**How SpringBoot solves problems of Spring -**

1) Spring Boot Auto Configuration Feature solves the problem of Spring where we have to do a lot of configurations. SpringBoot will automatically configure common spring beans whenever it will find a jar dependency in pom.xml / build.gradle file.

2) In Spring we need to explicitly configure the server and we need to deploy a spring application in that server. For ex lets say we want to use Tomcat server to deploy this spring based app, then we have to download it from internet and then setup in IDE and then we can able to deploy spring app in that Tomcat server. This is manual process.

Spring boot provides embedded tomcat server to quickly deploy application.

3) For Spring application, we have to manually manage the compatible versions of all the frameworks that we mention in pom.xml or build.gradle file.

SpringBoot provides a set of starter dependencies, it will internally manage the versions of all the framework.

Spring team developed Springboot on top of spring framework to quickly create and bootstrap spring based application.

Springboot takes opinionated approach for the configuration. For ex, whenever we add a springboot starter web dependency then springboot assumes that we are trying to create spring MVC project so springboot will configure all the spring beans related to spring MVC.

**Externalized Configuration -**typically we deploy spring applications in different environments such as production, testing or development env. So inorder to deploy spring app in different environments we have to externalize configurations based on environments. Springboot provides a good support to externalize the configurations based on different env.

**Spring Boot Actuator -**It provides out of box rest end points as a production ready features like we can use spring boot actuator provided rest APIs to view the application bean configuration details, the application URL mappings, environment details and configuration parameter values and to view the registered health check metrics as well.

**Easy-to-use embedded servlet container support -**Traditionally, in order to deploy the web application, we were building the war file and then deploying that war file in external server such as Tomcat server. But by using springboot, u can create a jar file out of the spring application and then u can deploy that jar file in a embedded servlet container.

spring-boot-starter-web provides Apache Tomcat as embedded default container.

within controller package we keep all the spring MVC controllers

**@Controller** annotation is used to make a Java class as a spring MVC controller. Within the rest controller we can create the rest APIs.

Whenever we develop the restful web services using spring MVC then all the rest API's return JSON to client. So inorder to convert a Java object into JSON, we have to use **@ResponseBody** annotation

**@RestController  = @Controller + @ResponseBody**

Note :- whenever u have the same variable name that is URI template variable name and method argument then u don't have to pass variable name in @PathVariable annotation, otherwise u need to pass it.

**@PathVariable** is used to bind the value of URI Template variable into method argument whereas we use **@RequestParam** to extract the value of query parameter in request URL

**@RequestBody** annotation internally uses Spring provided HttpMessageConverter to convert JSON into Java object.

**@ResponseStatus** annotation is used to return HTTP status to the client.

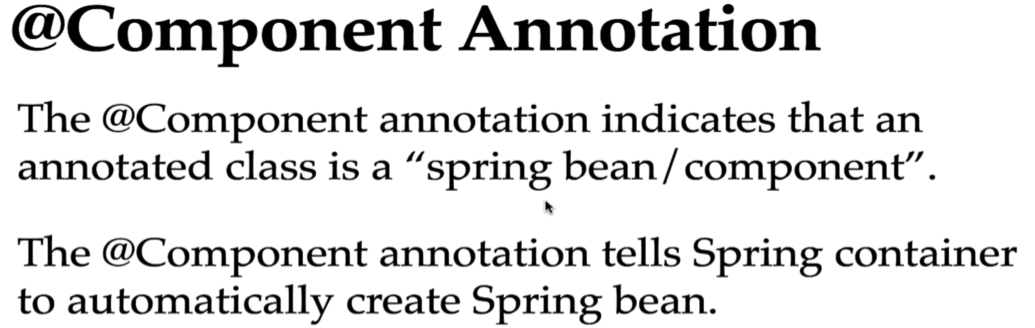
For post requests, code is 201 created.

put requests => 200 OK

Spring boot by default provide 200 status code



We can use **@RequestMapping** annotation at class level to configure the Base URL for REST APIs in a spring MVC controller.



Spring container take the control to automatically create the spring bean and manage that spring bean for us. That’s why spring IOC container also called **Inversion of Control**.

Ex-  This is called annotation based configuration.

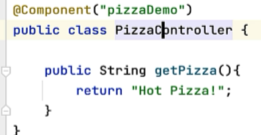
Inside main() method we have run() method which returns the application context object. **ApplicationContext** in Spring **acts as the Spring IoC (Inversion of Control) container**.

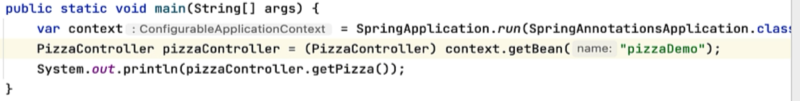


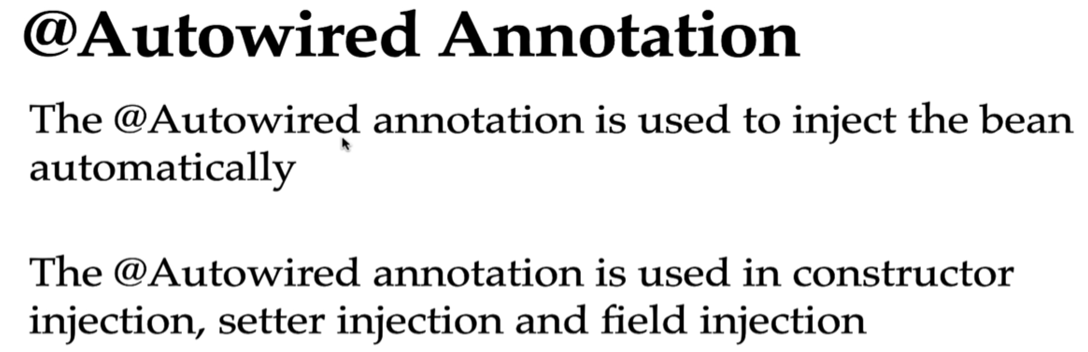
By default spring container will give name to the spring bean as the class name, but the first letter of class name in lower case.



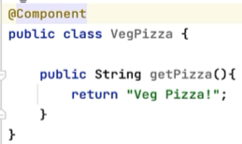
We can explicitly give name to spring bean by passing value to @Component annotation like –

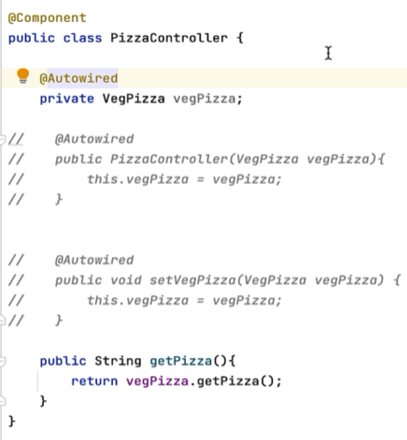


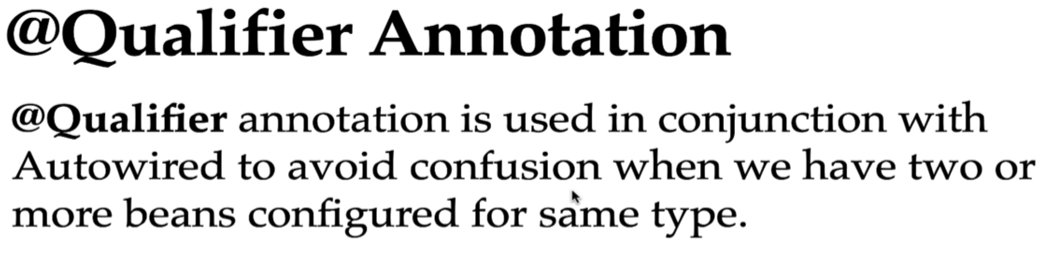


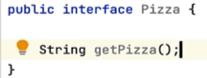


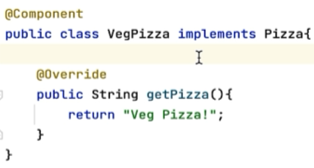
It is used in annotation based configuration.

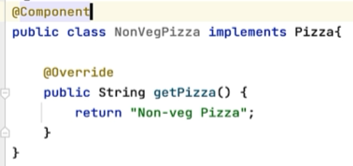
Ex - 

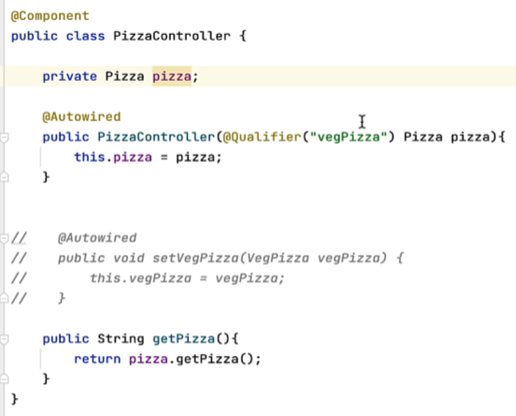




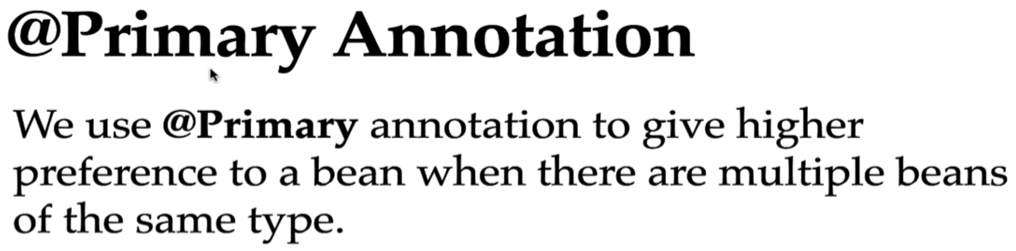
Ex – 



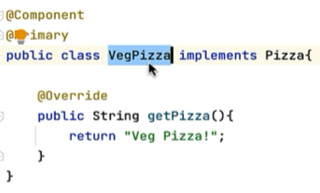


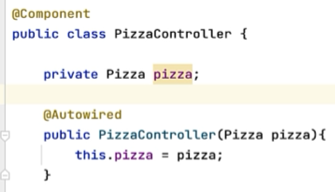


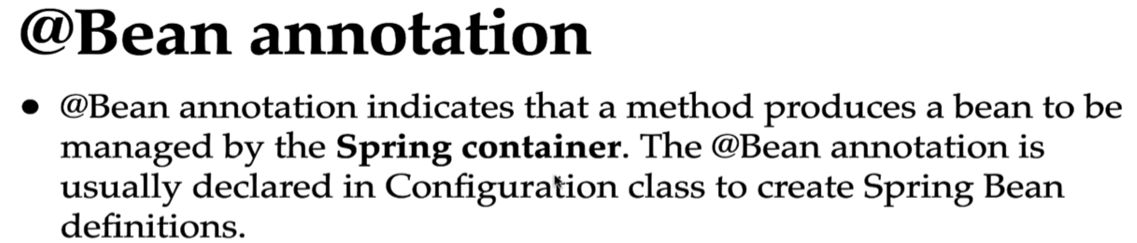
Without @Qualifier annotation, spring IOC container will get confused to inject which Pizza implementation Veg or Nonveg. So we explicitly mentioned it.



Its alternative to @Qualifier annotation.

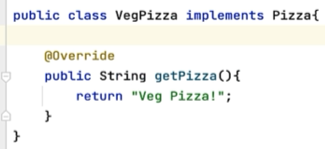
Ex – 





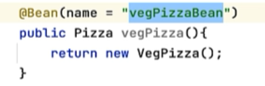
Whenever we use @Configuration annotation, that class becomes a configuration class and within that class we can define spring bean configurations using @Bean annotation. This is java based configuration.

A Spring Bean is an object that is managed by the Spring IoC container.

Ex – 

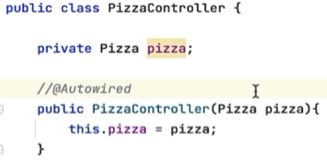


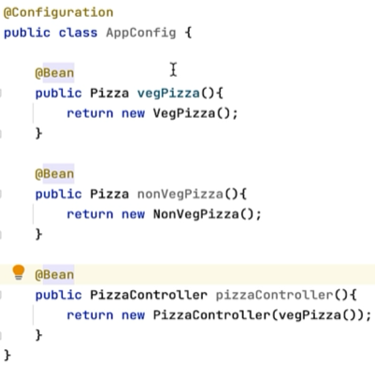
By default spring container will give name to this spring bean as method name. We can explicitly provide name like –

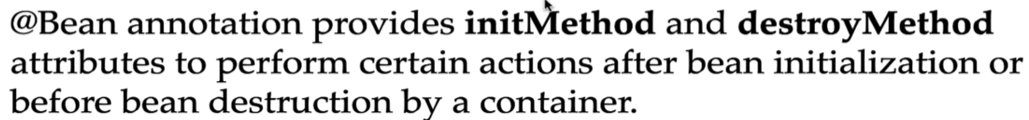




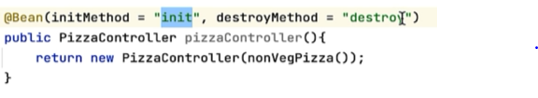
For PizzaController, remove @Autowired as we’ll use java based configuration –



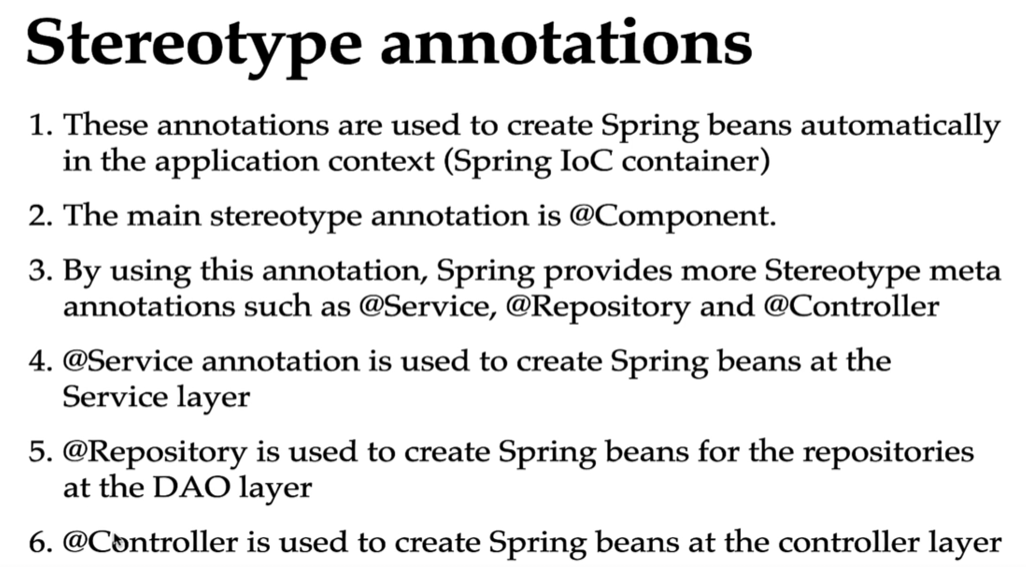


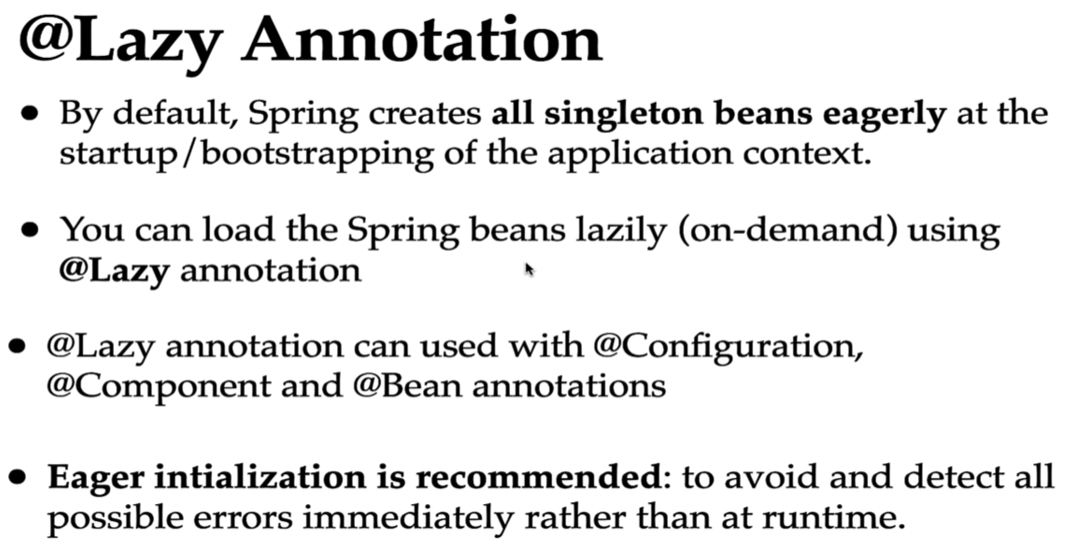


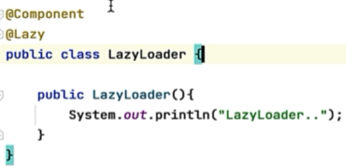
In value we pass the method name to call, these should be public methods defined inside PizzaController class –



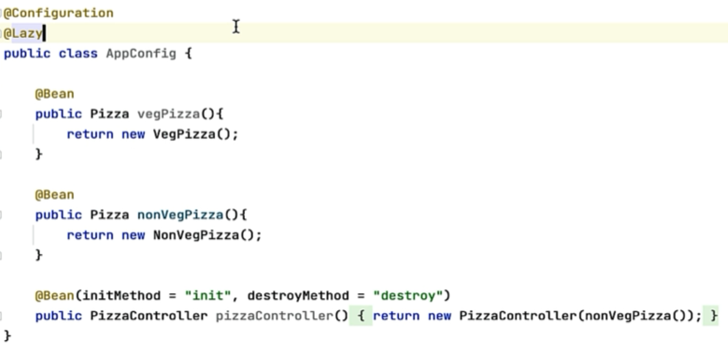
We can use it in scenarios like if we have to insert records in application before startup and destroy the records while application shutdown.



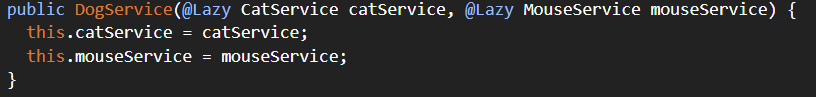




This way all the classes in configuration file are lazily loaded -



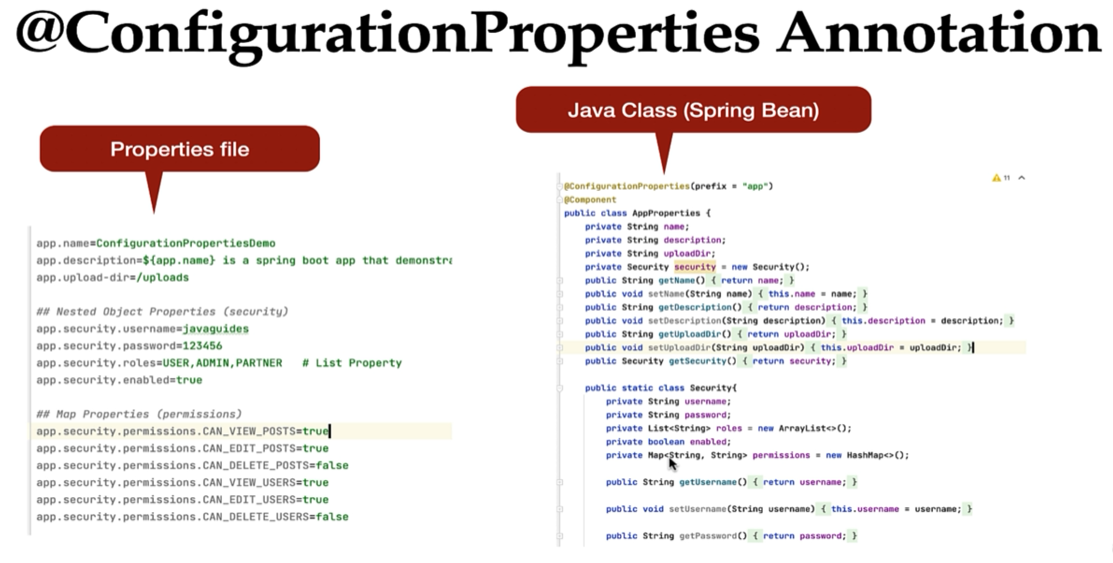
You can use @Lazy on autowired constructor argument –



Spring does not initialize CatService and MouseService immediately instead injects a proxy object. And when the proxy is accessed, the actual bean is created.



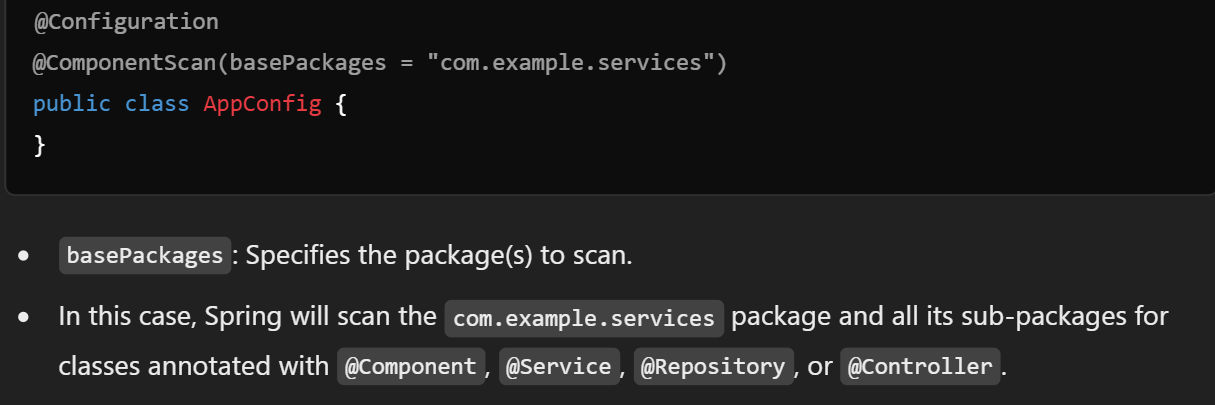
Read all the properties which are prefix with string “app”



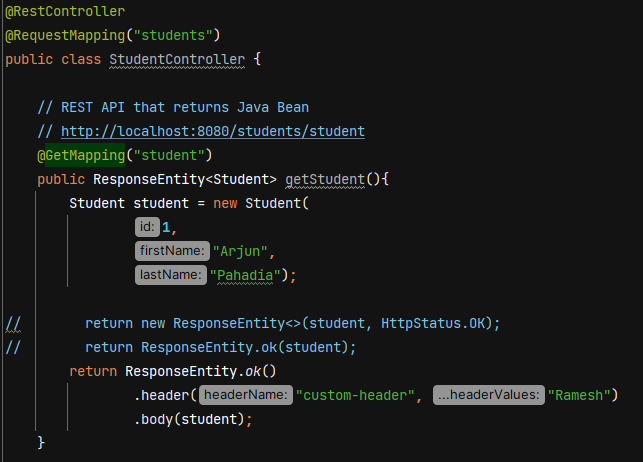
**@ComponentScan**

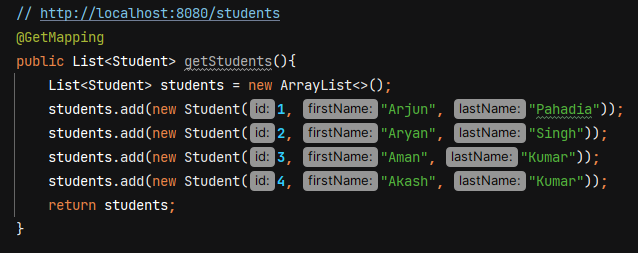
The @ComponentScan annotation in Spring is used to specify the packages that the Spring container should scan to detect and register beans annotated with stereotype annotations like @Component, @Service, @Repository, or @Controller.

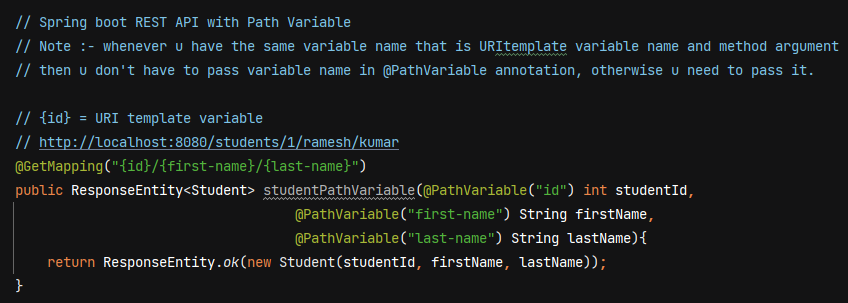
It is used in conjunction with Java-based configuration (@Configuration)

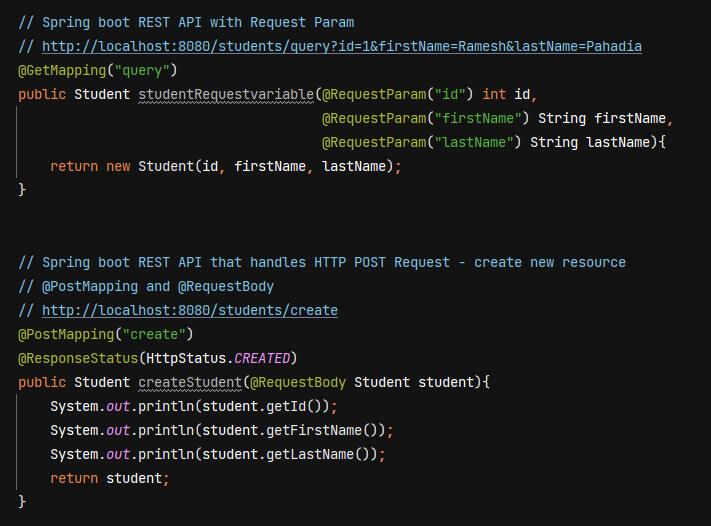


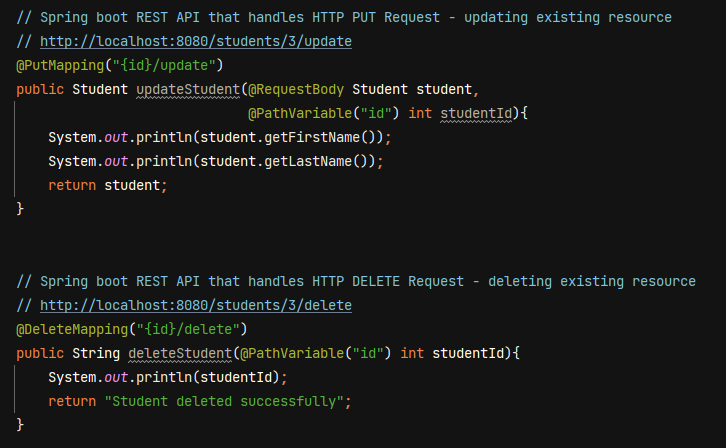
**REST APIs in SpringBoot**

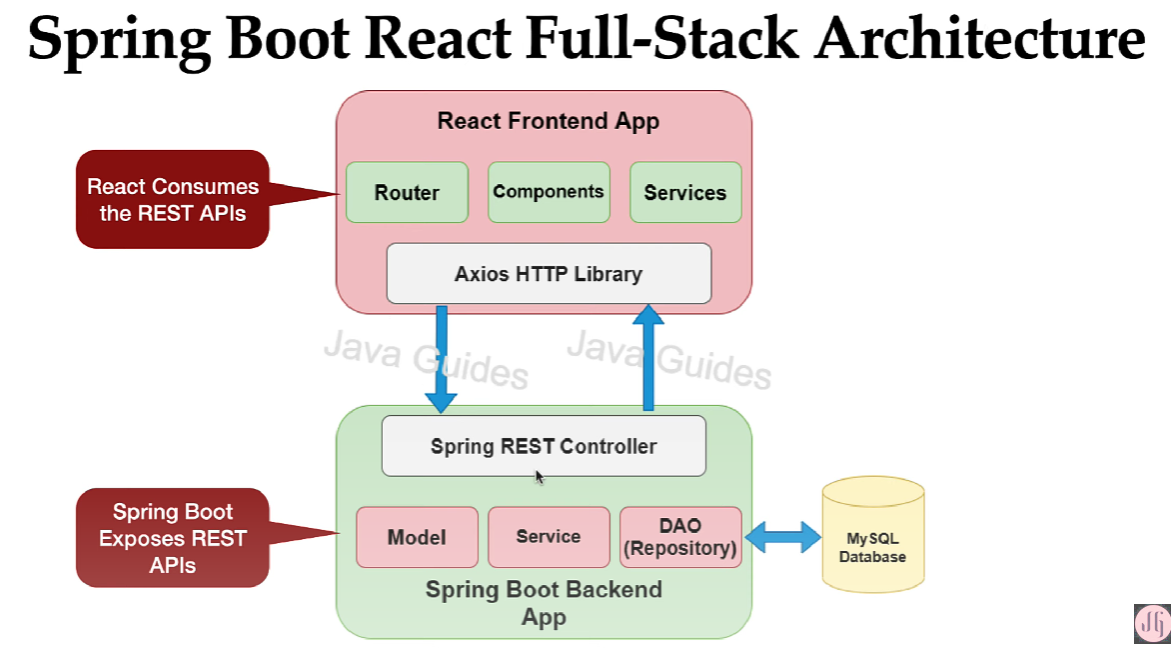












Both react frontend app and springboot backend app are loosely coupled.

For backend, we make 3 layer architecture – controler, service and DAO layer. The dao layer is responsible to talk with the DB and the service layer basically contains the business logic of app, and the controller basically contains the spring MVC controllers which exposes the REST APIs.

In react frontend app, we create components, services, router. We have used Axios http library to make a rest api call. We use a JSON format to exchange the data between react frontend and springboot backend.

**DTO (Data Transfer Object)**

It is widely used design pattern to transfer the data between client and server.

Server

Client

Client can create a DTO Object and it will send that DTO object in the HTTP request and server will extract the dto object from the request and it will use that DTO object. Similarlly server will create a DTO object and it will send that DTO object in the reponse of the rest API.

**Main advantage is to reduce the number of remote calls to the server**. For ex- in our employee managent system we have Organization inside which we have list of depts and within depts we have list of employees. Now if we want all the data, we have to make 3 individual rest api calls to get organization, list of depts and employees. But we can create a APIResponseDTO class having company, List<dept> and List<employeed> and send it back to the client.

Server can use DTO to transfer the only required amount of data to the client.

**How to use DTO pattern in Spring Boot**

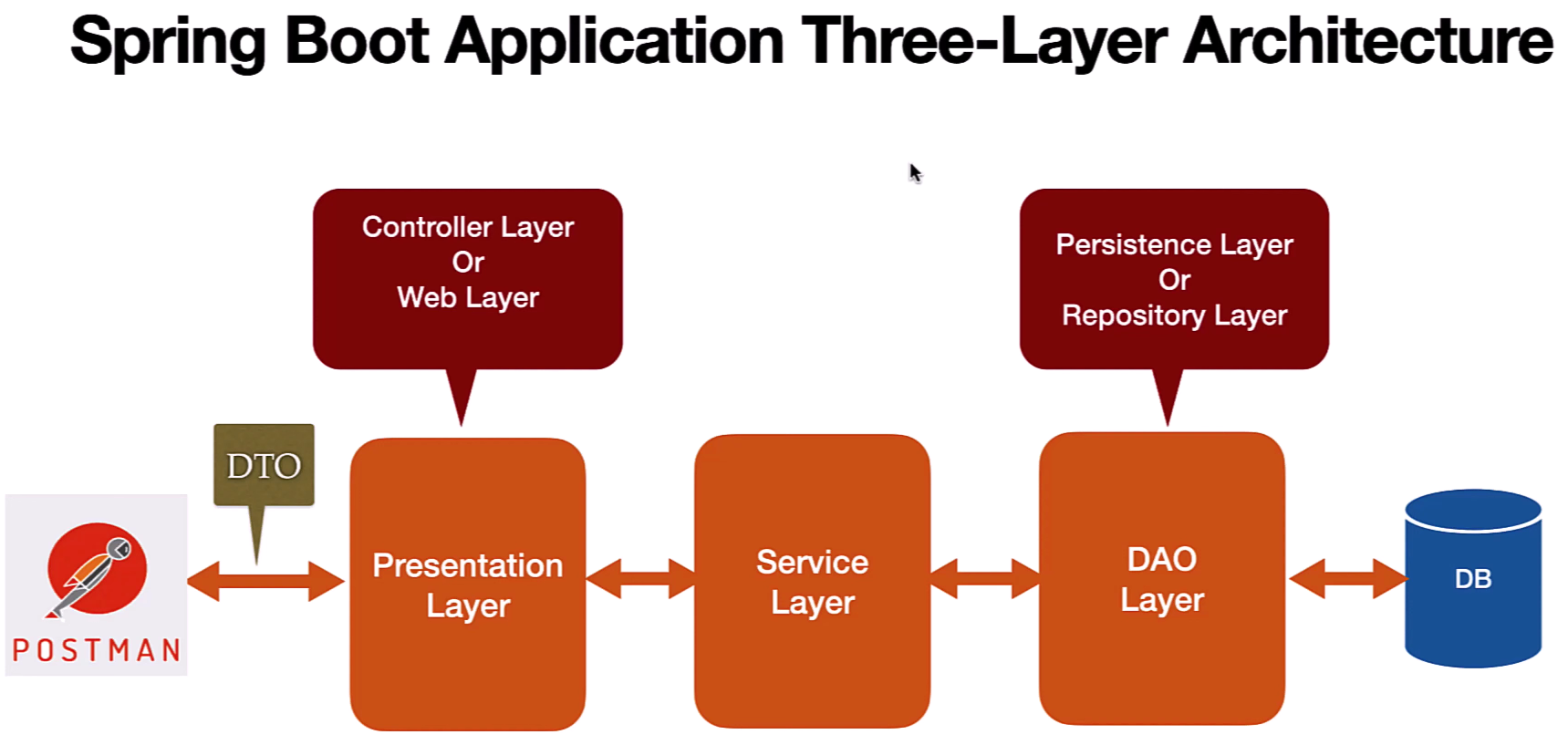
In springboot we have JPA entities and we use JPA entitity to map object to the relational database table. DAO / Repository layer uses JPA entities to store the data into a database and retrieve it.

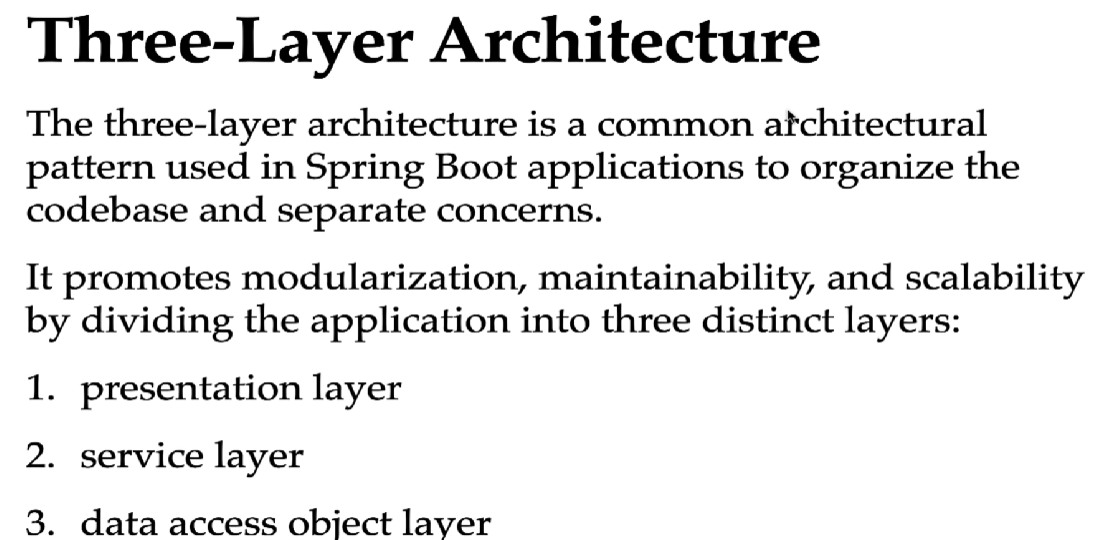
Client and Server uses DTO to transfer data. Some developers uses JPA entitites to transfer to data between client and server. But transfering JPA entitity has some disadvantages –

* Transporting the sensitive information. Consider our JPA entity has some fields like username, password, some codes. If we don’t handle this sensitive info and our REST API directly send the JPA entity to the client, the client will get the password and all the sensitive information.

To overcome this problem we can use DTO to transfer the data between client and server.

**In DTO we’ll keep only the required data that client expect in a response of the rest API.**

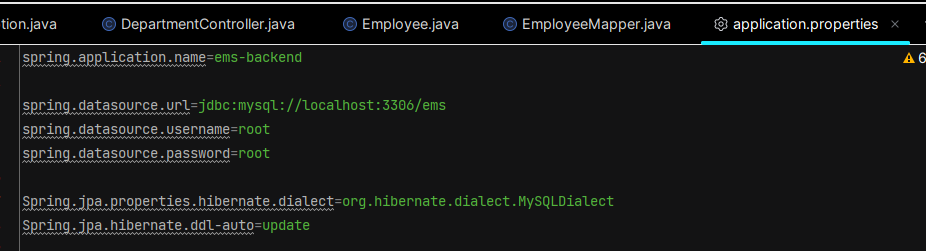




In application.properties file we configure database details

Hibernate uses **MySQLDialect** to create the SQL statement based on the database that we are using.

**Spring.jpa.hibernate.ddl-auto=update ->**  this property tell hibernate to automatically create the database tables based on our JPA entitied if the tables don’t exist in the database and if there are any changes in a JPA entities, then it will also tell Hibernate to update those changes in the DB tables as well.



Springboot by default uses Hikari data source and Hikari connection pool.

**Creating Employee JPA Entity**  - create a java class Employee.java in entity folder

@Entity - to specify a class as a JPA entity

@Table(name = ”employees”) – to specify the table name, if we don’t give name=” ” then jpa will take table name same as the class name

@Id – to configure the primary key

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

* to configure PK generation strategy
* IDENTITY generation strategy uses database autoincrement feature to automatically increment the PK

@Column(name = ”first\_name”, nullable = false, unique = true)

* to map a database table column with a class field. If u don’t mention it then JPA willl by default give column name as field name
* nullable = false makes the column value not null
* unique = true to make column value unique

After creating JPA entity, if we run the app then hibernate will automatically create a table in our database.



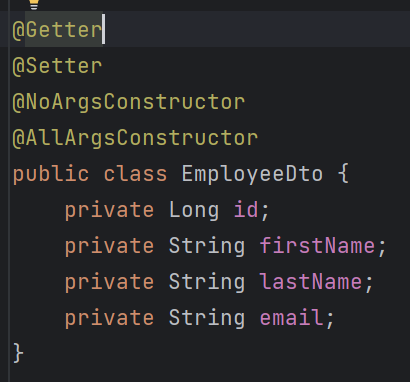
**Creating EmployeeRepository** – create interface EmployeeRepository in repository folder

 JpaRepositry is a generic interface, so pass 2 parameters – type of Entity and type of Primary Key

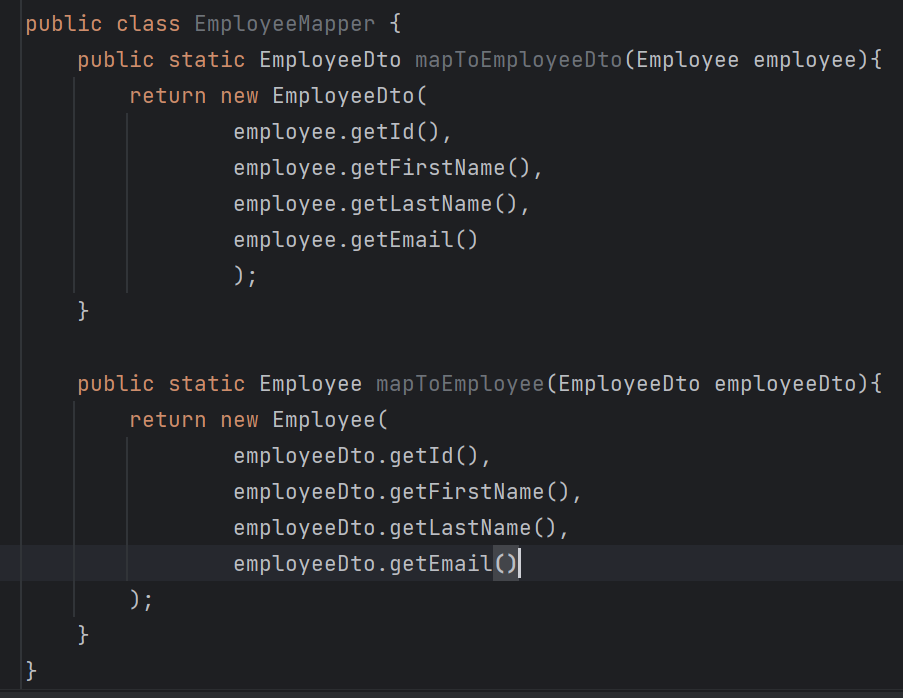
* EmployeeRepository will get methods to perform CRUD database operations on Employee Entity
* JpaRepository will inherit all the methods from all the entended interfaces. SimpleJpaRepository class of Spring Data JPA provide the impl for JpaRepository interface.
* We don’t have to annotate EmployeeRepository with @Repository annotation because the impl class SimpleJpaRepository is already annotated with @Repository annotation.
* SimpleJpaRepository class is also annotated with @Transactional. All the public methods in a SimpleJpaRepository are transactional so we don’t have to again use @Transactional to make these methods transactional.

**Create EmployeeDto and EmployeeMapper**

We’ll use EmployeeDto class to transfer data between client and server.

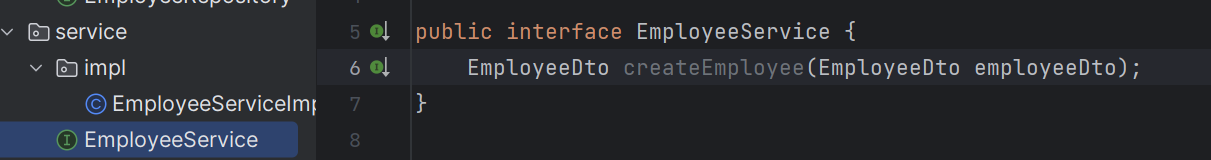


We create EmployeeMapper class to map Employee entity to EmployeeDto and EmployeeDto to EmployeeEntity.



**Build Add Employee REST API**

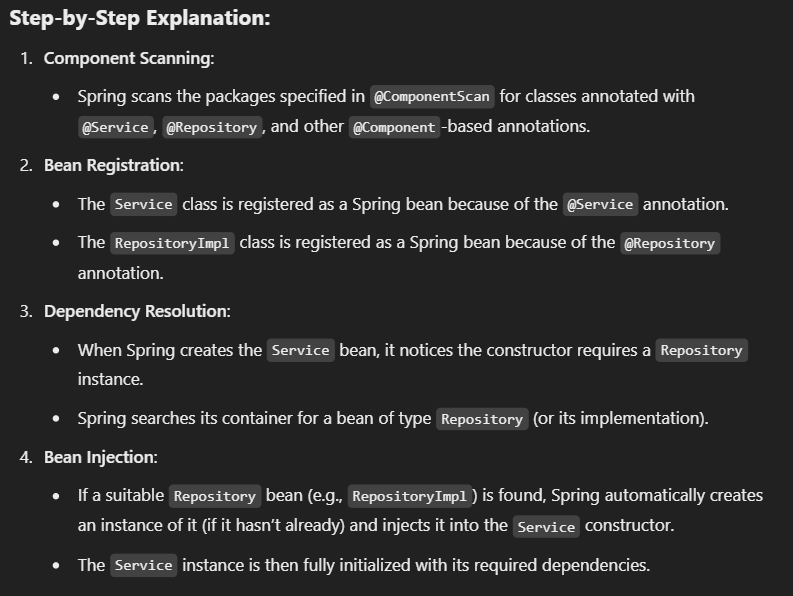
As Controller layer depends on Service layer, so we’ll first create service layer.



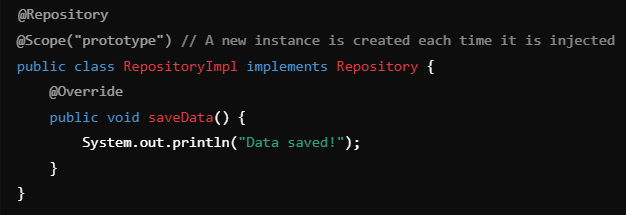
Create an impl of this – EmployeeServiceImpl

Use **@Service** on EmployeeServiceImpl, it will tell spring container to register bean for this class and u don’t need to manually declare beans in a @Configuration class.

We’ll use constructor based DI (when dependencies are provided through the class constructor) to inject the dependencies so annotate EmployeeServiceImpl with @AllArgsConstructor



In Spring, the default scope for beans is **singleton**. This means that only one instance of a bean is created and shared across the entire Spring application context. If you need separate instances of the Repository for each dependent Service, you can change the bean's scope to prototype. This is done using the @Scope annotation –



Inside createEmployee method, we need to first convert EmployeeDto into Employee entity becox we need to store the Employee entity in DB.



Now let’s create Controller –



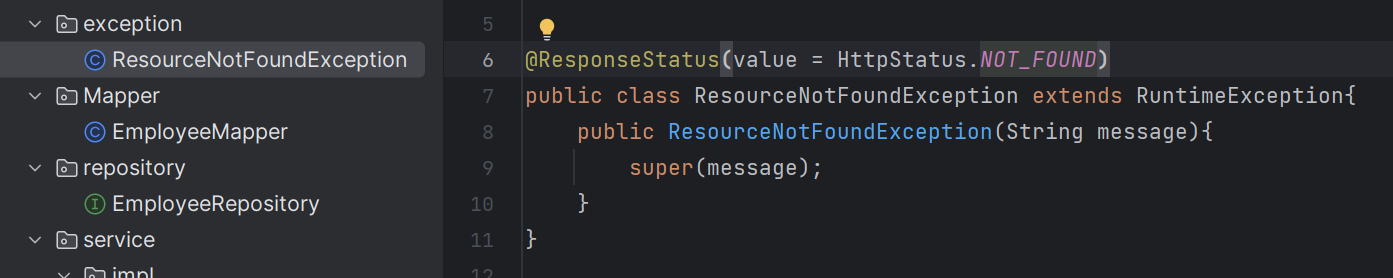
When testing throught Postman and passing JSON in body then json object properties name should match with the Dto class variables.

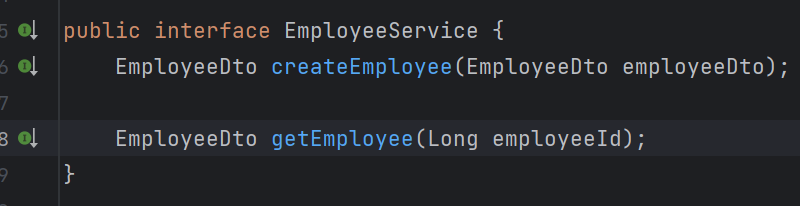


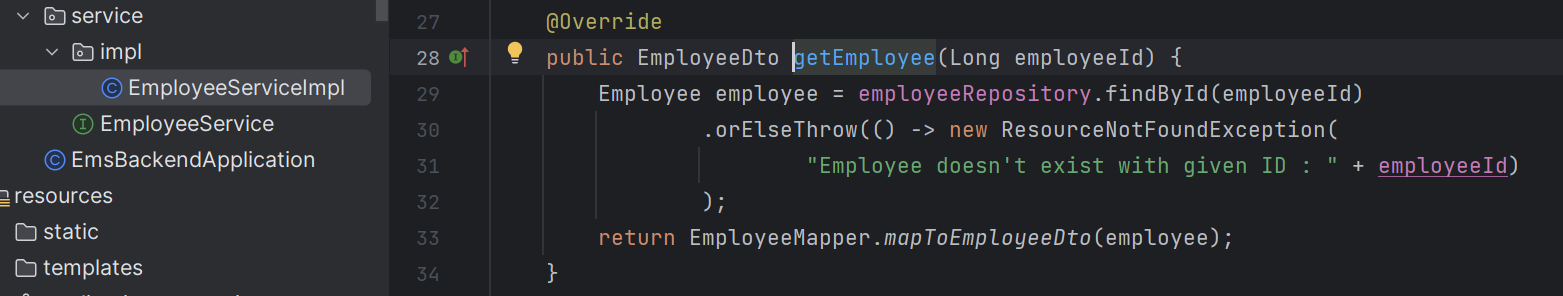
**Build Get Employee REST API**

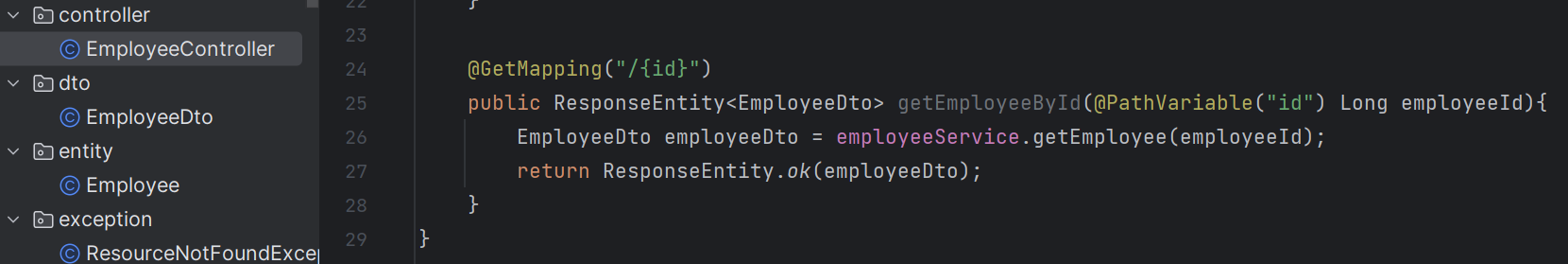
**@ResponseStatus** - used to mark a method or an exception class with the HTTP status code that should be returned in the response.

If a Employee with a given ID does not exist in DB, then throw custom ResourceNotFoundException and then springboot will catch this exception and will get the error message from exception, and it will send the error message along with the http status to client.

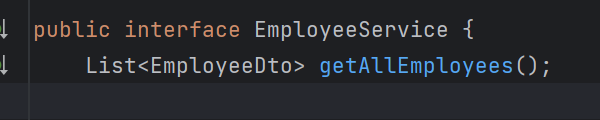




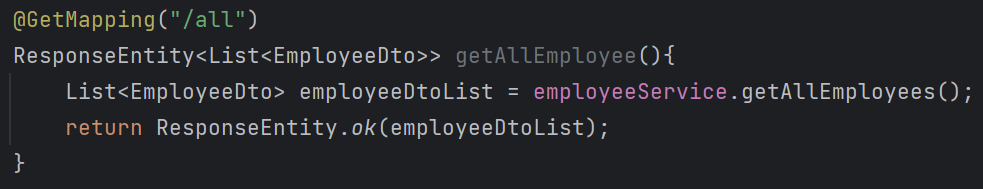




**Build Get All employee REST API**



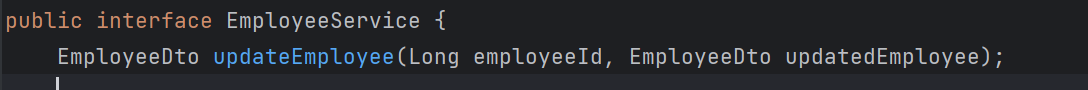




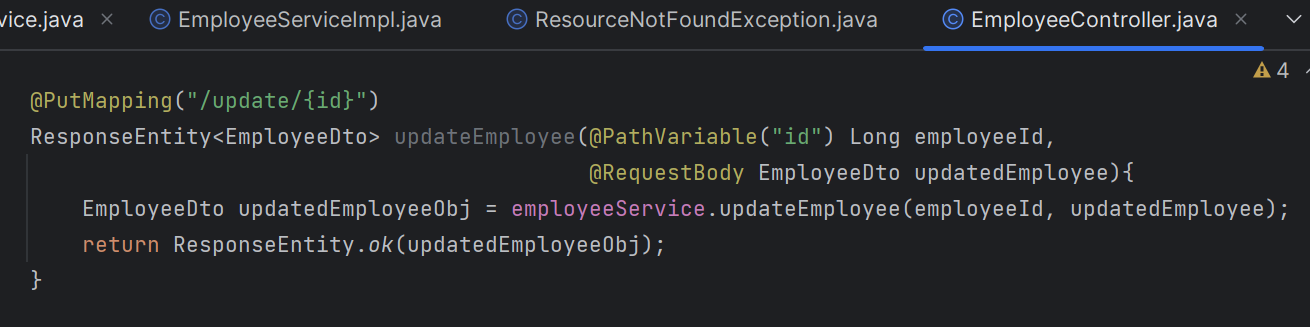
**Build Update Employee REST API**

Save() method of JpaRespository perform both save and update operation.

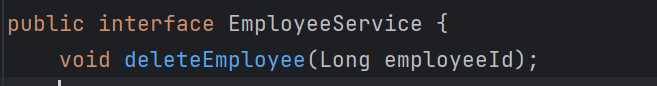
If employee object contains ID then the save method internally perform the update operation. And if employee doesn’t contain the Primary Key ID, then it will perform the insert operation.

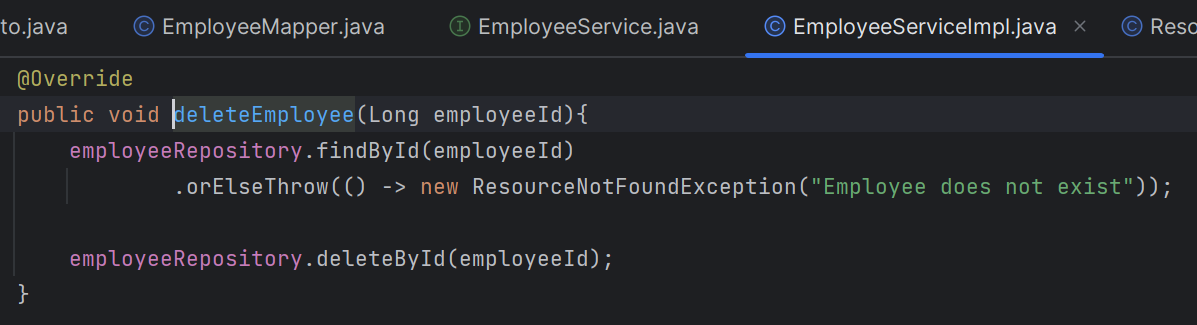


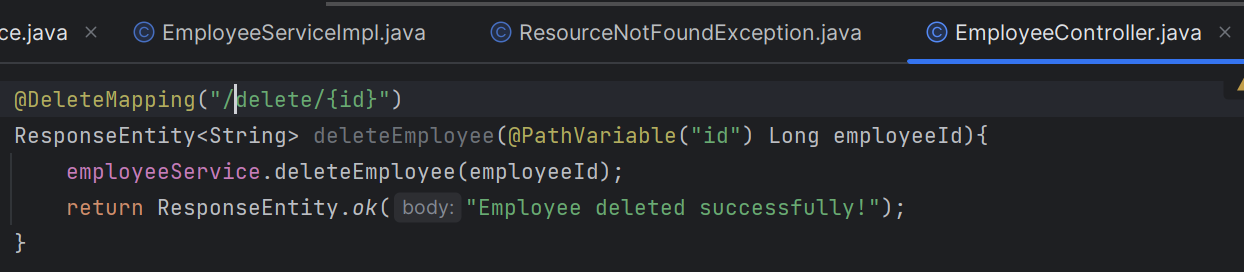




**Build Delete Employee REST API**



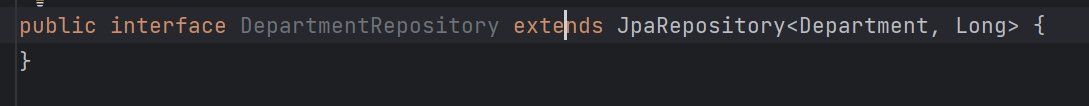




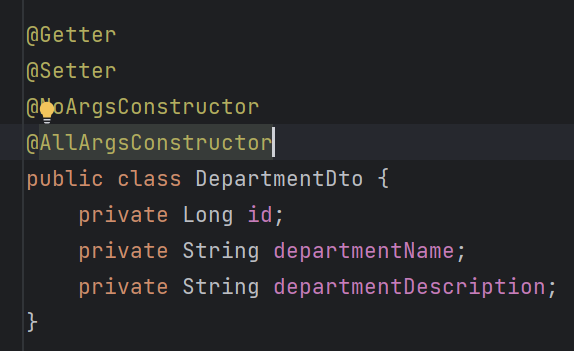
**Requirement 3 :- REST APIs For Department Management Module :**

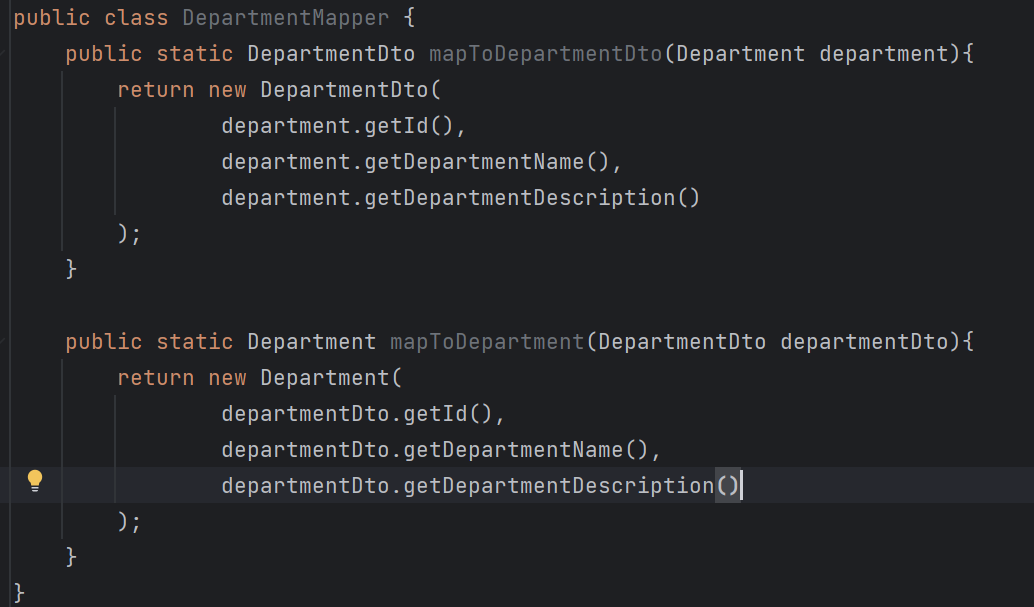
**Create Department Entity & DepartmentRepository**



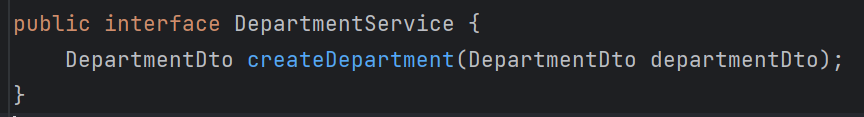


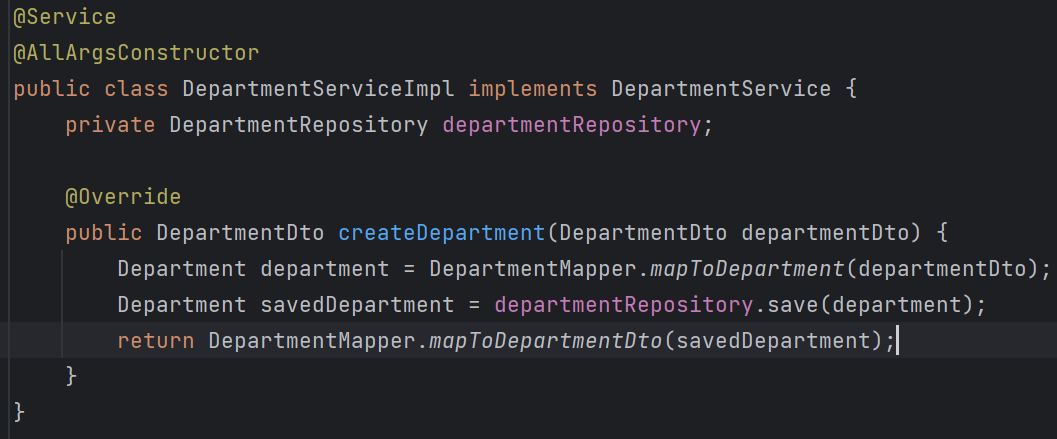
**Create DepartmentDto & DepartmentMapper**

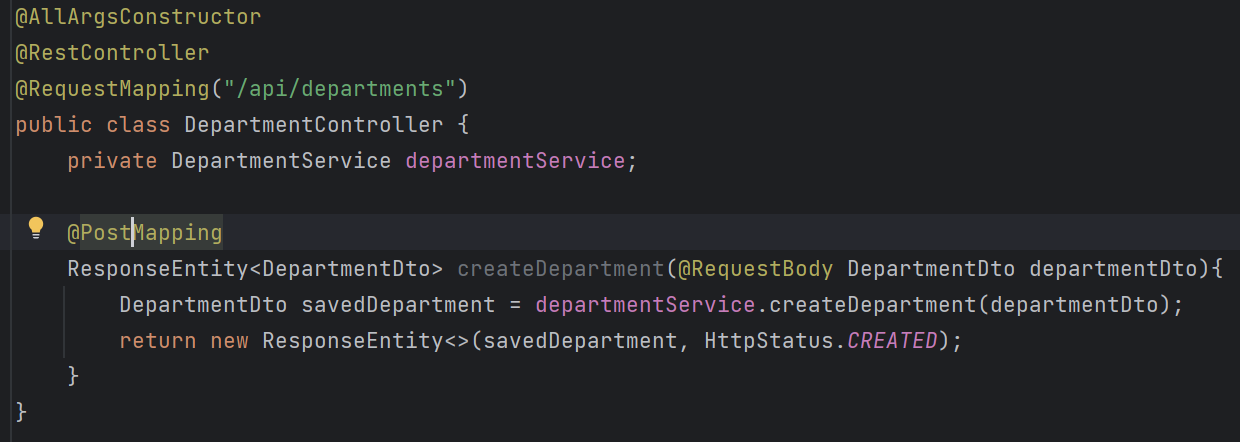




**Build Create Department REST API**







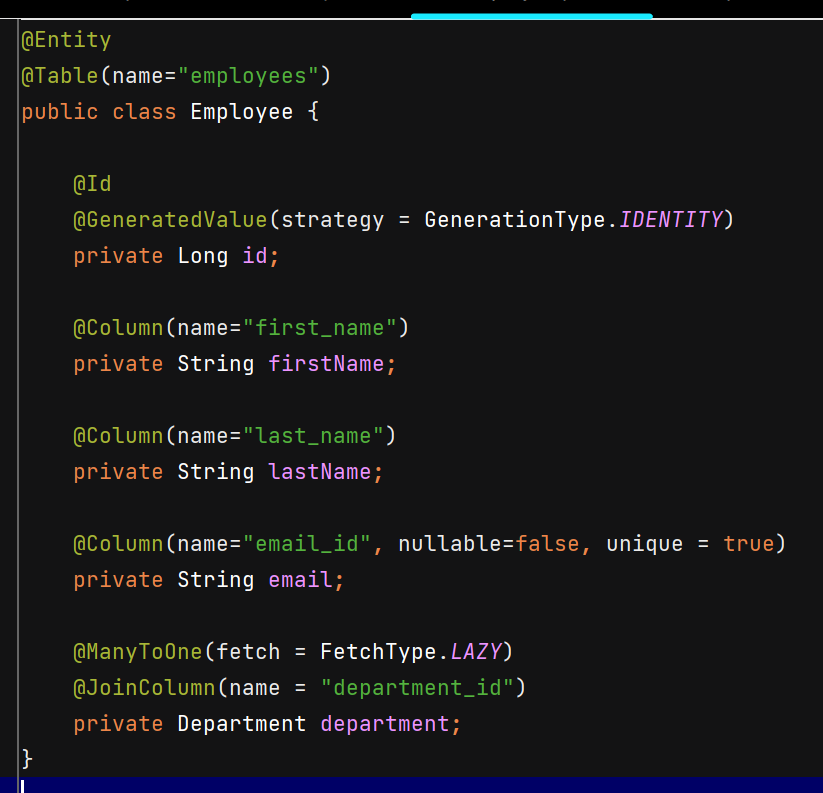
Similarly make REST APIs to get dept by Id, get all dept, update dept and delete dept.

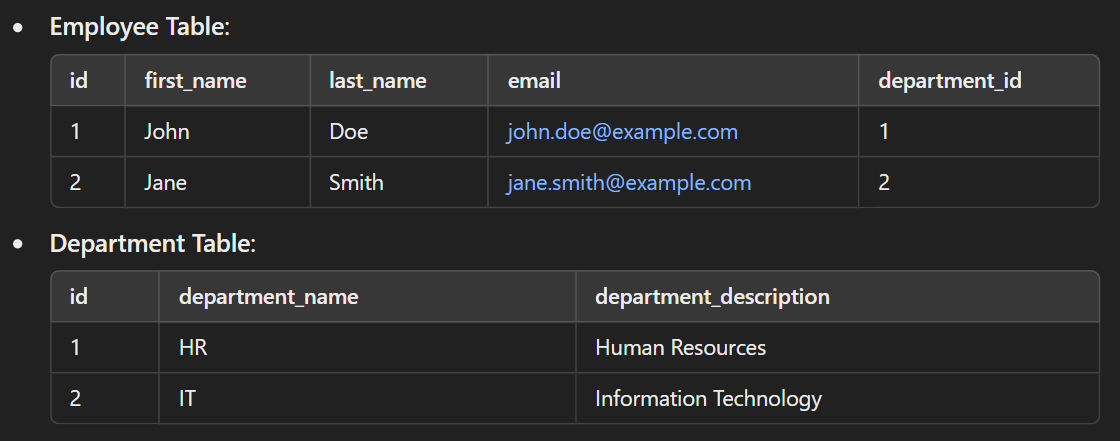
**Many To One Relationship between Employee & Deptarment JPA entities**

So go to Employee JPA entity and add Department instance variable.

Specify FetchType.LAZY becoz whenever we a get employee entity object from DB, the hibernate won’t load the department object immediately. We can get this department object lazily or on demand.

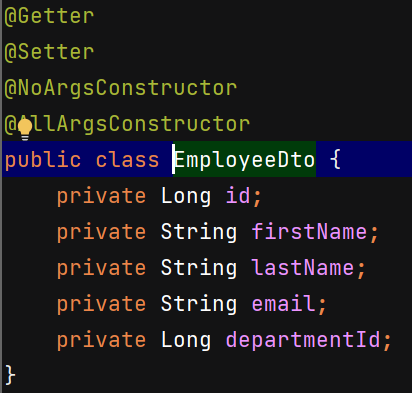
We have to maintain a foreign key in a employee table, specify it using @JoinColumn annotation. name attribute specifies the **name of the foreign key column** in the Employee table that will reference the primary key of the Department table.





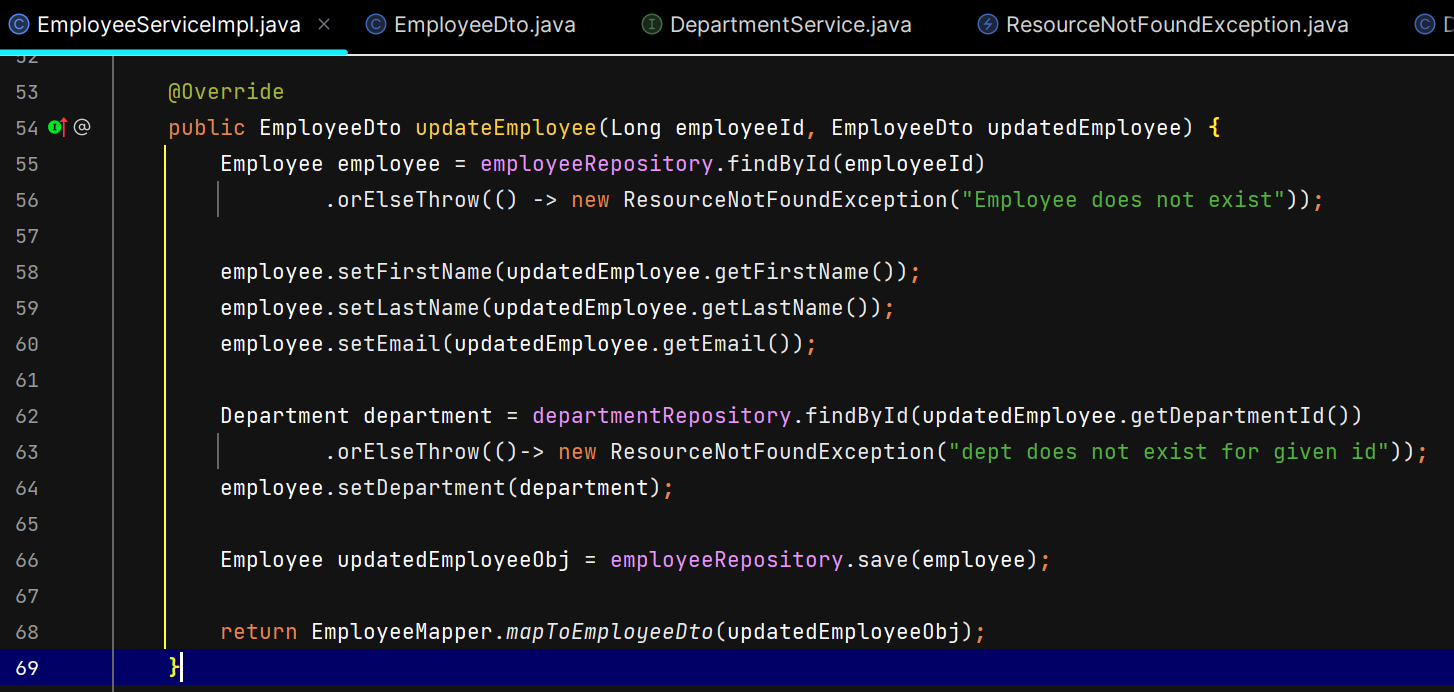
Whenever we save the Employee object in DB, we have to add Department to that Employee object, changes in EmployeeServiceImpl –

Add departmentId to EmployeeDto –





User can change department for particular employee –



Change implementation of mapToEmployee from constructor to setters, and for mapToEmployeeDto pass the department Id –

