**Why frameworks::**

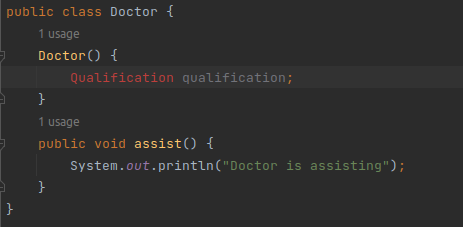
When building enterprise applications, we need to concentrate more towards business logic and don’t want to work on basic cross-cutting concerns.

This is where frameworks come into picture. It provides a lot of basic structure and functionalities and mainly works on business logic.

**Features of Spring Framework::**

* **IOC (Inversion of Control) ::** Giving control to the framework itself. For example, creating objects for thousands of classes in our application which is a tedious task.
  + **Dependency Injection ::** To allow above, Spring uses this. It means giving dependencies of different classes into different classes.
* **AOP (Aspect Oriented Programming) ::** There are certain things which we need to perform repeatedly like logging and security for every request coming from the server. With this approach, we separate all these cross-cutting concerns and invoke them when certain things happen.
* Ability to create web applications. (**Spring MVC**).
* Spring also provides many data libraries with which we can connect to the database easily.
* It also allows the implementation of other libraries and frameworks within it. This is why it is also known as **Frameworks of Frameworks.**

So, Spring adds a layer on the top of the Java applications to make the development easier.

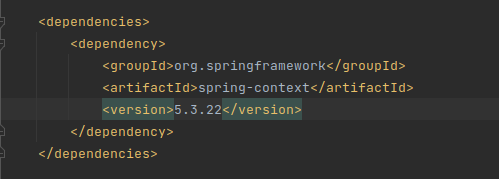


We cannot create a **Doctor** object unless the **Qualification** object is not created since both the classes are tightly coupled.

This concern is solved by Spring. For this we have to do **loose coupling.** With this testing becomes easier since we can do the separate testing on different components and together also.

The thing is that we should be able to create a **Doctor** object even if the **Qualification** object is not created.

This all thing above can be resolved using **Dependency Injection** from Spring. In this we just provide all classes to this that our application has. Also we provide what all things we need then Spring creates a **reference graph** and does all the needful. (We don’t need to create objects now).



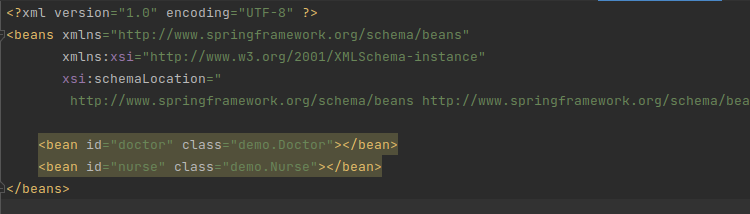
Now Maven fetches all the dependencies (.jar files) from the central repository and creates a local repo for them.

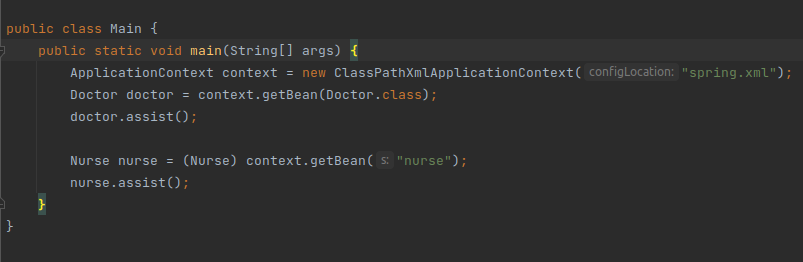
During the start of our application, Spring loads all the beans and stores them in the container.

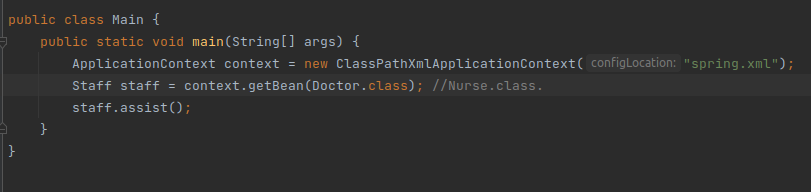
There are two interfaces :: **Bean Factory** & **Application context** from where we can get all the beans which are loaded and use them.

**Application Context** extends **BeanFactory** and it has lots of other features.

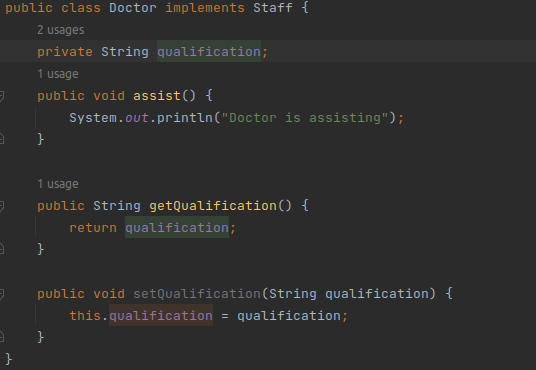
**Dependency Injection Using XML Configuration::**

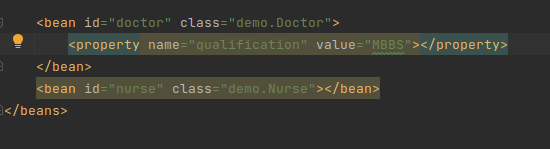


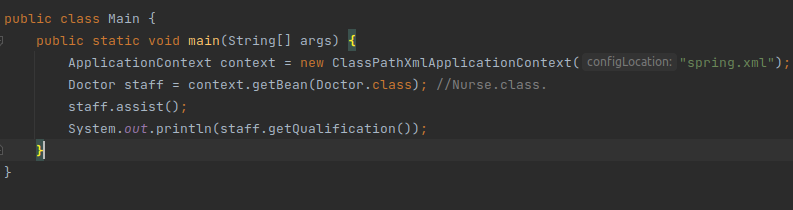




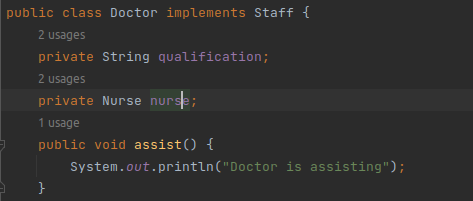
* **Injecting property of a class:: (Setter Injection)**

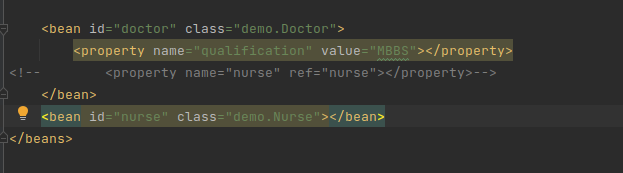
****

****

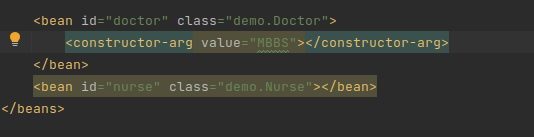
****

* **Injecting an entire object::**

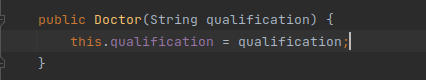
****

****

* **Constructor Injection::**

****

Here Spring expects a constructor inside **Doctor** class with one argument.

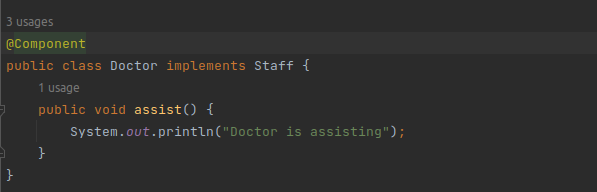


We will get the output as MBBS.

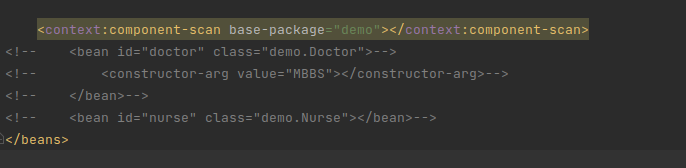
We read there are two ways we can inject values to our beans=

1. Setter Injection.
2. Constructor Injection.

**Dependency Injection using Annotation Based Configuration::**

****

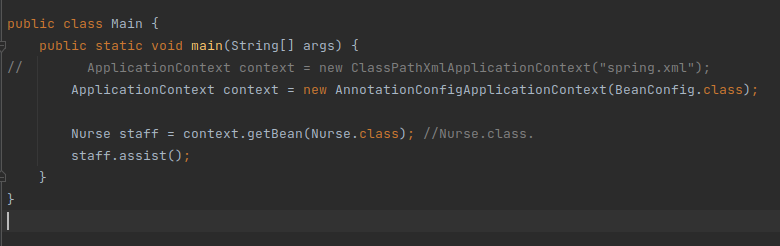
We are saying that through this component, we can create a bean out of it.



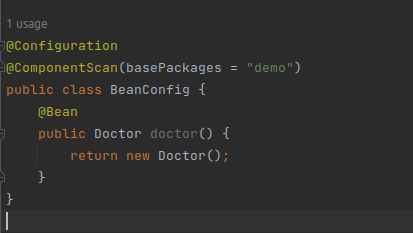
We need to tell Spring where to find the components.

**Dependency Injection Using Java Configuration::**





* **Injecting Properties::**

****

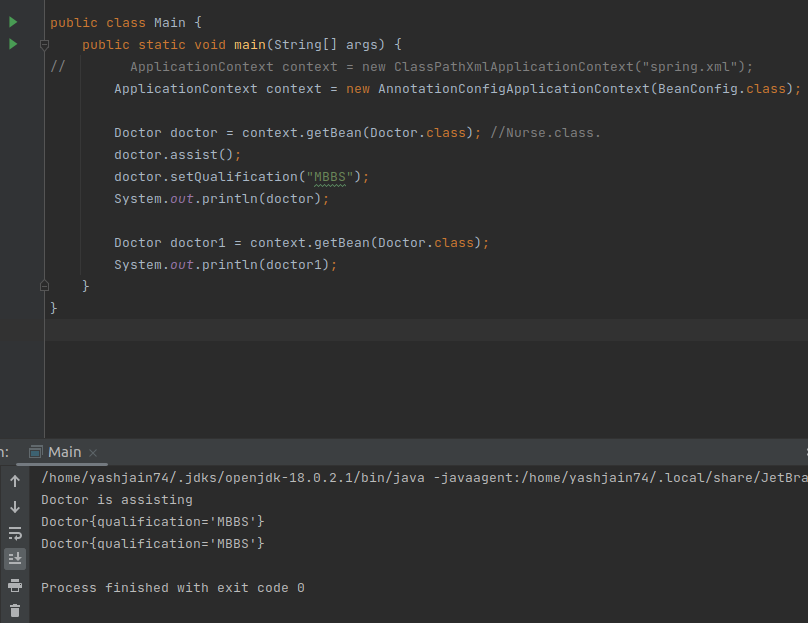
Remove **@Component** from the **Doctor** class since we have created our own bean.

**Scopes::**

Whenever Spring creates a bean, we need to define how those particular beans we want.

Five different types of Scopes::

1. **Singleton** :: (By Default). For every class, there will only one object available in the entire application.
2. **Prototype**
3. **Request**
4. **Session**
5. **Global Session**

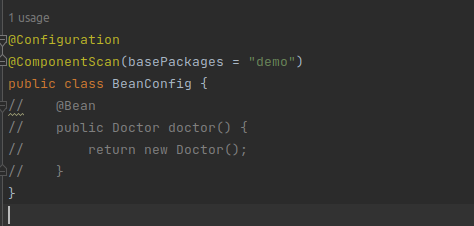
****

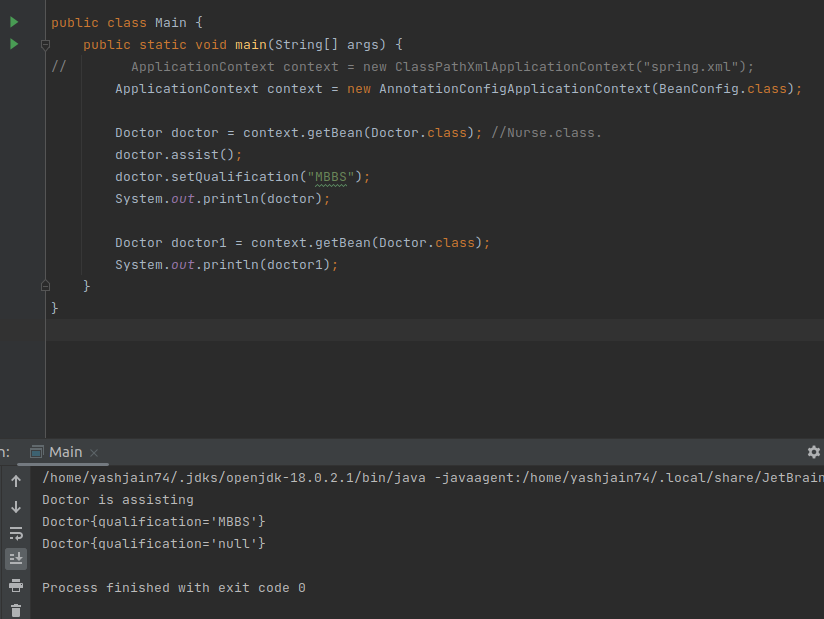
This is the example of **Singleton**.

**NOTE::** We remove @Component only when we have created the bean all by ourselves. But we have to use this annotation if we don’t because Spring won’t know what components to load.

For **Prototype** example::



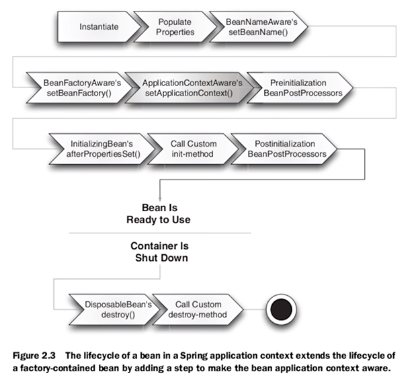




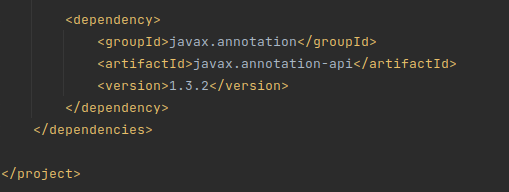
Rest other scopes are used for web context (Spring MVC).

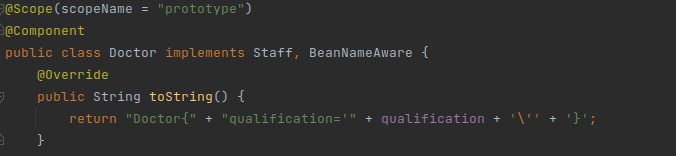
**Spring Lifecycle::**

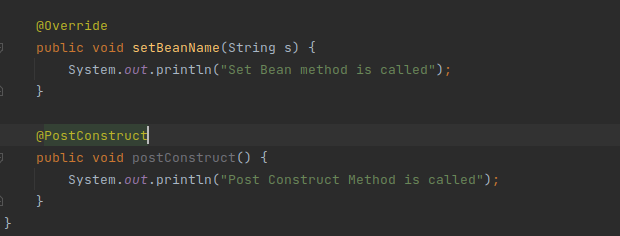
* We first need to define the beans that we need to create.
* After this, Spring will instantiate those beans.
* It then populates the properties. (Values, scope to the particular bean).
* Now the bean gets injected where it needs to and is available from the container.
* Define custom-implementation in the pre-destroy phase when destroying the bean.
* The bean is completely destroyed from the JVM.

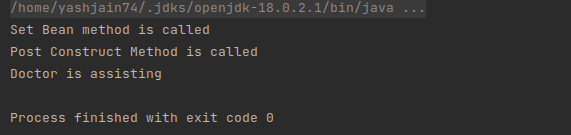


Custom methods are given to pass the default value to the bean like creating or destroying the connection etc.









**BeanNameAware** is used to modify the lifecycle of a bean.

**PostConstruct** is used to add custom methods before processing of the properties.

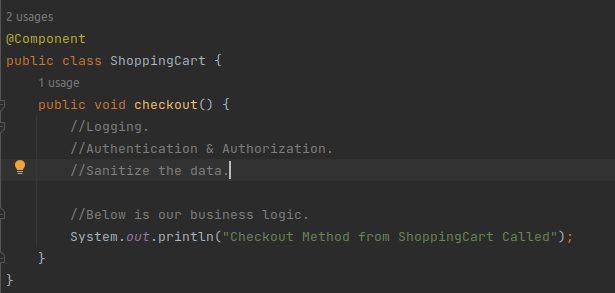
**AOP (Aspect Oriented Programming)::**

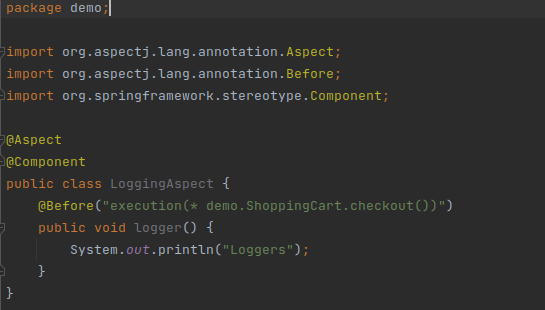
Separating all the cross-cutting concerns from the tasks to perform and execute them when needed. So with this business logic also gets separated.

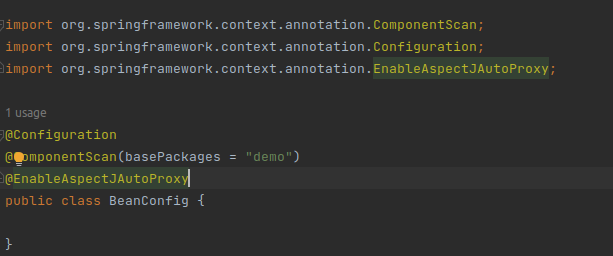
All the cross-cutting concerns are called **aspects.**

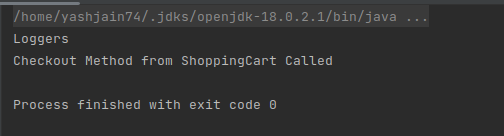
Now keep this question handy:: **What we want to call & where we want to call.**











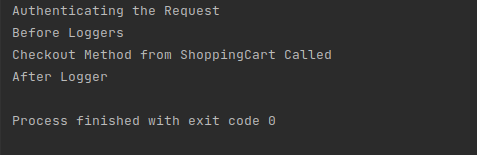
Like **@Before**, we have different **point-cuts** that we can call.



If for example, we add an argument to the **checkout()** method. Then in this case, those two logger functions won’t be executed because it doesn’t match with the signature mentioned.

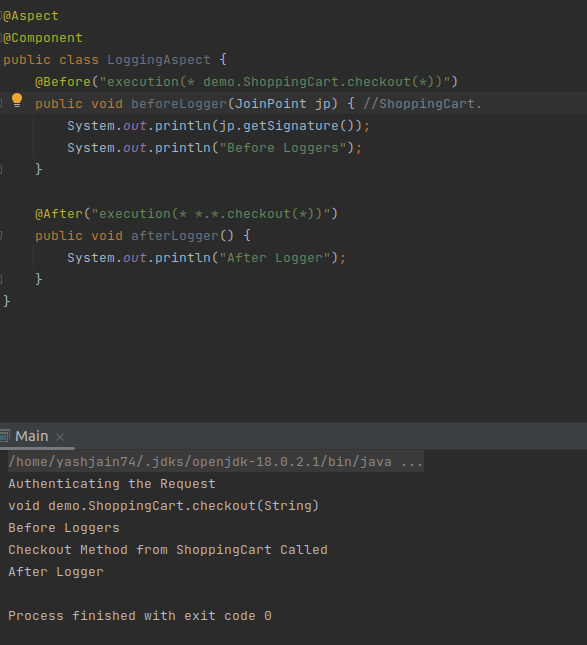
**AuthenticationAspect::**

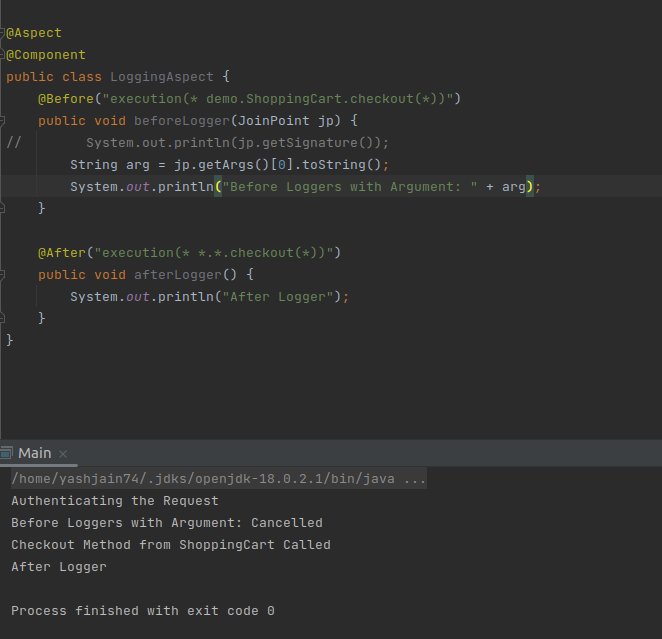
****

****

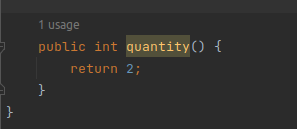
**within()::** It means that this pointcut is available for all the methods in all the classes defined inside the **demo** package.

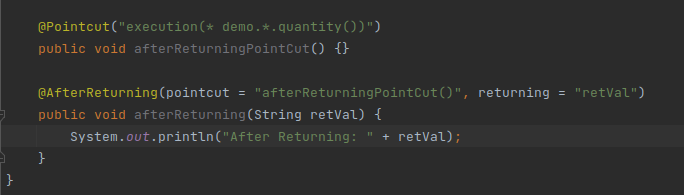
**Access Data::**





**@AfterReturning ::** Not working.





**Spring Boot::**

If we wanted to use any module or library within the Spring framework, we needed to do a lot of configurations for it to work.

To make developers focus mainly on business logic and not on configurations, Spring Boot was introduced. It is the extension layer for the Spring framework.

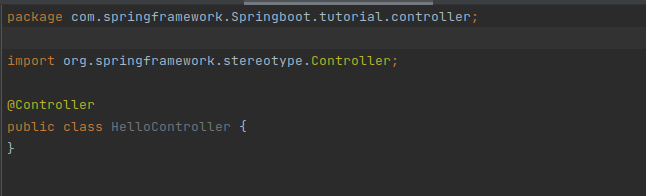
* Rapid application development.
* Earlier, we needed to add a lot of dependencies to perform certain tasks. With Spring Boot, it provides **starter templates** that include all the dependencies that we use to implement something.
* It has an auto-configuration feature meaning we don’t have to configure for all the dependencies or libraries that we are going to use. Just add that starter template for that library.
* Embedded server for the entire application. We don’t need to create .**war** files for our applications and then deploy those files in the application server. Spring Boot does all of this. Here we are just going to create a .**jar** file and then can run this file in any application server we want to.

Any starter template can be downloaded from <http://start.spring.io>.

With a **web starter template**, we have all the dependencies to run a web application and we don’t have to add those dependencies manually.

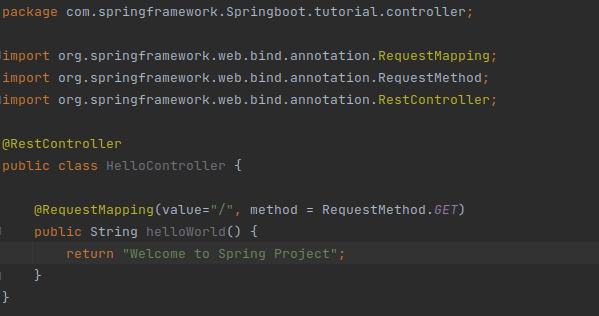
**@SpringBootApplication** is the annotation we use for Spring Boot applications

**Making REST APIs::**



**@Controller** annotation tells Spring that this class is a controller. This still behaves like a component.

To make a REST controller and not just a simple controller, we can use **@RestController** which ensures that this is a component and is going to return a response body.



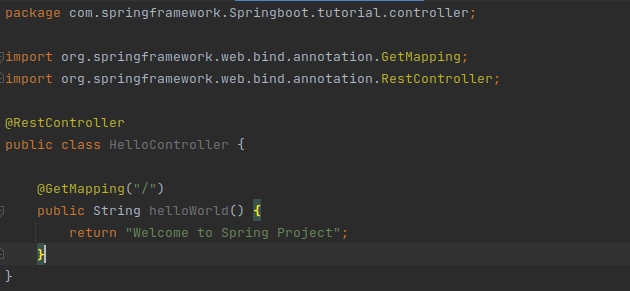
As we know, Spring does all the auto-configuration for us but if we want to change the configuration for something, we can do that in the **application.properties** file.

So for example, by default the auto-configuration for tomcat is done for us that the application will run on the **8080 port** by default. But we change that as well.



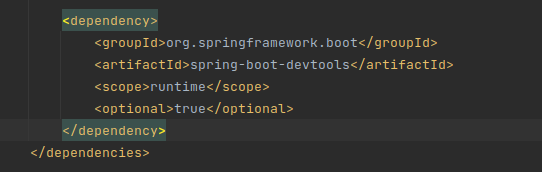
Now we will get the required output on the webpage.

**Shorter Syntax::**



We can also run the same from the terminal: **mvn spring-boot:run**

There is a dependency **dev tool** which we can use to restart the application automatically when any changes in the application is detected.



But we need to add some configurations to intelliJ to support live building functionality.

**For H2 database::**

# H2

spring.h2.console.enabled=true

# Datasource

#spring.datasource.url=jdbc:h2:file:./data/app\_db

spring.datasource.url=jdbc:h2:mem:spring\_boot\_h2

spring.datasource.username=sa

spring.datasource.password=pass

spring.datasource.driverClassName=org.h2.Driver

spring.jpa.hibernate.ddl-auto=update

<dependency>

<groupId>com.h2database</groupId>

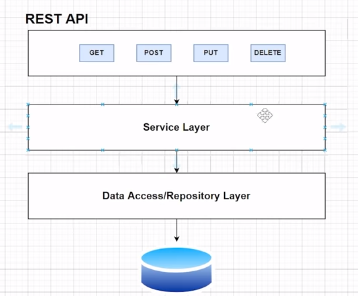
<artifactId>h2</artifactId>

</dependency>

For accessing it: **/h2-console/**

**For Swagger:**

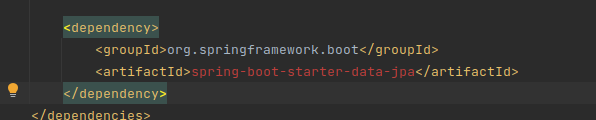
**Architecture & Example::**



There will be a **controller layer** for handling all our requests.

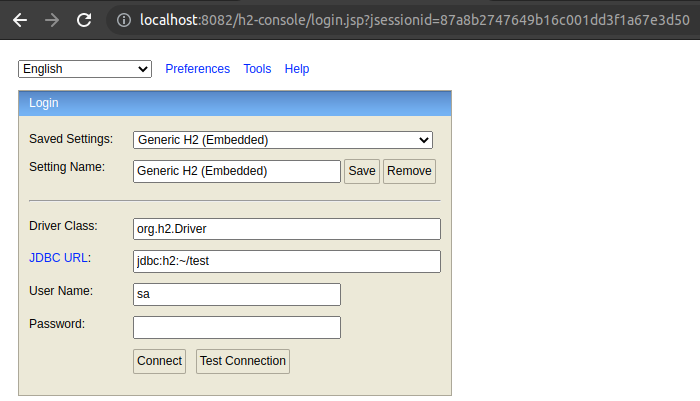
All the business logic that we want to add will be done inside the **service layer.**

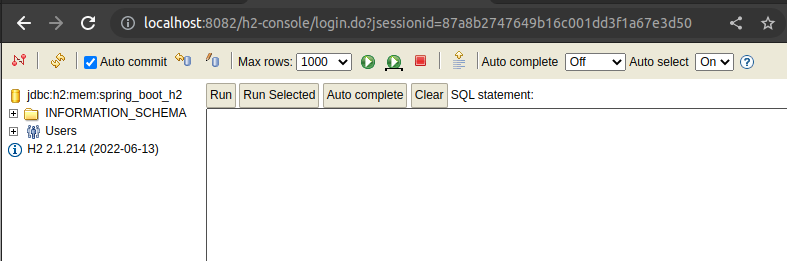
**Repository layer** is the one which interacts with our database. For this we are going to **spring data JPA** for which we need a couple of dependencies.



To add JPA.

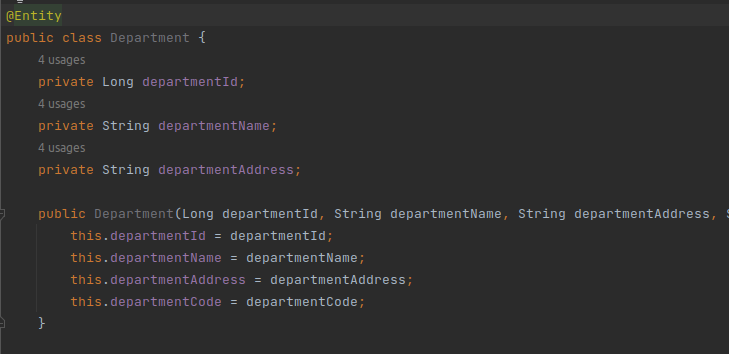
Now for the database, we need to add a couple of configurations to the application.propertiesfile.



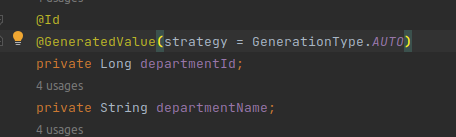


**Implementation of the Architecture Discussed Above::**

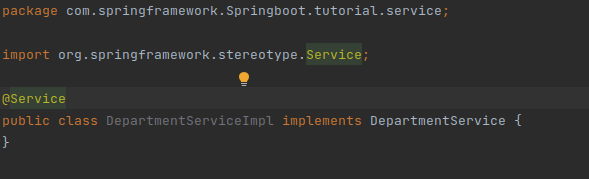
To make the entity for the **Department** that will interact with our database::

****

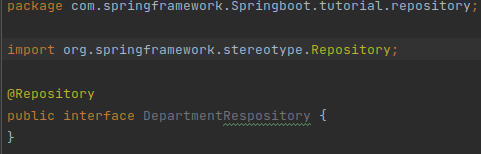
In relational database, we need a primary key for our table::



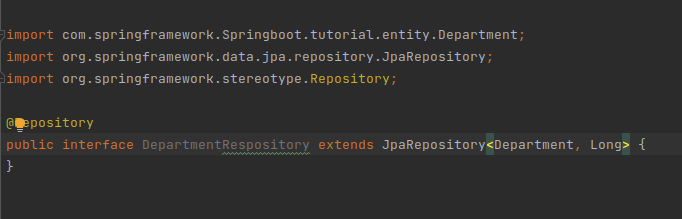
Now to make **services**, we create an **interface** and then create a class implementing that interface since one service can have multiple implementations.



Similarly we have to do it for **repositories** as well.

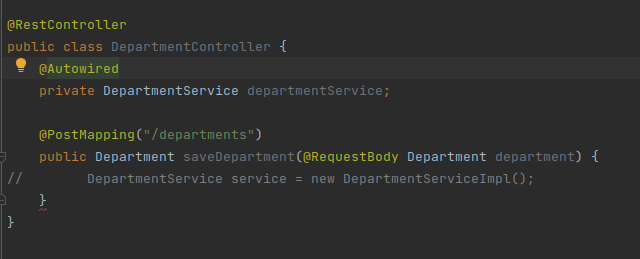


Here we won’t be creating a class that will implement this interface. Instead we will be extending the JPA repository because it comes with lots of methods that we can use to interact with all our entities with our database.



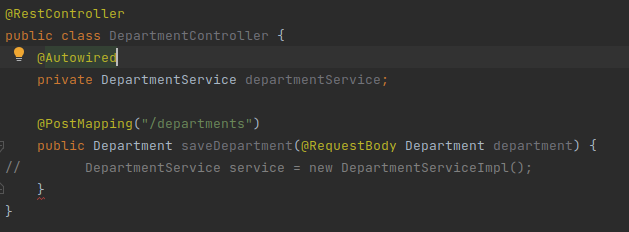
In this we need to pass the **<entity\_name, id\_type>**.

**REMEMBER::** All these annotations implement **@Component** as well and so these all classes we created are in the radar of Spring (since Spring only understands Component).



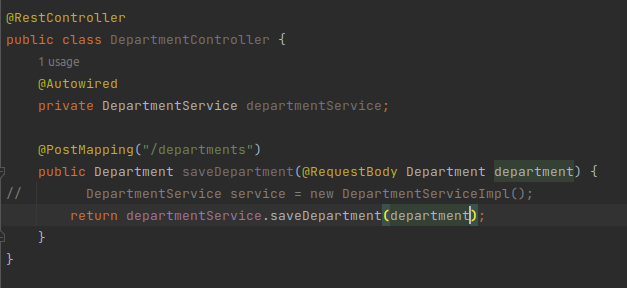
**@RequestBody** automatically converts the JSON object that we are passing to the object that we specified.

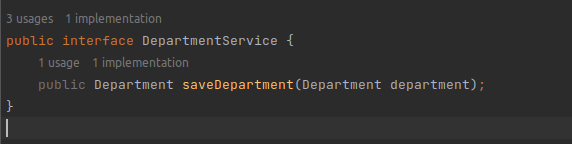
Here we will be calling **DepartmentService** to send the data so that we can implement the logic over there. But here we are again taking the control that we don’t want.

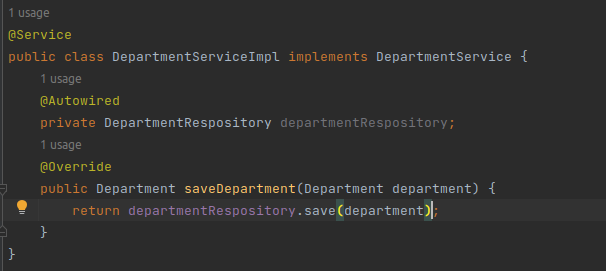


With **@Autowired** we are creating the objects from the components/beans that are stored in the spring container from the reference I created.

This is **property-based dependency injection.**

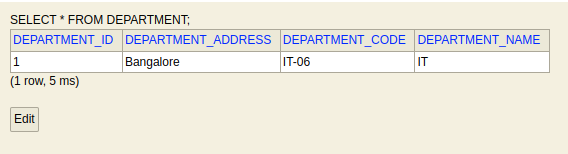






Spring JPA also creates the table for us in the database. If there is any update, then JPA does that automatically to our database.

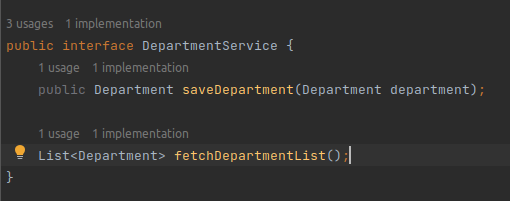
To test it, we can use POSTMAN for this.

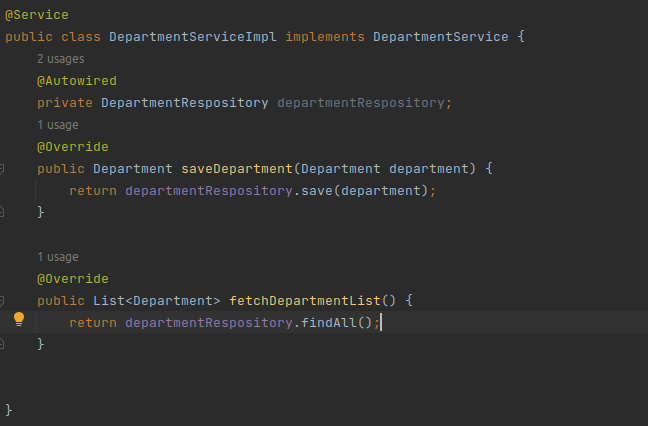


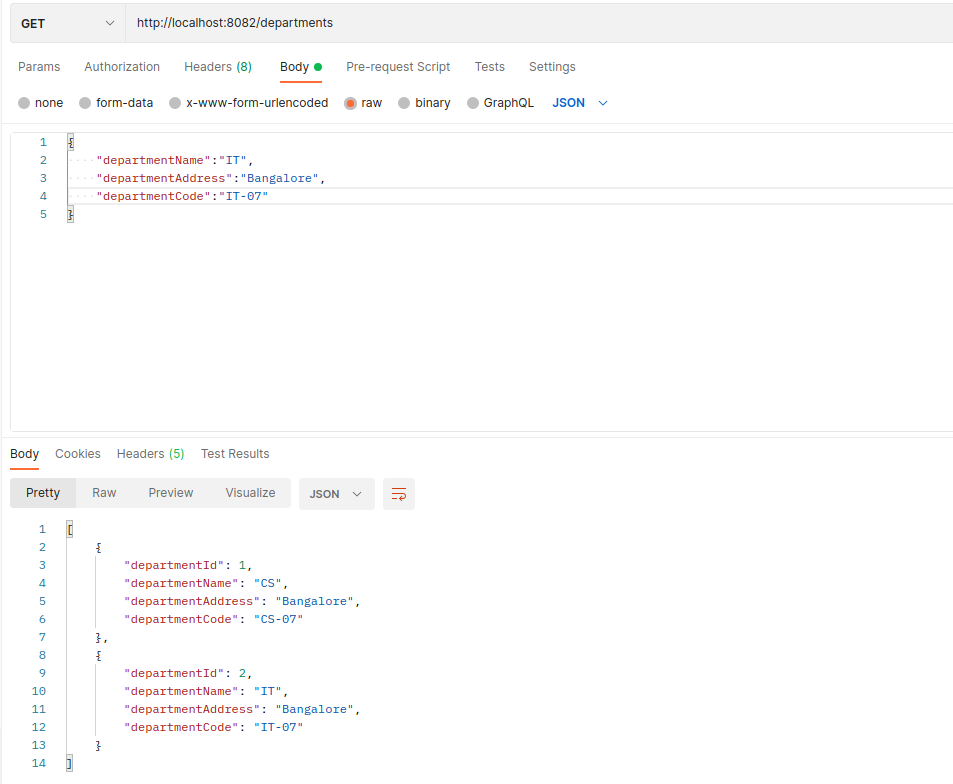
The data is successfully saved.

**GET ALL DEPARTMENTS From the Database::**

****

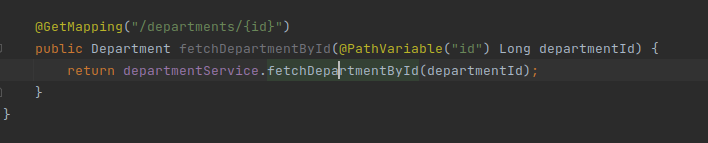
****

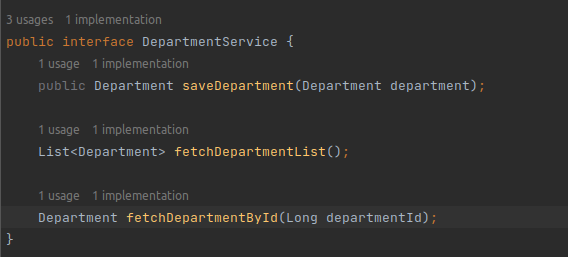
****

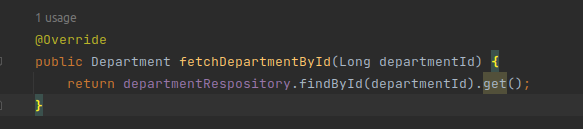
****

We get the required data from the API.

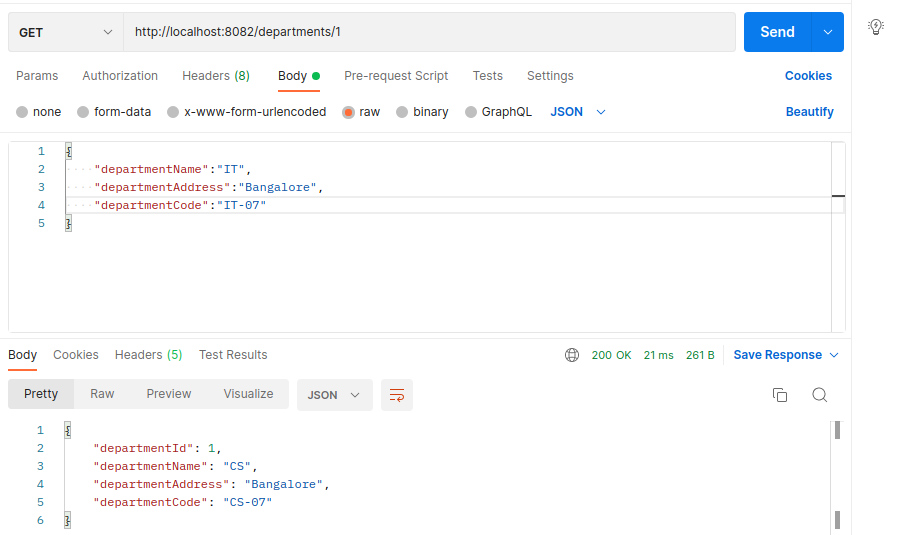
**FETCH DEPARTMENT BY ID From the Database::**

****

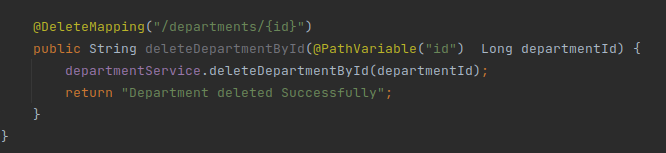
****

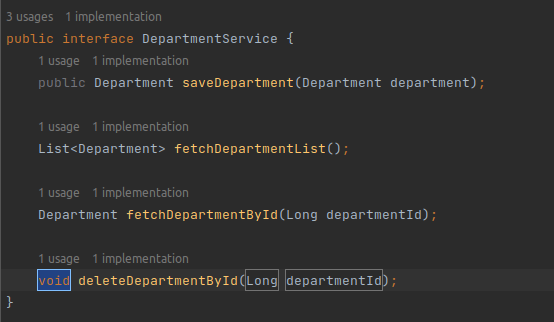
****

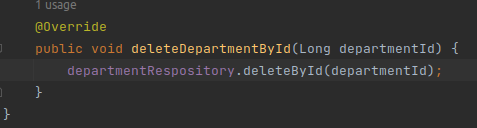
This returns an optional so to get the value we have to **get()**. method to it.

****

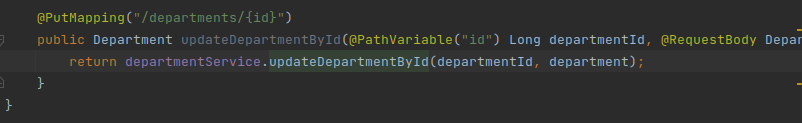
**DELETE DEPARTMENT from the Database::**

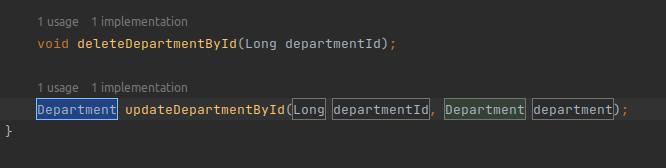
****

****

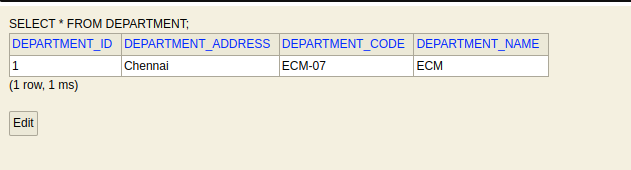
****

**UPDATING DEPARTMENT From the Database::**

****

****

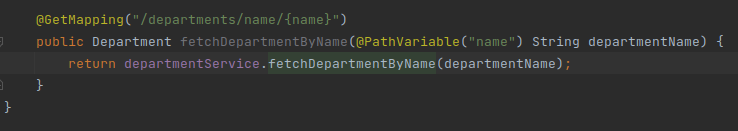
****

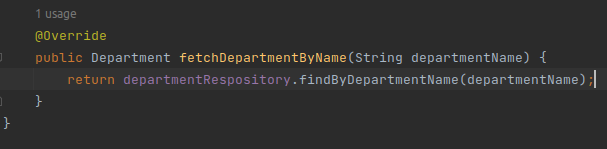
****

**These are all the CRUD functionalities we looked above.**

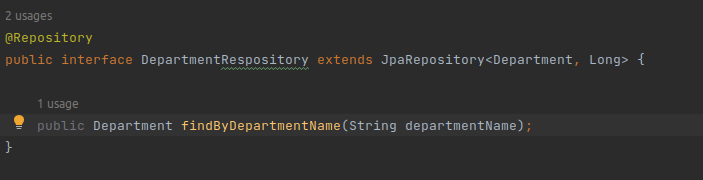
**Fetch Data By Name::**

For this we don’t have any specific method to fetch the data by **departmentName**. So, we need to create one custom method inside the **departmentRepository.**



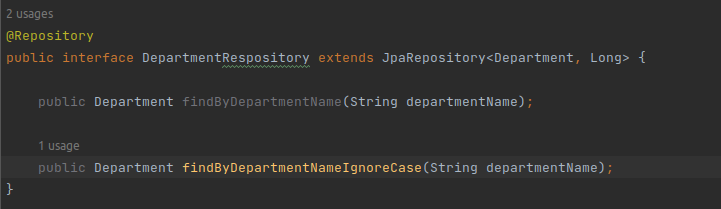


Here we have created custom method inside the repository as shown below::



**Proper naming convention has to be followed when naming methods inside the repository.**

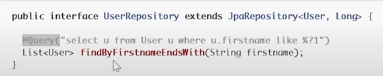
**GET DATA BY NAME (IGNORE CASES)::**

****

Now use this method instead of **findByDepartmentName**.

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

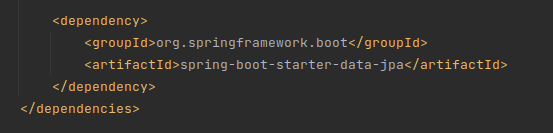
Now there are some queries which you want to write but are not possible from the above documentation methods available. In that case we can write our own JPQL queries or native SQL queries. (Just pass **, nativeQuery = true** to the **@Query** as well).

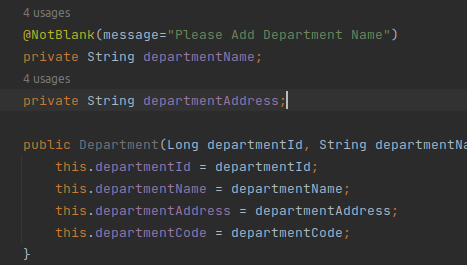


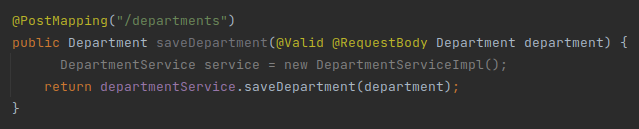
**Validations::**

For example when we are getting data in the form of JSON then we can say that the request is bad if I don’t get a specific field otherwise it’s good.

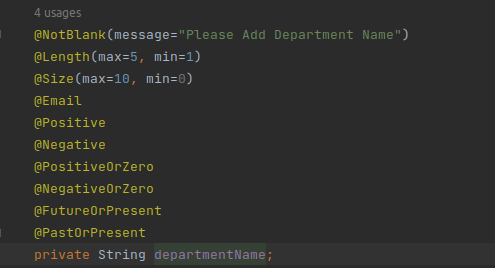
For adding validations, we use **hibernate** validation and for this, we need to add dependencies for it.





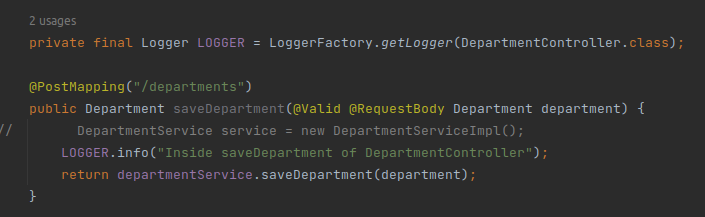


When running it will get a **bad request** error.



There are a bunch of validations that we can use from the above annotations that we can see.

**Adding Loggers::**

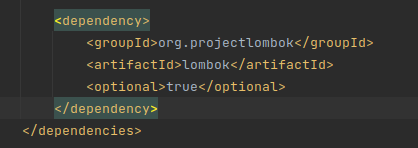
****

We can create policies as to how we want to aggregate the logs either in the console or in a file.

**Lombok::**

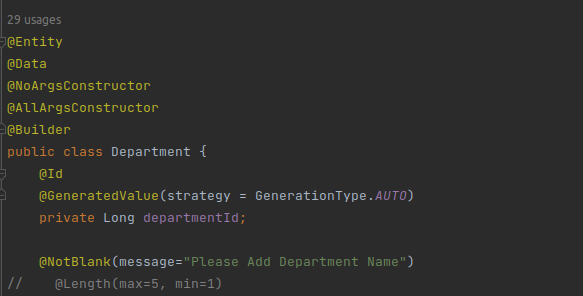
In an application, we tend to create a lot of POJOs like the **Department** we created. For every **POJO**, we have to write its own getter and setter methods along with other methods like **toString()** method etc.

This is where we can find a library called **Lombok** that can help to reduce the lines of code.



To tell the maven plugin that we are using lombok.



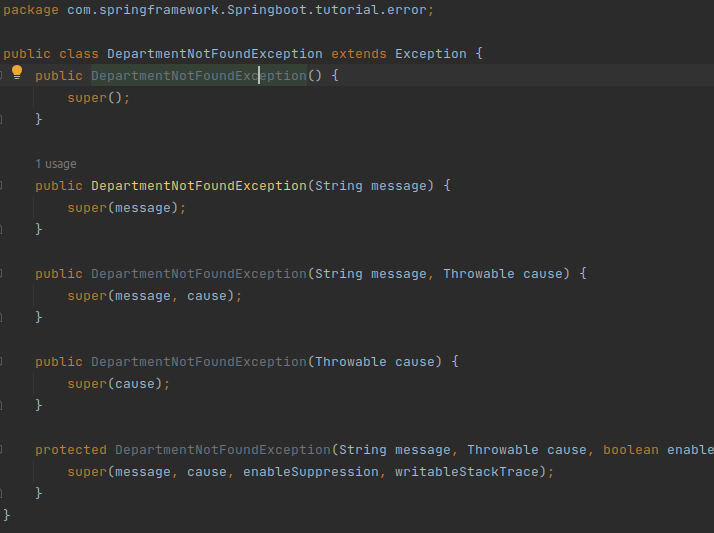


**@Data** annotation contains all the getters and setters methods and other methods too so we don’t have to add them.

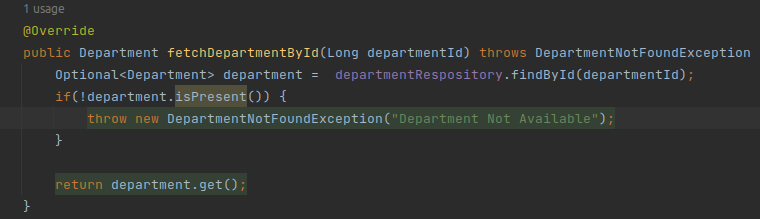
To implement the builder pattern we can do that with **@Builder.** Suppose we have 5 variables and want to initialize only 3 then the **builder** can be helpful to do that without using the constructor.

**Exception Handling::**

Suppose if we want to fetch the department by ID but we have supplied wrong ID to the path then it will give us the error.



We created a class which implements from **Exception** and overridden all the methods of it.



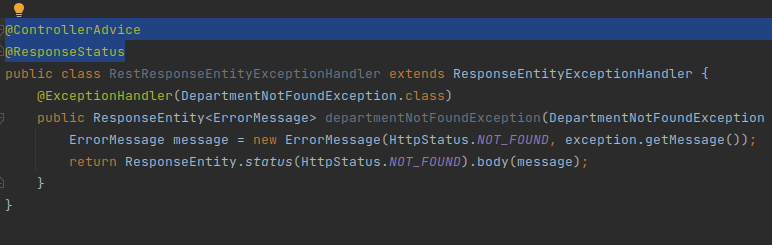
Since we are getting the data from the Service Layer, we need to add the error handling in this file only.

We also have to add the method signature to all other classes like **DepartmentService** interface and **DepartmentController**.

Still we will be getting the long error response from the API but get the required message we specified as well.

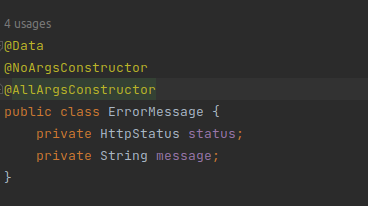
**To Send Meaningful Message as a Response::**

Whenever any exception is thrown in the **DepartmentController** then we need to identify which exception is there and then we can send data as a response for that specific exception which occurred.



This is the class that we are going to use to send the data as the response when an exception is thrown at the controller layer.

**@ControllerAdvice** is used for the classes which are used to handle the exceptions.



What data we need to send as the response, we write it here (**ErrorMessage** class).

We also need to wrap the **ErrorMessage** as the **ResponseEntity** as we are going to send this message as the response.

**Changing H2 -> MySQL::**

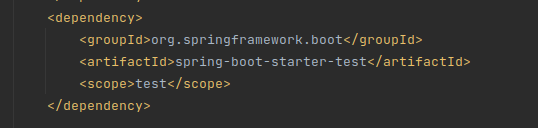
**<Skipping this part. We are going to use H2 file based only.>**

**Testing::**

**Integration testing** where we do the testing of our application end-to-end. Whenever we hit any endpoint, then how that particular endpoint is behaving, we can test it entirely.

We can also do testing for controller, service and repository separately or together (**integration**).

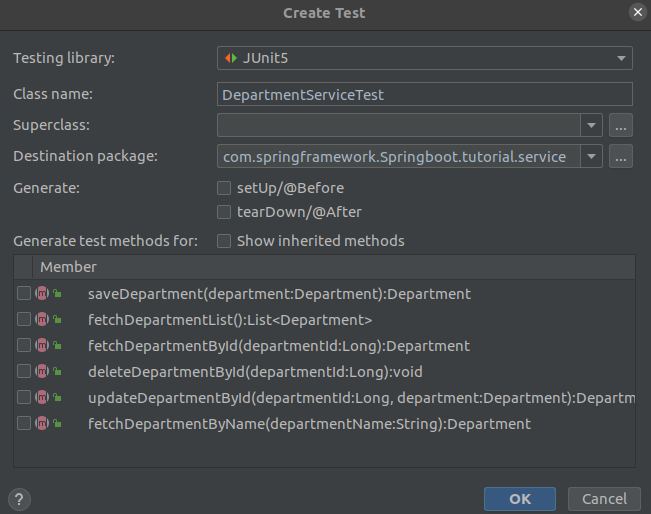
For this we are going to use **JUNIT 5** and **MOCKITO**.

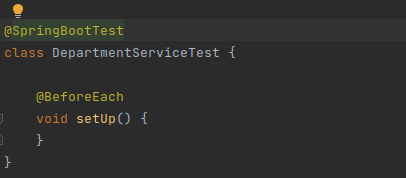


**Mocking::** Let’s say when doing testing of the controller layer, then it contains some calls to the service layer as well. So, the controller layer is dependent on the service layer. But we are testing only the controller layer. In that case, we mock the functionality available in that particular service layer.

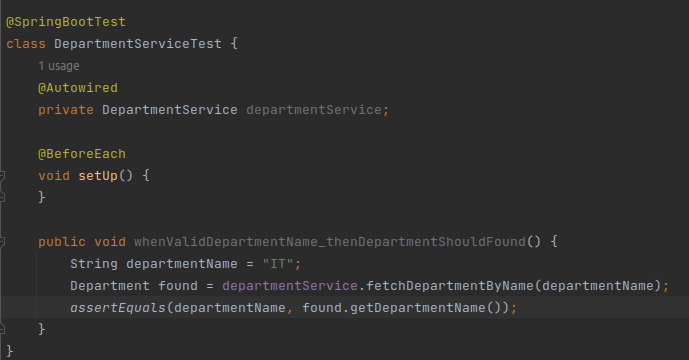
Whenever a call of a particular method is done on the service layer, give me the object that’s it, we don’t have to go to the service layer when testing for the controller layer. This is called **mocking.**

**Unit Testing : Service Layer::**





To tell Spring Boot that it is for testing purpose, we have to use the above annotations.

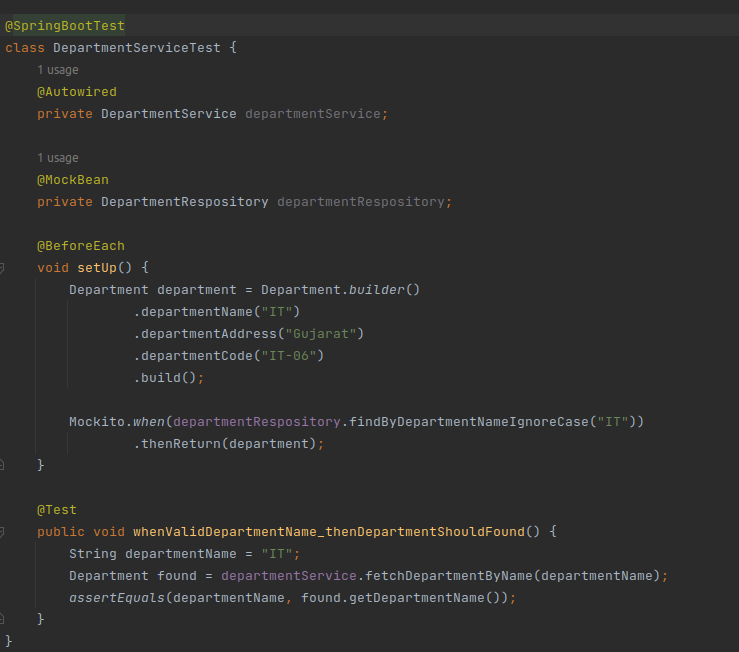


Now we need to cover both positive and negative scenarios.

In the above code, we can see we are calling **fetchDepartmentByName** which in turn calls the method defined inside the repository layer. But here we are doing testing for the service layer only.

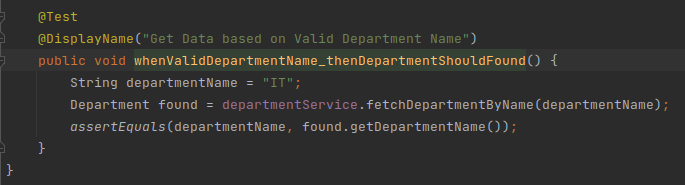
**@BeforeEach** method is called before every test case inside the above class.

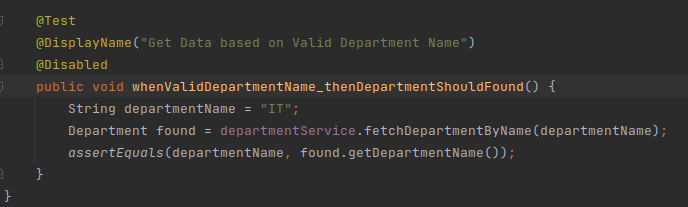
**@BeforeAll** method is called only once before all the test cases inside the above class.



**@MockBean** is used for the methods that we need to mock.

The test case will be passed for us.



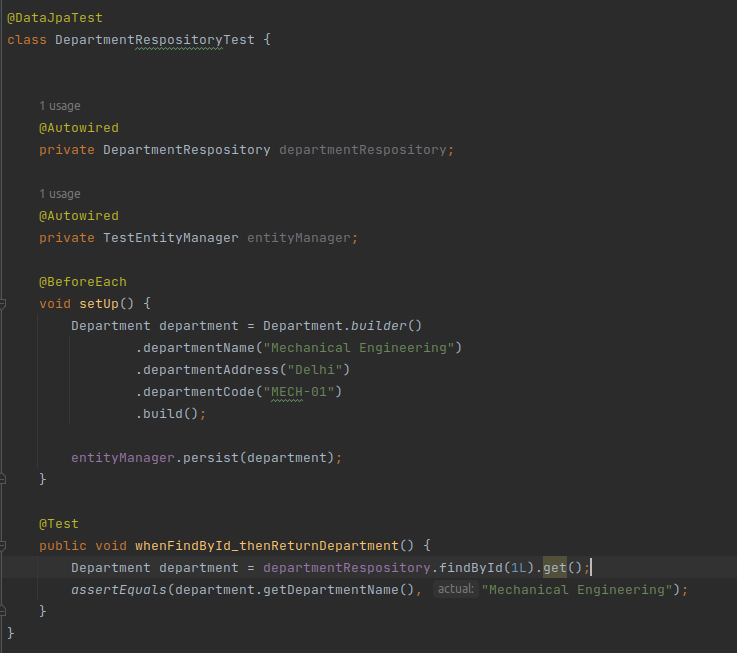


Whenever we are running multiple test cases and we want to disable some out of it then we can use the above approach.

**Unit Testing : Repository Layer::**

When testing a repository layer, we don’t wanna use junk data. Instead we can use JPA test annotation for this.

What it does is that whenever we are trying to persist data from the database, it persists the data in the database at the time of method execution. And when the execution is completed the data is flushed from the database.



**REMEMBER**:: No new records will be formed in the database. This all gets flushed from the database when method execution is completed.

**Unit Testing : Controller Layer::**

Controller layer testing is also different since it is hitting the endpoints and we see how our particular endpoint is behaving.