<u>How SpringBoot solves problems of Spring -</u>

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

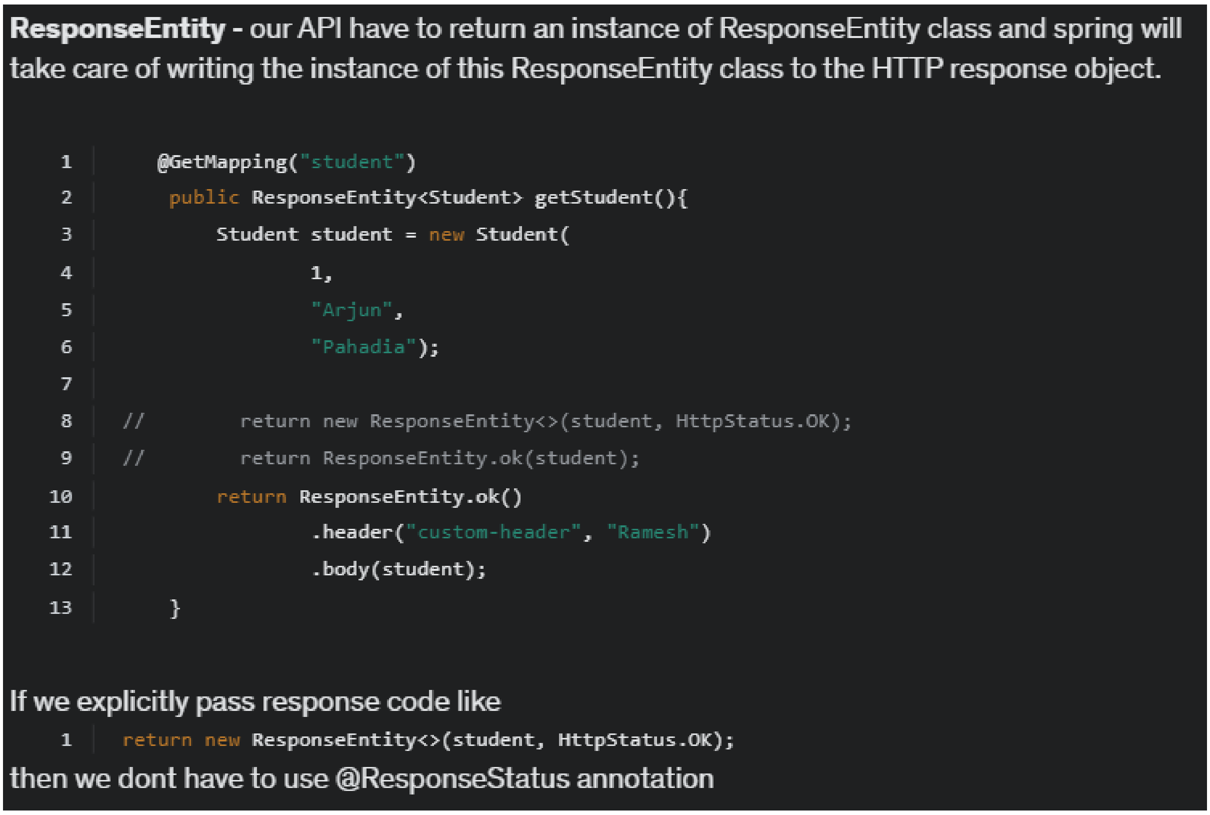
@RequestBody annotation internally uses Spring provided HttpMessageConverter to convert JSON into Java object.

@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



ResponseEntity - our API have to return an instance of ResponseEntity class and spring will take care of writing the instance of this ResponseEntity class to the HTTP response object.

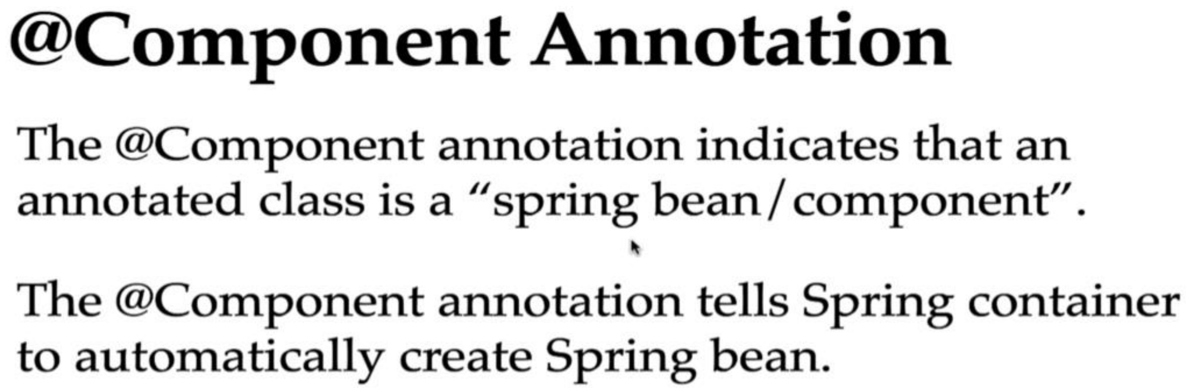
```
   @GetMapping("student")
    public ResponseEntity<Student> getStudent(){
        Student student = new Student(
                1,
                "Arjun",
                "Pahadia");

//          return new ResponseEntity<>(student, HttpStatus.OK);
//          return ResponseEntity.ok(student);
        return ResponseEntity.ok()
                .header("custom-header", "Ramesh")
                .body(student);
    }
```

If we explicitly pass response code like

```
return new ResponseEntity<>(student, HttpStatus.OK);
```

then we dont have to use @ResponseStatus annotation

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

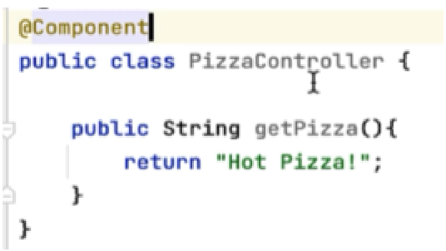


@Component Annotation

The @Component annotation indicates that an annotated class is a "spring bean/component".

The @Component annotation tells Spring container to automatically create Spring bean.

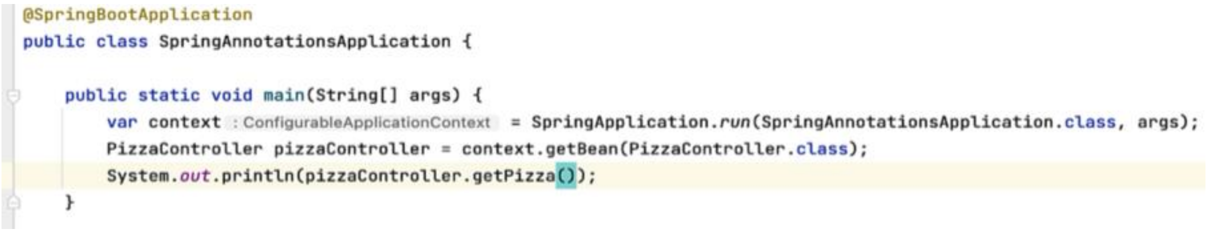
Spring container take the control to automatically create the spring bean and manage that spring bean for us. That's why spring IOC container also called **Inversion of Control**.

Ex- 

```
@Component
public class PizzaController {

    public String getPizza(){
        return "Hot Pizza!";
    }
}
```

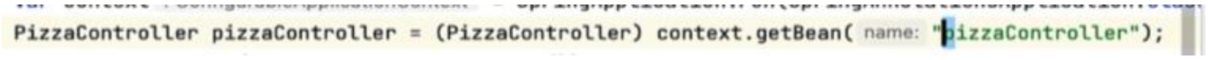
This is called annotation based configuration.

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**.



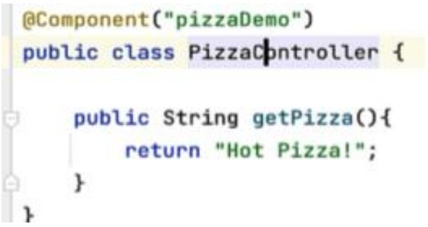
```
@SpringBootApplication
public class SpringAnnotationsApplication {

    public static void main(String[] args) {
        var context : ConfigurableApplicationContext  = SpringApplication.run(SpringAnnotationsApplication.class, args);
        PizzaController pizzaController = context.getBean(PizzaController.class);
        System.out.println(pizzaController.getPizza());
    }
```

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.

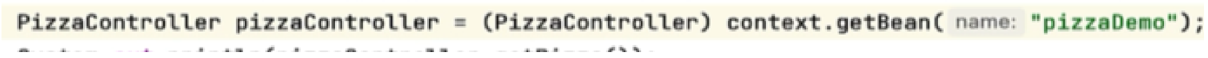


```
PizzaController pizzaController = (PizzaController) context.getBean( name: "pizzaController");
```

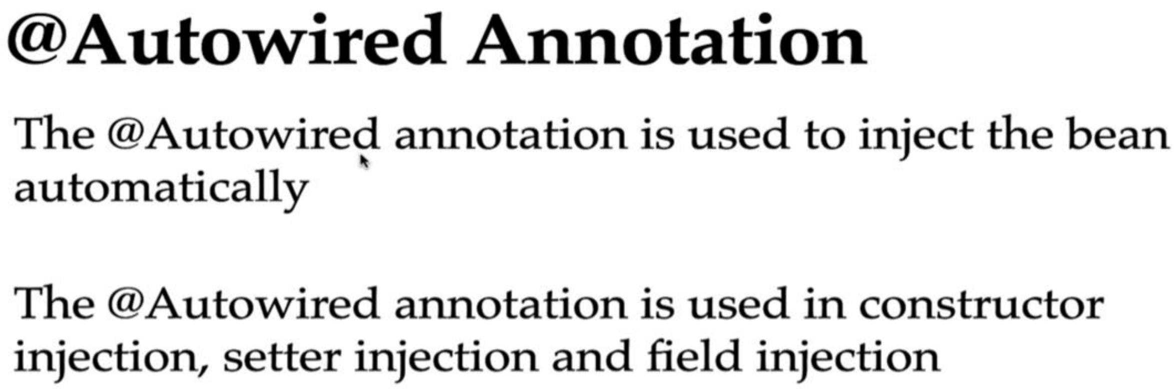
We can explicitly give name to spring bean by passing value to @Component annotation like –



```
@Component("pizzaDemo")
public class PizzaController {

    public String getPizza(){
        return "Hot Pizza!";
    }
}
```



```
PizzaController pizzaController = (PizzaController) context.getBean( name: "pizzaDemo");
```



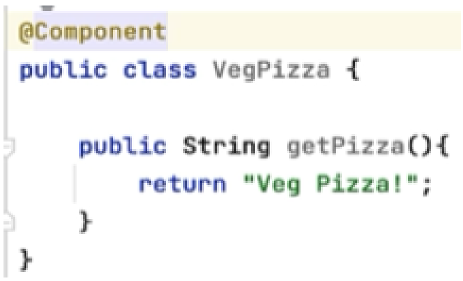
@Autowired Annotation

The @Autowired annotation is used to inject the bean automatically

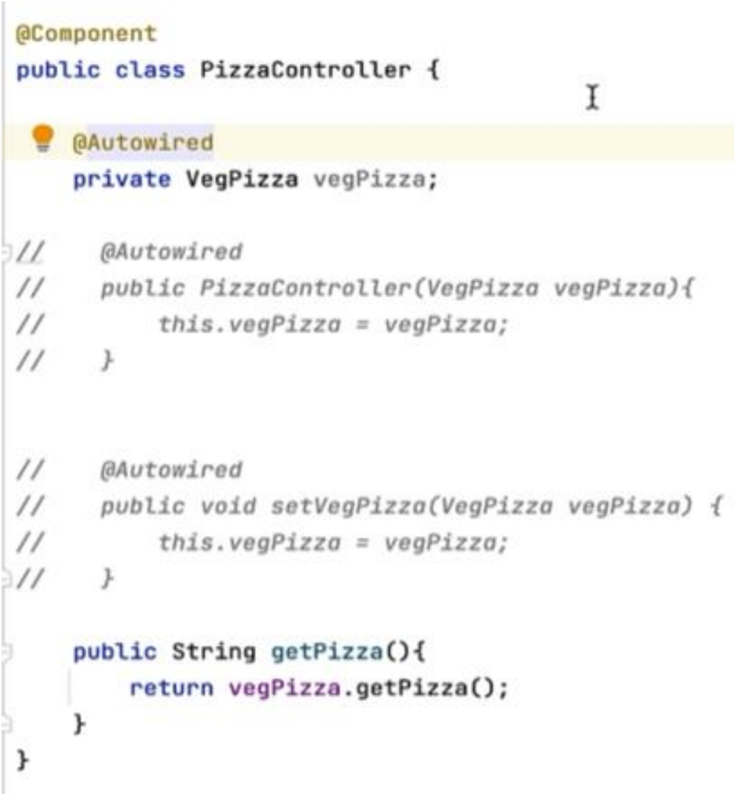
The @Autowired annotation is used in constructor injection, setter injection and field injection

It is used in annotation based configuration.

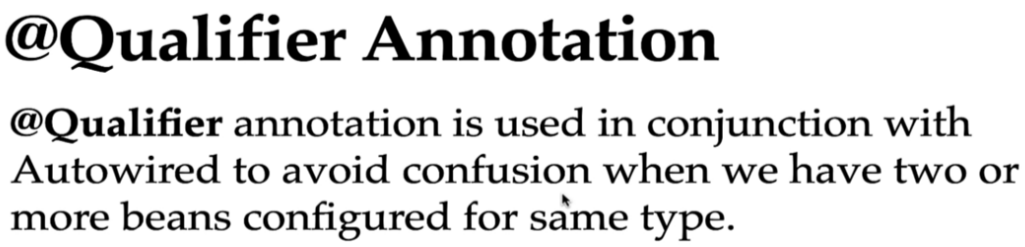
Ex -



```java
@Component
public class VegPizza {

    public String getPizza(){
        return "Veg Pizza!";
    }
}
```



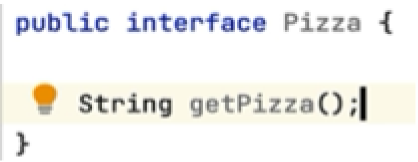
```java
@Component
public class PizzaController {

    @Autowired
    private VegPizza vegPizza;

//    @Autowired
//    public PizzaController(VegPizza vegPizza){
//        this.vegPizza = vegPizza;
//    }


//    @Autowired
//    public void setVegPizza(VegPizza vegPizza) {
//        this.vegPizza = vegPizza;
//    }

    public String getPizza(){
        return vegPizza.getPizza();
    }
}
```



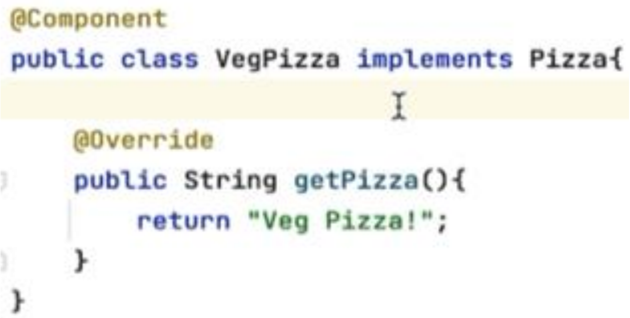
@Qualifier Annotation

@Qualifier annotation is used in conjunction with Autowired to avoid confusion when we have two or more beans configured for same type.

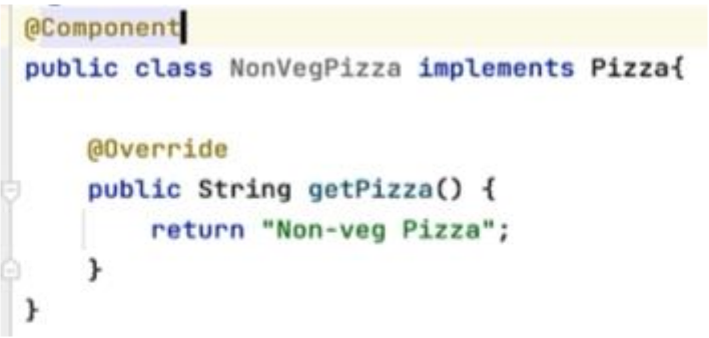
Ex –



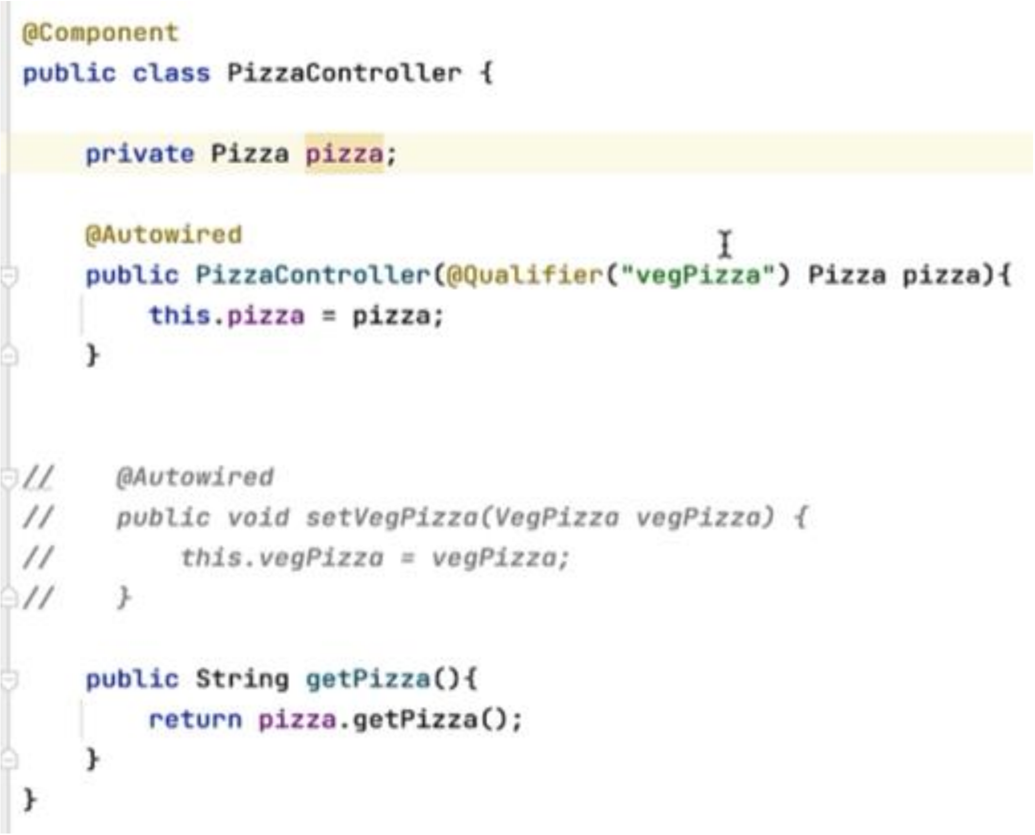
```java
public interface Pizza {

    String getPizza();
}
```



```java
@Component
public class VegPizza implements Pizza{

    @Override
    public String getPizza(){
        return "Veg Pizza!";
    }
}
```

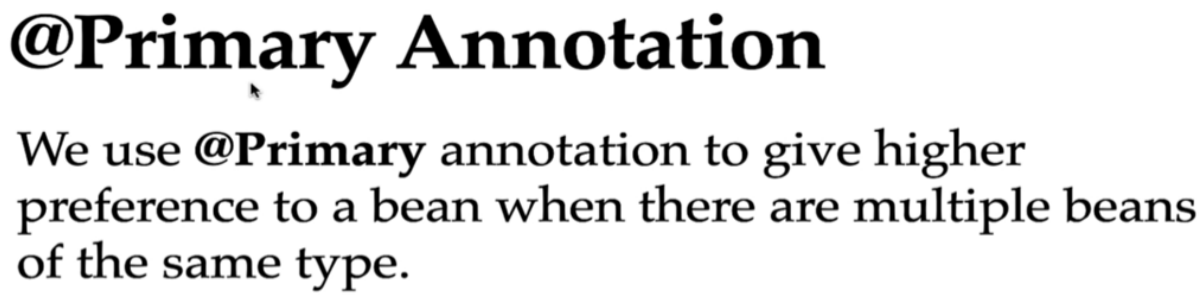


```java
@Component
public class NonVegPizza implements Pizza{

    @Override
    public String getPizza() {
        return "Non-veg Pizza";
    }
}
```



```java
@Component
public class PizzaController {

    private Pizza pizza;

    @Autowired
    public PizzaController(@Qualifier("vegPizza") Pizza pizza){
        this.pizza = pizza;
    }


//    @Autowired
//    public void setVegPizza(VegPizza vegPizza) {
//        this.vegPizza = vegPizza;
//    }

    public String getPizza(){
        return pizza.getPizza();
    }
}
```

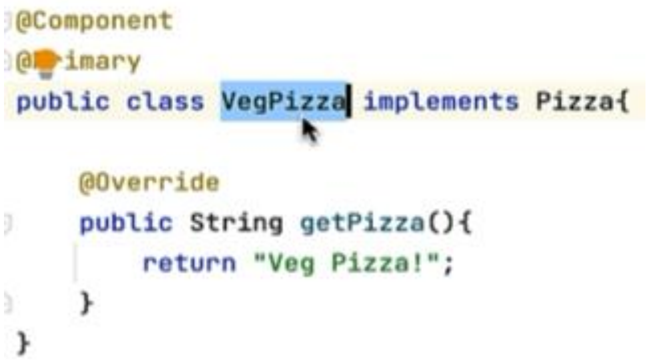
Without @Qualifier annotation, spring IOC container will get confused to inject which Pizza implementation Veg or Nonveg. So we explicitly mentioned it.



@Primary Annotation

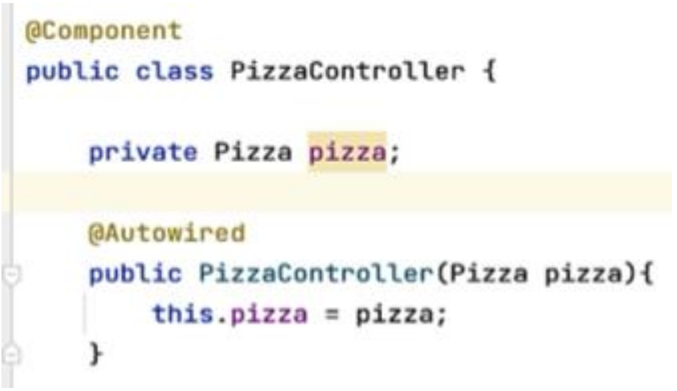
We use **@Primary** annotation to give higher preference to a bean when there are multiple beans of the same type.

Its alternative to @Qualifier annotation.

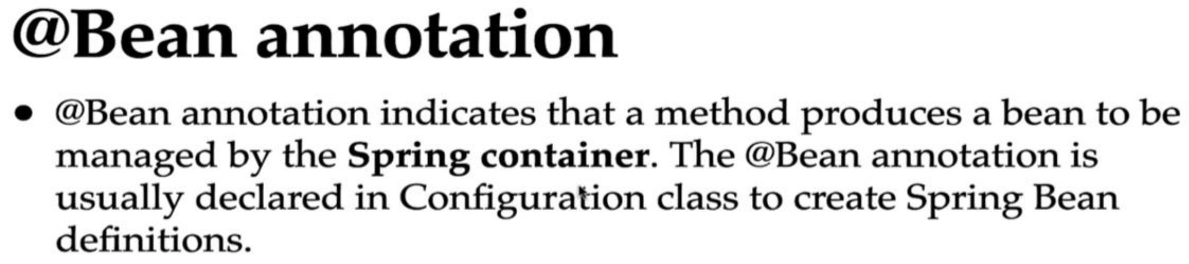


```
@Component
@Primary
public class VegPizza implements Pizza{

    @Override
    public String getPizza(){
        return "Veg Pizza!";
    }
}
```

Ex –



```
@Component
public class PizzaController {

    private Pizza pizza;

    @Autowired
    public PizzaController(Pizza pizza){
        this.pizza = pizza;
    }
```



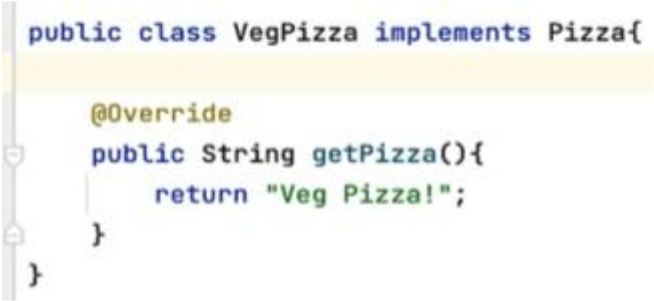
@Bean annotation

- @Bean annotation indicates that a method produces a bean to be managed by the **Spring container**. The @Bean annotation is usually declared in Configuration class to create Spring Bean definitions.

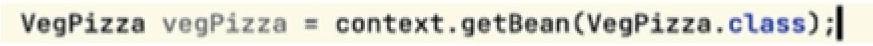
Whenever we use @Configuration annotation, that class becomes a configuration class and within that class we can define spring bean configurations using @Bean annotation. This is java based configuration.

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

Ex –

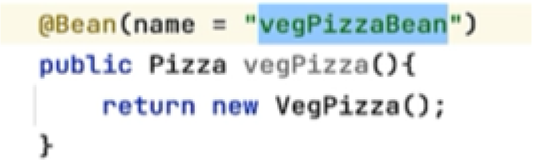


```java
public class VegPizza implements Pizza{

    @Override
    public String getPizza(){
        return "Veg Pizza!";
    }
}
```

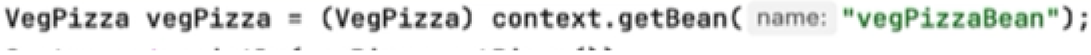


```java
VegPizza vegPizza = context.getBean(VegPizza.class);
```

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

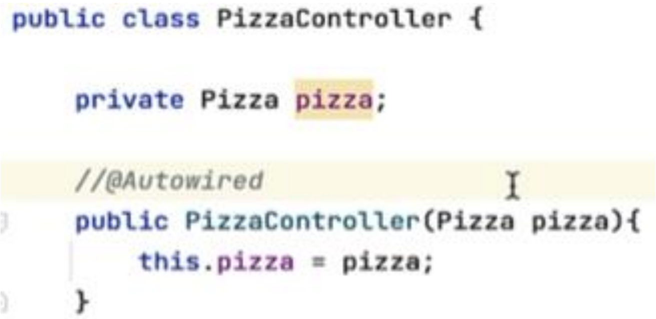


```java
@Bean(name = "vegPizzaBean")
public Pizza vegPizza(){
    return new VegPizza();
}
```

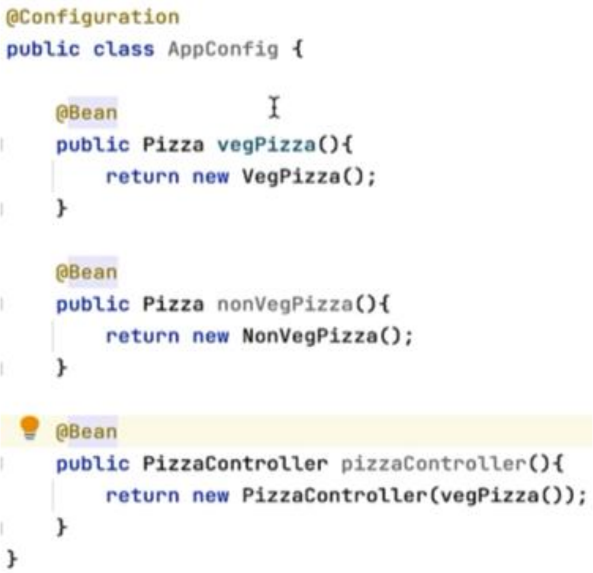


```java
VegPizza vegPizza = (VegPizza) context.getBean( name: "vegPizzaBean");
```

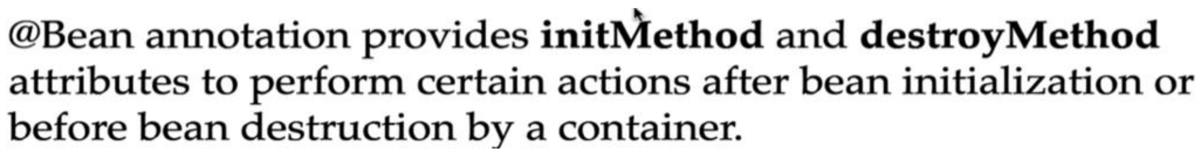
For PizzaController, remove @Autowired as we'll use java based configuration –



```java
public class PizzaController {

    private Pizza pizza;

    //@Autowired
    public PizzaController(Pizza pizza){
        this.pizza = pizza;
    }
```

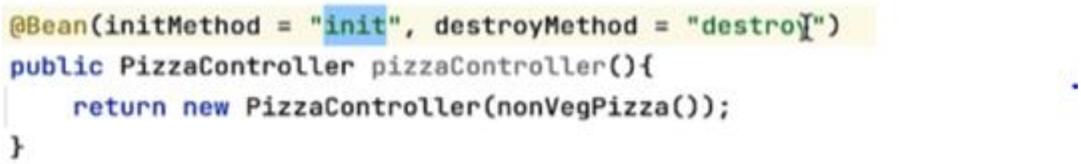


```java
@Configuration
public class AppConfig {

    @Bean
    public Pizza vegPizza(){
        return new VegPizza();
    }

    @Bean
    public Pizza nonVegPizza(){
        return new NonVegPizza();
    }

    @Bean
    public PizzaController pizzaController(){
        return new PizzaController(vegPizza());
    }
}
```



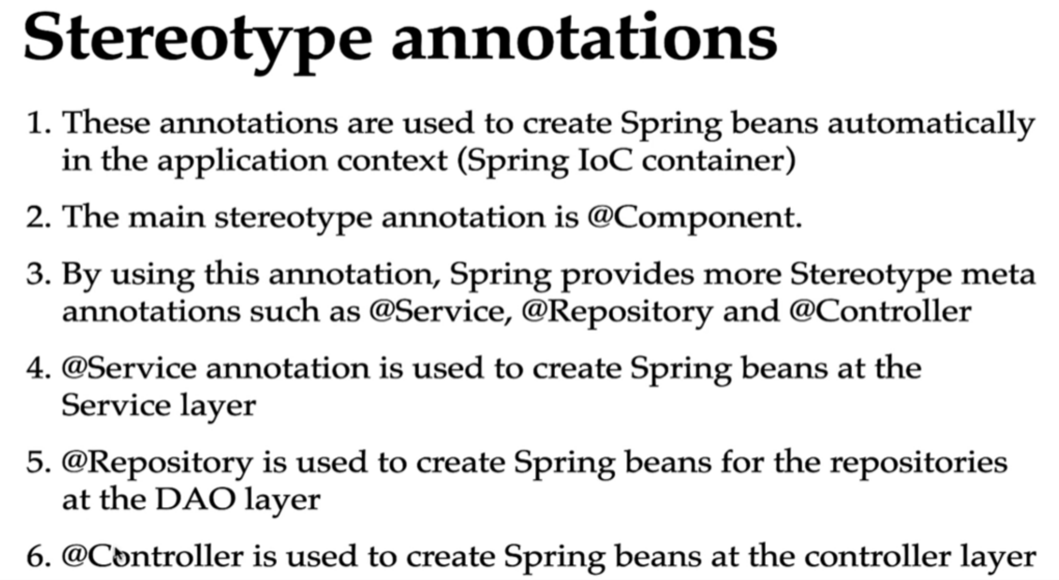
@Bean annotation provides **initMethod** and **destroyMethod** attributes to perform certain actions after bean initialization or before bean destruction by a container.

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



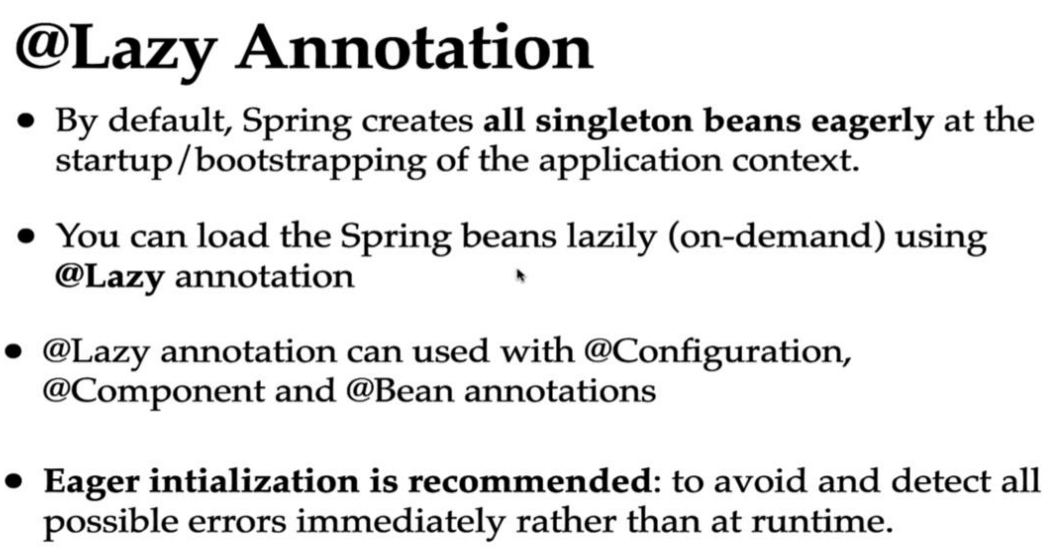
```
@Bean(initMethod = "init", destroyMethod = "destroy")
public PizzaController pizzaController(){
    return new PizzaController(nonVegPizza());
}
```

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



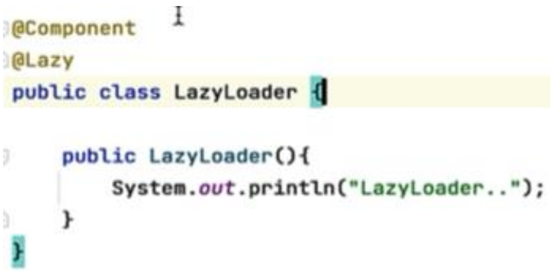
Stereotype annotations

1. These annotations are used to create Spring beans automatically in the application context (Spring IoC container)
2. The main stereotype annotation is @Component.
3. By using this annotation, Spring provides more Stereotype meta annotations such as @Service, @Repository and @Controller
4. @Service annotation is used to create Spring beans at the Service layer
5. @Repository is used to create Spring beans for the repositories at the DAO layer
6. @Controller is used to create Spring beans at the controller layer

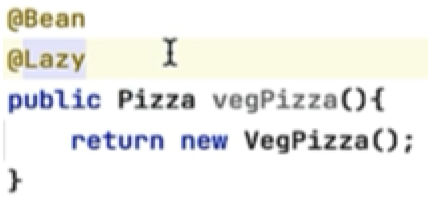


@Lazy Annotation

- By default, Spring creates **all singleton beans eagerly** at the startup/bootstrapping of the application context.
- You can load the Spring beans lazily (on-demand) using **@Lazy** annotation
- @Lazy annotation can used with @Configuration, @Component and @Bean annotations
- **Eager intialization is recommended**: to avoid and detect all possible errors immediately rather than at runtime.

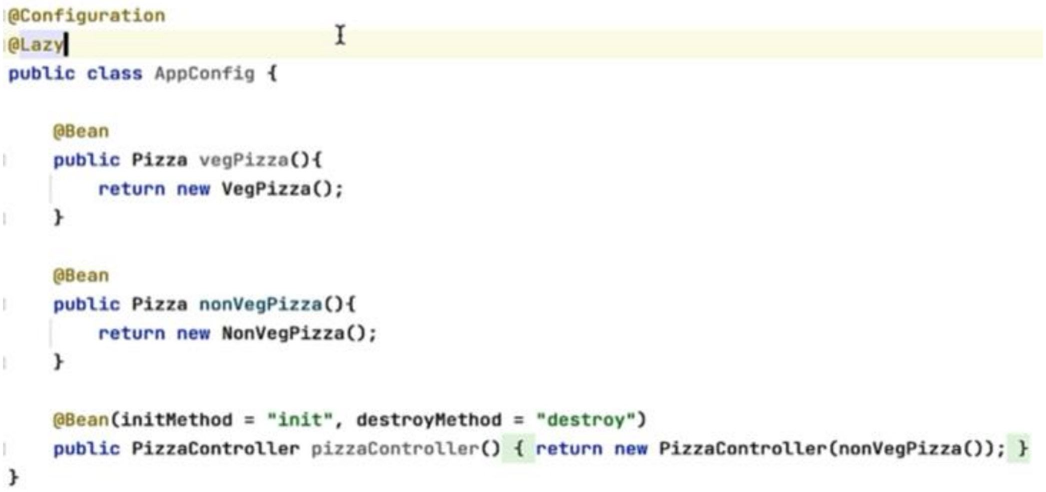


```
@Component
@Lazy
public class LazyLoader {

    public LazyLoader(){
        System.out.println("LazyLoader..");
    }
}
```



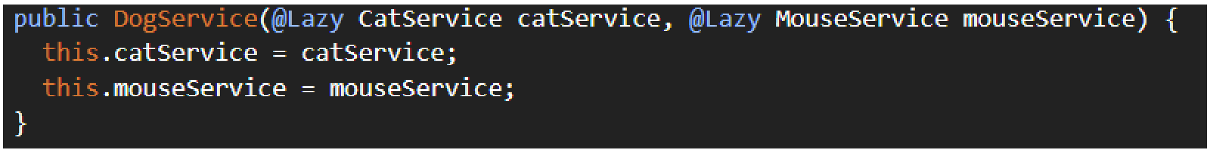
```
@Bean
@Lazy
public Pizza vegPizza(){
    return new VegPizza();
}
```

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



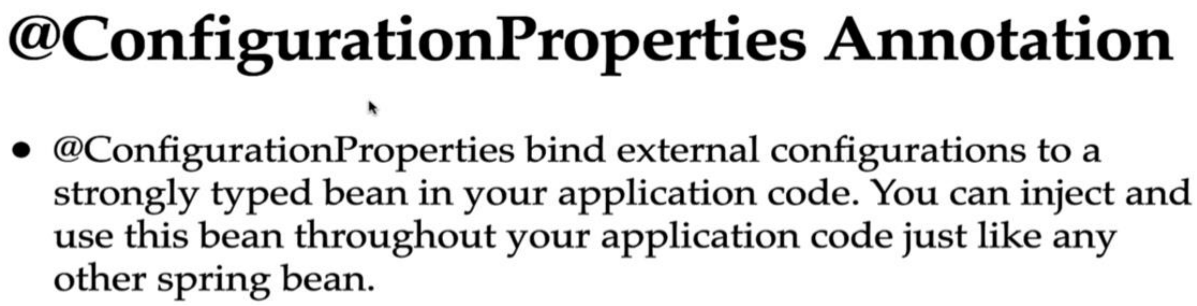
```
@Configuration
@Lazy
public class AppConfig {

    @Bean
    public Pizza vegPizza(){
        return new VegPizza();
    }

    @Bean
    public Pizza nonVegPizza(){
        return new NonVegPizza();
    }

    @Bean(initMethod = "init", destroyMethod = "destroy")
    public PizzaController pizzaController() { return new PizzaController(nonVegPizza()); }
}
```

You can use @Lazy on autowired constructor argument –



```
public DogService(@Lazy CatService catService, @Lazy MouseService mouseService) {
  this.catService = catService;
  this.mouseService = mouseService;
}
```

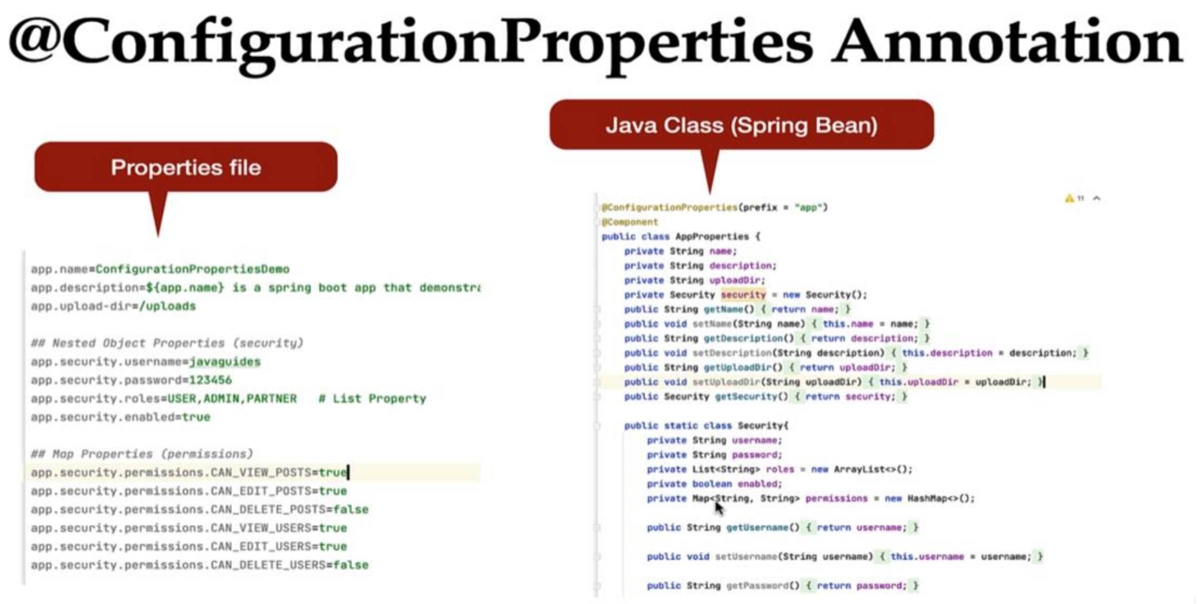
Spring does not initialize CatService and MouseService immediately instead injects a proxy object. And when the proxy is accessed, the actual bean is created.



@ConfigurationProperties Annotation

- @ConfigurationProperties bind external configurations to a strongly typed bean in your application code. You can inject and use this bean throughout your application code just like any other spring bean.

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



@ConfigurationProperties Annotation

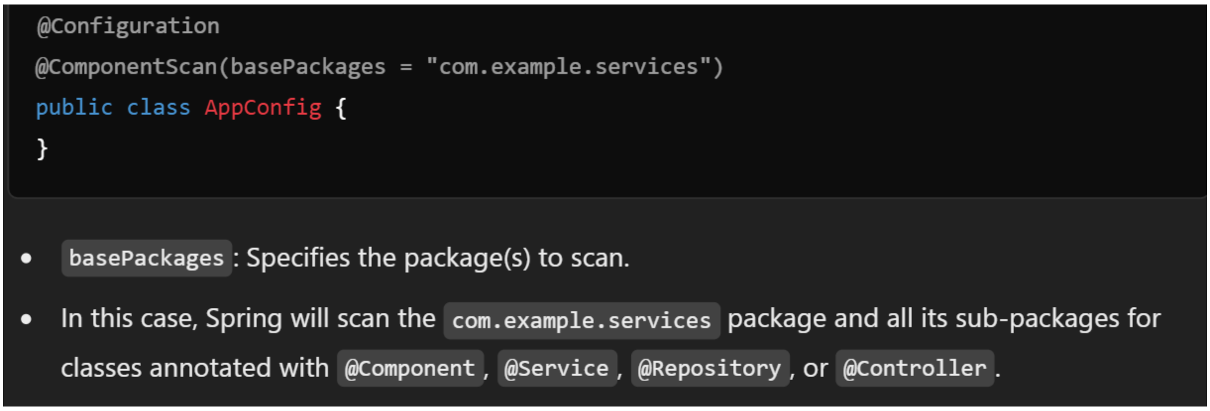
```
app.name=ConfigurationPropertiesDemo
app.description=${app.name} is a spring boot app that demonstra
app.upload-dir=/uploads

## Nested Object Properties (security)
app.security.username=javaguides
app.security.password=123456
app.security.roles=USER,ADMIN,PARTNER   # List Property
app.security.enabled=true

## Map Properties (permissions)
app.security.permissions.CAN_VIEW_POSTS=true
app.security.permissions.CAN_EDIT_POSTS=true
app.security.permissions.CAN_DELETE_POSTS=false
app.security.permissions.CAN_VIEW_USERS=true
app.security.permissions.CAN_EDIT_USERS=true
app.security.permissions.CAN_DELETE_USERS=false
```

```
@ConfigurationProperties(prefix = "app")
@Component
public class AppProperties {
    private String name;
    private String description;
    private String uploadDir;
    private Security security = new Security();
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
    public String getUploadDir() { return uploadDir; }
    public void setUploadDir(String uploadDir) { this.uploadDir = uploadDir; }
    public Security getSecurity() { return security; }

    public static class Security{
        private String username;
        private String password;
        private List<String> roles = new ArrayList<>();
        private boolean enabled;
        private Map<String, String> permissions = new HashMap<>();

        public String getUsername() { return username; }

        public void setUsername(String username) { this.username = username; }

        public String getPassword() { return password; }
```

@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.

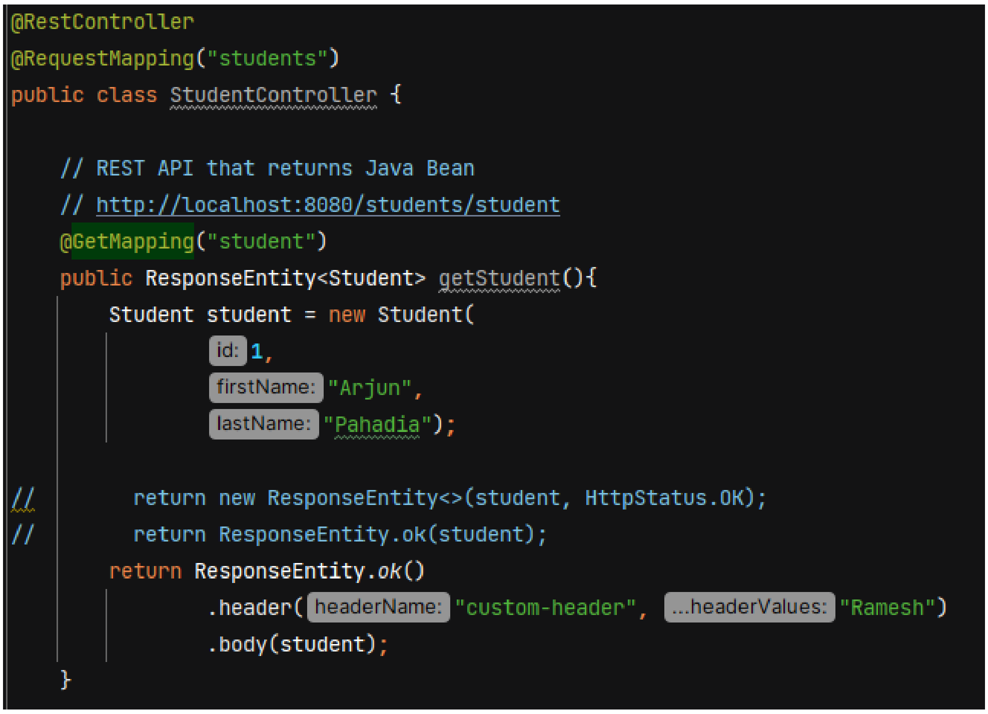
It is used in conjunction with Java-based configuration (@Configuration)



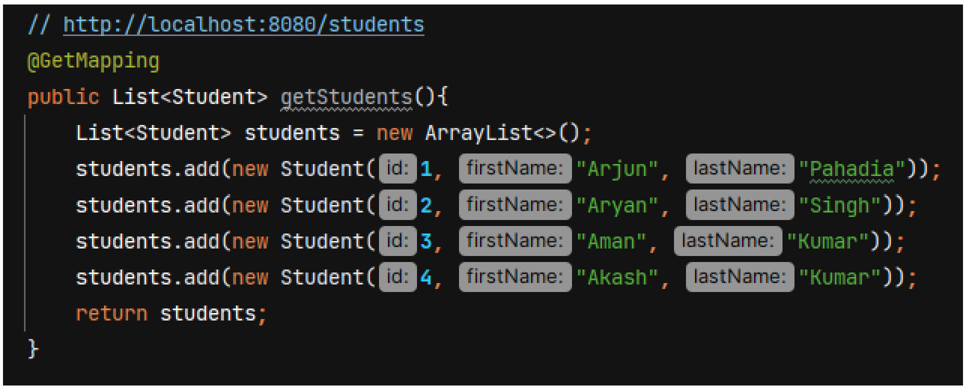
```
@Configuration
@ComponentScan(basePackages = "com.example.services")
public class AppConfig {
}
```

- `basePackages`: Specifies the package(s) to scan.
- In this case, Spring will scan the `com.example.services` package and all its sub-packages for classes annotated with `@Component`, `@Service`, `@Repository`, or `@Controller`.

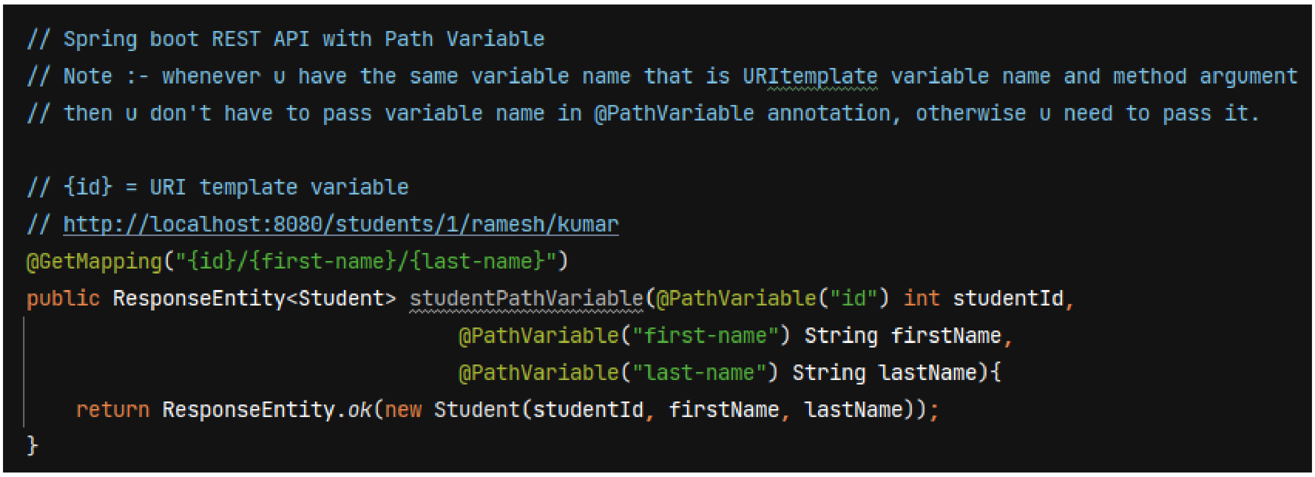
REST APIs in SpringBoot



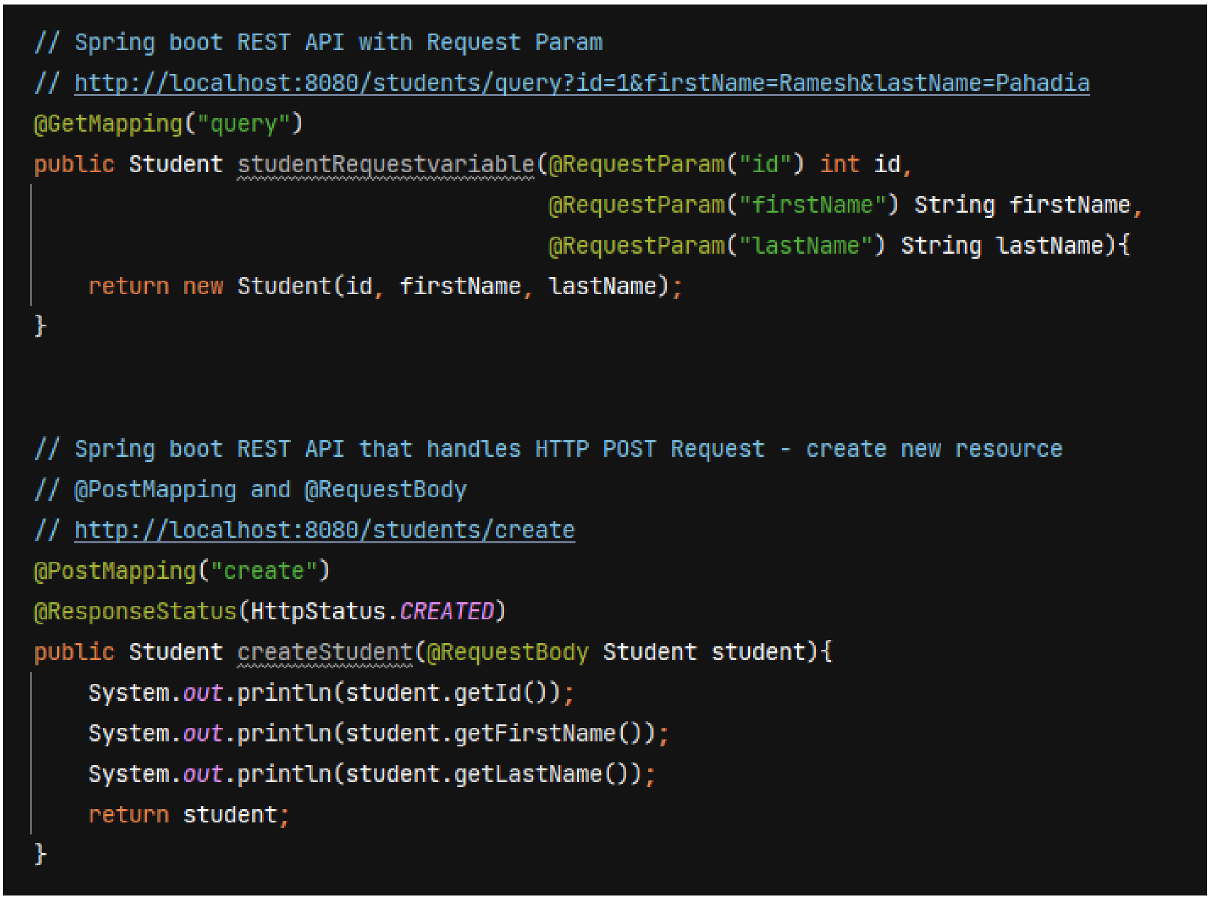
```
@RestController
@RequestMapping("students")
public class StudentController {

    // REST API that returns Java Bean
    // http://localhost:8080/students/student
    @GetMapping("student")
    public ResponseEntity<Student> getStudent(){
        Student student = new Student(
                id: 1,
                firstName: "Arjun",
                lastName: "Pahadia");

//          return new ResponseEntity<>(student, HttpStatus.OK);
//          return ResponseEntity.ok(student);
        return ResponseEntity.ok()
                .header( headerName: "custom-header",  ...headerValues: "Ramesh")
                .body(student);
    }
```



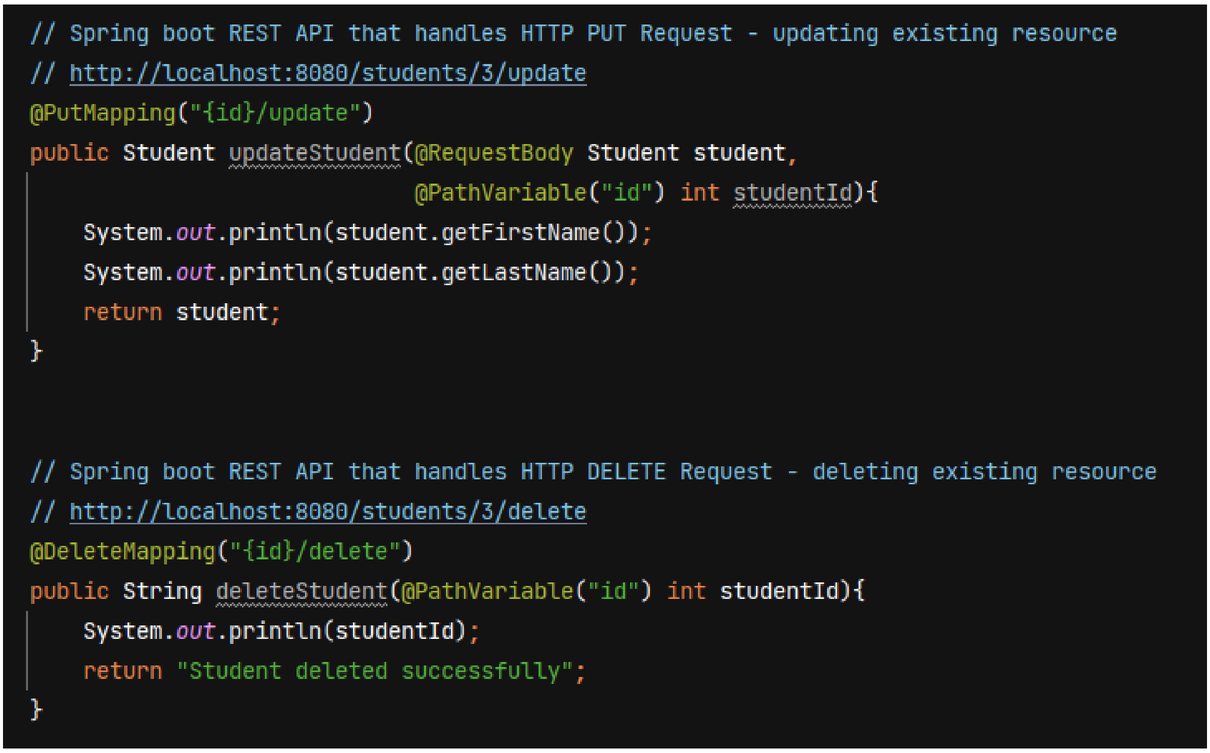
```
// http://localhost:8080/students
@GetMapping
public List<Student> getStudents(){
    List<Student> students = new ArrayList<>();
    students.add(new Student( id: 1,  firstName: "Arjun",  lastName: "Pahadia"));
    students.add(new Student( id: 2,  firstName: "Aryan",  lastName: "Singh"));
    students.add(new Student( id: 3,  firstName: "Aman",  lastName: "Kumar"));
    students.add(new Student( id: 4,  firstName: "Akash",  lastName: "Kumar"));
    return students;
}
```



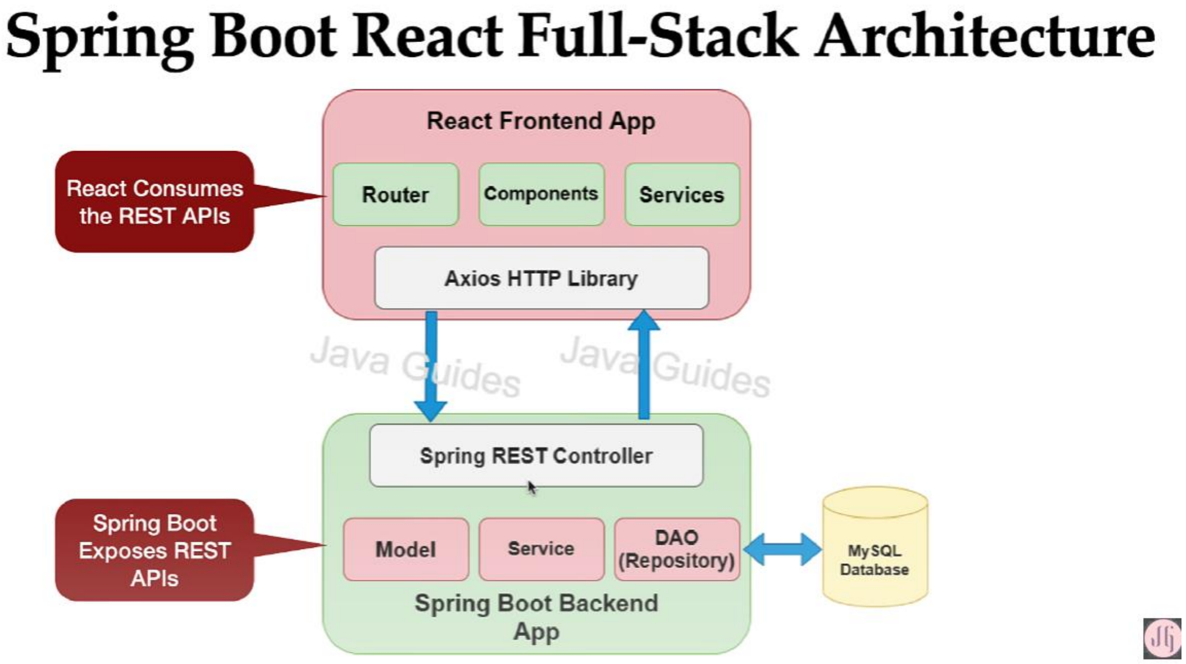
```
// Spring boot REST API with Path Variable
// Note :- whenever u have the same variable name that is URItemplate variable name and method argument
// then u don't have to pass variable name in @PathVariable annotation, otherwise u need to pass it.

// {id} = URI template variable
// http://localhost:8080/students/1/ramesh/kumar
@GetMapping("{id}/{first-name}/{last-name}")
public ResponseEntity<Student> studentPathVariable(@PathVariable("id") int studentId,
                                    @PathVariable("first-name") String firstName,
                                    @PathVariable("last-name") String lastName){
    return ResponseEntity.ok(new Student(studentId, firstName, lastName));
}
```



```java
// Spring boot REST API with Request Param
// http://localhost:8080/students/query?id=1&firstName=Ramesh&lastName=Pahadia
@GetMapping("query")
public Student studentRequestvariable(@RequestParam("id") int id,
                                      @RequestParam("firstName") String firstName,
                                      @RequestParam("lastName") String lastName){
    return new Student(id, firstName, lastName);
}


// Spring boot REST API that handles HTTP POST Request - create new resource
// @PostMapping and @RequestBody
// http://localhost:8080/students/create
@PostMapping("create")
@ResponseStatus(HttpStatus.CREATED)
public Student createStudent(@RequestBody Student student){
    System.out.println(student.getId());
    System.out.println(student.getFirstName());
    System.out.println(student.getLastName());
    return student;
}
```



```java
// Spring boot REST API that handles HTTP PUT Request - updating existing resource
// http://localhost:8080/students/3/update
@PutMapping("{id}/update")
public Student updateStudent(@RequestBody Student student,
                             @PathVariable("id") int studentId){
    System.out.println(student.getFirstName());
    System.out.println(student.getLastName());
    return student;
}


// Spring boot REST API that handles HTTP DELETE Request - deleting existing resource
// http://localhost:8080/students/3/delete
@DeleteMapping("{id}/delete")
public String deleteStudent(@PathVariable("id") int studentId){
    System.out.println(studentId);
    return "Student deleted successfully";
}
```



Spring Boot React Full-Stack Architecture

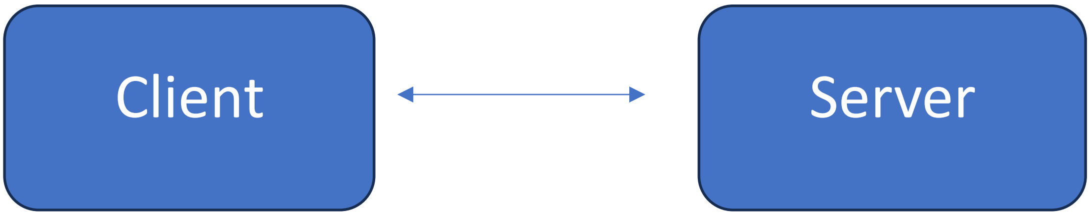
Both react frontend app and springboot backend app are loosely coupled.

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.

<u>DTO (Data Transfer Object)</u>

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



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.

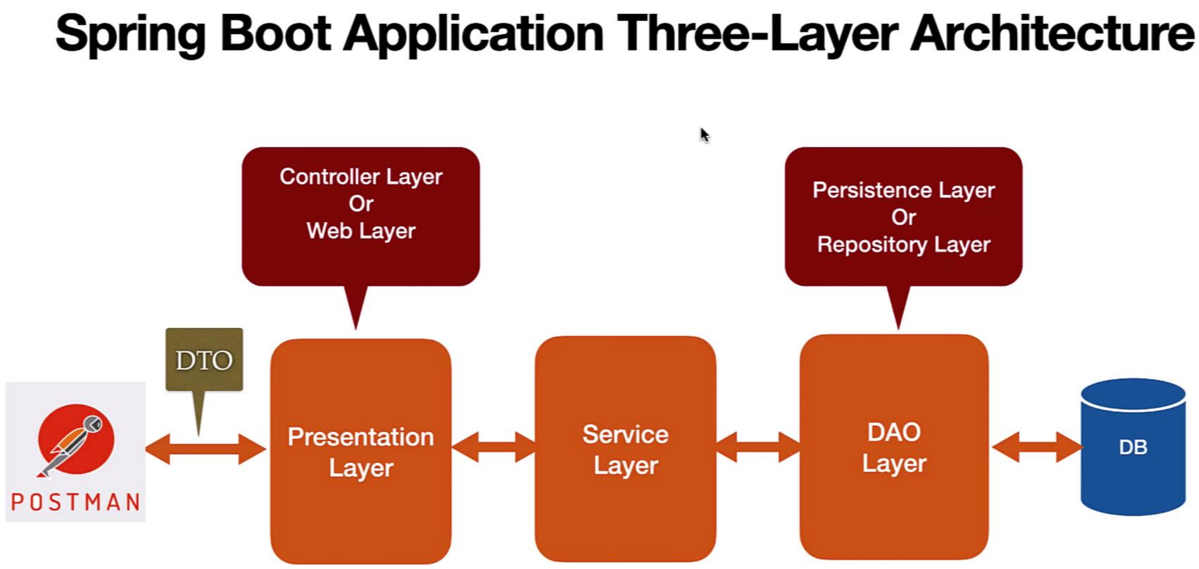
How to use DTO pattern in Spring Boot

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.

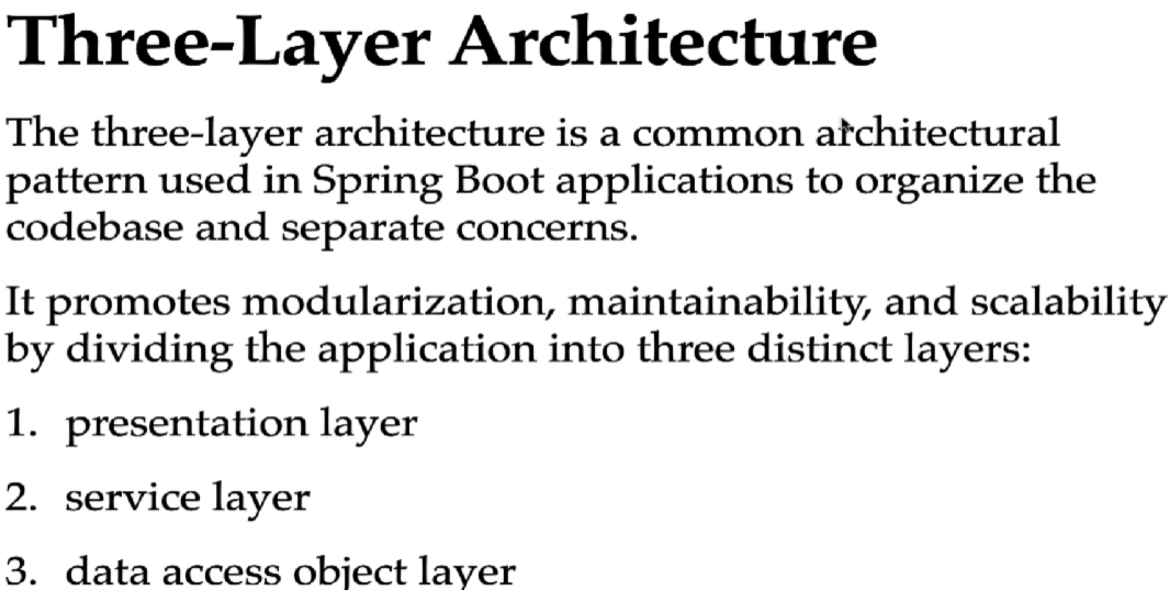
Client and Server uses DTO to transfer data. Some developers uses JPA entitites to transfer to data between client and server. But transfering JPA entitity has some disadvantages –

- 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.



Spring Boot Application Three-Layer Architecture



Three-Layer Architecture

The three-layer architecture is a common architectural pattern used in Spring Boot applications to organize the codebase and separate concerns.

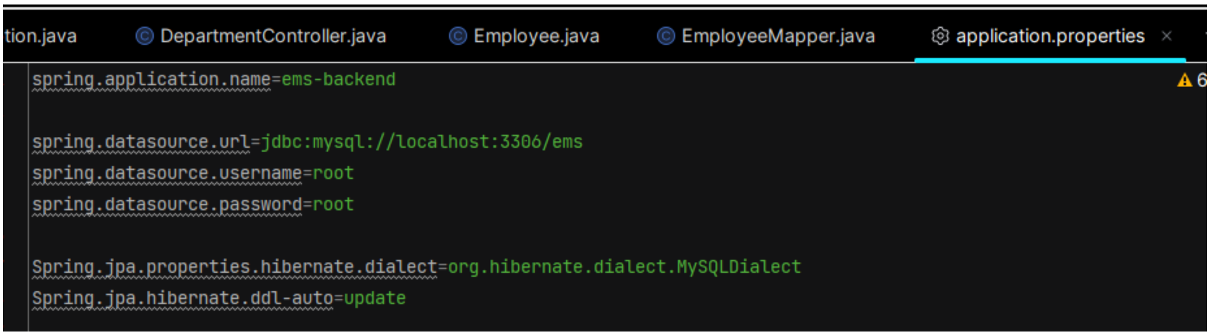
It promotes modularization, maintainability, and scalability by dividing the application into three distinct layers:

1. presentation layer
2. service layer
3. data access object layer

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.



```
spring.application.name=ems-backend

spring.datasource.url=jdbc:mysql://localhost:3306/ems
spring.datasource.username=root
spring.datasource.password=root

Spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
Spring.jpa.hibernate.ddl-auto=update
```

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

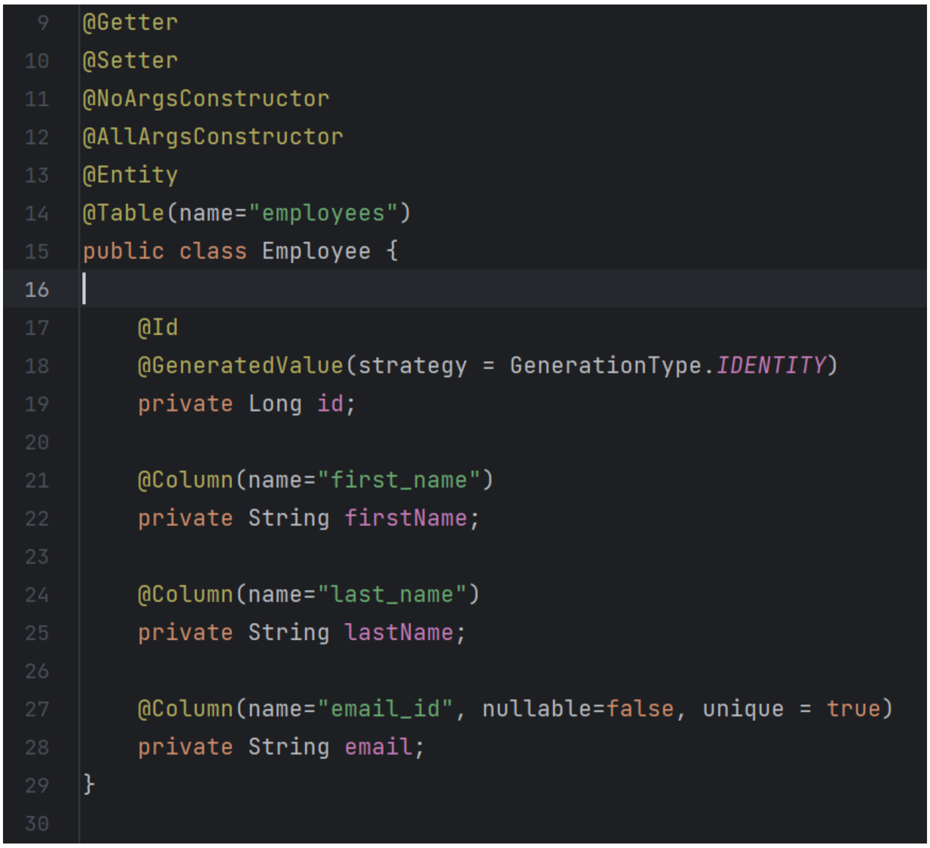
@GeneratedValue(strategy = GenerationType.*IDENTITY*)

- 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)

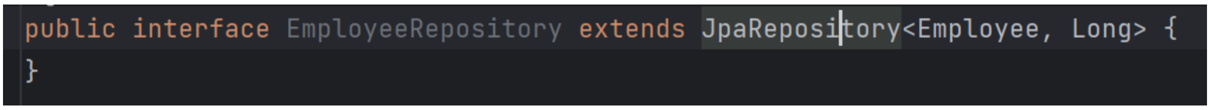
- to map a database table column with a class field. If u don't mention it then JPA willl by default give column name as field name
- nullable = false makes the column value not null
- unique = true to make column value unique

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



```java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name="employees")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name="first_name")
    private String firstName;

    @Column(name="last_name")
    private String lastName;

    @Column(name="email_id", nullable=false, unique = true)
    private String email;
}

```

<u>Creating EmployeeRepository</u> – create interface EmployeeRepository in repository folder



```java
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
}
```

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

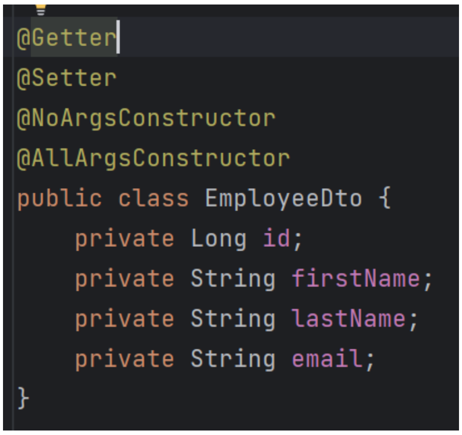
- JpaRepository will inherit all the methods from all the entended interfaces. SimpleJpaRepository class of Spring Data JPA provide the impl for JpaRepository interface.

- 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.

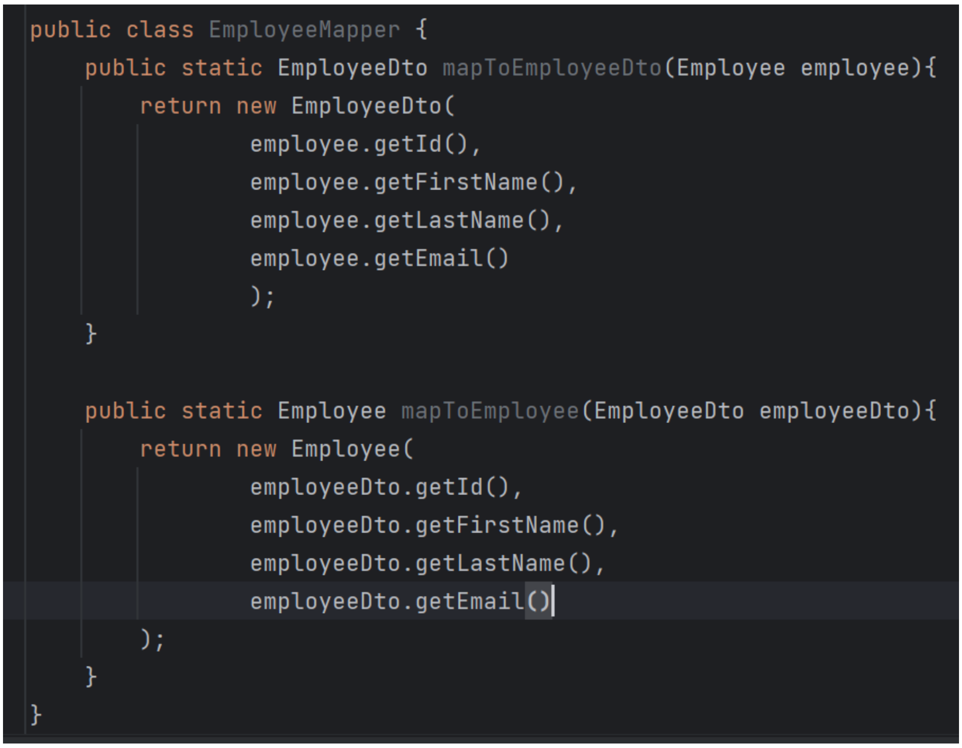
Create EmployeeDto and EmployeeMapper

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



```java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class EmployeeDto {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
}
```

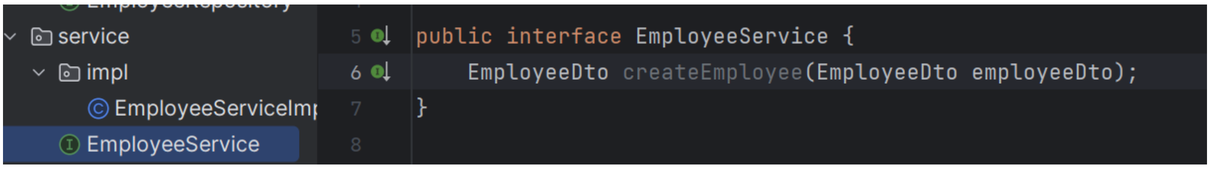
We create EmployeeMapper class to map Employee entity to EmployeeDto and EmployeeDto to EmployeeEntity.



```java
public class EmployeeMapper {
    public static EmployeeDto mapToEmployeeDto(Employee employee){
        return new EmployeeDto(
                employee.getId(),
                employee.getFirstName(),
                employee.getLastName(),
                employee.getEmail()
                );
    }

    public static Employee mapToEmployee(EmployeeDto employeeDto){
        return new Employee(
                employeeDto.getId(),
                employeeDto.getFirstName(),
                employeeDto.getLastName(),
                employeeDto.getEmail()
        );
    }
}
```

Build Add Employee REST API

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

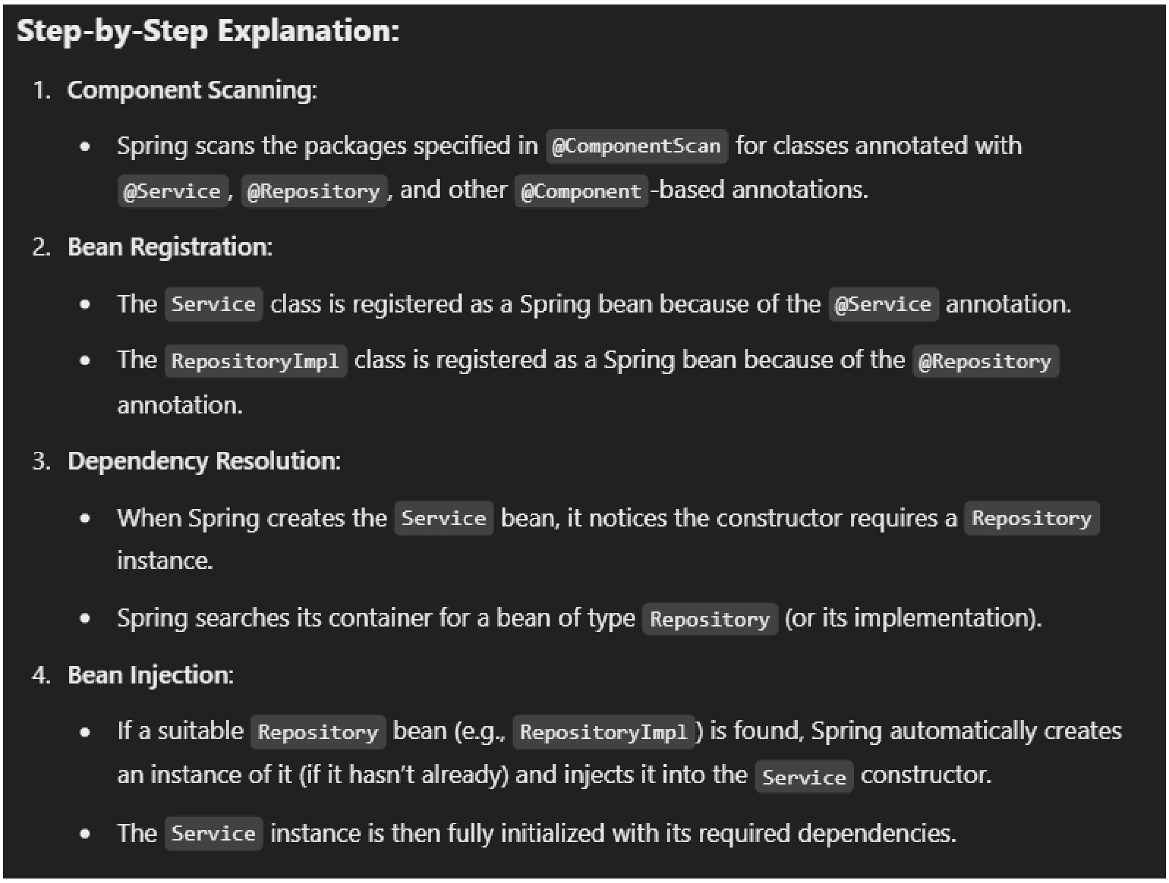


```java
public interface EmployeeService {
    EmployeeDto createEmployee(EmployeeDto employeeDto);
}
```

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.

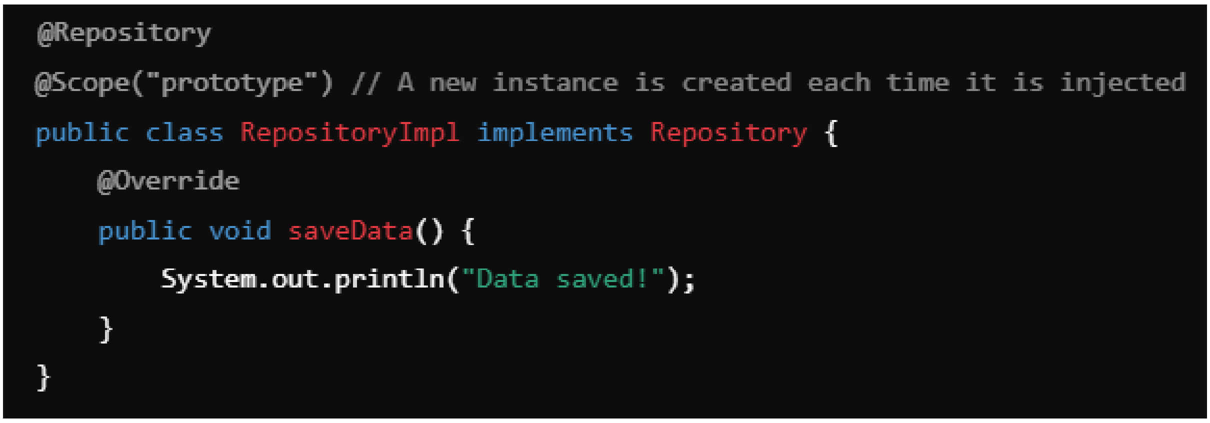
We'll use constructor based DI (when dependencies are provided through the class constructor) to inject the dependencies so annotate EmployeeServiceImpl with @AllArgsConstructor



Step-by-Step Explanation:

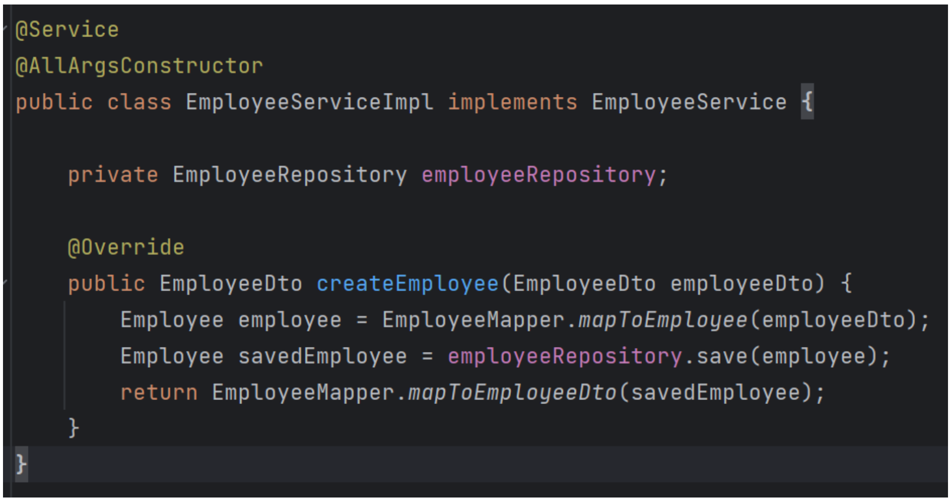
1. **Component Scanning**:
   - Spring scans the packages specified in `@ComponentScan` for classes annotated with `@Service`, `@Repository`, and other `@Component`-based annotations.
2. **Bean Registration**:
   - The `Service` class is registered as a Spring bean because of the `@Service` annotation.
   - The `RepositoryImpl` class is registered as a Spring bean because of the `@Repository` annotation.
3. **Dependency Resolution**:
   - When Spring creates the `Service` bean, it notices the constructor requires a `Repository` instance.
   - Spring searches its container for a bean of type `Repository` (or its implementation).
4. **Bean Injection**:
   - If a suitable `Repository` bean (e.g., `RepositoryImpl`) is found, Spring automatically creates an instance of it (if it hasn't already) and injects it into the `Service` constructor.
   - The `Service` instance is then fully initialized with its required dependencies.

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 –



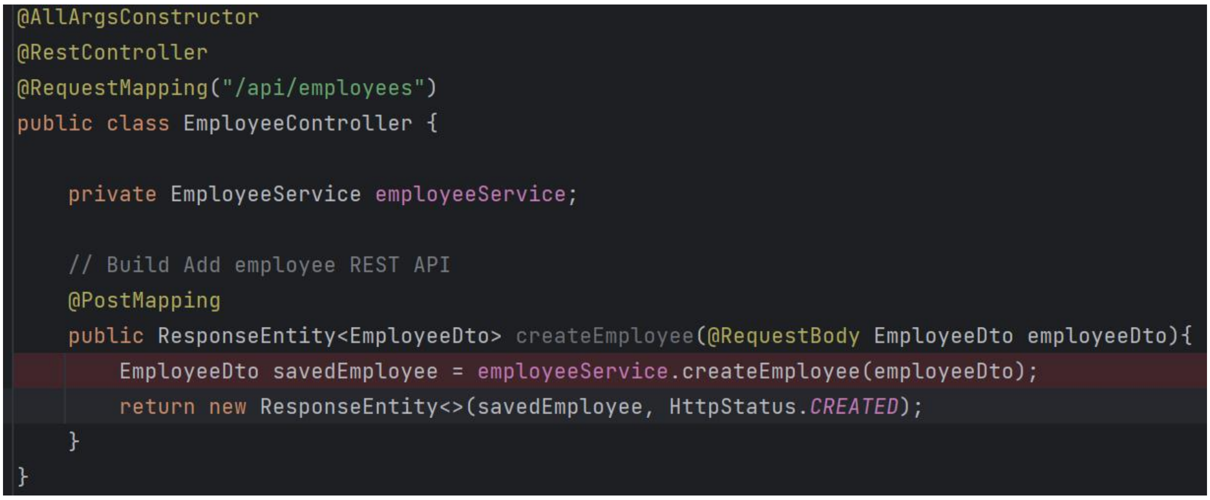
```java
@Repository
@Scope("prototype") // A new instance is created each time it is injected
public class RepositoryImpl implements Repository {
    @Override
    public void saveData() {
        System.out.println("Data saved!");
    }
}
```

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



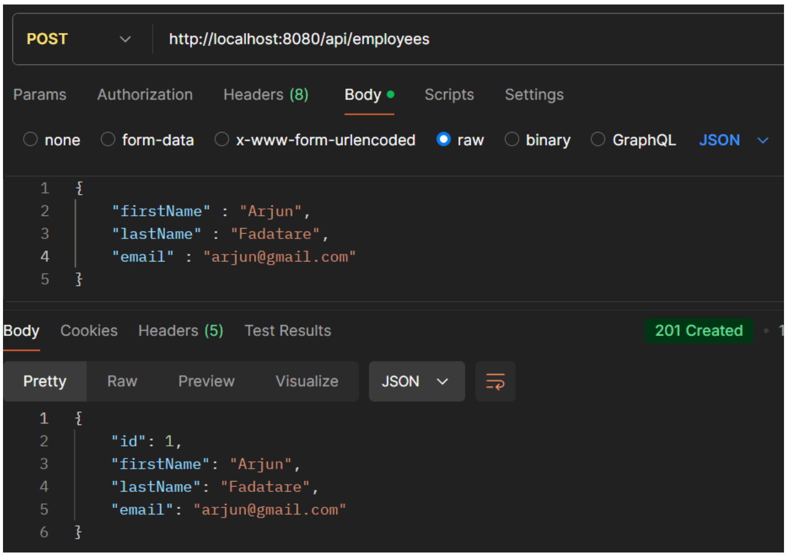
```
@Service
@AllArgsConstructor
public class EmployeeServiceImpl implements EmployeeService {

    private EmployeeRepository employeeRepository;

    @Override
    public EmployeeDto createEmployee(EmployeeDto employeeDto) {
        Employee employee = EmployeeMapper.mapToEmployee(employeeDto);
        Employee savedEmployee = employeeRepository.save(employee);
        return EmployeeMapper.mapToEmployeeDto(savedEmployee);
    }
}
```

Now let's create Controller –



```
@AllArgsConstructor
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    private EmployeeService employeeService;

    // Build Add employee REST API
    @PostMapping
    public ResponseEntity<EmployeeDto> createEmployee(@RequestBody EmployeeDto employeeDto){
        EmployeeDto savedEmployee = employeeService.createEmployee(employeeDto);
        return new ResponseEntity<>(savedEmployee, HttpStatus.CREATED);
    }
}
```

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



POST http://localhost:8080/api/employees

```
{
    "firstName" : "Arjun",
    "lastName" : "Fadatare",
    "email" : "arjun@gmail.com"
}
```

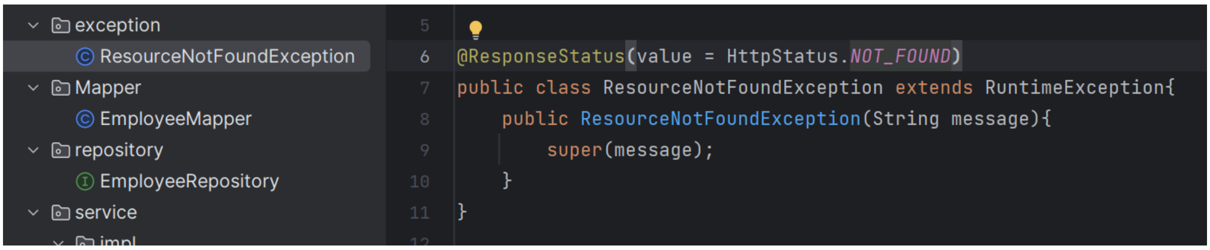
201 Created

```
{
    "id": 1,
    "firstName": "Arjun",
    "lastName": "Fadatare",
    "email": "arjun@gmail.com"
}
```

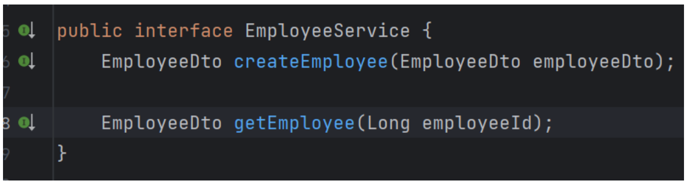
Build Get Employee REST API

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

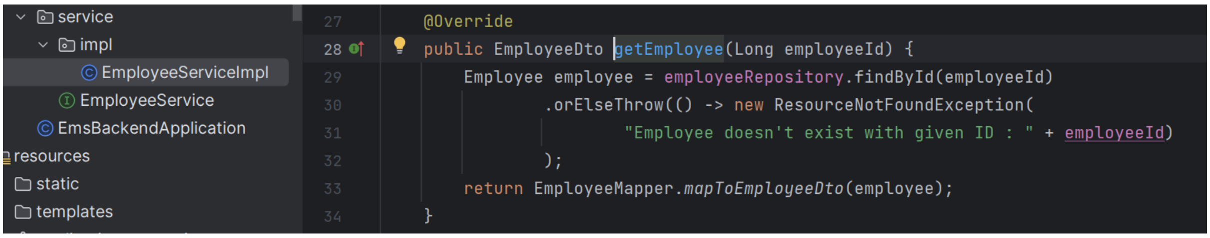
If a Employee with a given ID does not exist in DB, then throw custom ResourceNotFoundException and then springboot will catch this exception and will get the error message from exception, and it will send the error message along with the http status to client.



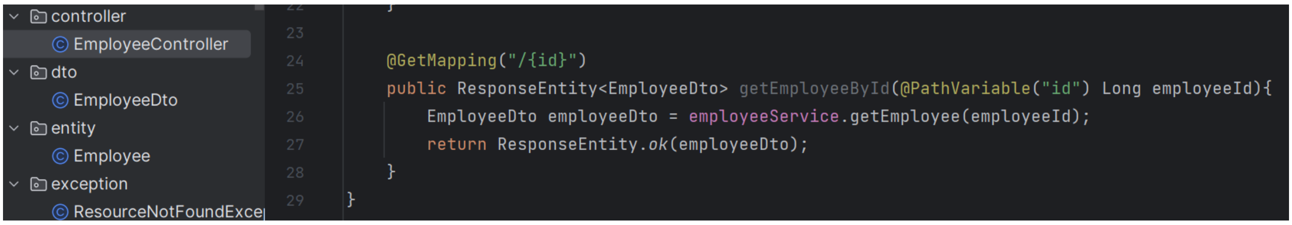
```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException{
    public ResourceNotFoundException(String message){
        super(message);
    }
}
```



```
public interface EmployeeService {
    EmployeeDto createEmployee(EmployeeDto employeeDto);

    EmployeeDto getEmployee(Long employeeId);
}
```

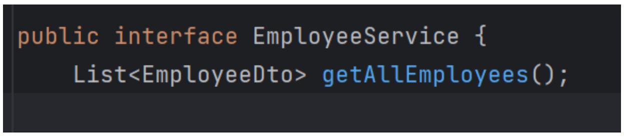


```
    @Override
    public EmployeeDto getEmployee(Long employeeId) {
        Employee employee = employeeRepository.findById(employeeId)
                .orElseThrow(() -> new ResourceNotFoundException(
                        "Employee doesn't exist with given ID : " + employeeId)
                );
        return EmployeeMapper.mapToEmployeeDto(employee);
    }
```

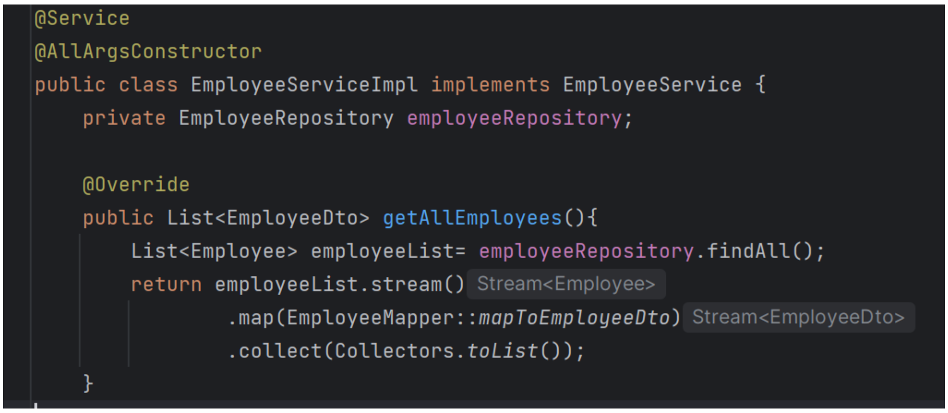


```
    }

    @GetMapping("/{id}")
    public ResponseEntity<EmployeeDto> getEmployeeById(@PathVariable("id") Long employeeId){
        EmployeeDto employeeDto = employeeService.getEmployee(employeeId);
        return ResponseEntity.ok(employeeDto);
    }
}
```

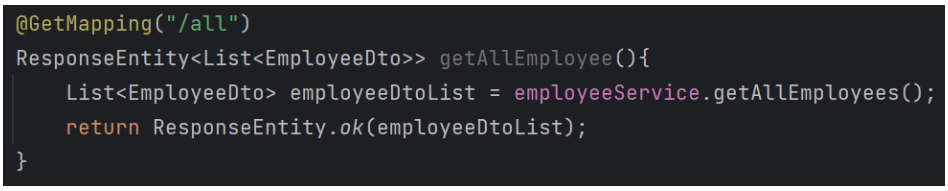
Build Get All employee REST API



```
public interface EmployeeService {
    List<EmployeeDto> getAllEmployees();
```



```
@Service
@AllArgsConstructor
public class EmployeeServiceImpl implements EmployeeService {
    private EmployeeRepository employeeRepository;

    @Override
    public List<EmployeeDto> getAllEmployees(){
        List<Employee> employeeList= employeeRepository.findAll();
        return employeeList.stream() Stream<Employee>
                .map(EmployeeMapper::mapToEmployeeDto) Stream<EmployeeDto>
                .collect(Collectors.toList());
    }
```

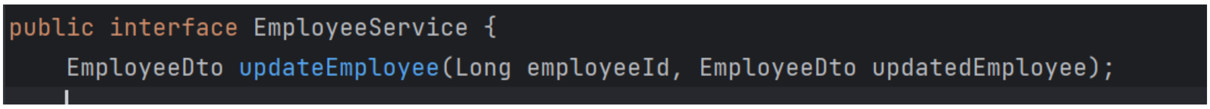


```
@GetMapping("/all")
ResponseEntity<List<EmployeeDto>> getAllEmployee(){
    List<EmployeeDto> employeeDtoList = employeeService.getAllEmployees();
    return ResponseEntity.ok(employeeDtoList);
}
```

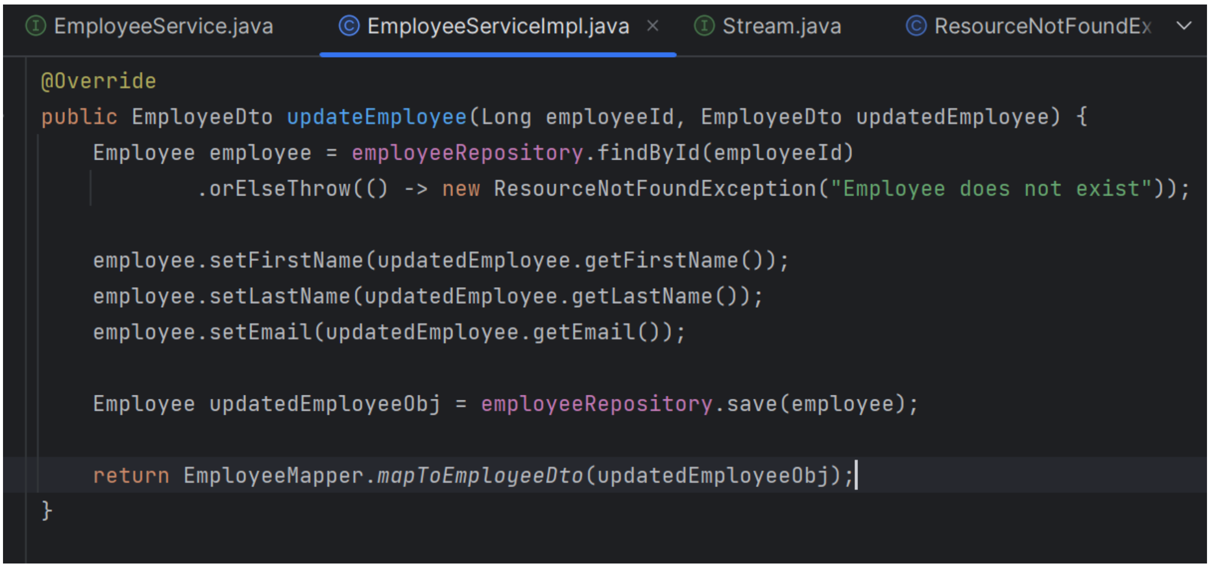
Build Update Employee REST API

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

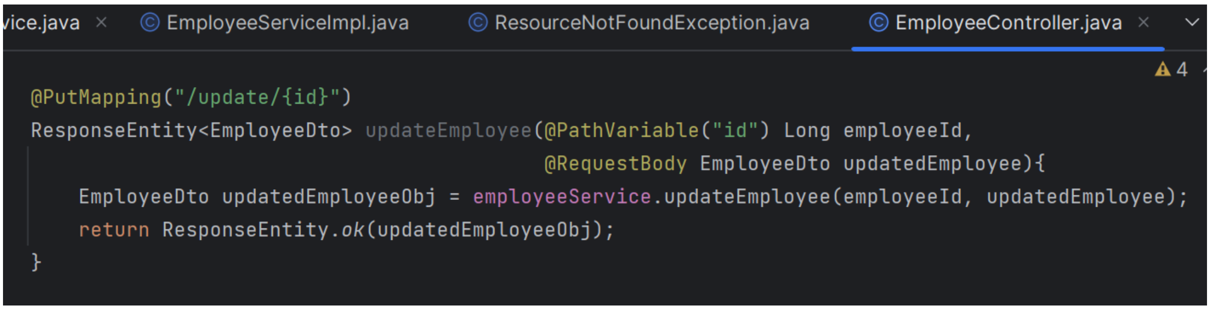
If employee object contains ID then the save method internally perform the update operation. And if employee doesn't contain the Primary Key ID, then it will perform the insert operation.



```
public interface EmployeeService {
    EmployeeDto updateEmployee(Long employeeId, EmployeeDto updatedEmployee);
```

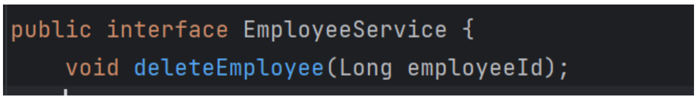


```
@Override
public EmployeeDto updateEmployee(Long employeeId, EmployeeDto updatedEmployee) {
    Employee employee = employeeRepository.findById(employeeId)
            .orElseThrow(() -> new ResourceNotFoundException("Employee does not exist"));

    employee.setFirstName(updatedEmployee.getFirstName());
    employee.setLastName(updatedEmployee.getLastName());
    employee.setEmail(updatedEmployee.getEmail());

    Employee updatedEmployeeObj = employeeRepository.save(employee);

    return EmployeeMapper.mapToEmployeeDto(updatedEmployeeObj);
}
```

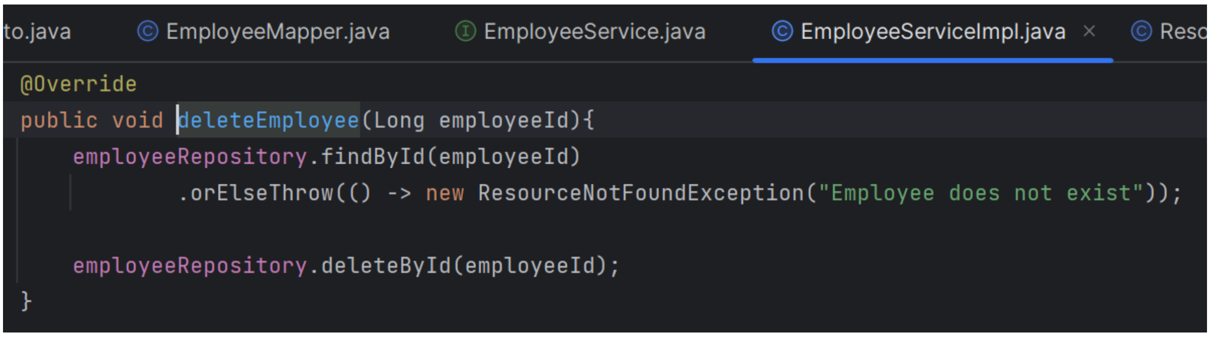


```
@PutMapping("/update/{id}")
ResponseEntity<EmployeeDto> updateEmployee(@PathVariable("id") Long employeeId,
                                           @RequestBody EmployeeDto updatedEmployee){
    EmployeeDto updatedEmployeeObj = employeeService.updateEmployee(employeeId, updatedEmployee);
    return ResponseEntity.ok(updatedEmployeeObj);
}
```

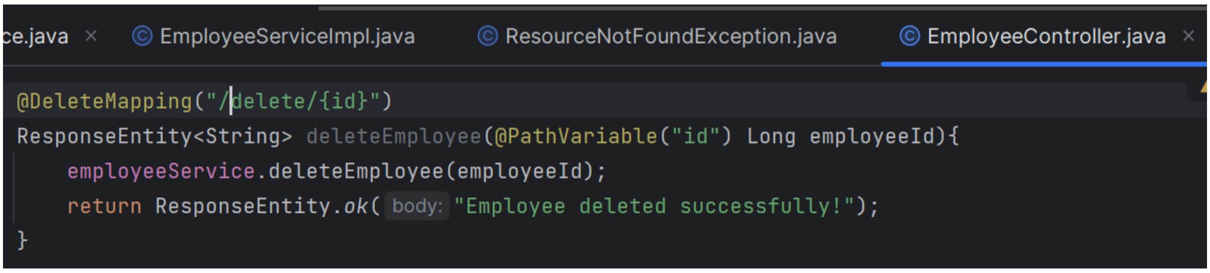
Build Delete Employee REST API



```
public interface EmployeeService {
    void deleteEmployee(Long employeeId);
```



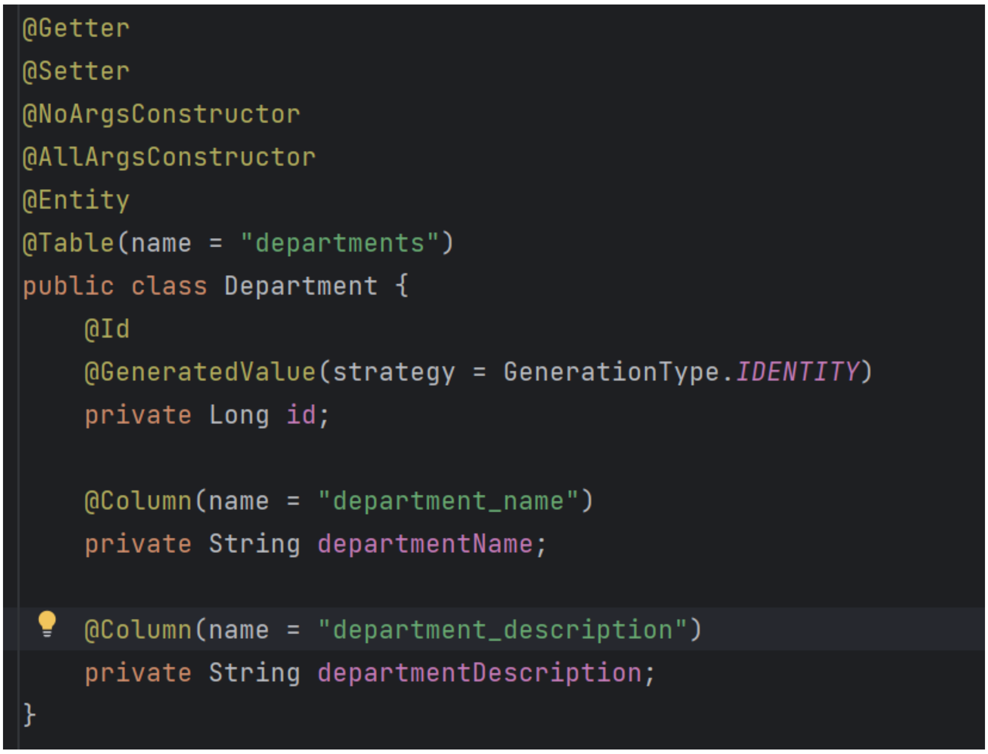
```
@Override
public void deleteEmployee(Long employeeId){
    employeeRepository.findById(employeeId)
            .orElseThrow(() -> new ResourceNotFoundException("Employee does not exist"));

    employeeRepository.deleteById(employeeId);
}
```



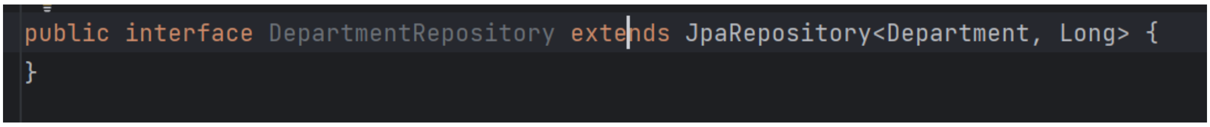
```
@DeleteMapping("/delete/{id}")
ResponseEntity<String> deleteEmployee(@PathVariable("id") Long employeeId){
    employeeService.deleteEmployee(employeeId);
    return ResponseEntity.ok( body: "Employee deleted successfully!");
}
```

Requirement 3 :- REST APIs For Department Management Module :

Create Department Entity & DepartmentRepository

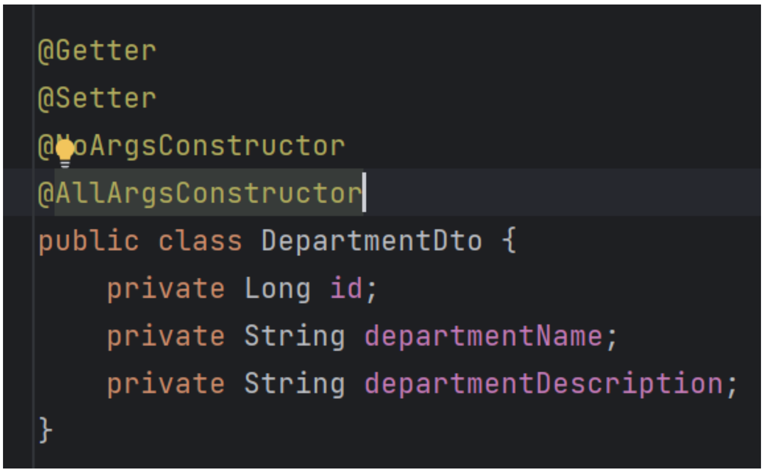


```java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "departments")
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "department_name")
    private String departmentName;

    @Column(name = "department_description")
    private String departmentDescription;
}
```

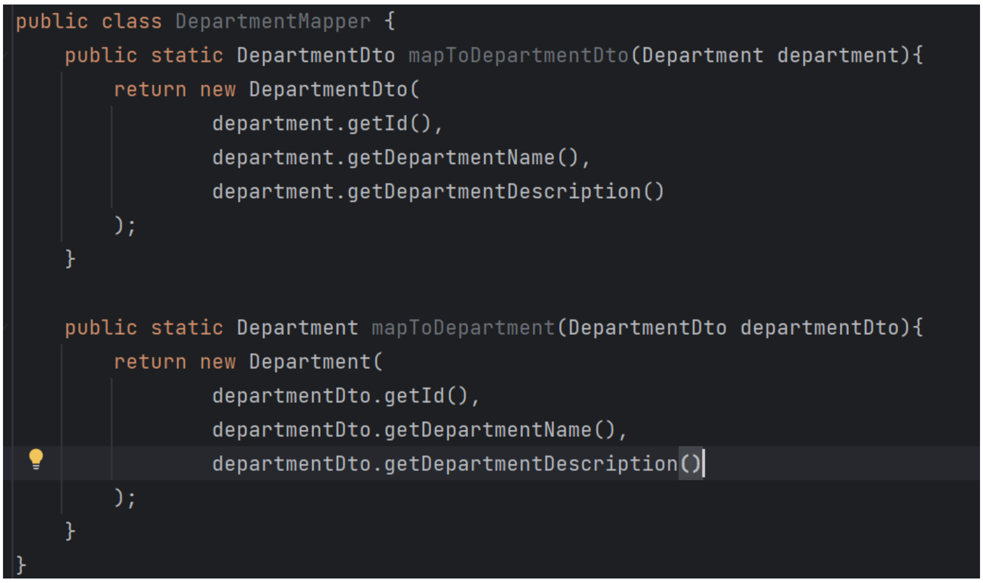


```java
public interface DepartmentRepository extends JpaRepository<Department, Long> {
}
```

Create DepartmentDto & DepartmentMapper

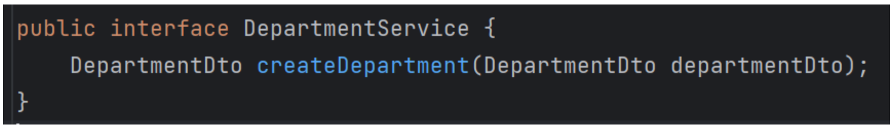


```java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class DepartmentDto {
    private Long id;
    private String departmentName;
    private String departmentDescription;
}
```

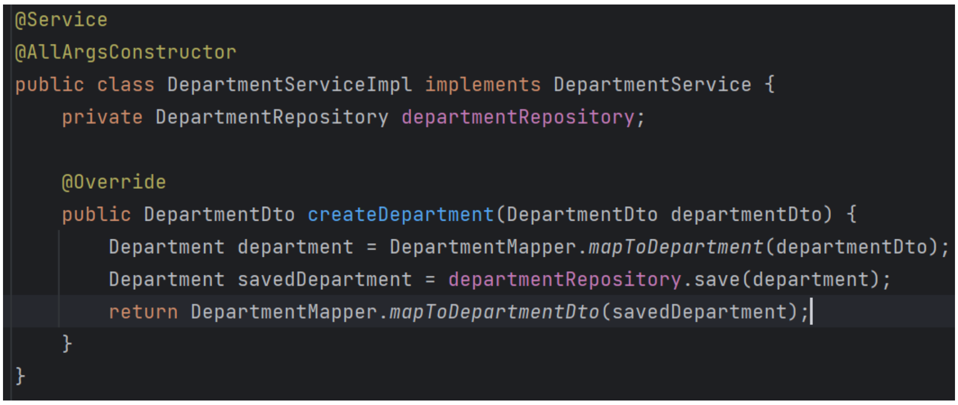


```
public class DepartmentMapper {
    public static DepartmentDto mapToDepartmentDto(Department department){
        return new DepartmentDto(
                department.getId(),
                department.getDepartmentName(),
                department.getDepartmentDescription()
        );
    }

    public static Department mapToDepartment(DepartmentDto departmentDto){
        return new Department(
                departmentDto.getId(),
                departmentDto.getDepartmentName(),
                departmentDto.getDepartmentDescription()
        );
    }
}
```

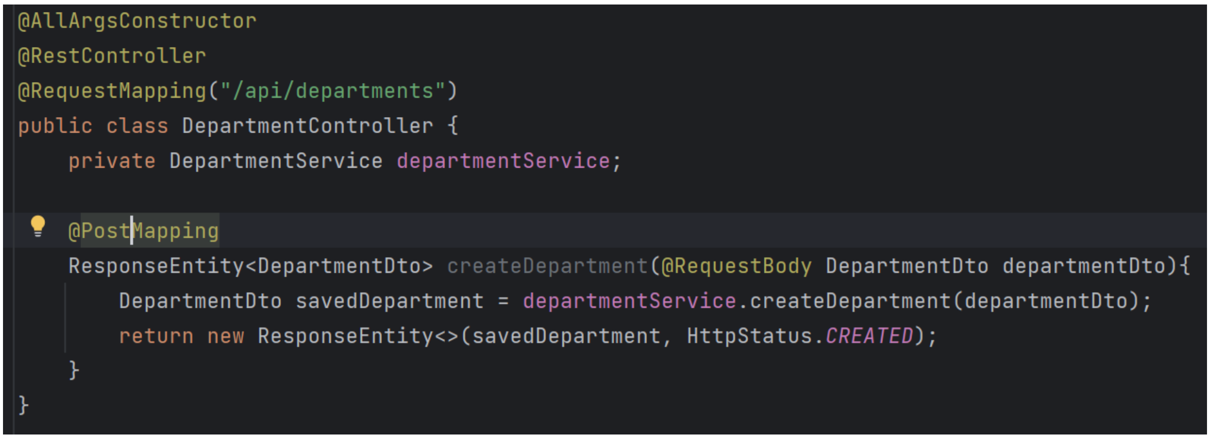
Build Create Department REST API



```
public interface DepartmentService {
    DepartmentDto createDepartment(DepartmentDto departmentDto);
}
```



```
@Service
@AllArgsConstructor
public class DepartmentServiceImpl implements DepartmentService {
    private DepartmentRepository departmentRepository;

    @Override
    public DepartmentDto createDepartment(DepartmentDto departmentDto) {
        Department department = DepartmentMapper.mapToDepartment(departmentDto);
        Department savedDepartment = departmentRepository.save(department);
        return DepartmentMapper.mapToDepartmentDto(savedDepartment);
    }
}
```



```
@AllArgsConstructor
@RestController
@RequestMapping("/api/departments")
public class DepartmentController {
    private DepartmentService departmentService;

    @PostMapping
    ResponseEntity<DepartmentDto> createDepartment(@RequestBody DepartmentDto departmentDto){
        DepartmentDto savedDepartment = departmentService.createDepartment(departmentDto);
        return new ResponseEntity<>(savedDepartment, HttpStatus.CREATED);
    }
}
```

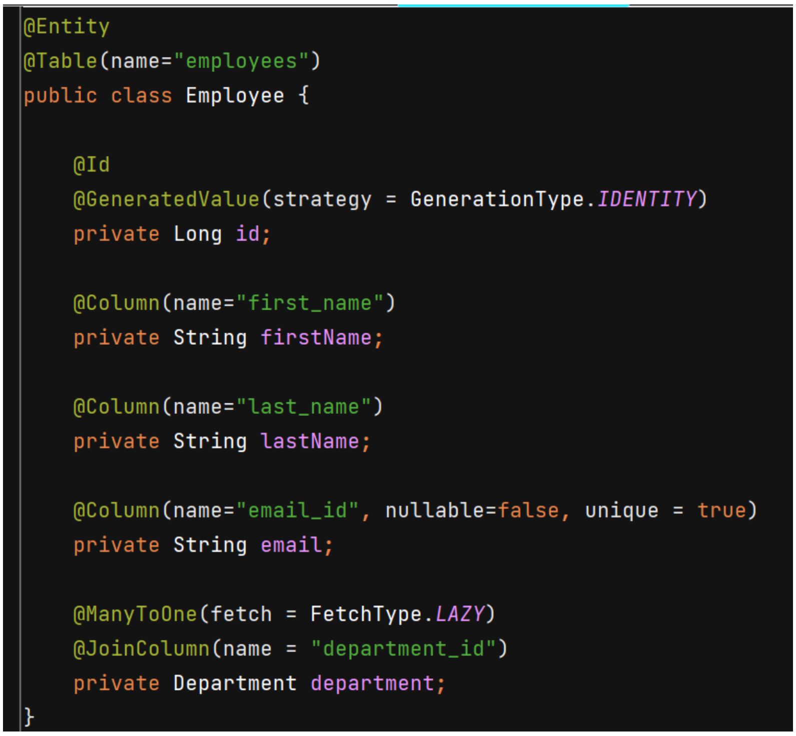
Similarly make REST APIs to get dept by Id, get all dept, update dept and delete dept.

Many To One Relationship between Employee & Deptarment JPA entities

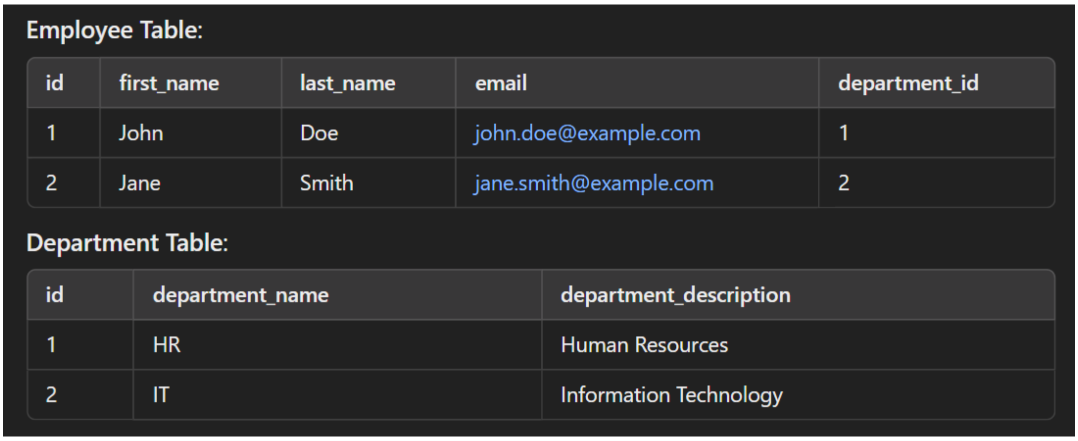
So go to Employee JPA entity and add Department instance variable.

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.



```
@Entity
@Table(name="employees")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name="first_name")
    private String firstName;

    @Column(name="last_name")
    private String lastName;

    @Column(name="email_id", nullable=false, unique = true)
    private String email;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "department_id")
    private Department department;
}
```



Employee Table:

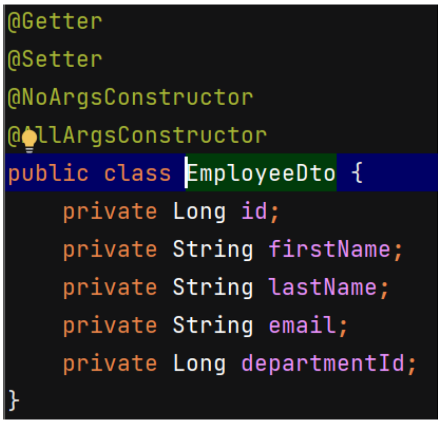
id	first_name	last_name	email	department_id
1	John	Doe	john.doe@example.com	1
2	Jane	Smith	jane.smith@example.com	2

Department Table:

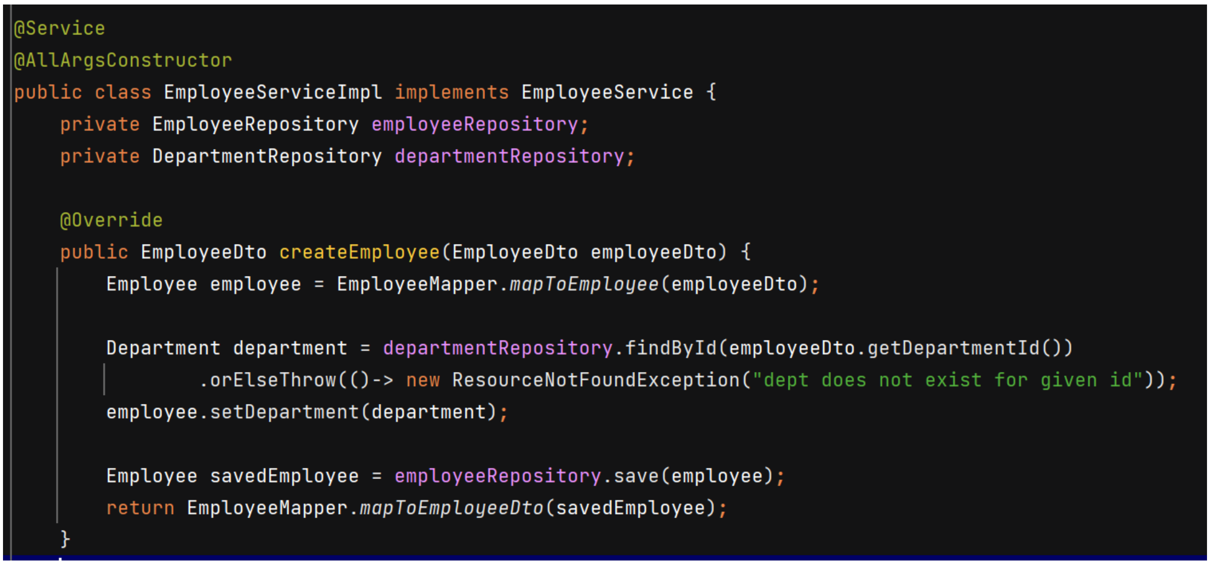
id	department_name	department_description
1	HR	Human Resources
2	IT	Information Technology

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

Add departmentId to EmployeeDto –

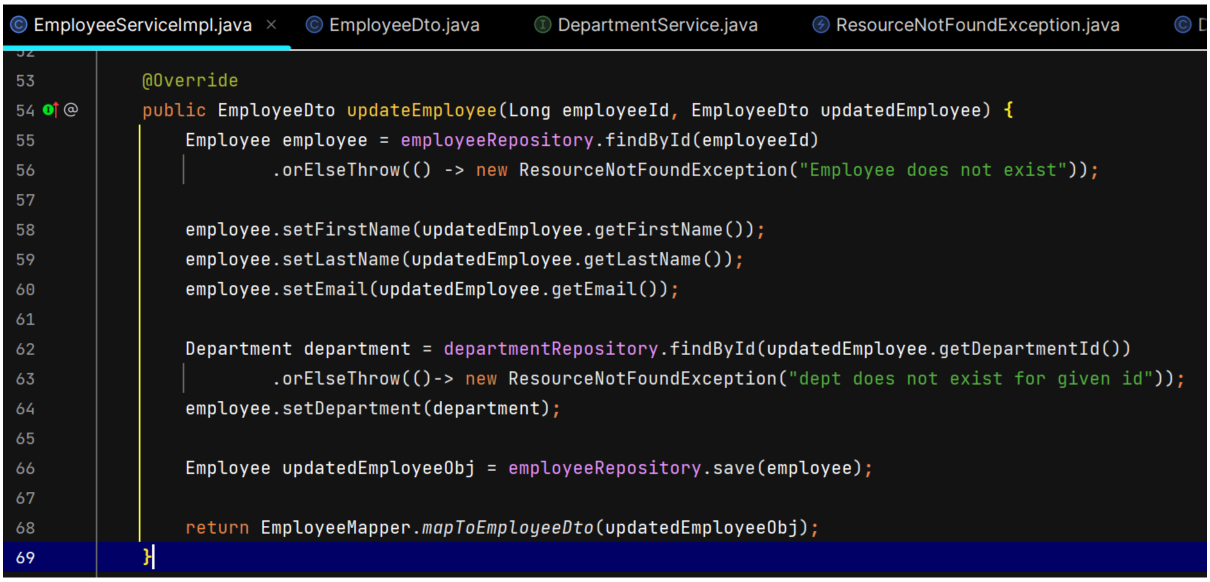


```java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class EmployeeDto {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private Long departmentId;
}
```



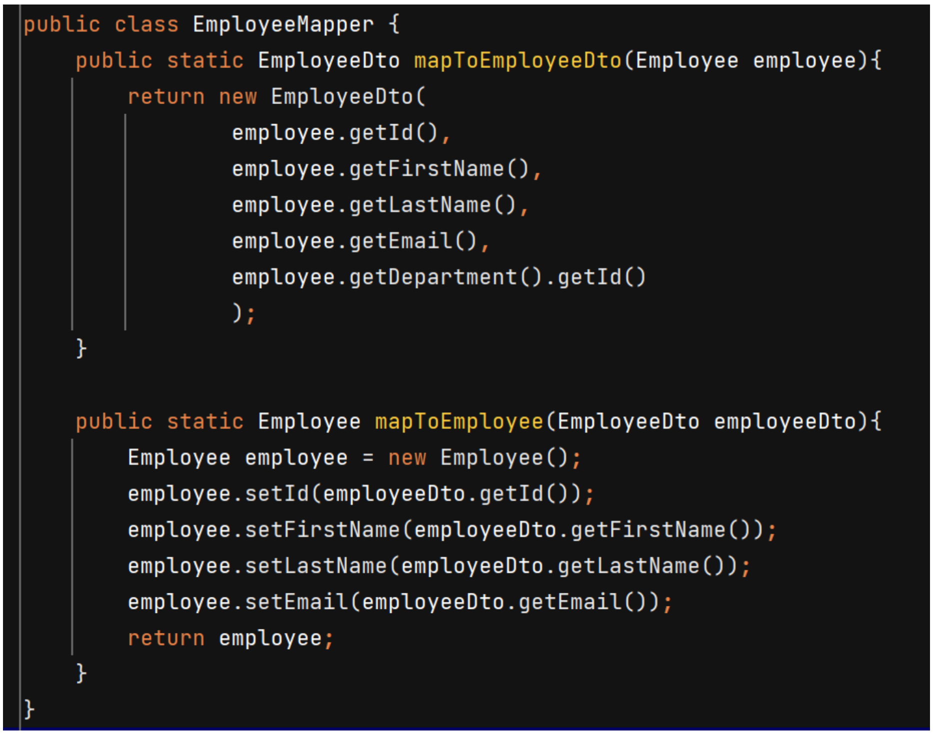
```java
@Service
@AllArgsConstructor
public class EmployeeServiceImpl implements EmployeeService {
    private EmployeeRepository employeeRepository;
    private DepartmentRepository departmentRepository;

    @Override
    public EmployeeDto createEmployee(EmployeeDto employeeDto) {
        Employee employee = EmployeeMapper.mapToEmployee(employeeDto);

        Department department = departmentRepository.findById(employeeDto.getDepartmentId())
                .orElseThrow(()-> new ResourceNotFoundException("dept does not exist for given id"));
        employee.setDepartment(department);

        Employee savedEmployee = employeeRepository.save(employee);
        return EmployeeMapper.mapToEmployeeDto(savedEmployee);
    }
```

User can change department for particular employee –



```java
    @Override
    public EmployeeDto updateEmployee(Long employeeId, EmployeeDto updatedEmployee) {
        Employee employee = employeeRepository.findById(employeeId)
                .orElseThrow(() -> new ResourceNotFoundException("Employee does not exist"));

        employee.setFirstName(updatedEmployee.getFirstName());
        employee.setLastName(updatedEmployee.getLastName());
        employee.setEmail(updatedEmployee.getEmail());

        Department department = departmentRepository.findById(updatedEmployee.getDepartmentId())
                .orElseThrow(()-> new ResourceNotFoundException("dept does not exist for given id"));
        employee.setDepartment(department);

        Employee updatedEmployeeObj = employeeRepository.save(employee);

        return EmployeeMapper.mapToEmployeeDto(updatedEmployeeObj);
    }
```

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



```
public class EmployeeMapper {
    public static EmployeeDto mapToEmployeeDto(Employee employee){
        return new EmployeeDto(
                employee.getId(),
                employee.getFirstName(),
                employee.getLastName(),
                employee.getEmail(),
                employee.getDepartment().getId()
                );
    }

    public static Employee mapToEmployee(EmployeeDto employeeDto){
        Employee employee = new Employee();
        employee.setId(employeeDto.getId());
        employee.setFirstName(employeeDto.getFirstName());
        employee.setLastName(employeeDto.getLastName());
        employee.setEmail(employeeDto.getEmail());
        return employee;
    }
}
```