Solution:





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:

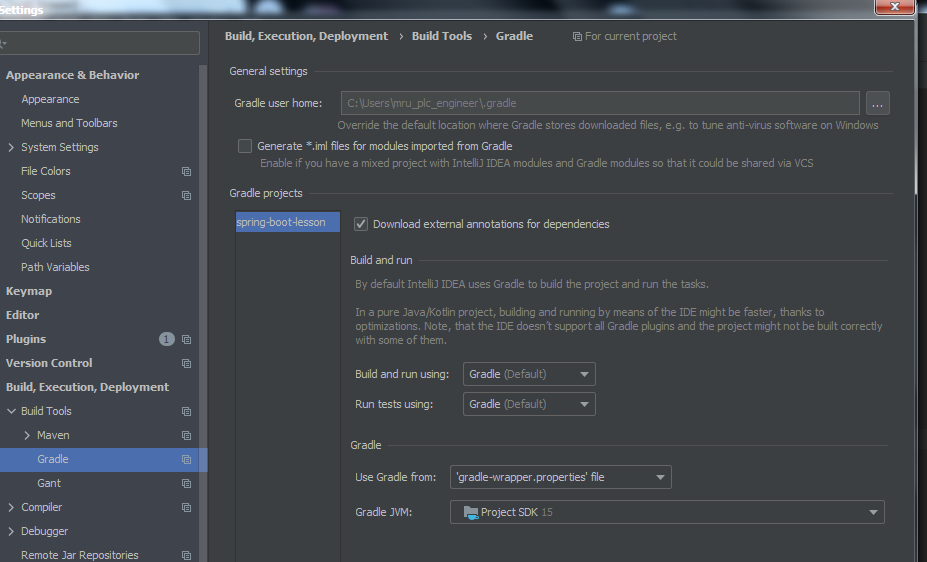
Shift-shift->enter Presentation Mode . And then mouse up View->…->Project. We have nothing but code))

To exit : View->Exit Presentation mode.

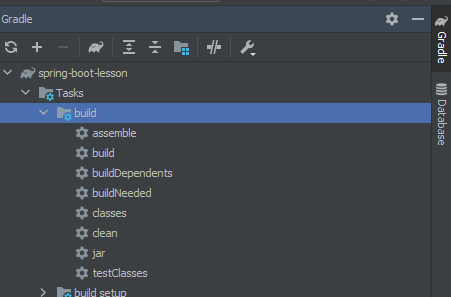
Gradle or Maven wrappers allow us to move our application from one pc to another without bounding to build tool version. We can change wrapper version in property file and it will be pulled up.



We check Gradle JVM version and change if needed.

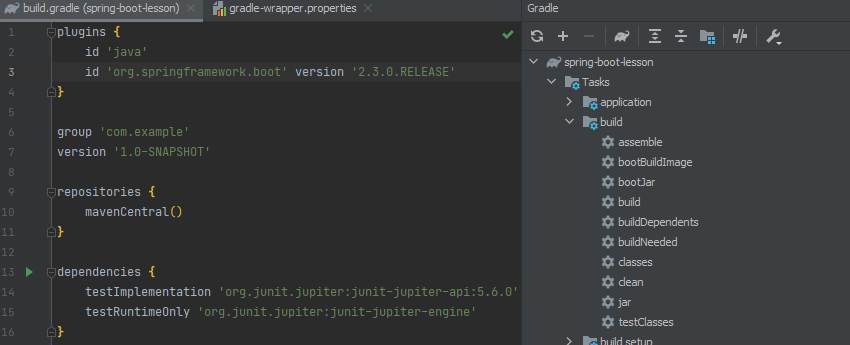


So far we have following application lifecycle:



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:

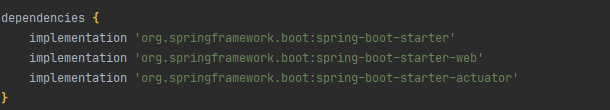


Second plugins helps to avoid collisions of dependency versions:

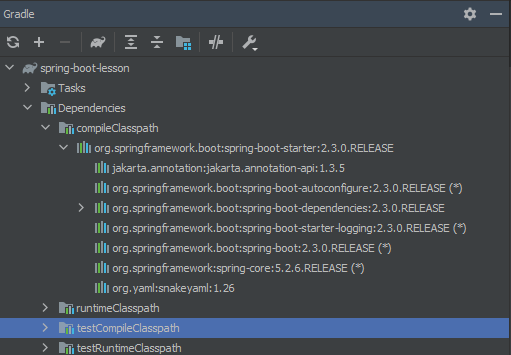


With that plugin we don’t even need to specify versions for most of our libraries.

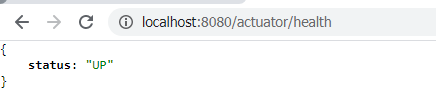
For beginning we add following dependencies:



IDEA with Gradle shows all used dependencies grouped by scope:



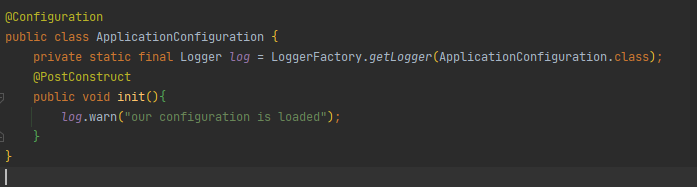
We add default springboot class with main method and run the app:



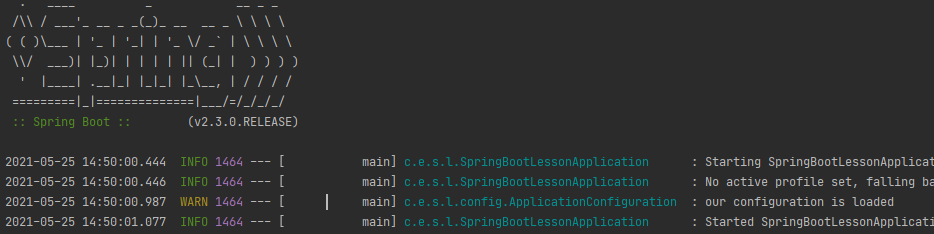
## Lesson2. Conditional and autoconfigure

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.

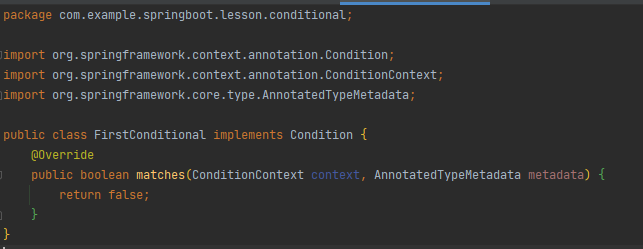
We make our first configuration class and delete all dependencies except spring-boot-starter:

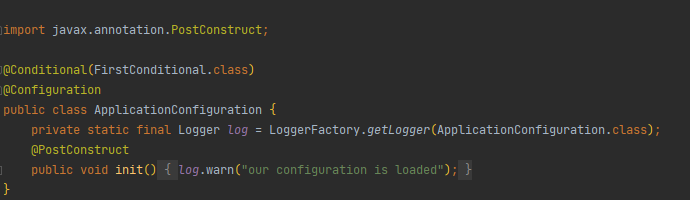


If we start our app it will be loaded:

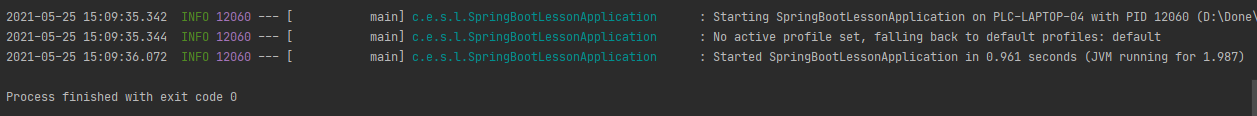


Now let’s use @Conditional. @Conditional is annotation on which autoconfiguration is based. Conditional can get custom or standard conditions. Let’s create a custom:

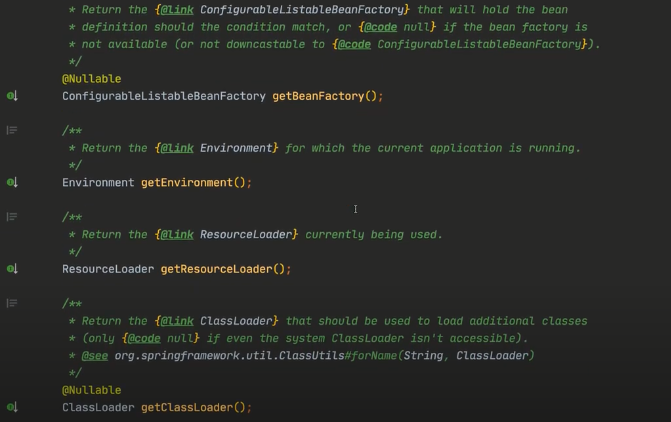


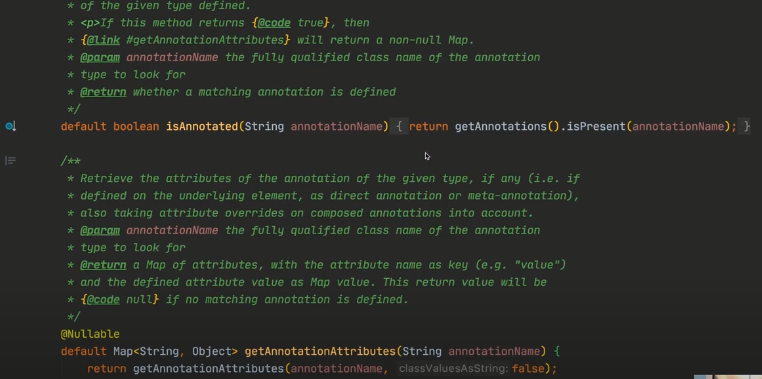


“return false” is ok for us so far. With it we can demonstrate that our configuration will not be loaded anymore.

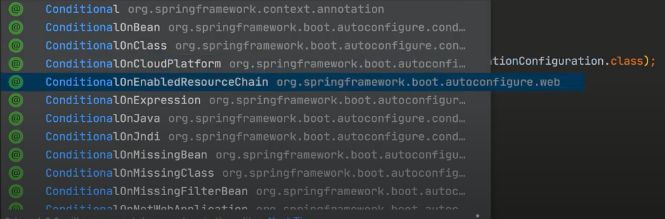


But we have to note that method matches of Condition takes context and metadata. From that we can get access to everything we need to decide if our configuration should be loaded.



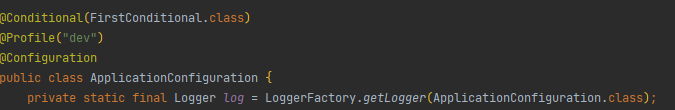


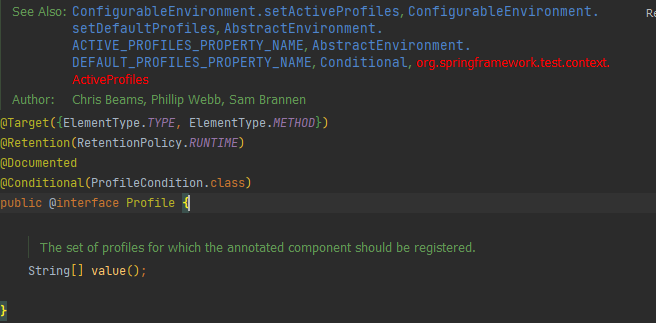
And we have a lot of standard conditionals:



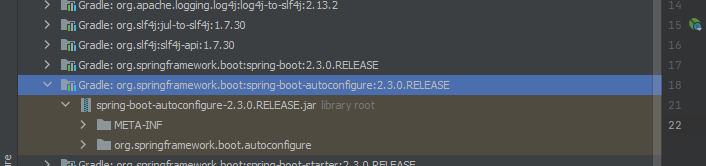
And they are enough. In real practice it probably will not be needed to write custom one.

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

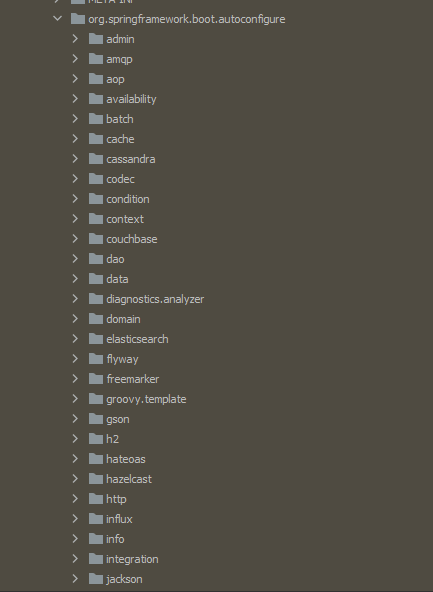




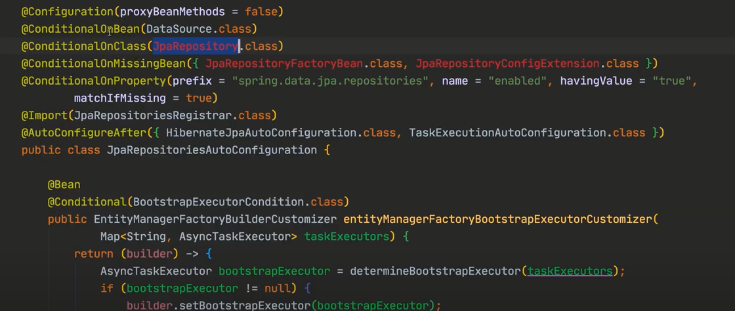
Here is autoconfiguration package:



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



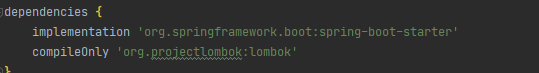
For instance it’s how JpaRepositoryConfiguration looks:



It will be loaded if all conditions are satisfied.

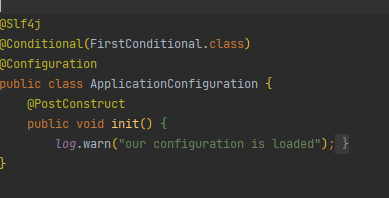
## Lesson3. Lombok

Add Lombok:



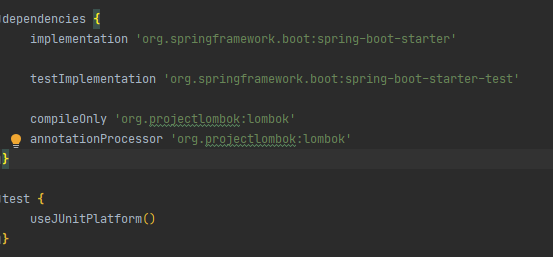
And as usual we activate AnnotationProcessing in project settings and download Lombok plugin.

Now we can replace logger field with annotation:

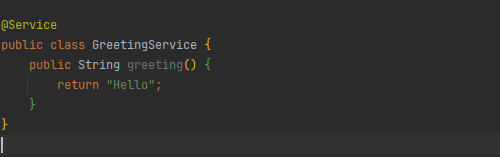


## Lesson4. Test starter

Add Junit5 dependency (useJunitPlatform specifies 5th version):

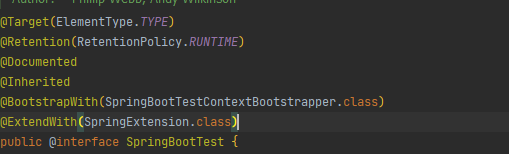


Create simple class:

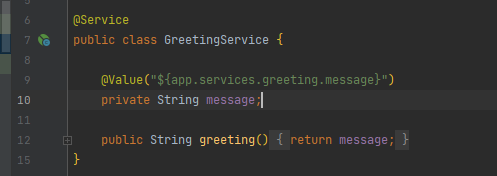


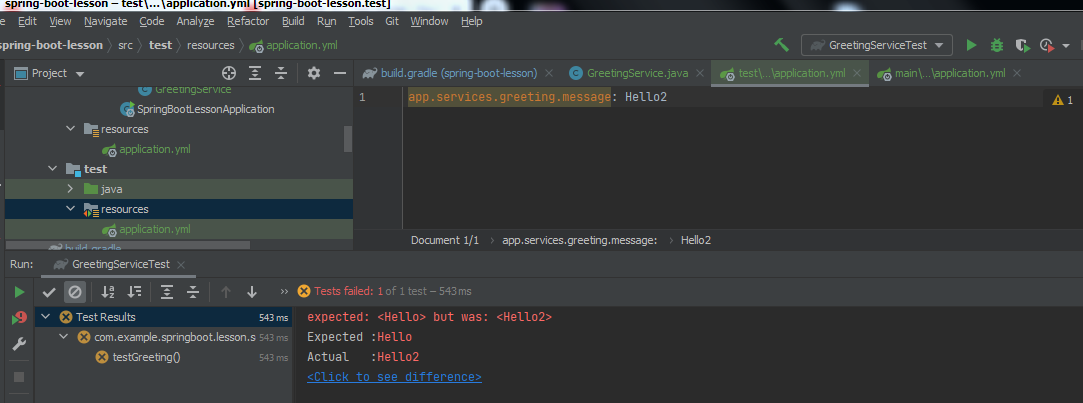
And then we generate a test for the class.

Annotation @ SpringBootTest is used when we need to load Spring Context.

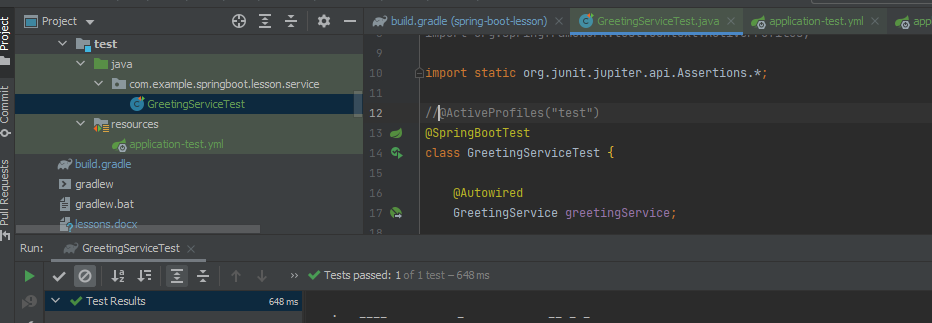


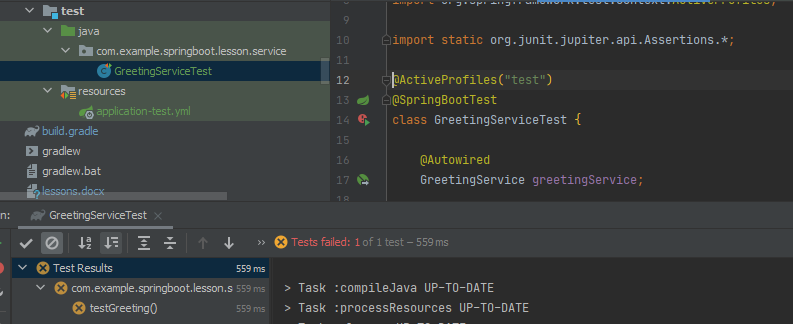
Often we need to override some properties during test phase (for instance db url). To make this we just create the same property file in test directory and put there properties we want to override:





We need to understand that in example spring override not only one property but the whole property file. To avoid the situation and override only specified properties we need to use profiles:





## Lesson5. Data JPA starter: preparations

In beginning of the lesson author shows how to run Postgres in Docker container.

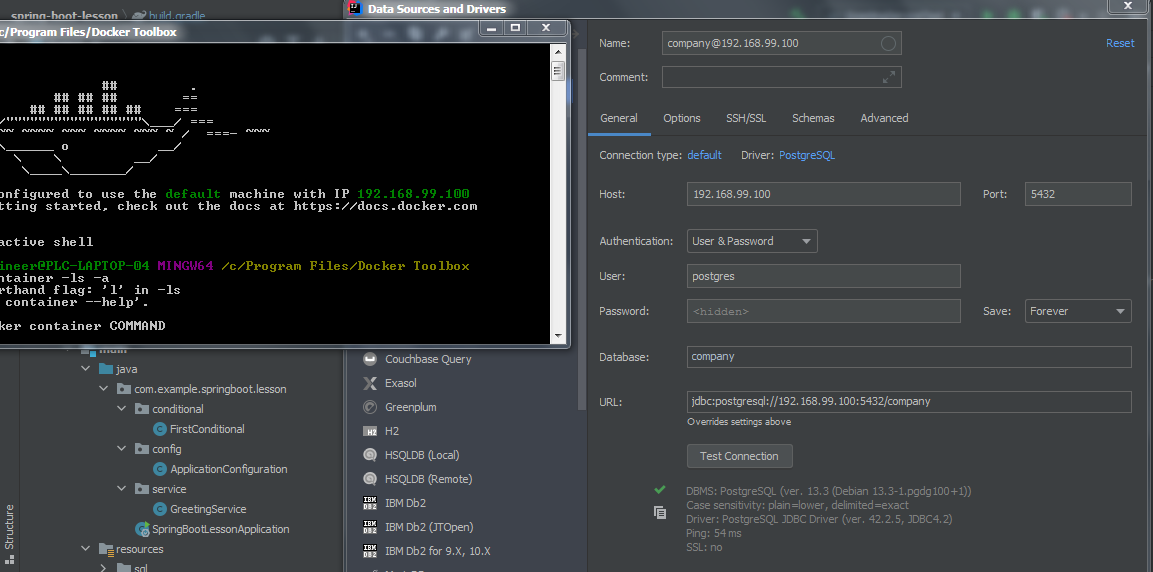
How did I install Docker on Windows 7:

* <https://github.com/docker/toolbox/releases> - download Docker toolbox and install it
* Turned on virtualization in BIOS

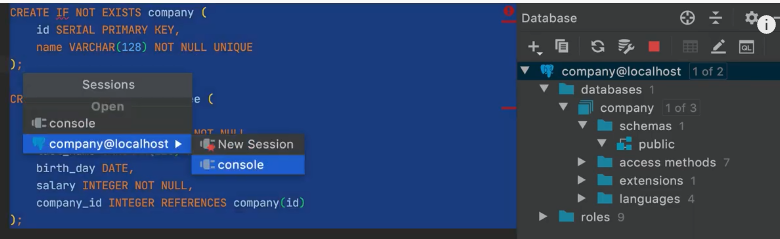
short command list enough for managing postgres container:

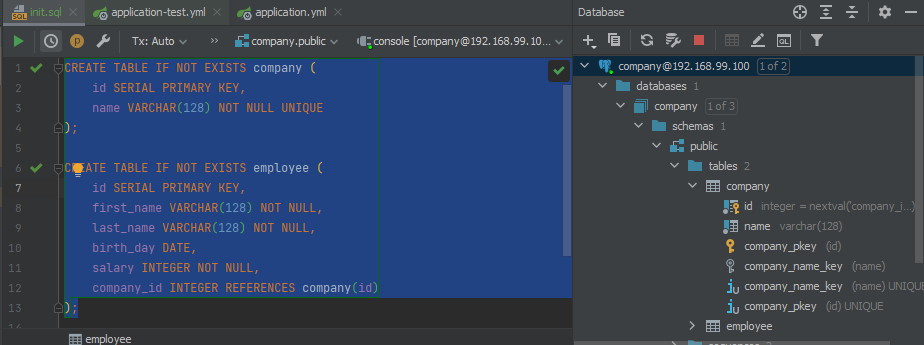
* docker run --name company-postgres -e POSTGRES\_PASSWORD=postgres -e POSTGRES\_USER=postgres -e POSTGRES\_DB=company -p 5432:5432 -d postgres:13
* docker images
* docker images rm <image\_name>
* docker container ls –a
* docker start <container id>
* docker rm <container id>

Note that in my machine Docker uses not LOCALHOST ip:



Then we run our script and get databases created:





## Lesson6. Data JPA starter: adding entities and repositories

First we add jpa starter dependency to byild.gradle.

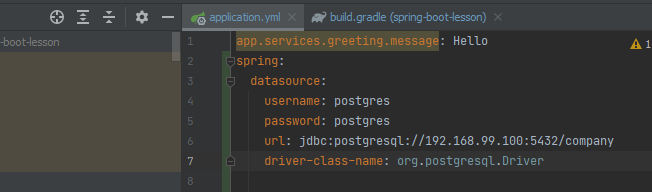
About entity naming:

If we going to use postfix dto for our dto classes, then we can call our entities “company” and “employee”. Otherwise it’s better to call them “companyEntity” and “employeeEntity” (it will be clear from the service layer that those objects are entities).

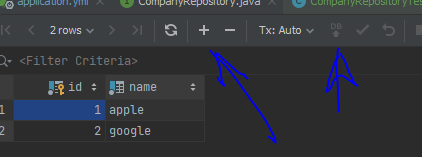
Then we create to entity classes and two repository classes. As usual.

## Lesson7. Data JPA starter: testing repositories

First we add postgres driver to dependencies and added settings to application.yml:



Manual adding to database:



After pressing right button ids are generated automatically (because we used SERIAL datatype).

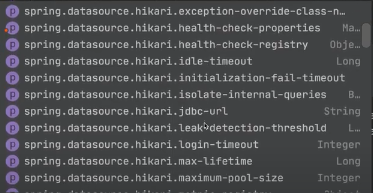
Then we add test for our repository:



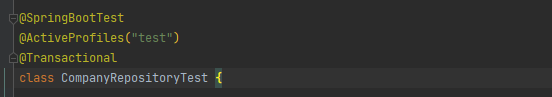
HikariPool is connection pull for our database:



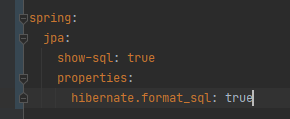
Its settings can be changed in property file:



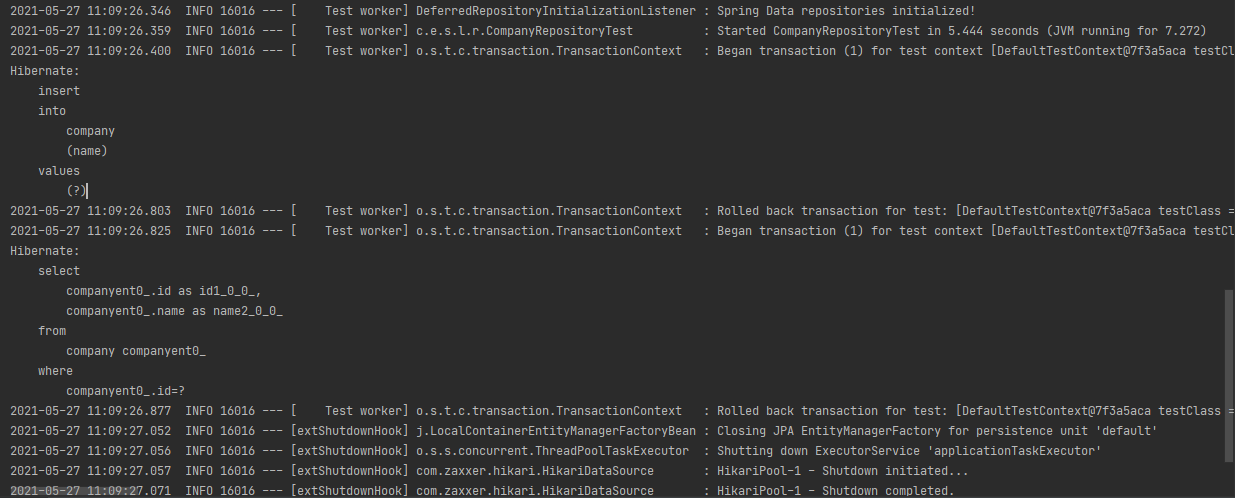
@Transactional allow to rollback transaction after each test and not to save data to our database when we test our CRUD operations:



We don’t want to see our queries to database during runtime, but we do want to see them during tests, so we put new props to application .yml file in our test directory:

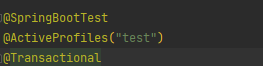


After that we see our queries:

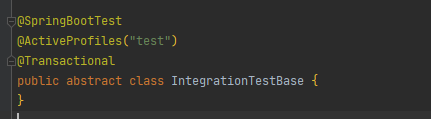


## Lesson8. Data JPA starter: custom querries

At the beginning we create test for EmployeeRepository. And we will have to put on it the same annotations:



To avoid doing this each time, and to make our tests more easy to read we put them on base abstract test class and will extend the class each time:



About custom queries with data JPA:

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

@Querry allows to override querry created by method name by JPQL or native querry:

