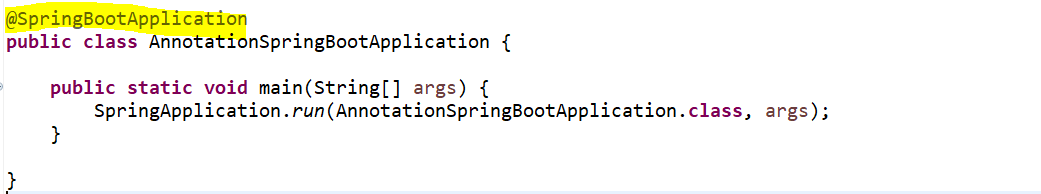
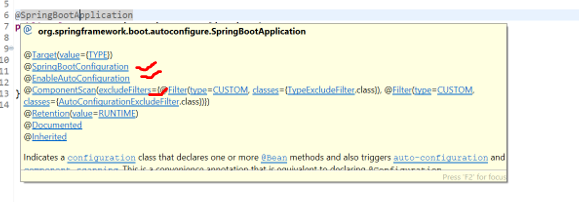
Since we know Spring Boot starts with main method so let’s begin with the main class annotation i.e **@SpringBootApplication**





**@SpringBootApplication**

* @**EnableAutoConfiguration**
* @**ComponentScan**
* @**Configuration**

This is a convenience annotation that combine other annotations like @**EnableAutoConfiguration**, @**ComponentScan** and @**Configuration**. Now let’s understand the role of each annotation.

>> @**EnableAutoConfiguration** : this automatically configure the Spring application based on the jar dependencies that we have added in our pom.xml. for example, if the H2 database jar is present in the class path and we have not configured any Beans related to that database manually then Spring Boot auto configuration feature automatically configured it in our project.

>> @**ComponentScan** : this is just a simple annotation where it will just scan your Bean so that it can be visible to the IOC Container. let’s say you define a different package structure, or you just want to load specific package and class to IOC container then you can use @ComponentScan, and you can provide the Base Packages or Base classes.

>> **@Configuration** : we use this annotation for java base configuration where we can define the Bean definition inside this class so that Spring IOC can load it.

So, these are the 3 annotations Spring or Spring Boot developers combines these 3 annotations in a root annotation called @**SpringBootApplication**.

**Stereotype annotation**

* @**Component**
* @**Service**
* @**RestController/ @Controller**
* @**Repository**

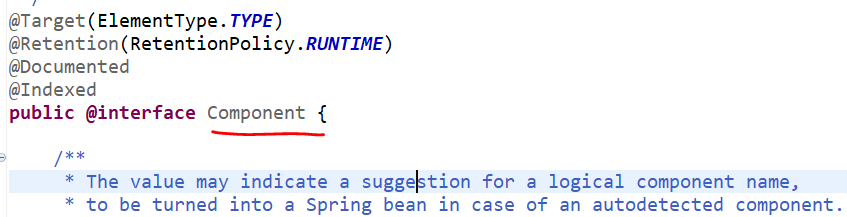
Spring Framework provides some special kind of annotations like @**Component**, @**Service**, @**RestController** and @**Repository**. these are called Stereotype annotations.

Basically, these annotations are used to create Spring Bean automatically in the application context if you annotate any of these annotations on top of your Spring Bean then Spring will scan that Bean and managed its Lifecycles begin from object creation to object destroyed.

@**Component** annotation is the main or base Stereotype annotation you can say and other 3 are derived from the @**Component** annotation.

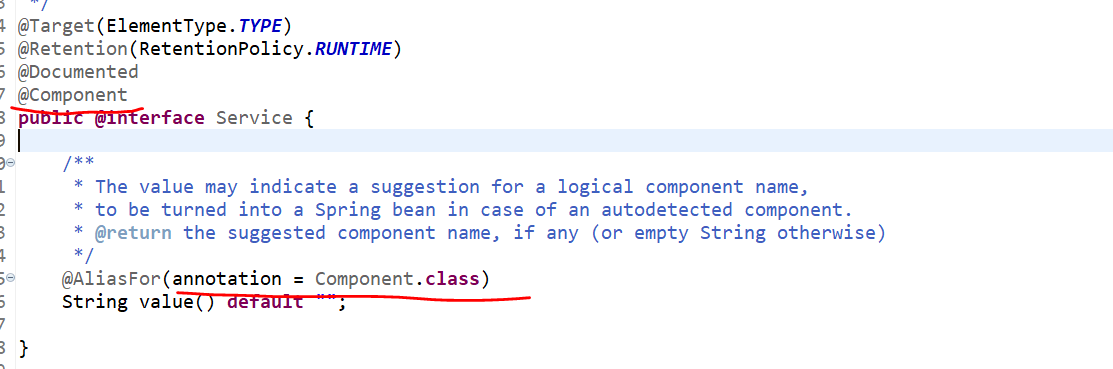
In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled and managed by a Spring IoC container.

As you can see this @**Component** is the root annotation .

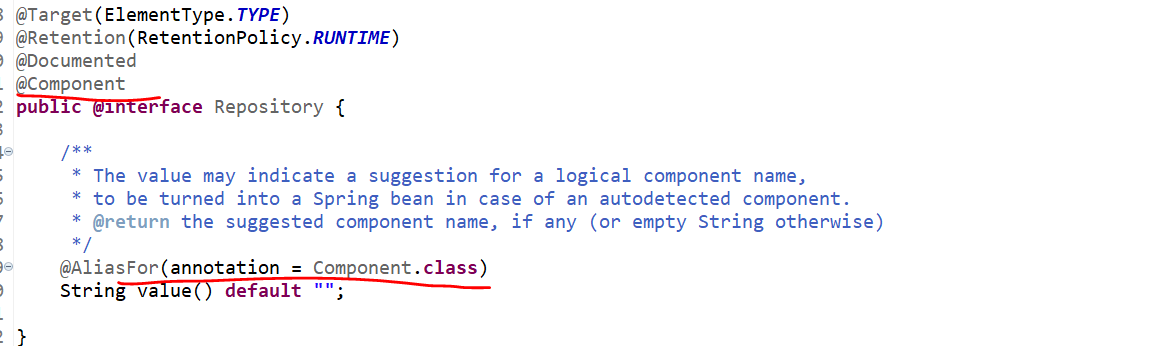


Now let me show you other all Stereotype annotations are derived from @**Component** annotation.

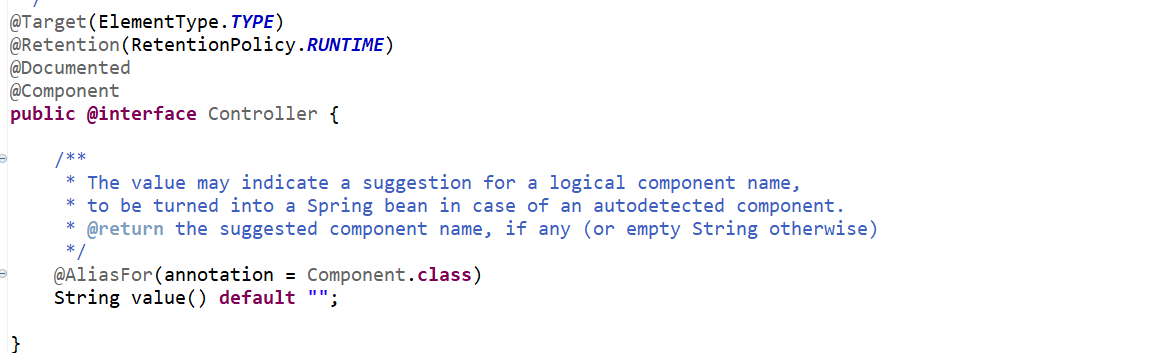
If I open the @**Service,** this is what derived from the @**Component** class.



Similarly, @**Repository** also derived from @**Component** class.



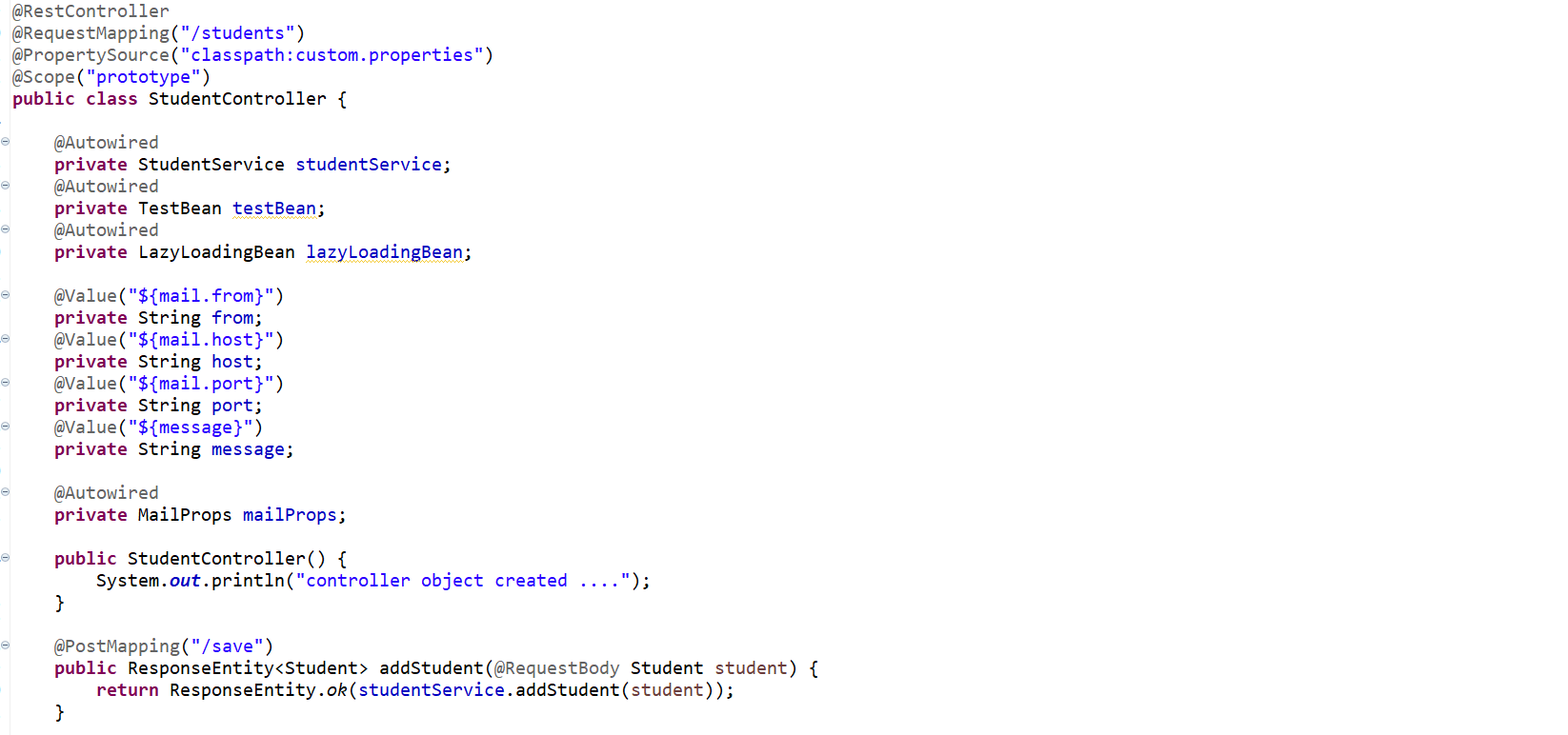
And @**Controller** this is also derived from the @**Component** class…



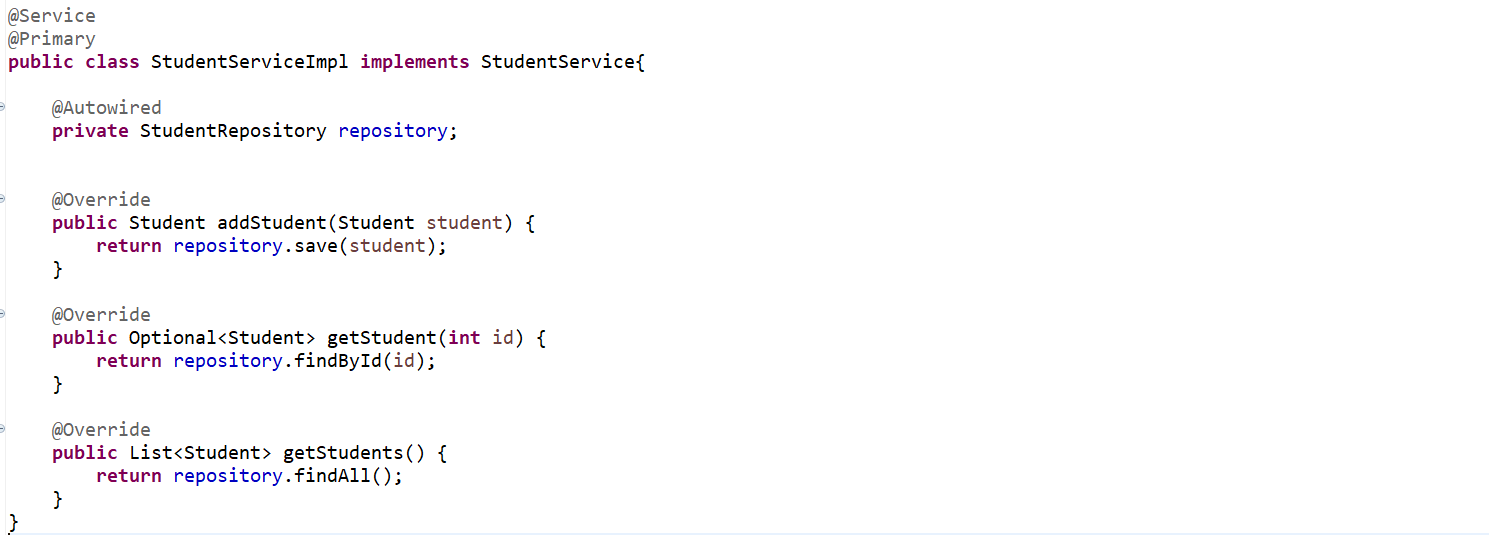
Why there are 4 annotations like @**Component**, @**Service**, @**RestController** and @**Repository** as we know @**Component** is a parent annotation and everywhere we can use it?

**This is absolutely correct we can use it, but the main purpose of the other annotations will tell the Roles of that classes**.

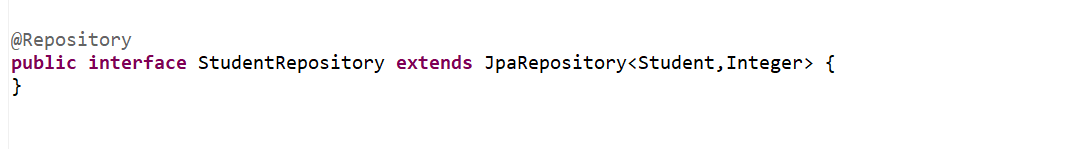
If you annotate @**Controller** someone easily identified that ok this is what the Web Layer where I can expose the Rest API. I defined @**RestController** so that easily someone can identify this is what the class where I can expose my Rest endpoint.



Similarly, if I will go to the Service class by seeing this @**Service** annotation someone can easily identified that ok this is what that class where I need to write my business logic.



Similarly, if I will go to the Repository class by seeing this @**Repository** annotation someone can easily identified that this is where I need to write the database logic.



So that’s why there are 3 different annotations to just defines the role of the classes.

Also, this is best practice to specify meaningful annotations in each layer.

**Spring Core Annotations**

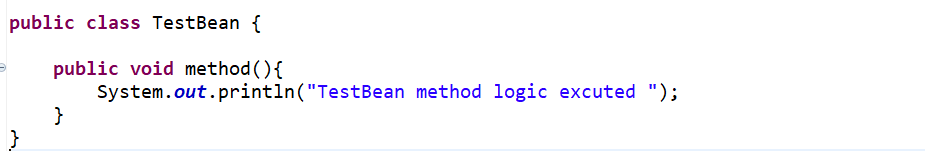
* **@Configuration**
* **@Bean**
* **@Autowired**
* **@Qualifier**
* **@Primary**
* **@Lazy**
* **@Value**
* **@PropertySource**
* **@ConfigurationProperties**
* **@Profile**
* **@Scope**

**==================== @Configuration & @Bean =====================**

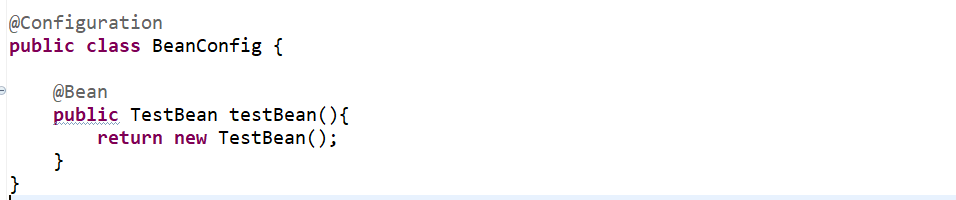
Usually @**Configuration** and @**Bean** these 2 annotations used when u want to use java base configuration when I say java base configuration which means we don’t want to handle the Spring bean lifecycle either using **Xml** or **Annotations**.

So, when you write @**Configuration** it indicates that the class can be used by the Spring IOC container as a source of bean definition.

When we annotate any class with @**Configuration** now Spring IOC will expect from this class to define couple of @**Bean** annotation. Which mean just define couple of beans which will return the object. So, let’s create a **TestBean** with a random method just print something.

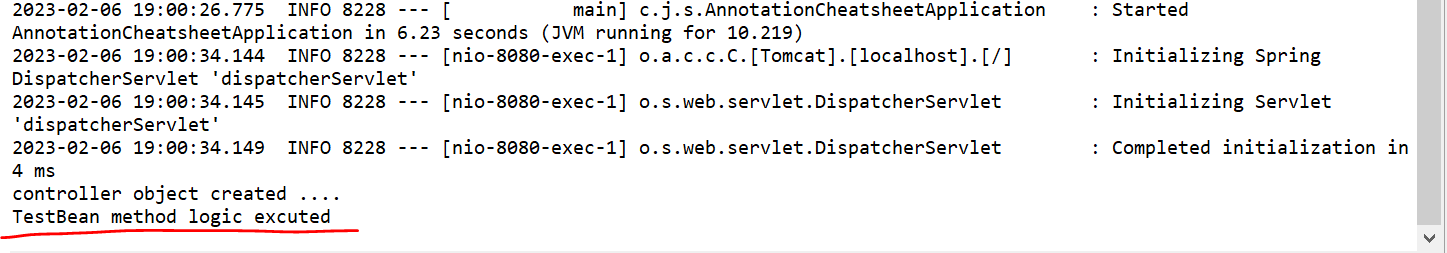


Now how can I define this as a Spring Bean there is 2 ways…I can use the XML otherwise simply I can annotate this with @**Component** . but I don’t want to do that I just want to define this Bean definition inside my @**Configuration** Config class. How can we define simply just create a object of it and annotate that method with @**Bean**.



So, here when you start your application Spring **IOC** will search an annotation with @**Configuration** then we search for an annotation @**Bean** inside the @**Configuration** classes. Once it will find @**Bean** then immediately Spring **IOC** will create an object of it and then maintain the lifecycle by himself. So let me inject this in our Controller and let’s hit the rest endpoint.





So, I can see when I call the rest-endpoint **/all** its showing TestBean method logic executed.

Now we didn’t create the object for **TestBean** Its just created by Spring IOC by reading this Bean class. If u can remember we usually creates the **Kafka Configuration** and **RestTemplate** **Configuration** inside @**Configuration** class. This is how we can define a Bean inside Configuration class.

**Application : annotation-configuration-app**

<http://localhost:8080/students/all>

**[**

**{**

**"id": 101,**

**"name": "Kaushal",**

**"rollNo": 14,**

**"dept": "Science"**

**},**

**{**

**"id": 102,**

**"name": "Santosh",**

**"rollNo": 48,**

**"dept": "Arts"**

**},**

**{**

**"id": 103,**

**"name": "Rajesh",**

**"rollNo": 16,**

**"dept": "Commerce"**

**},**

**{**

**"id": 104,**

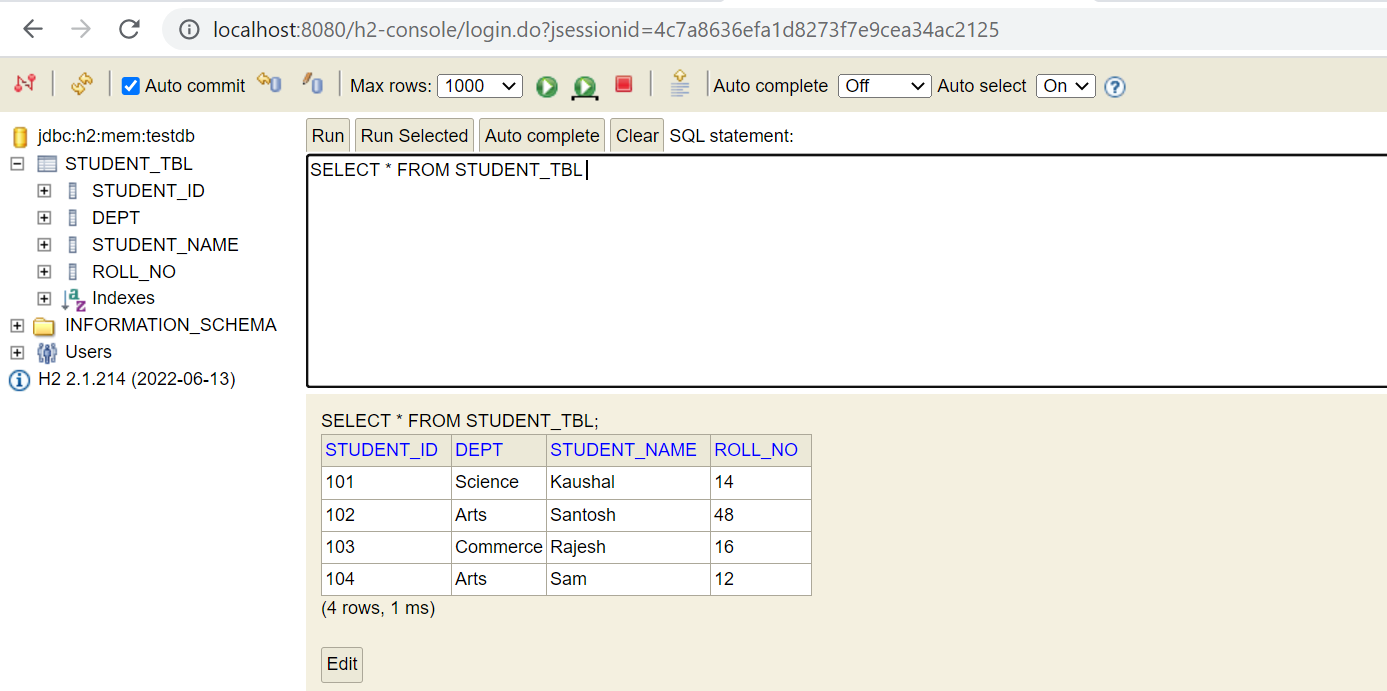
**"name": "Sam",**

**"rollNo": 12,**

**"dept": "Arts"**

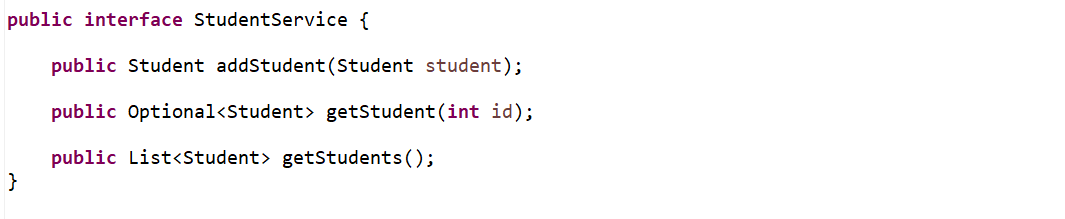
**}**

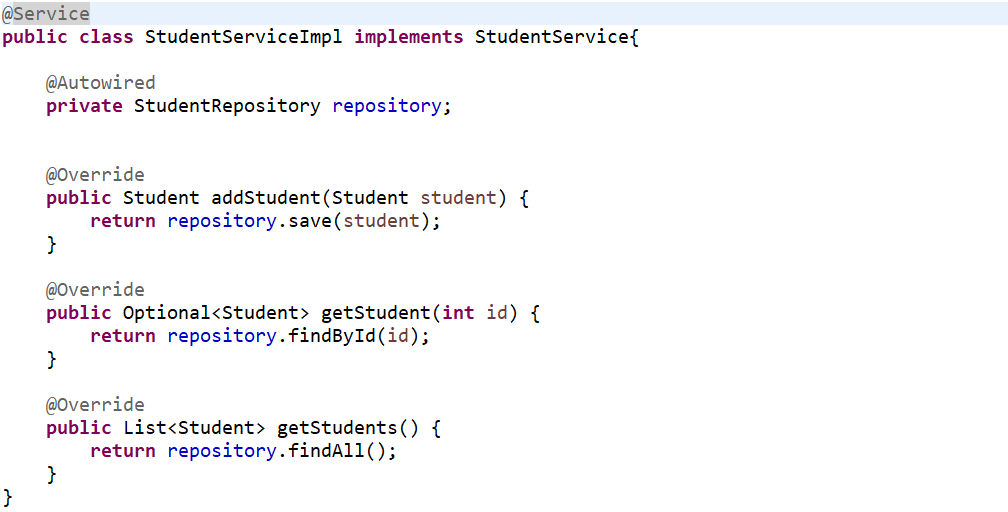
**]**



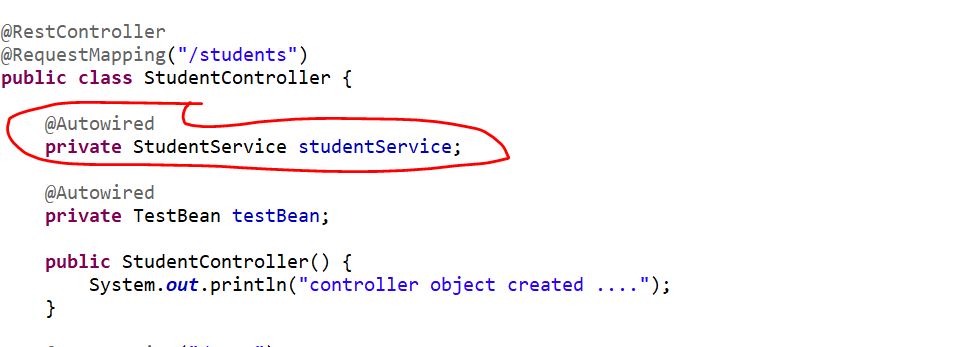
**================== @Autowired, @Quilifier, @Primary =================**

If I will go to the any class let me go to the Controller class. So, if you observe inside this controller class, I want to use the object of **StudentService** class. If you observe this StudentService is an Interface. And it’s having the implementation **StudentServiceImpl**.



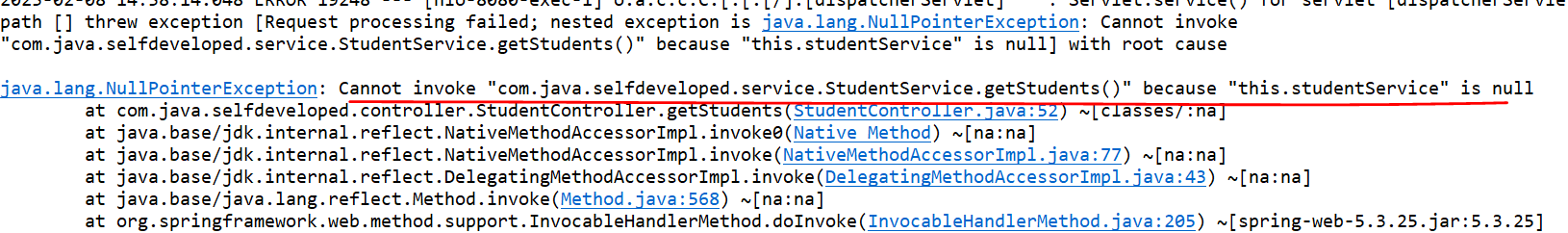


So, this object I want to use in my Controller class …



That is where I just want to tell the spring boot, I just Inject using @**Autowired** you just create the object of this class and give it to me so that I can use it. To show you let me just remove @**Autowired** so if I removed it spring will not be able to create the object of it bcz we are not creating object manually.

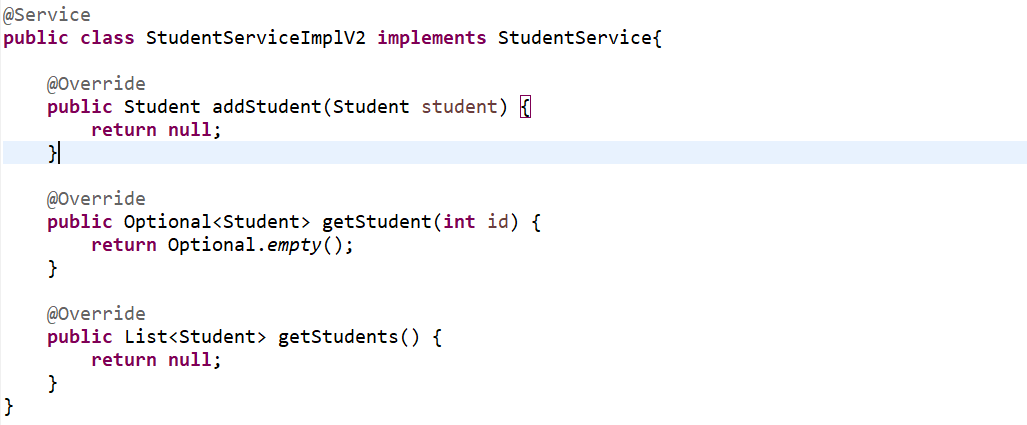
<http://localhost:8080/students/all>



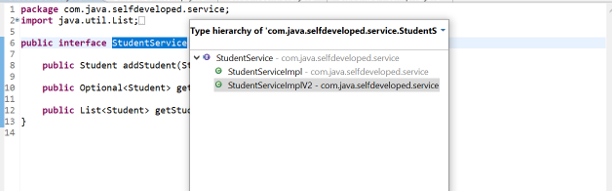
I am getting a **NullPointerException.**

So, you just need to tell the Spring Framework, or Spring Just Inject this Bean I used @**Autowired** annotation. You just inject this object in my class wherever I want to use it.

if you observe there is a challenge so, in **StudentController** I just injected an Interface called **StudentService** and if you observed there is only one implementation class. Now just assume in future I want to define another V2 version of this implementation class. Let’s say StudentServiceImplV2 class.

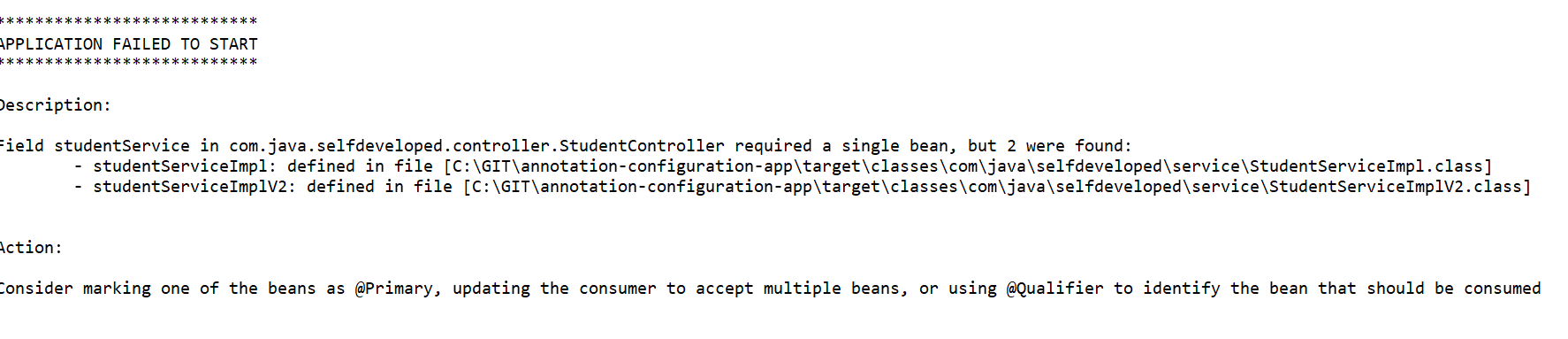


Now for this StudentService I have 2 implementation class. StudentServiceImpl v1 version and StudentServiceImplV2 with the v2 version along with some enhancement. Now if I run this application it should work. let me run and see.



We got an Exception:-

Field studentService in com.java.selfdeveloped.controller.StudentController required a single bean, but 2 were found.



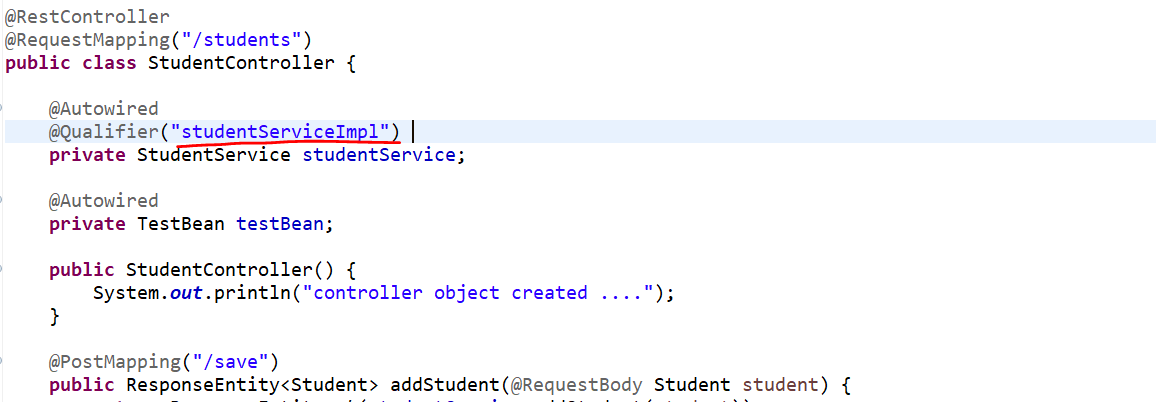
This controller class expecting one Bean, but it found 2 Bean class 1. StudentServiceImpl and 2. StudentServiceImplV2.

**To Solve this:**

Consider marking one of the beans as @**Primary**, updating the consumer to accept multiple beans, or using @**Qualifier** to identify the bean that should be consumed.

So, you need to tell the spring boot I have an interface and that interface having the 2 implementations now you need to tell the Spring which exact implementation you want to load here whether StudentServiceImpl or StudentServiceImplV2 you need to tell that Spring Boot. How you can tell it you can tell using **Qualifier**

**@Qualifier**: here you can tell to that Spring Boot please load the Bean the name which I am giving. So, for now if I will give **studentServiceImpl** . you just need to provide the Alias name of it to **Qualifier**.

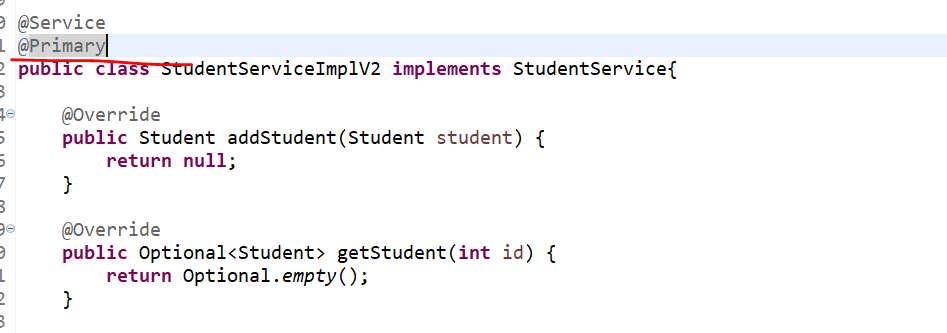


So, I am telling the Spring Boot for this particular interface there is 2 implementations just ignore it. Please load that particular Bean what I am specifying in **Qualifier**.

If I run there is no exception, and I am able to access the rest api.

**@Primary:** now if you observed for this particular StudentService there are 2 qualified Beans. If you don’t want to use @**Qualifier** and always you want to give a priority to a Single Bean either StudentServiceImpl or StudentServiceImplV2. Whatever the scenarios I just want to load this particular bean I don’t know anything then u can simply use **@Primary**.

So just go to that particular class which Bean you want to give a priority. Just annotate here @**Primary**. Let me remove @Qualifier bcz I want to give a priority to a specific bean.



Let’s run and hit api call.

If I run it, I won’t get any response and there is no error.

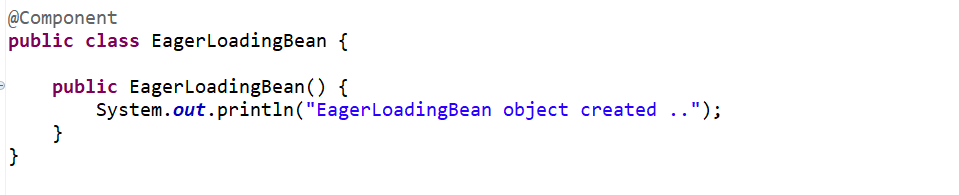
**Best Use-Case for @Primary**

When you have multiple Data Source or database configuration Bean in your application you can give a Primary to a specific connection either in **MYSQL** or in **MongoDB** you can define a DB connection and you can give a @**Primary**.

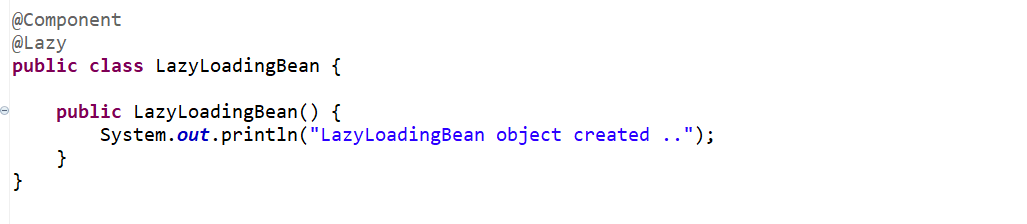
**============================== @Lazy ============================**

**@Lazy :** As you know by default Spring Beans are Eager Loading. I mean if u define Bean in your application, then Spring IOC will create By Default its Object. However, it does not matter whether you are using that Bean or not. Which really increase the Heap now how we can avoid that. So, there you can use @**Lazy** annotation. We can tell the Spring /Spring Boot hey just create a Bean or Load a Beans on demand only otw don’t do anything.

Let’s see example for both. Since these 2 are the Beans so either we can define a xml or annotation, so, let’s use @**Component**.

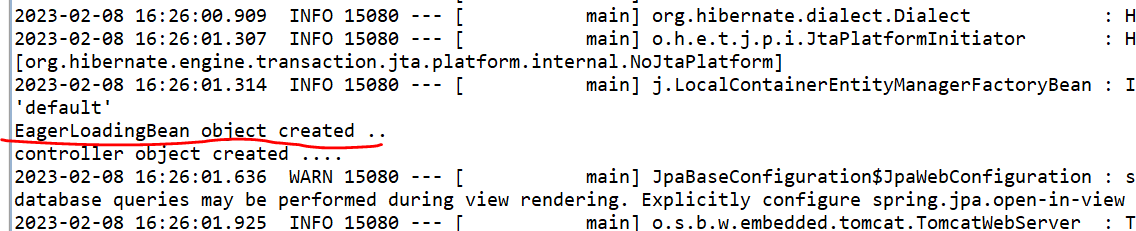


------------------------------------------------------------------------------------



If I will define @**Component** by default this object will be created.

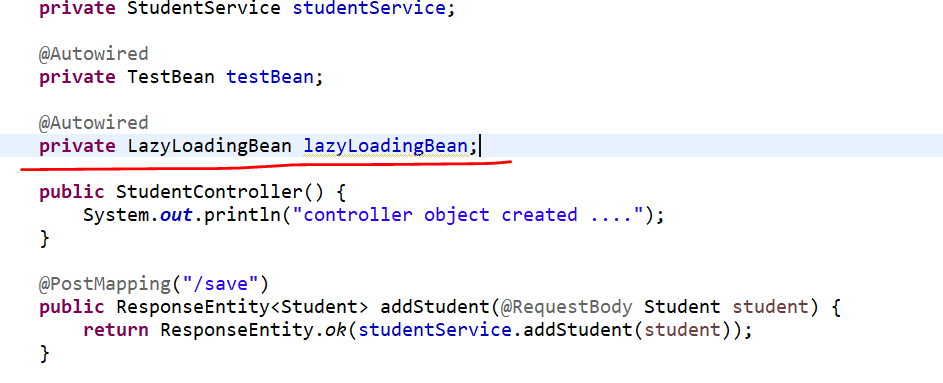
Since I have been defined both Bean @**Component** for LazyLoading and EagerLoading both objects will be created by Spring IOC. Now LazyLoading I want to load it lazily that means whenever I want it, I can tell Spring to Load it. So, I annotated @**Lazy** let’s run and see which object Spring has created.



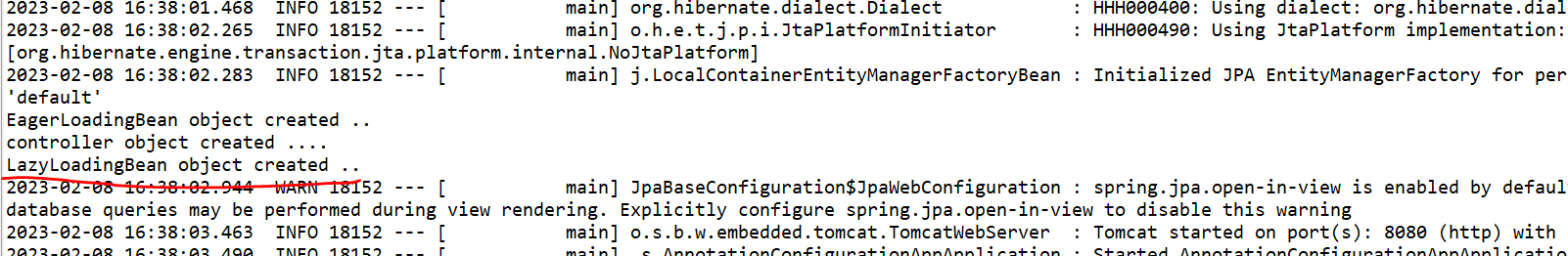
Now if u observed its only created an Eager Loading Bean.

Now it’s proved that Spring boot is Eager Loading. So, if u don’t want to use Eager Loading you just want to load a Bean wherever you need it then you can define @**Lazy**.

Now when I Inject this Bean or use this bean then only Spring will create an Object of it.



As I have injected this Bean in Controller so Spring will create an object bcz he thinks someone is using this bean so let’s create an object for it.

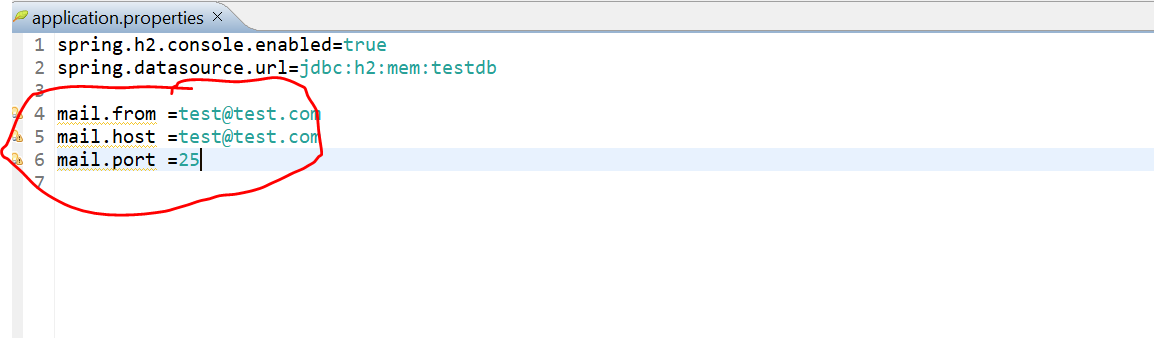


So, once we are using lazy Loading Bean then only Spring will create an object. Bcz of @Lazy Annotation.

**=============@Value, @PropertySource, @ConfigurationProperties=========**

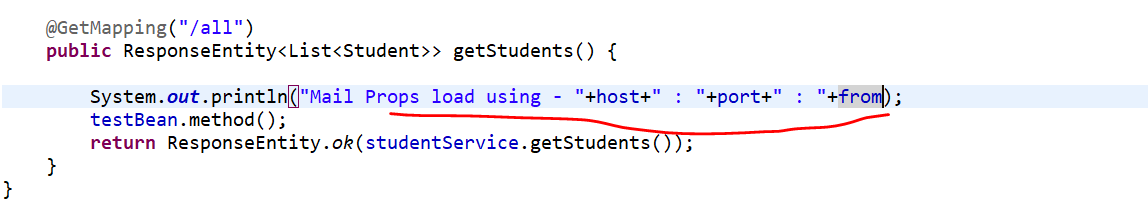
Whenever you want to load some metadata from properties file then you can use @**Value** annotation.

For example, let me define some key values in our **application.properties** file

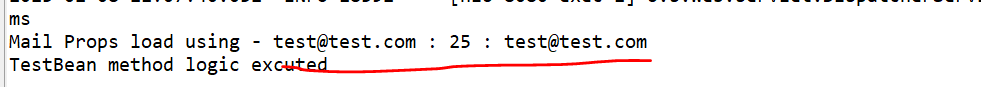


I just want to load these key values for mail. I just want to build an API which will trigger a mail using Java Mail Sender. So, for that I just need these 3 fields from, host and port. So, I can do that using @**Value** annotation.



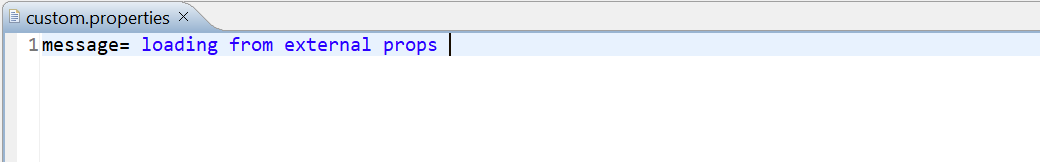


As we can see console

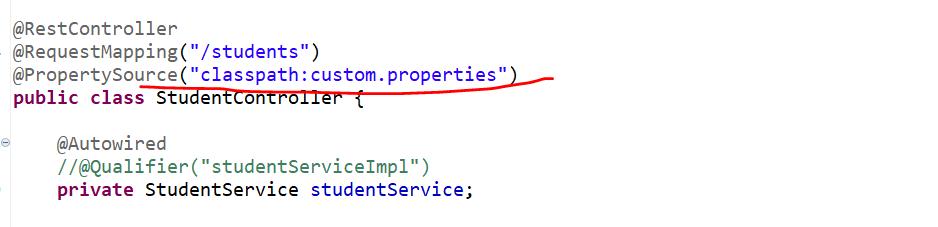


Now you can see we didn’t give the name of our properties file to the spring boot like from which properties file need to load the key by default spring boot loading it form the **application.properties** file.

But if u want to define your custom properties file like **custom.properties** then Spring Boot won’t load these particular properties.

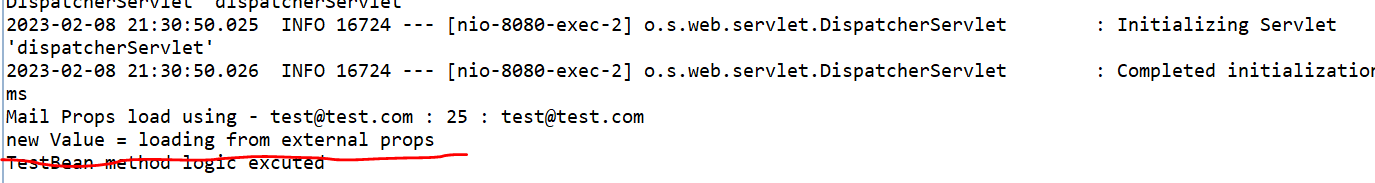


To show you that if you are not using an **application.properties** file how u can tell the spring boot that load this custom properties. So, in our Controller I am telling Spring boot load the properties file which I am giving using @**PropertySource.**



And then we will just load the value from external properties file.





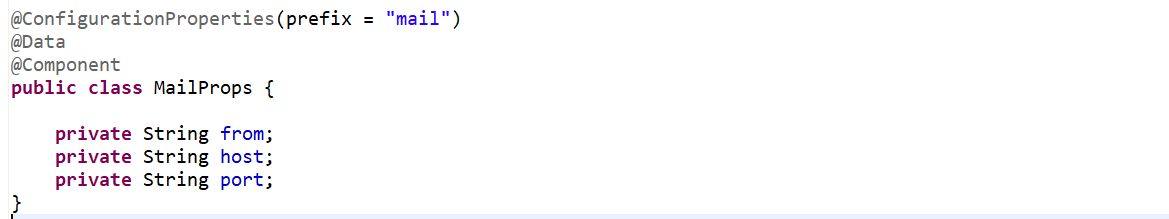
**@ConfigurationProperties (Interesting)**

Let’s say you have bunch of keys and values in your application.properties file or your custom properties file. So rather than load it one by one. Let’s say u have data source properties file with driver name, username, password, and URL like all the n number of fields you have, and you want to bind them a single DTO class or Bean class and you want to map those keys to that particular DTO class. As of now let’s try for this mail.from, mail.host, mail.port.

Let’s annotate this DTO with **@ConfigurationProperties(prefix = “”)** and inside this params you need to provide the prefix. Now if you see in application.properties file the keys are mail.from, mail.to, mail.port… so the unique common match is **mail** only. So, I can consider this as a prefix.

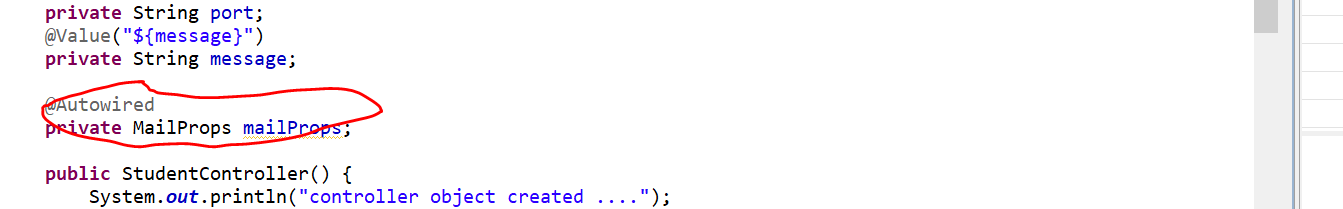


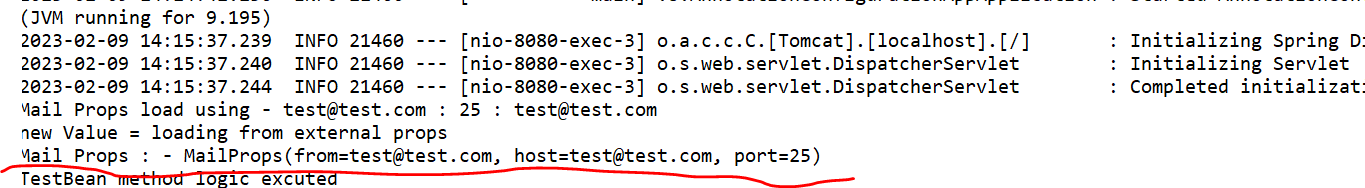
Now I don’t want to load below property values using @**Value** or manually we want to load using @**ConfigurationProperties**.



This is really important and interesting bcz I used it many times. So, once you understand it completely u will must use it whenever you want to load keys or data values from an external property. To tell the Spring that this is my Bean u can annotate with @**Component**. since I used Lombok so I can use @**Data** otw u can use Getter and Setter. So now from prefix it will load from, host and port.

Now to see how to use just go to the controller Inject the MailProps and just print the object and see.





**Now you can see all you keys Map to your DTO class which is MailProps.**

**=========================== @Profile ===============================**

**@Profile(Very Useful)**

This is very useful I believe everyone must be used in their projects. We usually used this annotation whenever we want to load some configuration specific to the environment.

Let’s take an example we are using database in our project and each environment specific data source properties or database connection properties are different.

For development driver name, URL, username, and password are different like SYS, UAT(Stage) and PROD.

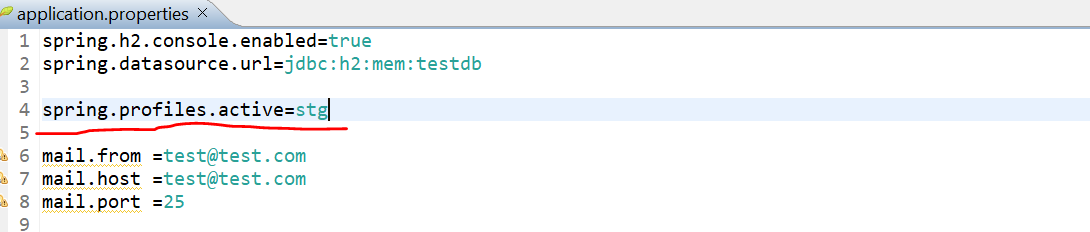
As a developer when I write and test my code, I should use development environment specific database connection and when you want to debug some bug, we need to use Stage database connection. Now how we can switch from one environment to another that is where @**Profile** annotation came into the picture.

Now if you go to resources folder you can see **application-dev.properties**, **application-prod.properties, application-stg.properties** so, here I defined 3 different properties specific to the environments.

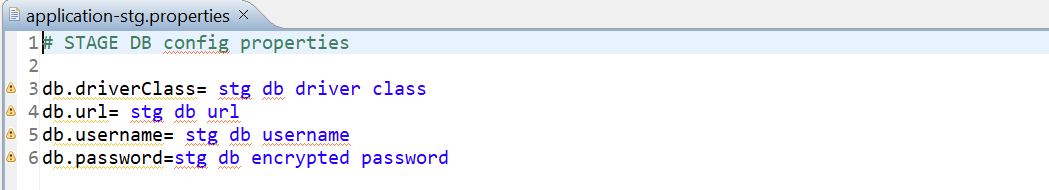
**UseCase:-**

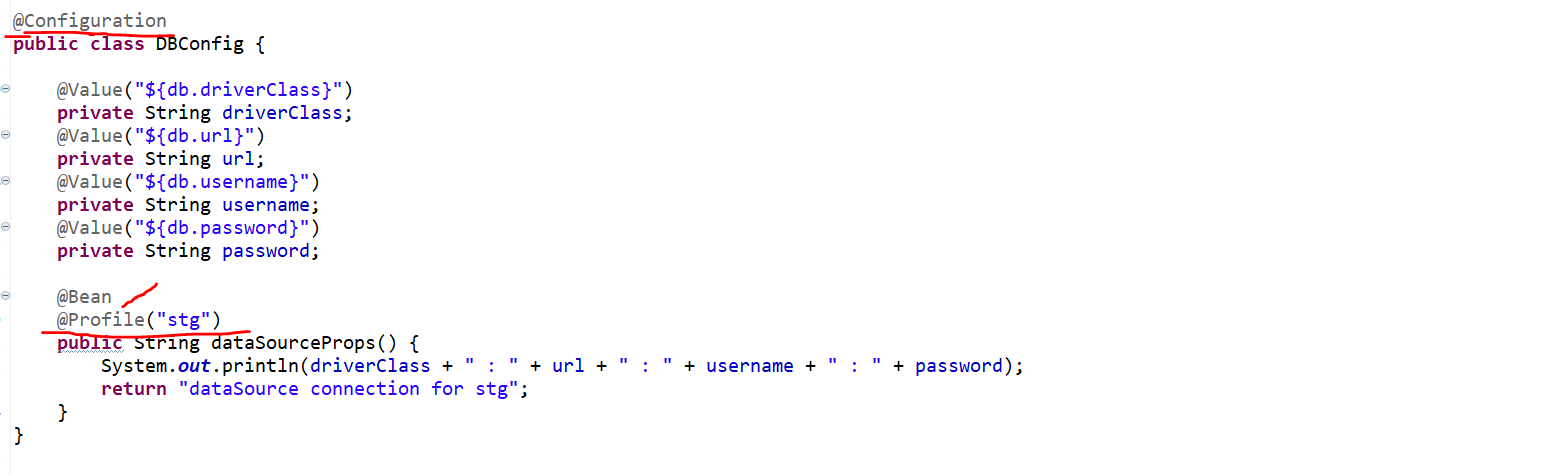
Now I am just debugging the code one of the bugs I found, and I just want to debug it in lower environment in stage environment. Then how I can tell the spring boot that there is a 3 environment specific properties dev, stage, and prod. So, just load the stage for me so that I can continue the testing. That is where we can use @**Profile** annotation. Also, you need to specify the active profile in application.properties file like **spring.profile.active = stg**

**Now if we go to Config package, I have created a class called DBConfig.**

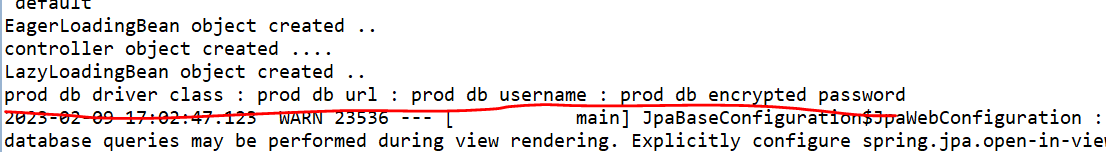


-------------------------------------------------------------------------------------------------------------





Now let’s check for prod



**=============================@Scope===============================**

**@Scope:** This is the last annotation of Spring Core related annotations. This particular annotation indicates the Scope of Beans such as **Singleton**, **Prototype**, **Session,** or **Request**.

So, how can we use @**Scope**. So just go to that Bean where you want to specify the scope of that Bean.so

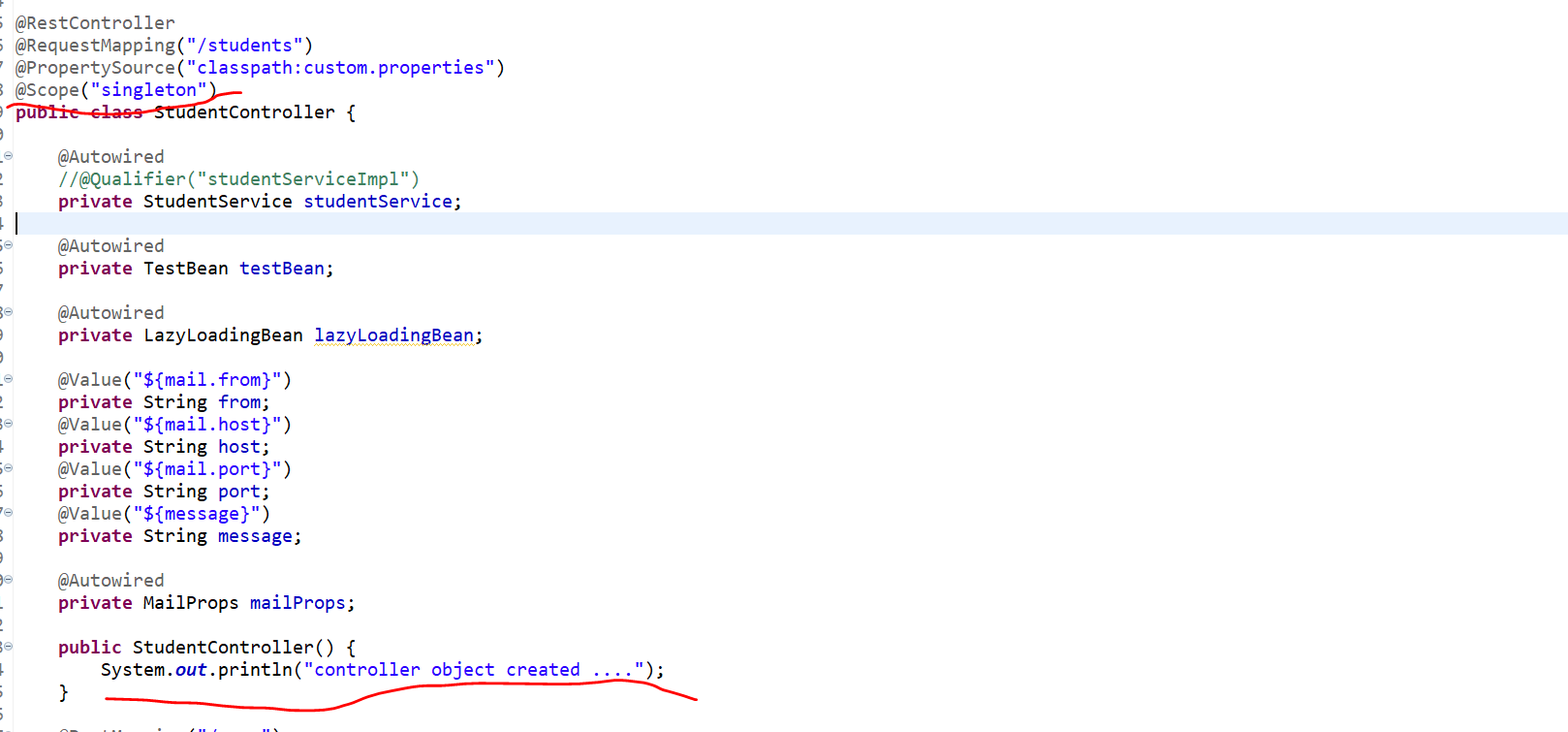
* if you define a Scope Bean as a Singleton then it will just create a single object for our Application Context.
* if you specify a Scope Bean as a Prototype based on the number of request or based on the how many times you will use that instance it will create a different-different object.

**@Singleton**

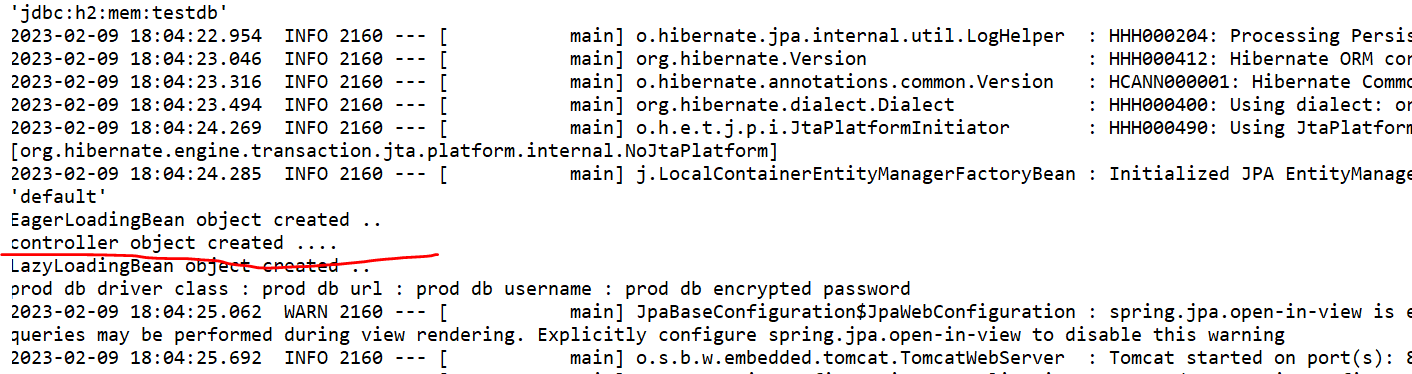
To prove that let’s go to Controller…

Let’s do first for Singleton. And let’s create a constructor of it so I will see that either object is getting created or not.

It will create only one object as many as request we are going to send it doesn’t matter but it will create only one object.



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

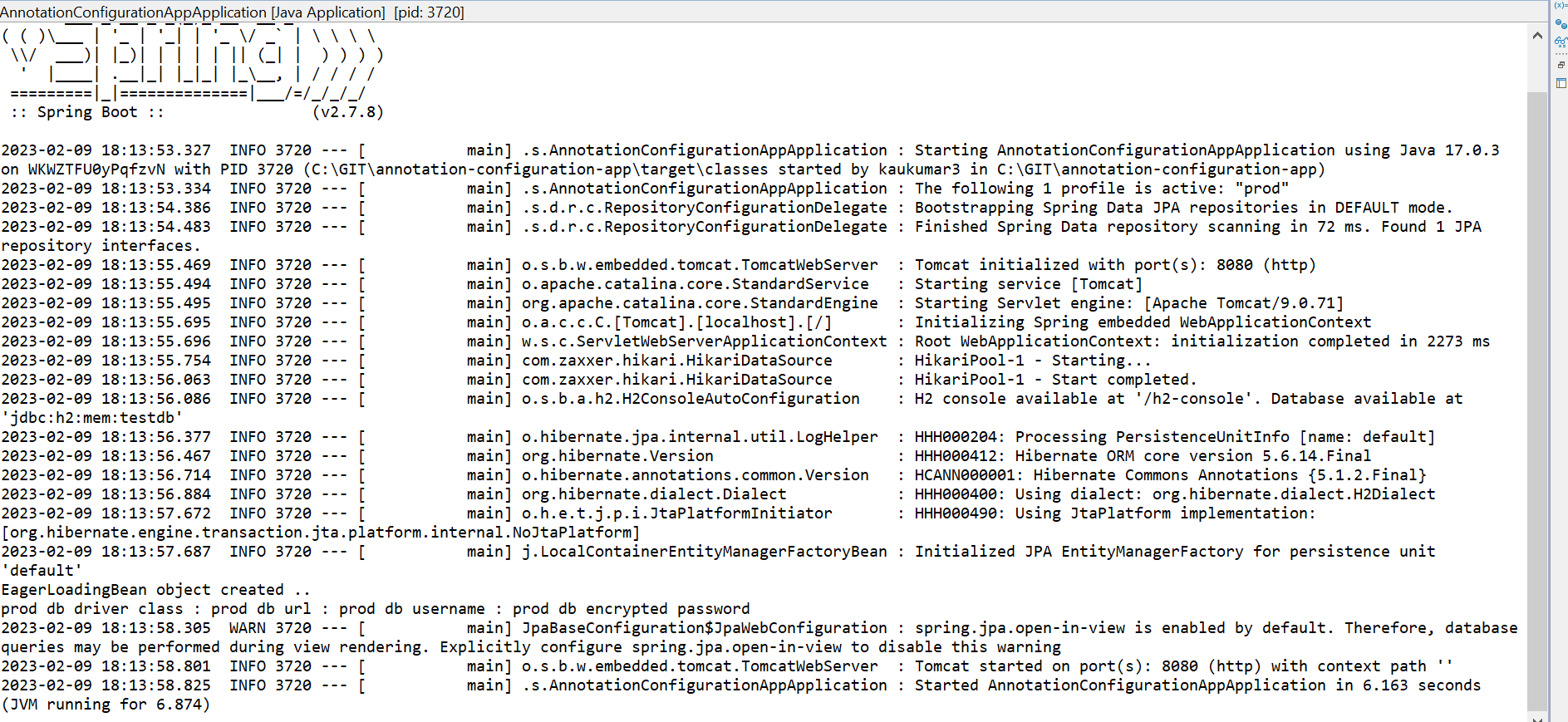


If we clear the console and send multiple controller request, we can’t see that statement. Which means object will create once and then object will be store in an application context and that is what the spring will re-used it.

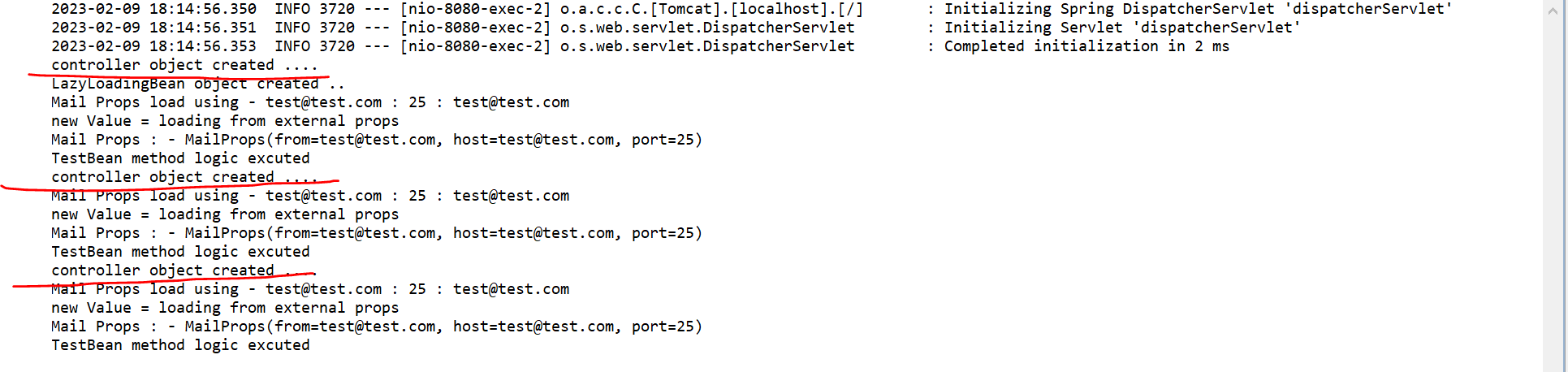


**@Prototype**

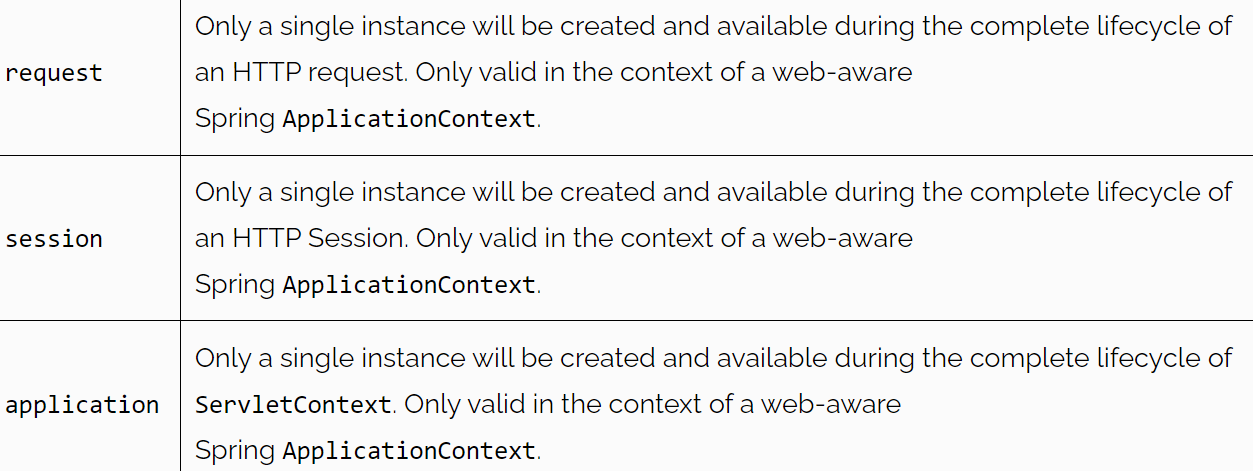
now let’s define the scope as a Prototype. I will just re-start my application



If you observed here, there are no object created yet. Now let me hit the endpoint 3 times.



So, as many times I hit the request controller objects got created. That is what the difference between **singleton** and **prototype**.



**REST API Related Annotations**

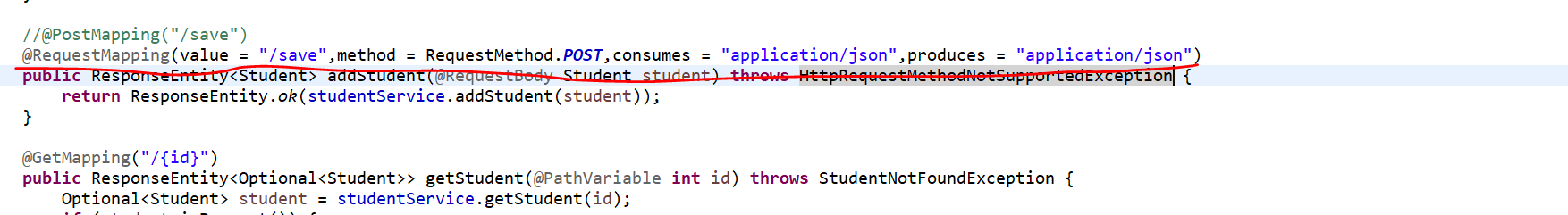
* **@RestController**
* **@RequestMapping**
* **@GetMapping**
* **@PostMapping**
* **@PutMapping**
* **@DeleteMapping**
* **@RequestBody**
* **@PathVariable**
* **@RequestParam**
* **@ControllerAdvice**
* **@ExceptionHandler**

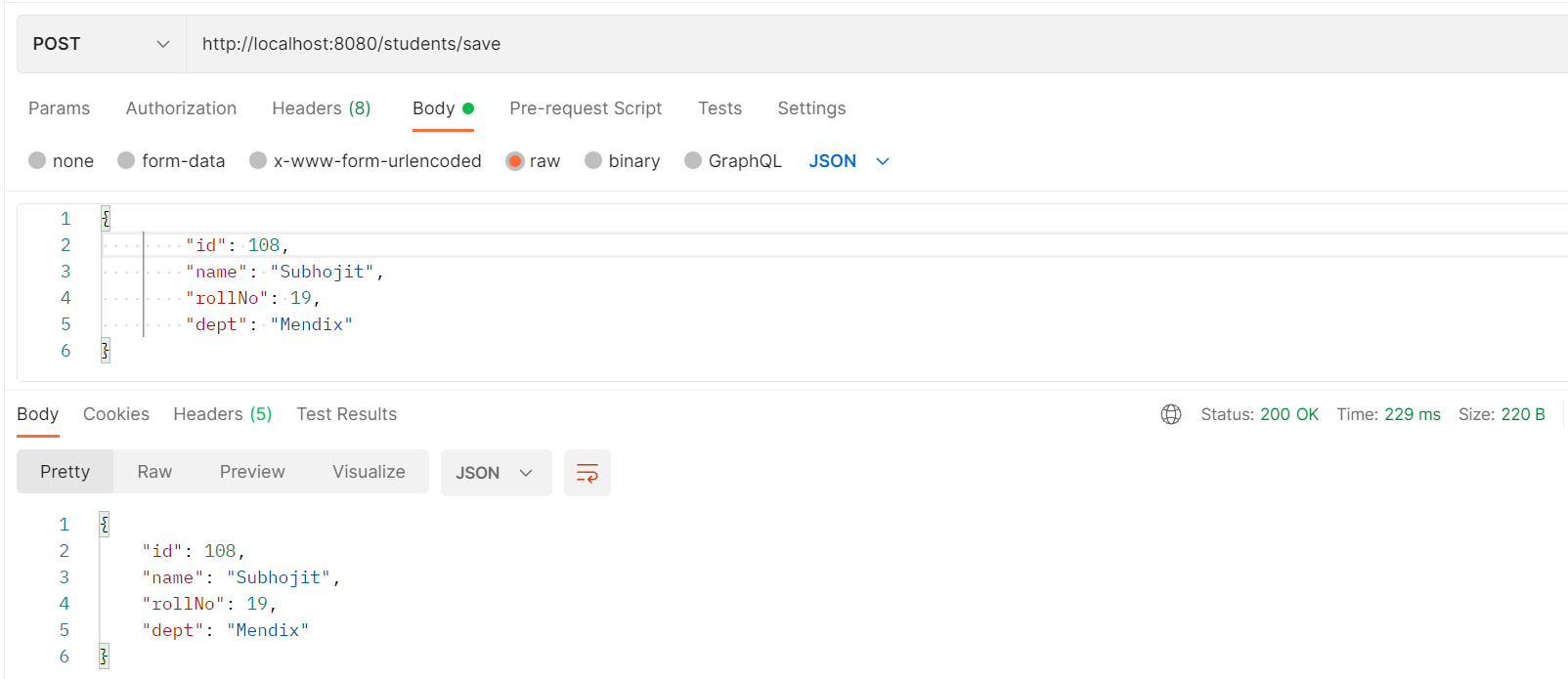
When you say REST API first annotation which will come into our mind is @**RestController** with this annotation we define that this is the class where we exposed our rest endpoints, and this is one of Stereotype annotation which we already discussed.

===**@RequestMapping**,( @**GetMapping**, @**PostMapping**, @**PutMapping** and @**DeleteMapping)===**

**@RequestMapping**

Now the next annotation is @**RequestMapping** this is quite old annotation which introduced to defined HTTP methods. Using this annotation, you can define what HTTP methods you want to expose your endpoints. For example

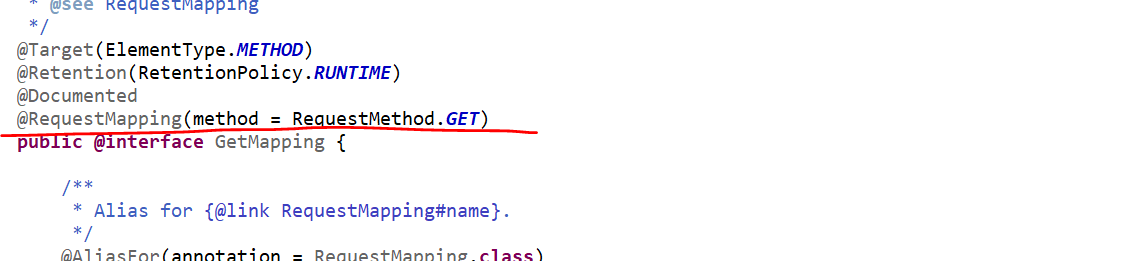




But usually in latest Spring Version we are not using this annotation bcz spring Developers provide separate annotations for each HTTP methods.

Like @**GetMapping**, @**PostMapping**, @**PutMapping** and @**DeleteMapping.**

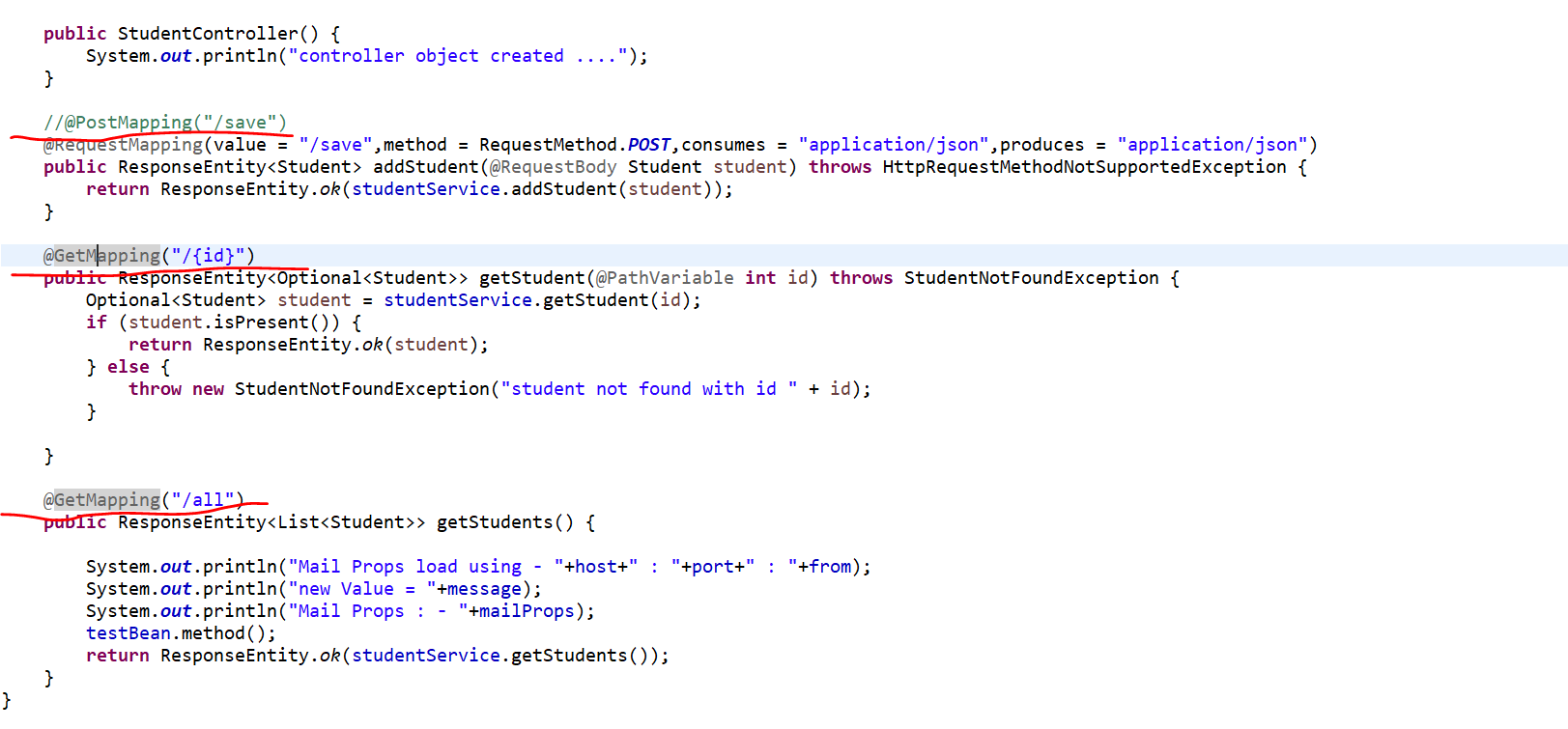
Now if you observed these all annotation also derived from @**RequestMapping**.



Now why there are 4 annotations?

Each annotations defines specific roles here.

* If you want to Fetch something or retrieve something, then you can use @**GetMapping**.
* If you want to save data or want to perform create operation to DB, then you can use @**PostMapping**.
* If you want to perform update operation to DB, then you can use @**PutMapping**.
* If you want to delete something , then you can use @**DeleteMapping**.

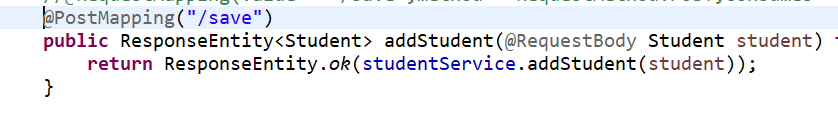


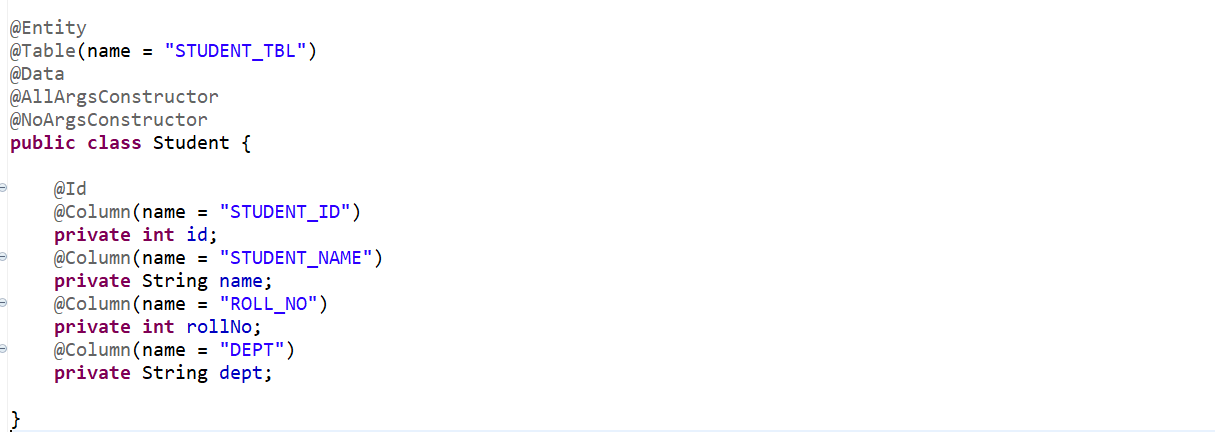
**==============@RequestBody, @PathVariable, @RequestParam==============**

**@RequestBody**

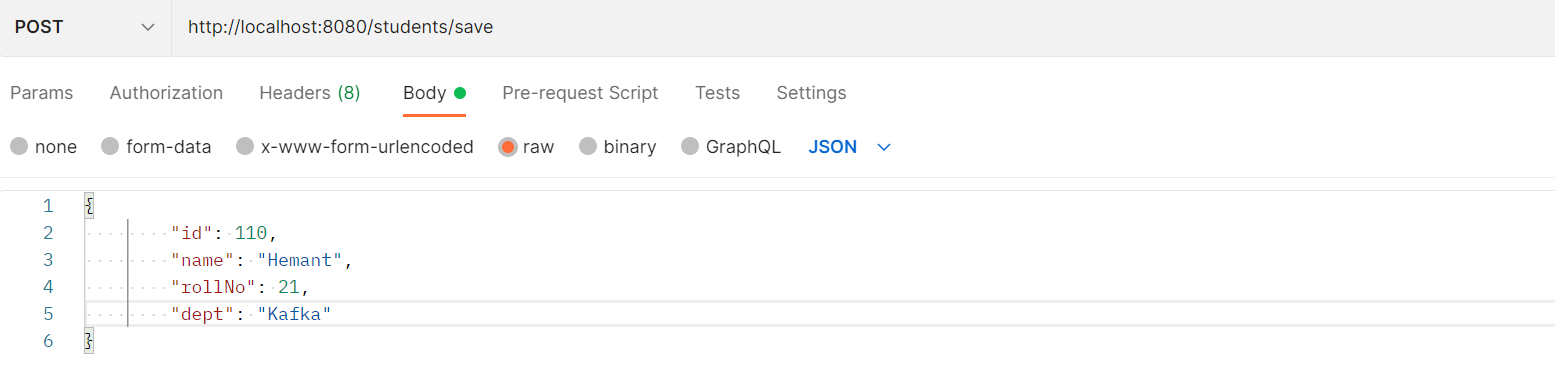
**This annotation we used to pass the Payload.**

If you observed, I have just added @**RequestMapping** annotation in POST method argument. So, here I want to perform the POST operation means I want to save this Student object to our DB.

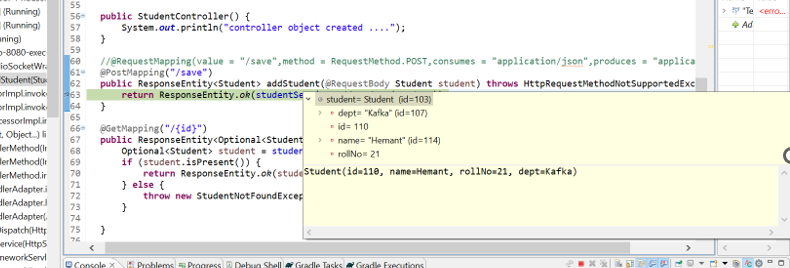




So, that is what the code I write in the particular endpoint. Now how can I access this endpoint. I just need to define this URL and then I just need to pass this student json as a Request Body from the Postman.



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

\

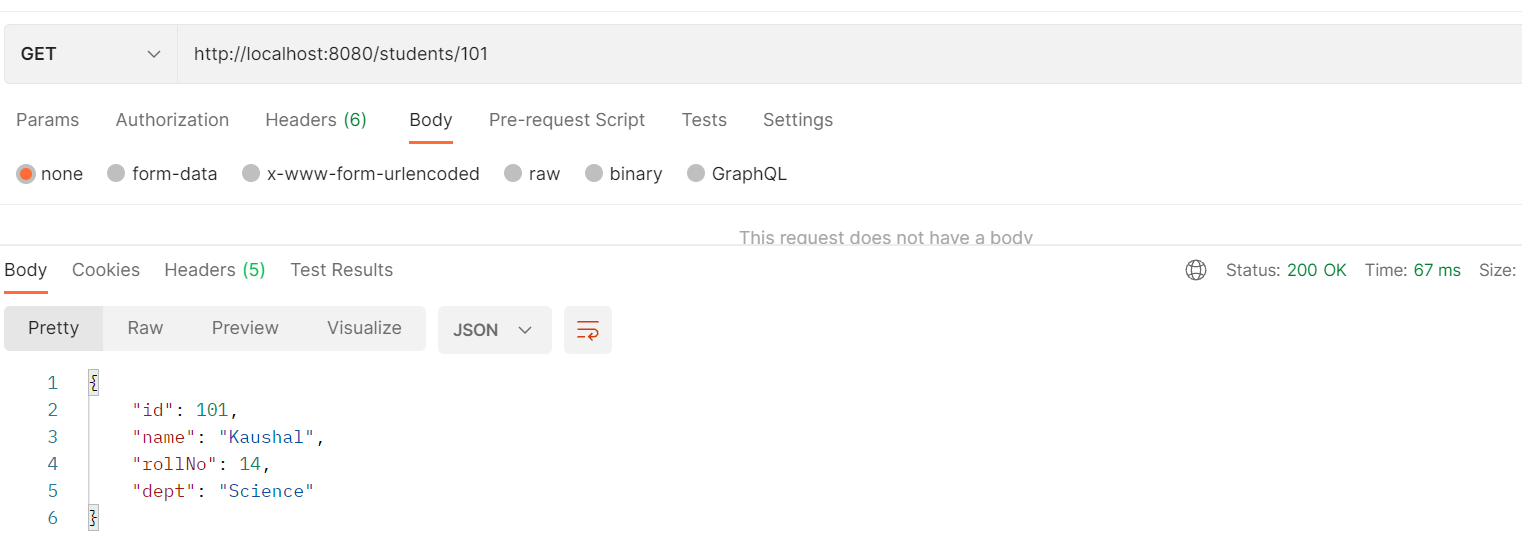
Now the question here when I pass a Json Object through Postman then how does it mapped to the Student object which is there in my method argument and then that Student Object is going to save in the DB.

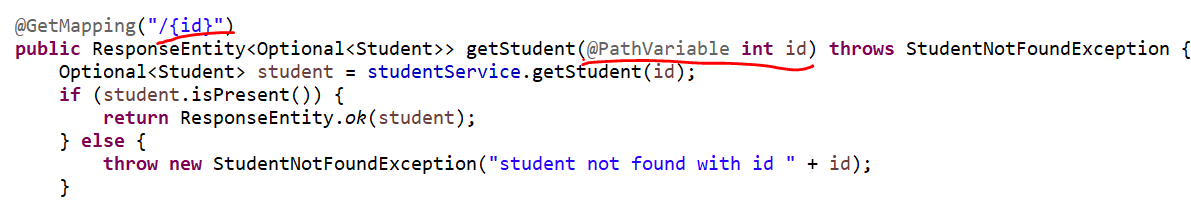
Now once I pass the json object now if u observed whatever the json I passed here that mapped to this particular Student object. The Json getting De-Serialized to this Student Object. That is where this @**RequestBody** help us. **It will automatically de-serialized the Body of HTTP Request into the DTO or Model whatever we defined here.**

**@PathVariable**

@**RequestBody** annotation we used to pass the Payload but there may be a possible scenario that you need to pass the input as part of Request URL itself. That is the case you can use @**PathVariable**. Now if u observed I just want to get the student based on id and this id I will pass as part of request URL itself. So, give the id of students and fetch the Student detail from the DB.

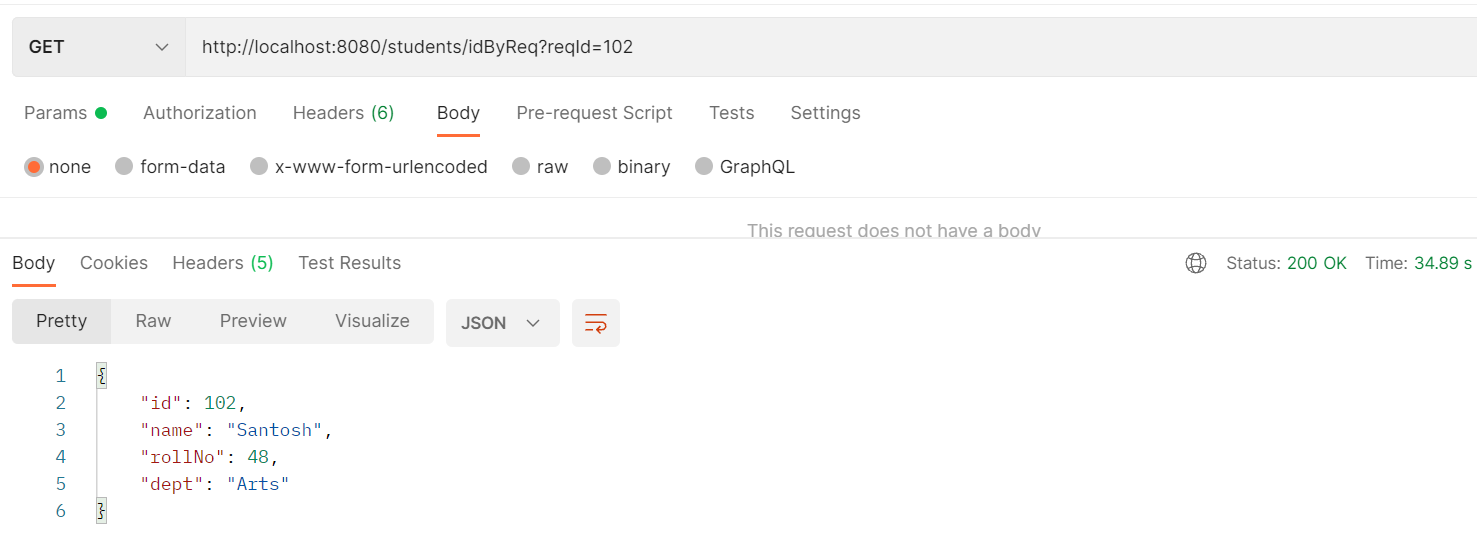
<http://localhost:8080/students/101>

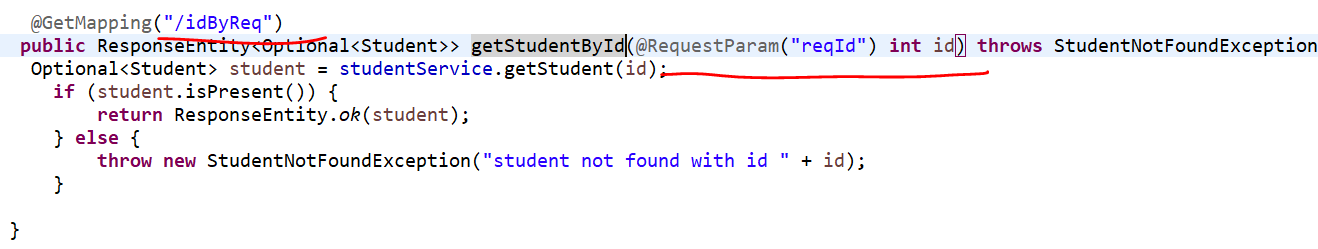




So, this is how you can use the @**PathVariable**.

If you want to pass input as part of Request URL also you can use @**RequestParam** rather than using the @**PathVariable**. In @**RequestParam** parameter u need to specify the parameter which you are going to pass as part of request.





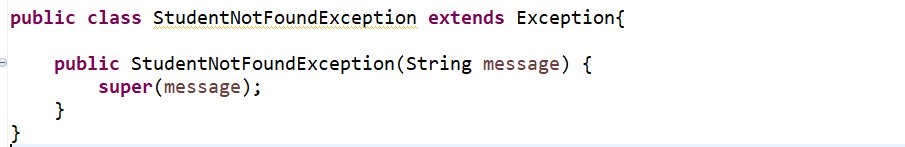
**So, what there is 2 annotations to pass the input of Request URL?**

Either u can use @**PathVariable** or @**RequestParam** the main difference behind it is if you are using a @**PathVariable** and then you are not giving an input then you will get 404. So, if you use **@ PathVariable** then it will be forcing or mandate to pass the input otw you will get 404 but in case of @**RequestParam** it is optional if u won’t give the input there is no harm you won’t get the exception.

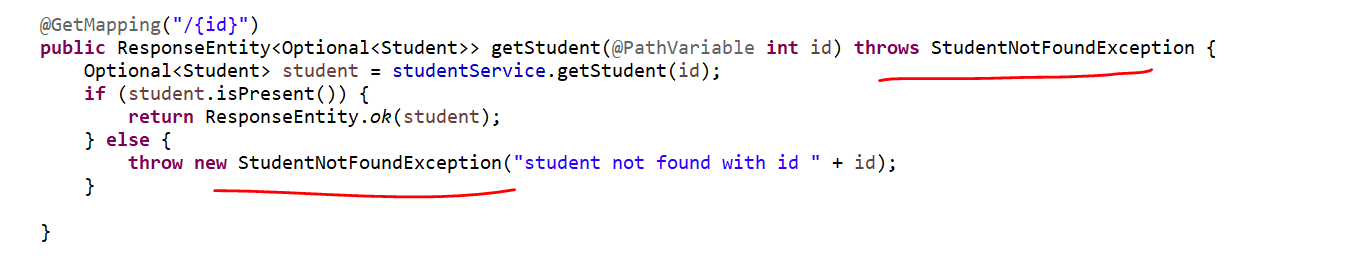
So, @**RequestParam** won’t force you to pass the input as part of Request URL. So based on your business you need to decide either your field is mandatory or optional.

**==========@ControllerAdvice and @ExceptionHandler=============**

Usually, these 2 annotations used to handle exceptions in our application. Now if I will show you in our code.



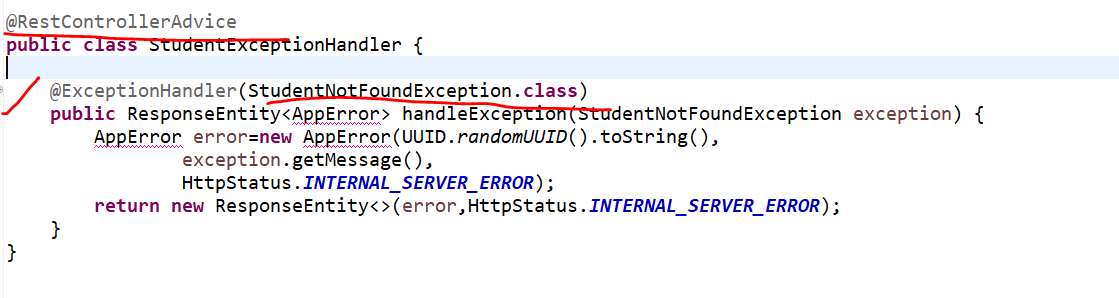
---------------------------------------------------------------------------------------------------------------------------



If Student is not present in the service, we are throwing the exception. Now when your controller throws these particular exception then immediately Spring or Spring Boot will Search for a class which is annotated with @**RestControllerAdvice** or **@ControllerAdvice.**

**Once he found a class immediately, he will delegate a request to that corresponding class and inside that class it will just find the matched annotation or matched Exception.**

If you observed in Controller, we are throwing **StudentNotFoundException** and if u observed in **RestControllerAdvice** we are already mapped the **StudentNotFoundException**.



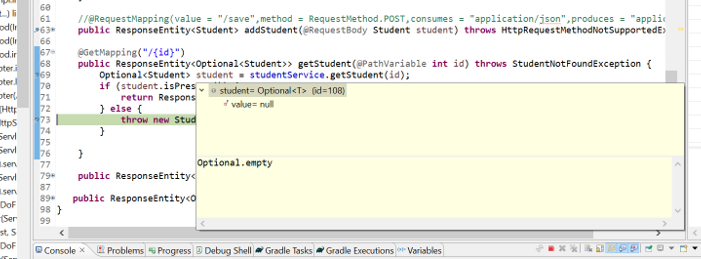
Already we found the matched exception which we are throwing from our controller and we annotate with @**ExceptionHandler**.

Now in this class simply we are telling the Spring this is the method I will build my Custom Error message and I will returned it back to the end-user. so, that is where the @**ControllerAdvice** and @**ExceptionHandler** is required to handle the exception in your Rest API.

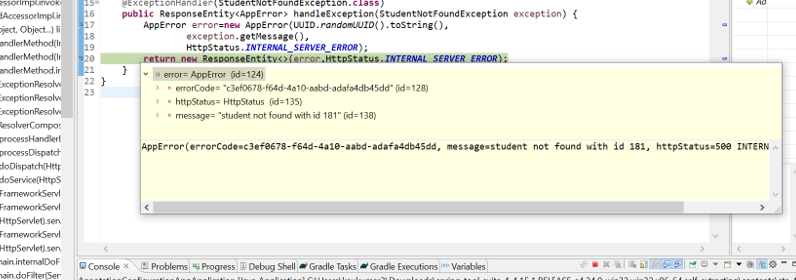


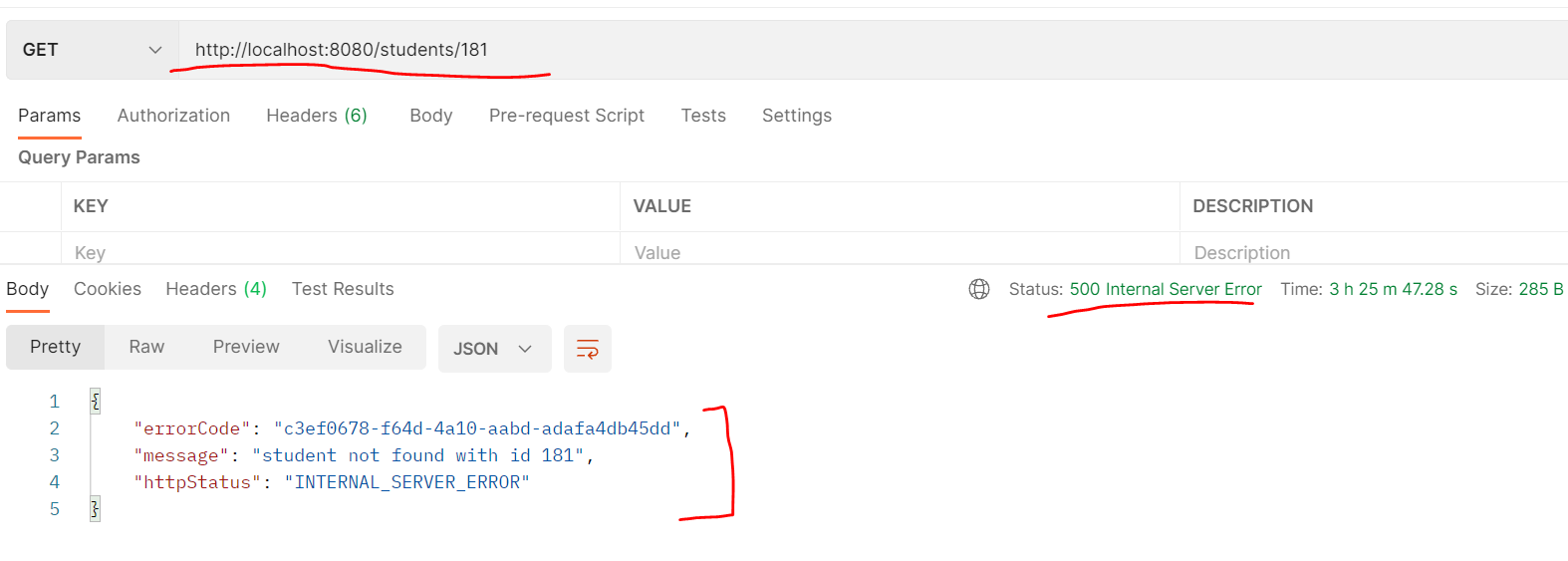
Let me add a breakpoint and show you…

I will give some id which is not there in my database.



Now I can see the student object is empty. Once the Request will throw the exception immediately the request will delegate to ControllerAdvice class. I have created a **AppError** it is just a particular DTO class to define the **ErrorCode**, **ErrorMessage** and **HttpStatus**. And simply it will just returned it to your Postman.





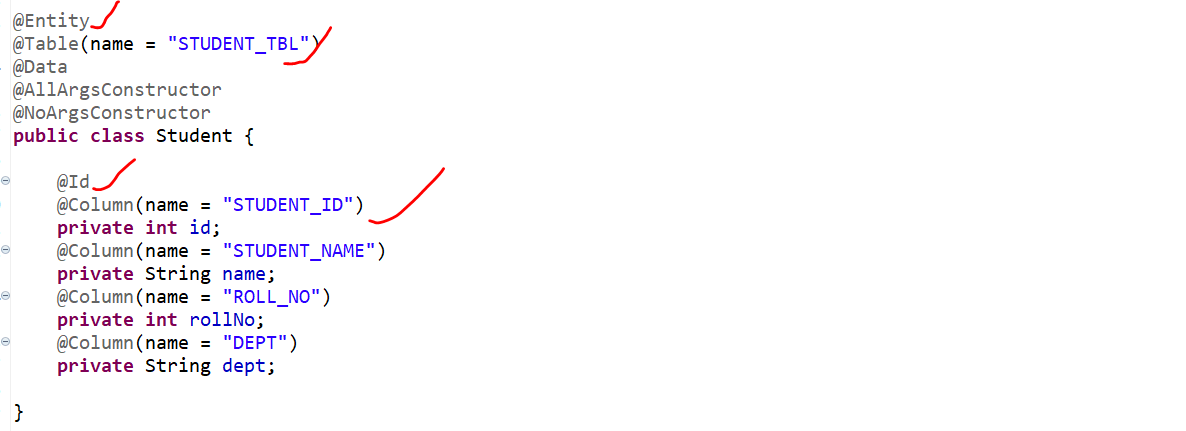
That is how these 2 annotations(@**ControllerAdvice/@RestControllerAdvice** and @**ExceptionHandler)** helps us to handle the exception in our Restful webservices here.

**Spring Data JPA Related Annotation**

* **@Entity**
* **@Table**
* **@Id**
* **@GeneratedValue**
* **@Column**
* **@Transactional**

Entity Class relationships:-

* **@OnetoOne**
* **@OnetoMany**
* **@ManytoOne**
* **@ManytoMany**



**AOP Related Annotations**

* **@Aspect**
* **@PointCut**
* **@JoinPoint**
* **@Advice**
* **@Before**
* **@After**