I

Spring & SpringBoot Annotations

1. @Component
2. @Configuration
3. @Bean
4. @ComponentScan
5. @EnableAutoConfiguration
6. @SpringBootApplication
7. @Controller
8. @RestController
9. @RequestBody
10. @ResponseBody
11. @RequestMapping
12. @PostMapping
13. @GetMapping
14. @PutMapping
15. @DeleteMapping
16. @Valid
17. @PathVariable
18. @RequestParam
19. @Entity
20. @Service
21. @Repository
22. @Id
23. @GeneratedValue
24. @Autowired
25. @Qualifier
26. @Primary
27. @Required
28. @Transactional
29. @PatchMapping
30. @Mock
31. @injectmock
32. @BeforeForEach
33. @ExceptionHandeler
34. @RestControllerAdvice
35. @transactional(timeOut=30)
36. @transactional(readOnly=true)

Spring Boot Main annotations

----

1. @SpringBootApplication
2. @ComponentScan
3. @EanbleAutoConfiguration
4. @Configuration

Stereotype annotation

----

1. @Component
2. @Service
3. @RestController / @Controller
4. @Repository

Spring Core related Annotations:

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

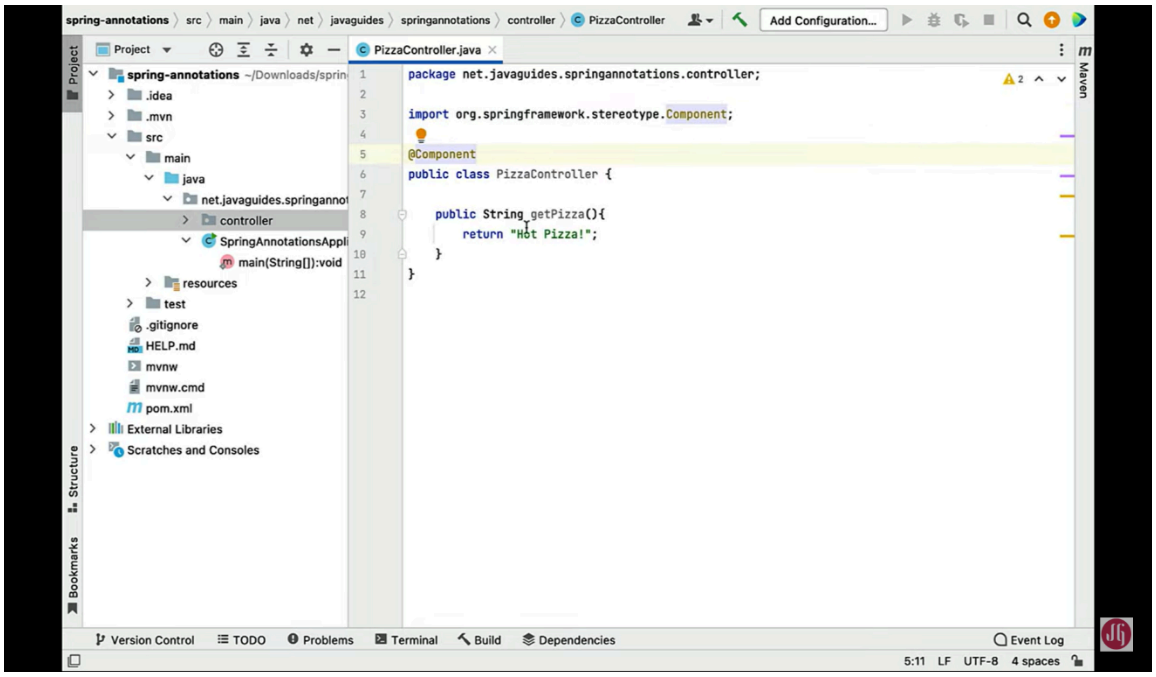
REST API related Annotations:

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

Spring Data JPA related annotations:

@Component:

The @Component annotation is a fundamental annotation in the Spring Framework and is used to mark a Java class as a Spring component. When the Spring container starts up, it automatically scans for classes annotated with @Component and creates instances of those classes to manage them as Spring beans in a container.



```java
package net.javaguides.springannotations.controller;

import org.springframework.stereotype.Component;

@Component
public class PizzaController {

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

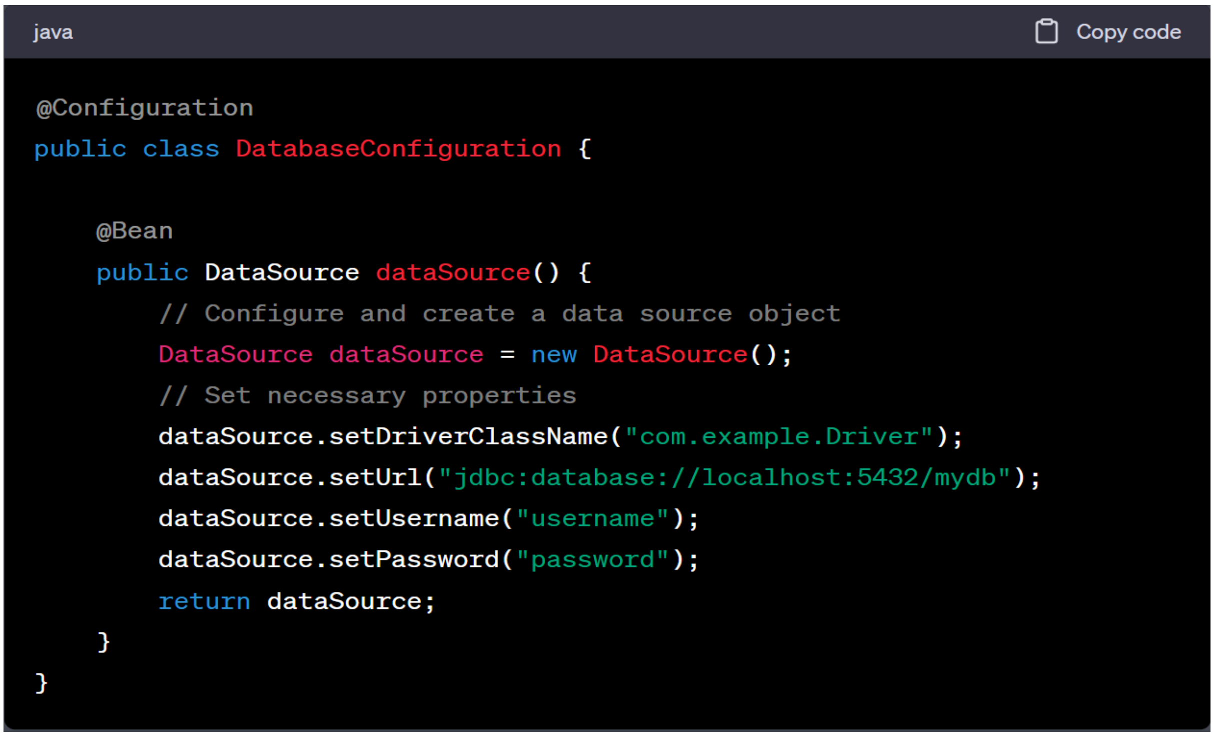
@Configuration:

@Configuration annotation indicates that the class has @Bean definition methods. So Spring container can process the class and generate Spring Beans to be used in the application.

@Bean:

In Spring Boot, the @Bean annotation is used to mark a method that produces a bean which will be managed by the Spring container. When the Spring container starts up, it looks for methods annotated with @Bean and executes them to create the corresponding bean instances

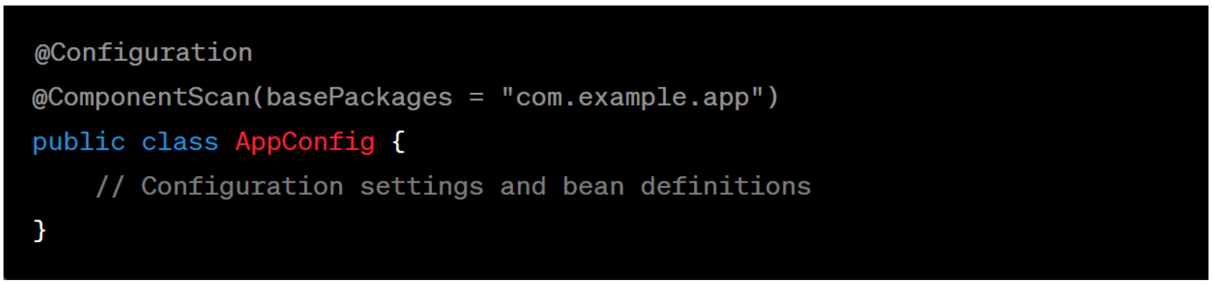
UseCase - Let's consider a scenario where you have a Spring application that requires configuration for connecting to a database. You can create a configuration class annotated with @Configuration to provide the necessary bean definitions for the database connection. This configuration class can contain methods annotated with @Bean to define beans for the data source, transaction manager, and other related components.



```java
@Configuration
public class DatabaseConfiguration {

    @Bean
    public DataSource dataSource() {
        // Configure and create a data source object
        DataSource dataSource = new DataSource();
        // Set necessary properties
        dataSource.setDriverClassName("com.example.Driver");
        dataSource.setUrl("jdbc:database://localhost:5432/mydb");
        dataSource.setUsername("username");
        dataSource.setPassword("password");
        return dataSource;
    }
}
```

@ComponentScan:

This annotation says that particular package is the default package and it responsible to scan all the sub packages, classes under that base package and finds out all the beans so IOC can create the related beans.



```java
@Configuration
@ComponentScan(basePackages = "com.example.app")
public class AppConfig {
    // Configuration settings and bean definitions
}
```

@EnableAutoConfiguration:

This annotation enables spring-boot to auto configure the application context. Therefore it automatically creates and registers beans based on the jar files included on the classpath.

When you add @EnableAutoConfiguration to your application's main class, Spring Boot will scan the classpath for dependencies and automatically configure beans and other necessary components based on the detected dependencies.

- For example, when we define spring-boot-starter-web dependency in our classpath, springboot autoconfigures Dispatch servlet, Tomcat, Jetty.
- For example, if you have added the Spring Data JPA starter, Spring Boot will configure the EntityManagerFactory and DataSource automatically.

@SpringBootApplication:

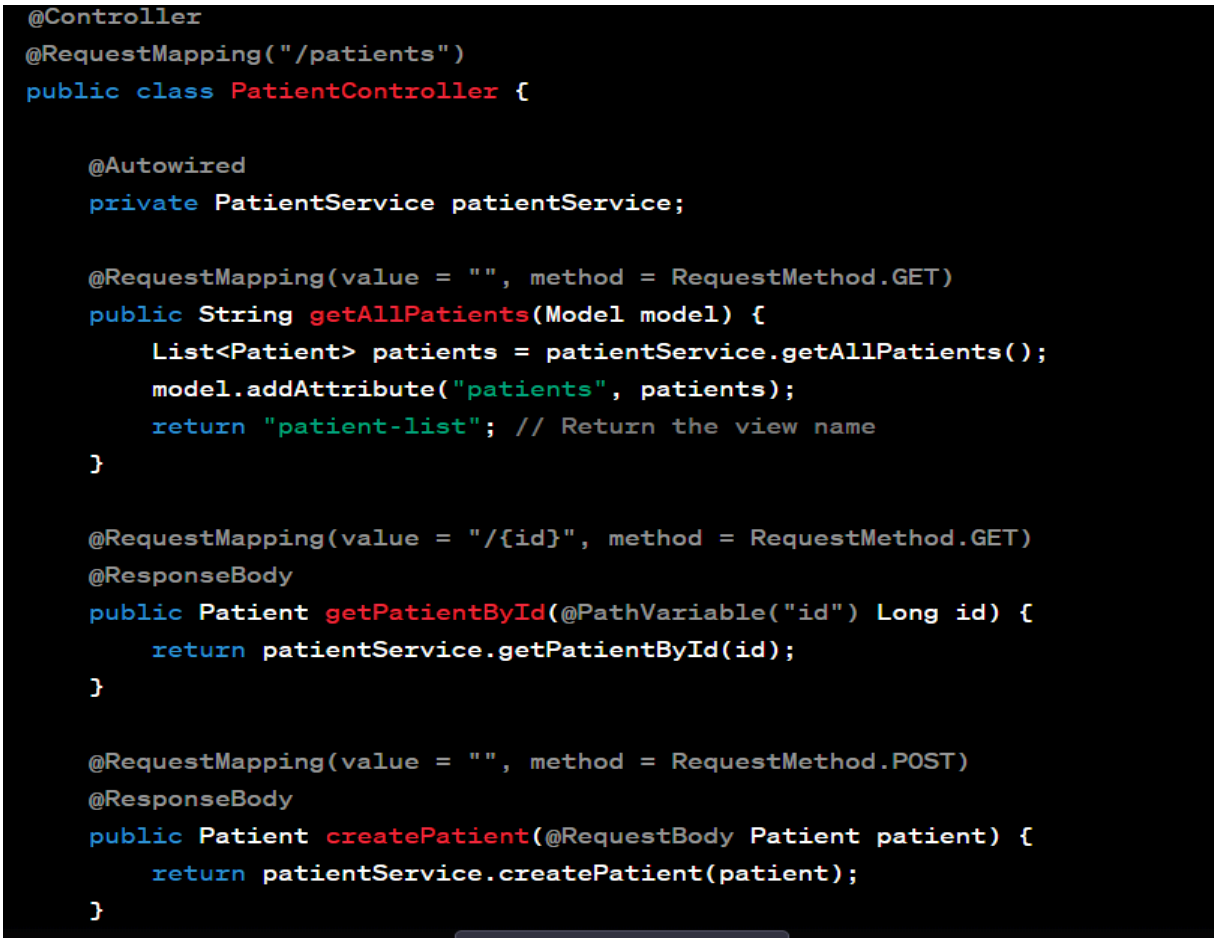
When we create a spring boot application by default one java class will be created with the name 'ApplicationName.java' which is annotated with /@SpringBootApplication. It is an entry point of our application. Execution starts from this class. This @ SpringBootApplication is a combination of 3 annotations;

1)@Configuration
2)@EnableAutoConfiguration
3)@ComponentScan

Controller:

For example, when you are working on spring application we used to mark our controller class with @Controller. Whatever mappings we have in controller class returns UI basically an html page like JSP/Thymleaf.

Usually @Controller in traditional MVC architecture returns a view /model. And when working with REST API's we need to mark each API with @ResponseBody Annotation to return the http response.



```java
@Controller
@RequestMapping("/patients")
public class PatientController {

    @Autowired
    private PatientService patientService;

    @RequestMapping(value = "", method = RequestMethod.GET)
    public String getAllPatients(Model model) {
        List<Patient> patients = patientService.getAllPatients();
        model.addAttribute("patients", patients);
        return "patient-list"; // Return the view name
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    @ResponseBody
    public Patient getPatientById(@PathVariable("id") Long id) {
        return patientService.getPatientById(id);
    }

    @RequestMapping(value = "", method = RequestMethod.POST)
    @ResponseBody
    public Patient createPatient(@RequestBody Patient patient) {
        return patientService.createPatient(patient);
    }
```

@RestController:

@RestController is a combination of @Controller & @ResponseBody. @Controller annotation is used to mark a class as a controller which will handle incoming HTTP requests from UI and provide a response back. As @Controller returns a view page in traditional Spring MVC but @RestController in Springboot returns a model object.

@RestController sets model object directly into the http response and then it will be converted to Json/xml automatically.

@ResponseBody:

This annotation tells to the SpringBoot that return value of the method should be serialized directly into HTTP response as JSON/XML and return the response back. When we use @ResponseBody annotation, spring converts the value and writes it into Http response automatically.

Note - For example, if we have 20 APIs in our controller class then we need to mention @ResponseBody for all API's. So from Spring 4.0 onwards we need not explicitly mention @ResponseBody . As @RestController internally uses @ResponseBody.

```
@RestController
@RequestMapping("/users")
public class UserController {

    private final UserService userService;

    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return userService.getUserById(id);
    }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User user) {
        return userService.updateUser(id, user);
    }
```

@RequestBody:

Any web applications processes the client requests over http. This annotation is used to bind or retrieve the Http request body into the required model object. Spring framework automatically de-serializes incoming http request to the java object using 'HTTP Message Converters.

@RequestMapping:

@RequestMapping annotation is used to map HTTP requests to specific methods in a controller class. When a request is received by the SpringBoot application, the @RequestMapping annotation is used to determine which method should be invoked to handle that particular request. The annotation can be applied at both the class level and the method level.

@PostMapping:

This annotation maps HTTP POST requests onto specific handler methods. It binds the method to a particular URL or endpoint, allowing the method to handle requests sent to that URL.

@GetMapping:

It is used to handle HTTP GET requests for a specific URI (Uniform Resource Identifier) or URL path. This annotation helps to map a method or a controller to a specific URL, allowing the method or controller to handle requests sent to that URL.

@PutMapping:

It is used to map HTTP PUT requests to a particular method in a controller class. When a PUT request is made to a specific URL, the method annotated with @PutMapping will be invoked to handle the request.

@DeleteMapping:

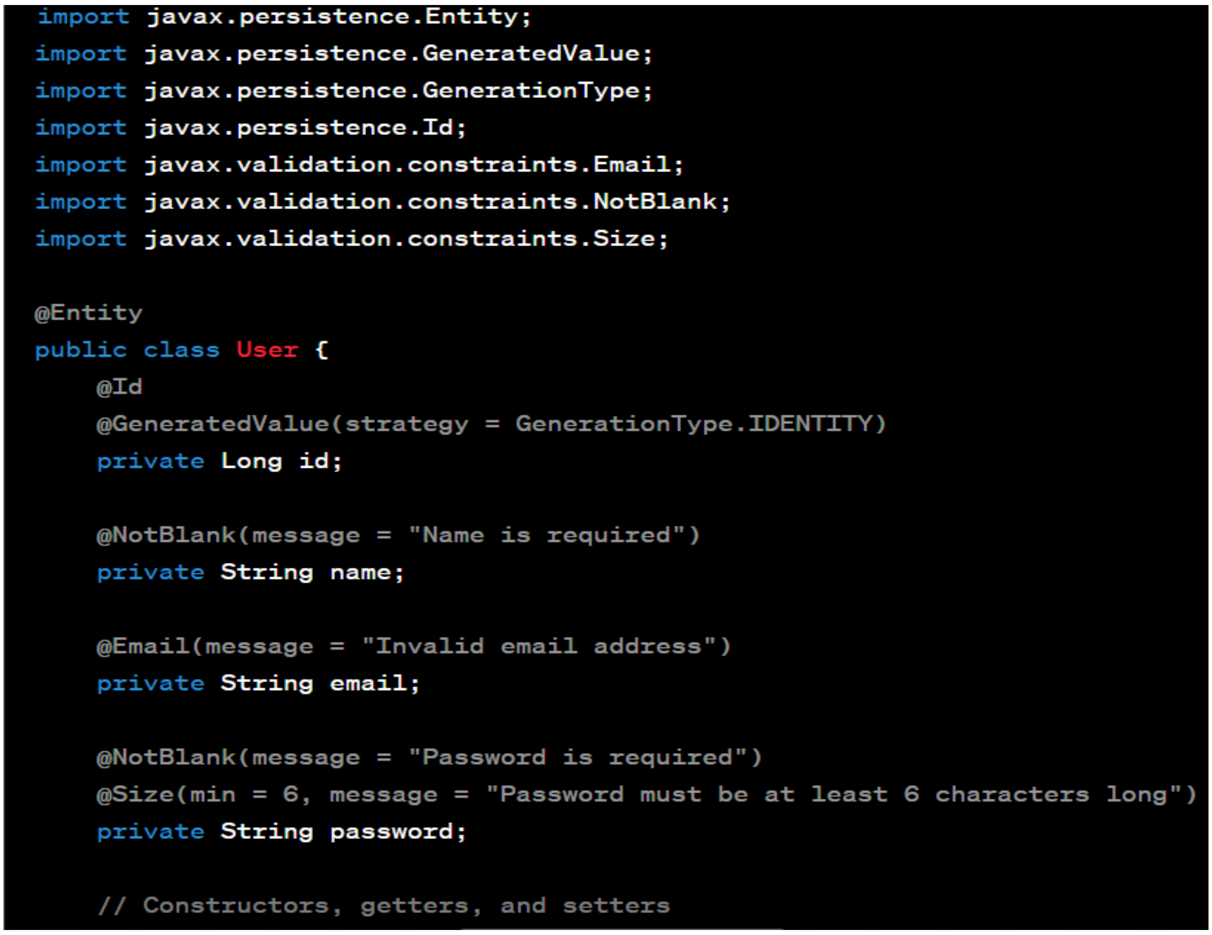
When you annotate a method in a controller class with @DeleteMapping, it indicates that this method will be invoked when a DELETE request is sent to the specified URL.

@Valid:

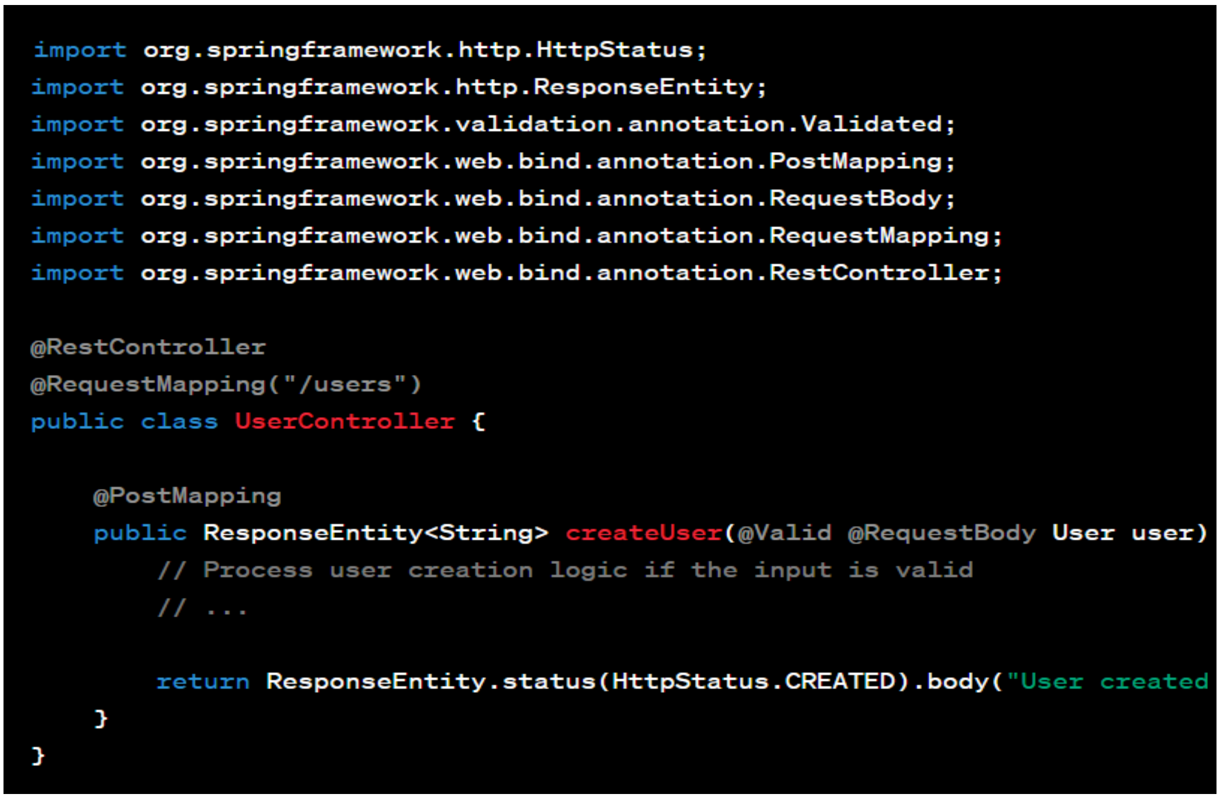
This annotation is commonly used in web applications to validate user input or data before processing it.

Before using the @Valid annotation, you need to define validation rules for your data objects using the Java Bean Validation API. This API provides a set of annotations such as @NotNull, @Size, @Email etc., that you can use to annotate fields or properties of your data objects.

- Suppose we have an entity class called User that represents a user in our application. We want to validate certain properties of the User object.



```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Name is required")
    private String name;

    @Email(message = "Invalid email address")
    private String email;

    @NotBlank(message = "Password is required")
    @Size(min = 6, message = "Password must be at least 6 characters long")
    private String password;

    // Constructors, getters, and setters
```

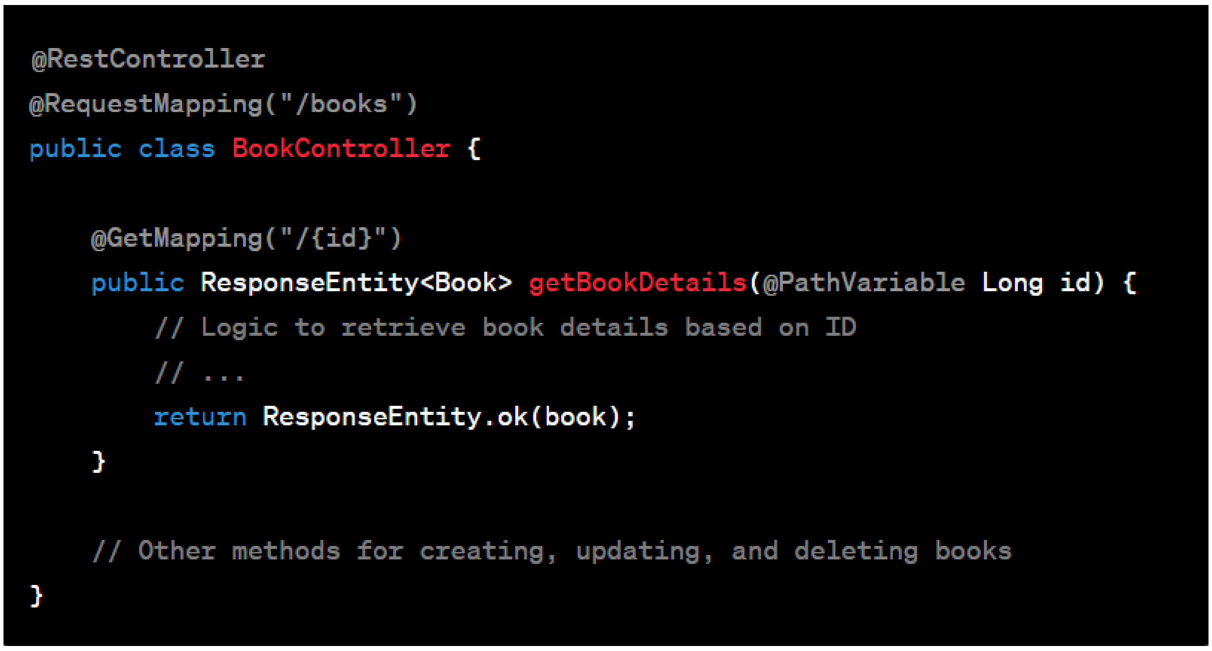


```java
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class UserController {

    @PostMapping
    public ResponseEntity<String> createUser(@Valid @RequestBody User user)
        // Process user creation logic if the input is valid
        // ...

        return ResponseEntity.status(HttpStatus.CREATED).body("User created
    }
}
```

@PathVariable:

@PathVariable annotation is used to bind a method parameter to a URI template variable. It allows you to extract values from the path of a URL and use them as method arguments in your Spring Boot controller methods.

Ex: Suppose you have a RESTful API for managing books, and you want to retrieve details about a specific book based on its ID.



```java
@RestController
@RequestMapping("/books")
public class BookController {

    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookDetails(@PathVariable Long id) {
        // Logic to retrieve book details based on ID
        // ...
        return ResponseEntity.ok(book);
    }

    // Other methods for creating, updating, and deleting books
}
```

The @PathVariable annotation is applied to the id parameter of the getBookDetails method. This annotation tells Spring to bind the value from the URI template variable to the method parameter.

In this case, the {id} segment in the mapping indicates that there is a path variable named "id" in the URL. The @PathVariable("id") annotation is used to bind the value of this path variable to the userId method parameter.

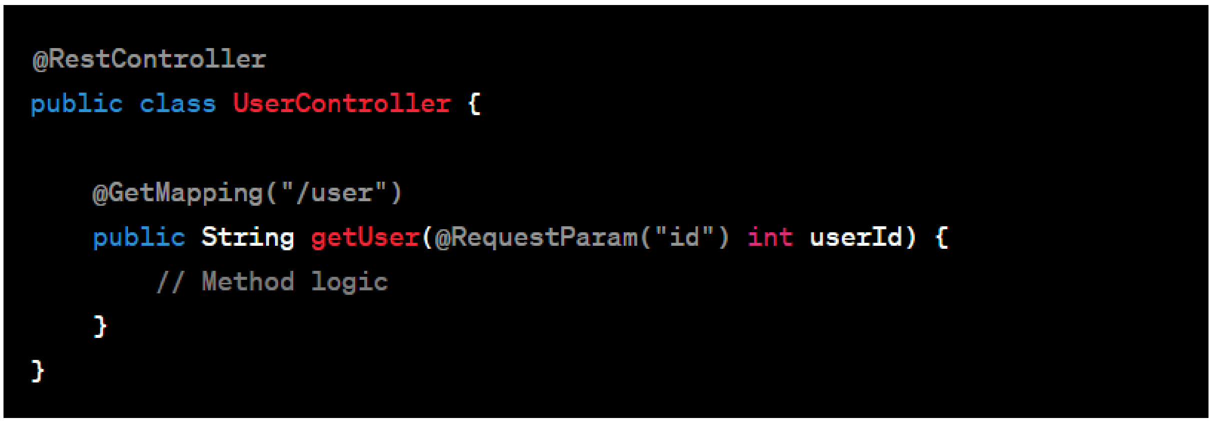
Request flow:

- The client sends a GET request to the URL /books/123.
- Spring Boot matches the request to the getBookDetails method based on the @GetMapping("/{id}") mapping.
- Spring extracts the value 123 from the URL and assigns it to the id parameter.
- The method logic executes, retrieving the book details with the ID 123.
- The book details are returned as an HTTP response.

@RequestParam:

@RequestParam annotation is used to bind request parameters to method parameters in a Spring Boot controller. It allows you to extract and use query parameters sent in an HTTP request.

When a request is made to a Spring MVC controller method that uses the @RequestParam annotation, Spring Boot automatically maps the value of the specified request parameter to the corresponding method parameter.



```java
@RestController
public class UserController {

    @GetMapping("/user")
    public String getUser(@RequestParam("id") int userId) {
        // Method logic
    }
}
```

Request Flow:

- The client makes an HTTP request, specifying the URL and the required request parameters.
- The request is received by the server, which is running a Spring Boot application.
- The controller method with the appropriate request mapping (e.g., @GetMapping, @PostMapping) and @RequestParam annotations is identified.
- Spring Boot examines the method's parameters and their corresponding annotations to determine how to bind the request parameters.
- Spring Boot extracts the values of the specified request parameters from the request URL.
- The extracted values are assigned to the corresponding method parameters.
- The controller method executes its logic using the assigned parameter values.
- The method generates a response, which is returned to the client.

Diff btw @PathVariable & @RequestParam:

While both @PathVariable and @RequestParam annotations are used to extract data from incoming requests in Spring Boot, they serve different purposes and operate on different parts of the request.

@PathVariable: This annotation is used to extract values from the path variables in the URL.

URL - http://example.com/users/123

@RequestParam: This annotation is used to extract values from request parameters in the query string or form data. Request parameters are key-value pairs added to the URL query string.

URL - http://example.com/users?id=123

@Entity:

We need to store the data in the database through repository layer. For that we need to define an entity class which is annotated with @Entity telling to the spring container that, this class is a bean of Entity type.

- Entity represents a table in the database.

@Service:

@Service annotation helps us specify that this class is a bean of service type. Service class usually talk to the dao or data access layer to store & retrieve the data from the database.

In our application, business logic resides inside service layer. So we use @Service annotation to indicate that class belongs to service layer.

@Repository:

This annotation is used to indicate that class provides the mechanism for storage, retrieval, update, delete operations on objects. By annotating the class with @Repository then SpringBoot Repository classes are autodetected by spring framework through classpath scanning where Repository is responsible for providing CRUD operations on database tables.

@Id:

This annotation is used to define particular field as a primary key in the entity class.

@GeneratedValue:

The @GeneratedValue annotation is used to specify how the values for primary keys should be generated automatically. It is typically used in combination with the @Id annotation, which marks a field or property as the primary key of the entity.

@AutoWired:

@Autowired annotation enables you to inject the object dependency implicitly. You do not need to tell the IOC container specially, whenever it sees @Autowired annotation IOC container automatically injects the object to the required classes.

- By default, Spring uses constructor injection if the target class has one or more constructors, and it uses setter injection if there is a default (no-argument) constructor available.
- Setter injection provides flexibility in terms of changing dependencies dynamically. You can modify the dependencies at runtime, which can be useful in certain scenarios.
- Constructor injection promotes immutability because the dependencies are set at the time of object creation and cannot be changed afterward. This can help ensure the object's integrity and avoid inconsistent states.

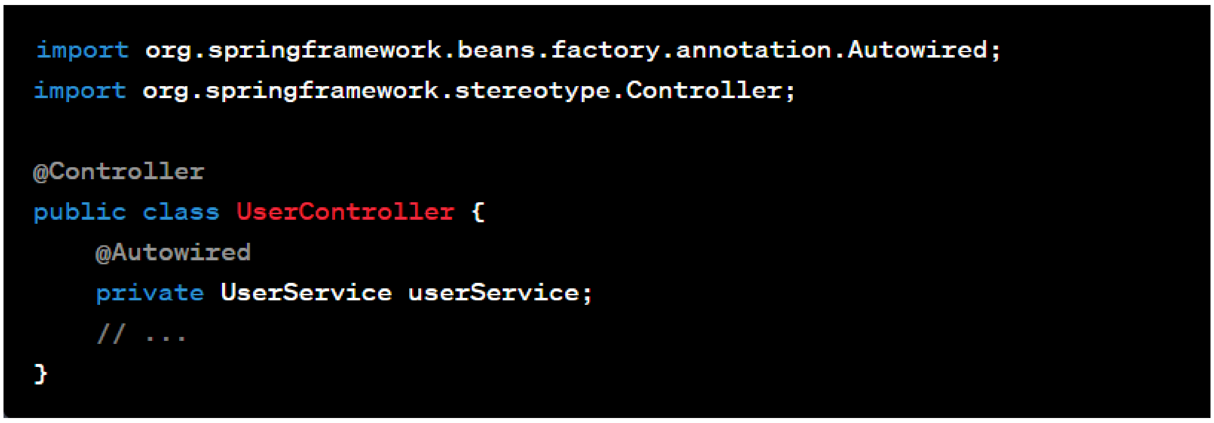
*We can perform Autowiring in 3 levels;

1) Field Level

2) Constructor Level

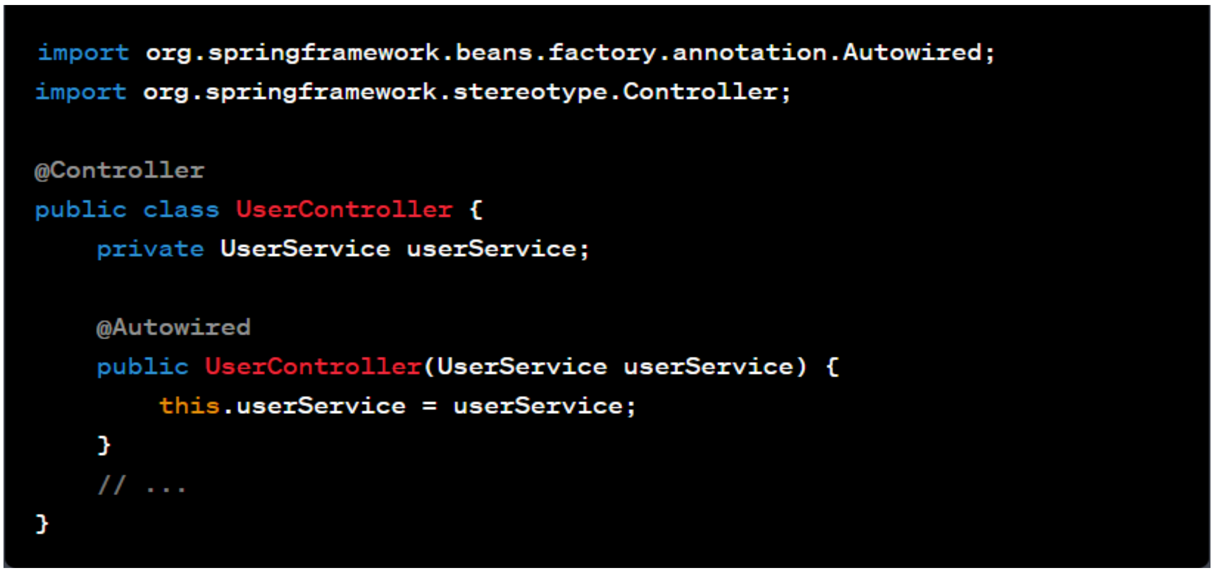
3) Setter level

1**) Field Level:**



```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Controller
public class UserController {
    @Autowired
    private UserService userService;
    // ...
}
```

2) Constructor Level:



```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Controller
public class UserController {
    private UserService userService;

    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }
    // ...
}
```

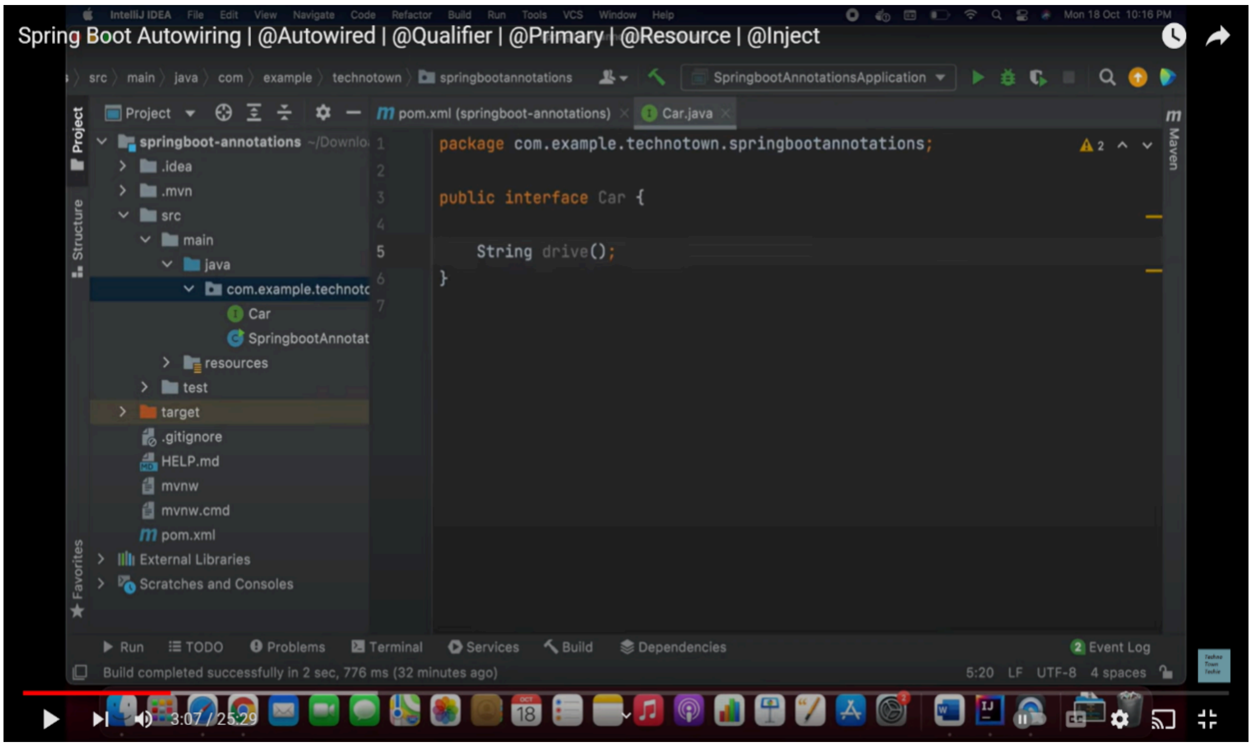
3) Setter Level:

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Controller
public class UserController {
    private UserService userService;

    @Autowired
    public void setUserService(UserService userService) {
        this.userService = userService;
    }
    // ...
}
```

@Qualifier:

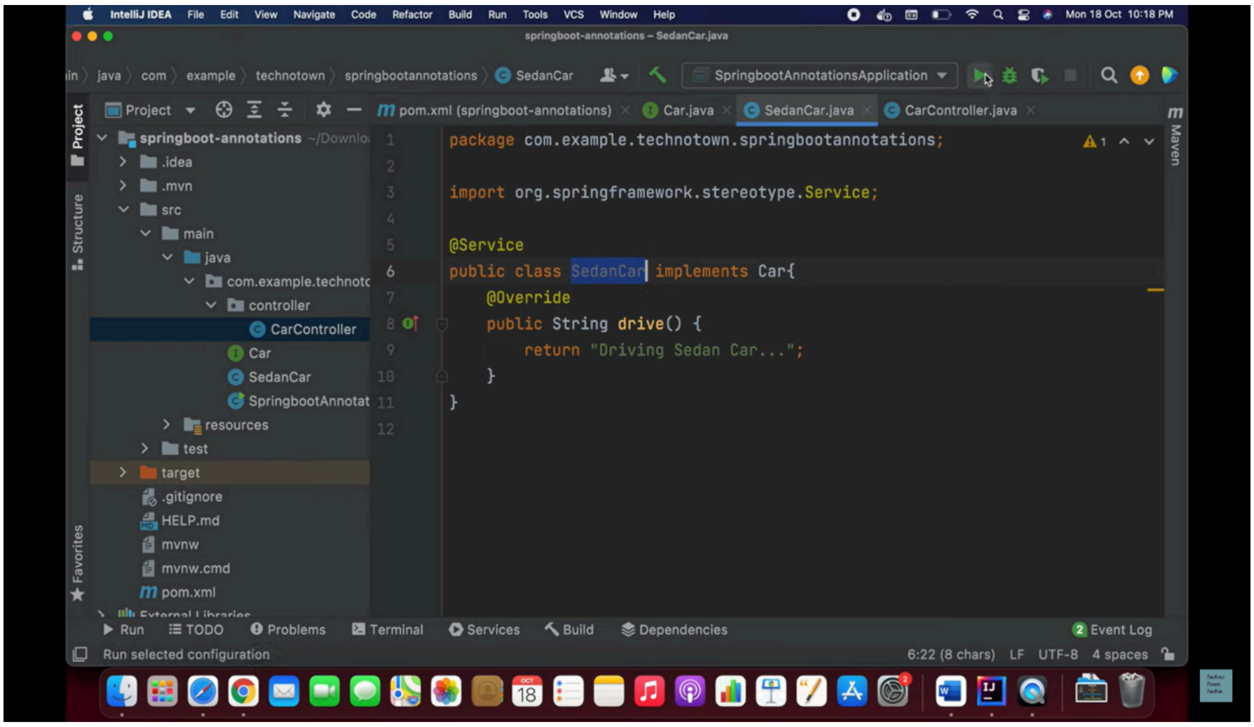
If more than one bean of the same type is available in the container, the framework will throw a NoUniqueBeanDefinitionException.

So when there are multiple beans of the same type, it's a good idea to use @Qualifier to avoid ambiguity.

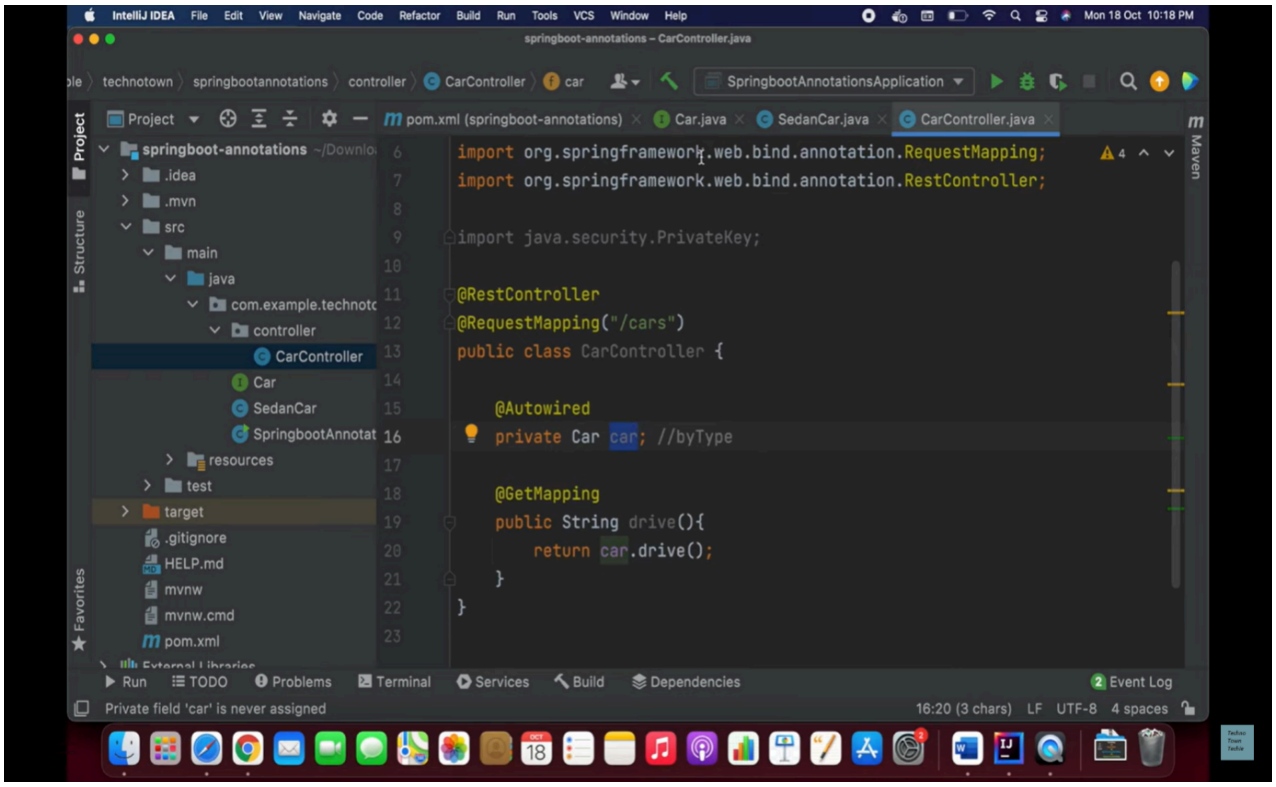


```java
package com.example.technotown.springbootannotations;

public interface Car {

    String drive();
}
```

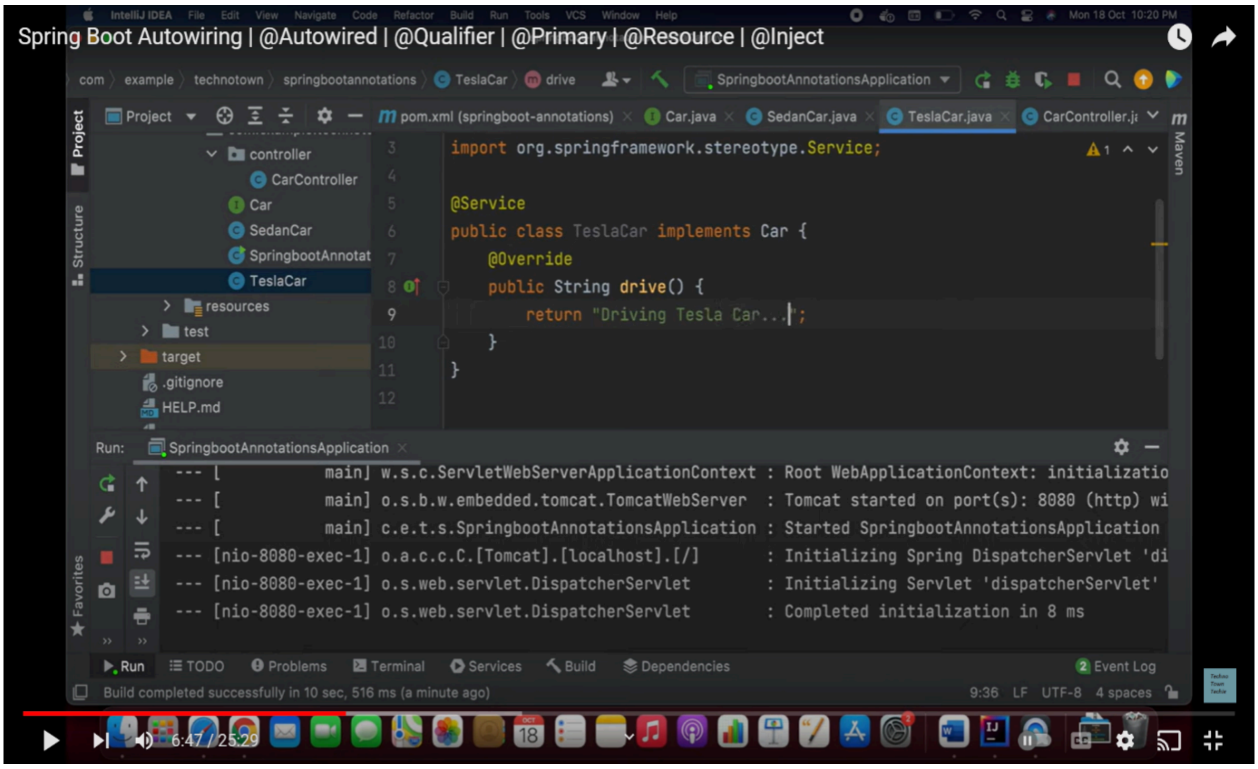
Build completed successfully in 2 sec, 776 ms (32 minutes ago)


springboot-annotations – SedanCar.java
pom.xml (springboot-annotations)
Car.java
SedanCar.java
CarController.java
Run selected configuration
6:22 (8 chars) LF UTF-8 4 spaces

```java
package com.example.technotown.springbootannotations;

import org.springframework.stereotype.Service;

@Service
public class SedanCar implements Car{
    @Override
    public String drive() {
        return "Driving Sedan Car...";
    }
}
```


springboot-annotations – CarController.java
pom.xml (springboot-annotations)
Car.java
SedanCar.java
CarController.java
Private field 'car' is never assigned
16:20 (3 chars) LF UTF-8 4 spaces

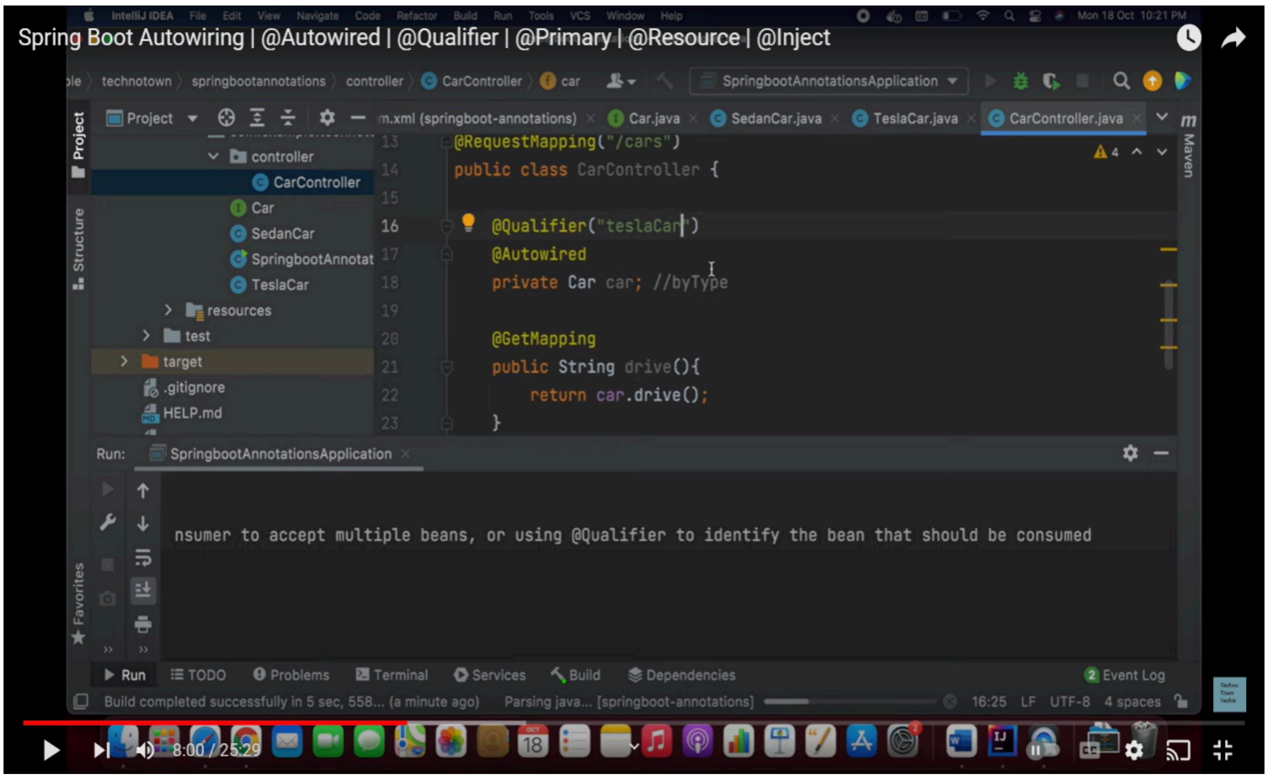
```java
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.security.PrivateKey;

@RestController
@RequestMapping("/cars")
public class CarController {

    @Autowired
    private Car car; //byType

    @GetMapping
    public String drive(){
        return car.drive();
    }
}
```



```java
import org.springframework.stereotype.Service;

@Service
public class TeslaCar implements Car {
    @Override
    public String drive() {
        return "Driving Tesla Car...";
    }
}
```

```
--- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initializatio
--- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) wi
--- [           main] c.e.t.s.SpringbootAnnotationsApplication : Started SpringbootAnnotationsApplication
--- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'di
--- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
--- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 8 ms
```

Build completed successfully in 10 sec, 516 ms (a minute ago)



```java
@RequestMapping("/cars")
public class CarController {

    @Qualifier("teslaCar")
    @Autowired
    private Car car; //byType

    @GetMapping
    public String drive(){
        return car.drive();
    }
```

```
nsumer to accept multiple beans, or using @Qualifier to identify the bean that should be consumed
```

Build completed successfully in 5 sec, 558... (a minute ago)   Parsing java... [springboot-annotations]

@Primary:

We can also use @Primary annotation if not @Qualifier. What @primary annotation does? -->When u annotate the class with @Primary, it means that particular class/bean will have highest priority given with respect to autowired concept when there are multiple implementations are available.

@Required:

@Required annotation is used to indicate that a particular dependency or property is mandatory and must be provided.

- If the dependency is not provided or cannot be resolved, an exception will be thrown, indicating that the required dependency is missing.
- required = true (default behavior): If the required attribute is set to true, it means that the field or property must be set. If the required dependency is not provided, the Spring container throws a BeanInitializationException during bean initialization, indicating that the dependency is missing.
- required = false: If the required attribute is explicitly set to false, it means that the field or property is optional. If the dependency is not provided, the Spring container does not throw an exception. Instead, the field or property remains uninitialized or null, and the application code must handle this situation accordingly.

@Transactional:

Spring supports 2 types of transaction management;

1) Programmatic transaction Management.
2) Declarative transaction management.

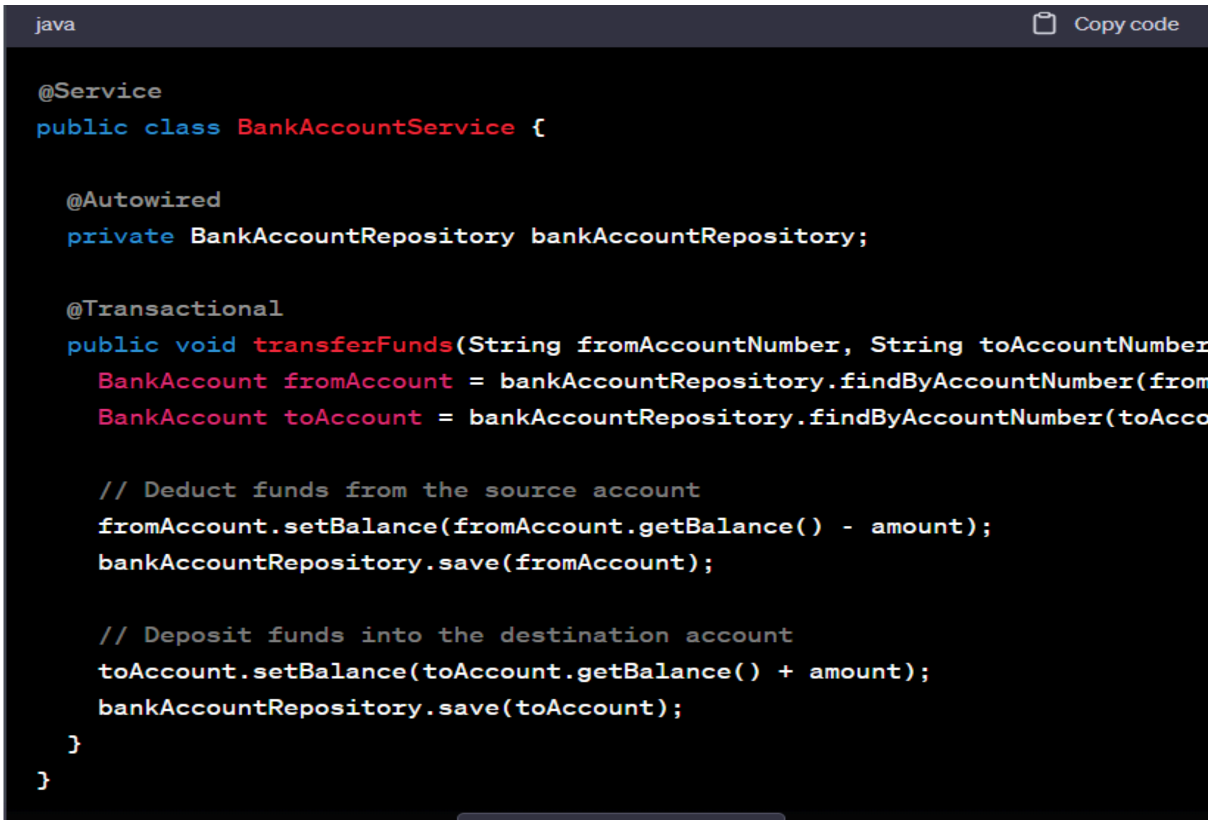
- Programmatic transaction is used when you have small number of transactional operations.
- In case of large number of transactions, it is better to use Declarative transaction management.

1) **Programmatic Transaction Management:** Programmatic transaction management involves explicitly managing transactions in your code. You manually begin and commit or rollback transactions using transaction APIs.
2) **Declarative Transaction Management:** Declarative transaction management allows you to manage transaction using annotations or XML configuration. You let Spring manage transactions based on these declarative rules.

* You annotate your methods or classes with transactional annotations, such as @Transactional.

The @Transactional annotation is a feature in Spring Framework that allows developers to simplify the management of database transactions in their Java applications.

By adding the @Transactional annotation to a method, Spring will automatically manage the transaction for that method, handling things like opening and closing the transaction, rolling back the transaction if an exception is thrown, and committing the transaction if everything succeeds.



```java
@Service
public class BankAccountService {

  @Autowired
  private BankAccountRepository bankAccountRepository;

  @Transactional
  public void transferFunds(String fromAccountNumber, String toAccountNumber
    BankAccount fromAccount = bankAccountRepository.findByAccountNumber(from
    BankAccount toAccount = bankAccountRepository.findByAccountNumber(toAcco

    // Deduct funds from the source account
    fromAccount.setBalance(fromAccount.getBalance() - amount);
    bankAccountRepository.save(fromAccount);

    // Deposit funds into the destination account
    toAccount.setBalance(toAccount.getBalance() + amount);
    bankAccountRepository.save(toAccount);
  }
}
```

Stereotype Annotations:

The Spring Framework provides us some special annotations. These annotations are used to create Spring beans automatically in the application context.

In spring there are 4 types of stereotype annotations are there,

1.@Component

2.@service

3.@Repository

4.@controller

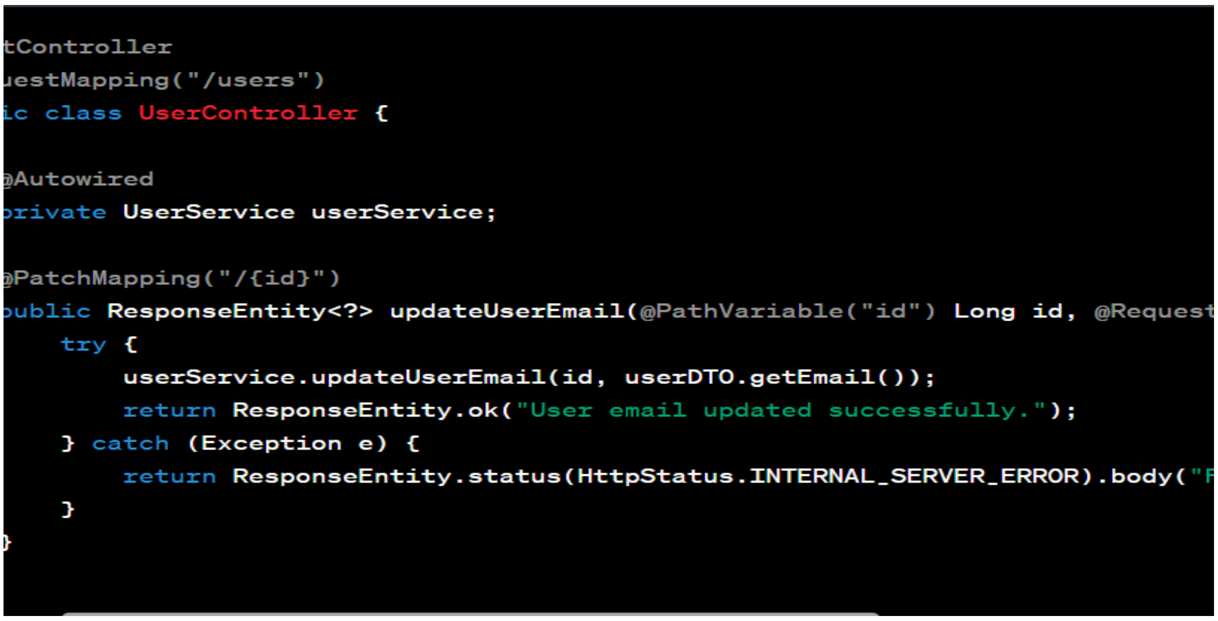
-> The main stereotype annotation is @Component. @Component is a Spring Framework annotation used to indicate that a class is a component and should be automatically detected and registered as a bean in the Spring context.

-> When Spring scans the classpath for classes annotated with @Component, it automatically detects them and creates instances of the classes as beans. These beans can be used for dependency injection by other Spring-managed components.

-> Also Spring provides more Stereotype annotations such as @Service, used to create Spring beans for the Service layer, @Repository, which is used to create Spring beans for the repositories at the DAO layer, and @Controller, which is used to create Spring beans at the controller layer.

@PatchMapping:

In Spring Boot, the PATCH API is used to partially update a resource. It allows you to update specific fields of an object without sending the entire object in the request payload. The @PatchMapping annotation is used to handle PATCH requests in a Spring Boot controller.



```
tController
uestMapping("/users")
ic class UserController {

@Autowired
private UserService userService;

@PatchMapping("/{id}")
public ResponseEntity<?> updateUserEmail(@PathVariable("id") Long id, @Request
    try {
        userService.updateUserEmail(id, userDTO.getEmail());
        return ResponseEntity.ok("User email updated successfully.");
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("F
    }
}
```

@transactional(timeOut=30) trans should be complete in 30 sec

@transactional(readOnly=true) allow only read operations like get()

@Mock : create a mock object of a class or interface. Once the mock object is created, you can use it in your test methods to define its behavior using Mockito's stubbing methods, such as when() and thenReturn(). This allows you to specify the expected behavior of the mock object during the test.

@InjectMock The @InjectMocks annotation in JUnit Mockito is used to automatically inject mock objects into the fields of a tested object. It is typically used in conjunction with the @Mock annotation to create and inject mock objects into the class being tested.

@test The @Test annotation in JUnit is used to mark a method as a test method. Test methods are the actual test cases that verify the behavior of the code under test.

@Beforeforeach: that is used to annotate a method that needs to be executed before each test method in a test class**.**

@ExceptionHandeler: annotation is used in Spring to handle exceptions within a controller or a handler method. It is part of Spring's exception-handling mechanism and allows you to define methods that will be invoked when a specific exception occurs during the execution of a request

@ControllerAdvice: This annotation can be used to apply advice to all controllers in an application. This can be useful for applying global security or logging configuration.

@RestControllerAdvice: This annotation can only be used to apply advice to REST controllers. This can be useful for applying global exception handling or response formatting configuration.

@PathVariable annotation is used to extract the id value from the URI template, while the @RequestParam annotation is used to extract the page and size values from the query string.