# Core Features

02 May 2024 12:52

IOC -Does the object lifecycle of java Objects, when they are called and destroyed

AOP - handles application spanning functions, Logging, caching and auth etc

DAF - Does database communication, JVC, JPA also does resource management and exception handling

MVC - Allows developers to use MVC, handles requests and building of controllers and responses via view.

Spring Bean - An object handled by the Spring framework

Spring beans get there lifecycle and dependencies managed by Spring.

Spring beans can be configured using an XML or Java code

Spring bean lifecycle:

Handles how it is created, how it behaves, when and how its destroyed

Beans are all managed within IOC containers

# Beans & Components

05 May 2024 16:20

Spring Bean - An object handled by the Spring framework

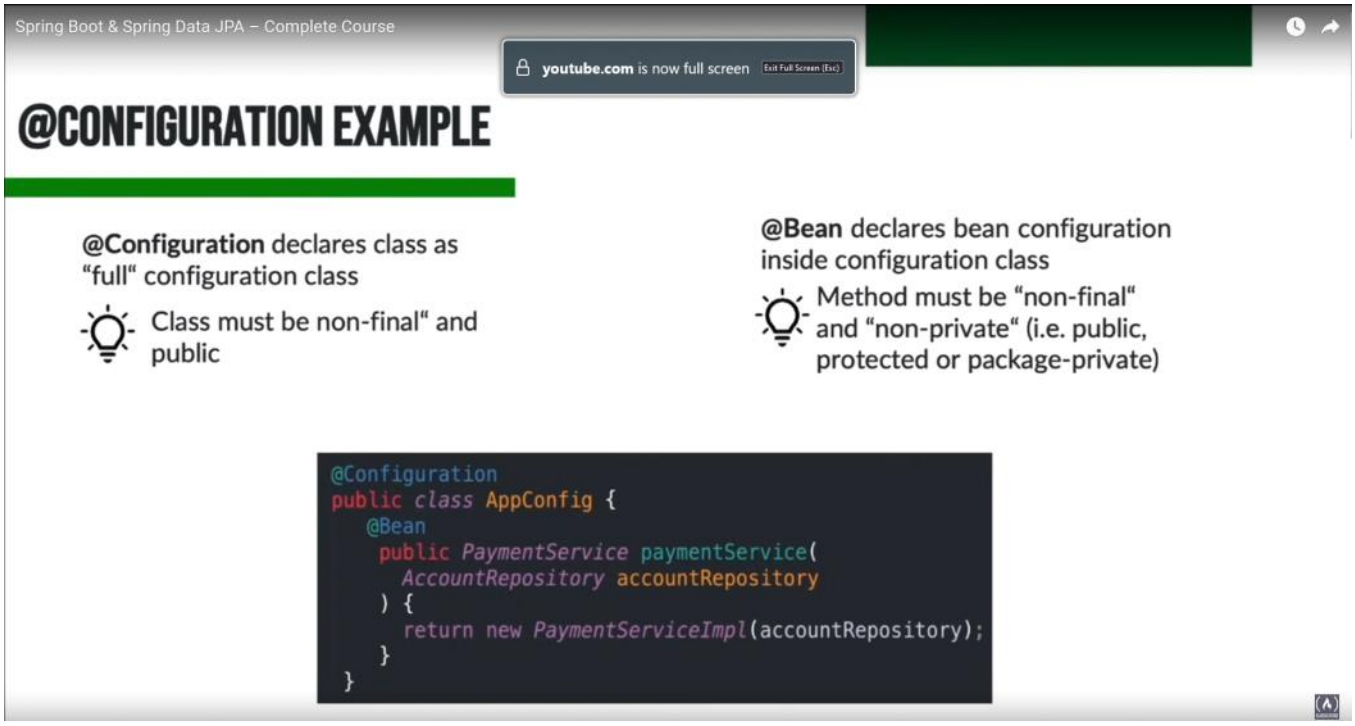
Spring beans get there lifecycle and dependencies managed by Spring.

Spring beans can be configured using an XML or Java code

Spring bean lifecycle:

Handles how it is created, how it behaves, when and how its destroyed

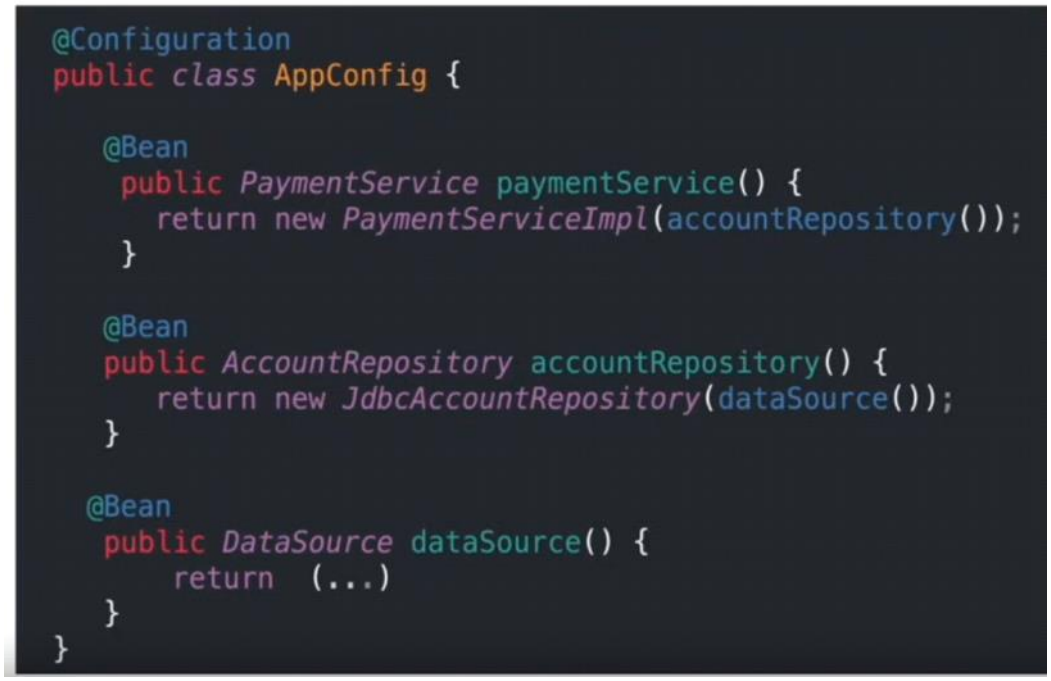
Beans are all managed within IOC containers




To declare a BEAN CONFIG use the @Configuration annotation, must be a public class

Use the @Configrutation annotation if you want to declare lots of beans

Use @Bean to declare to bean. Cannot be a private (singleton).




## Components:

@Component creates a bean for the class and managed by container (cuz bean)

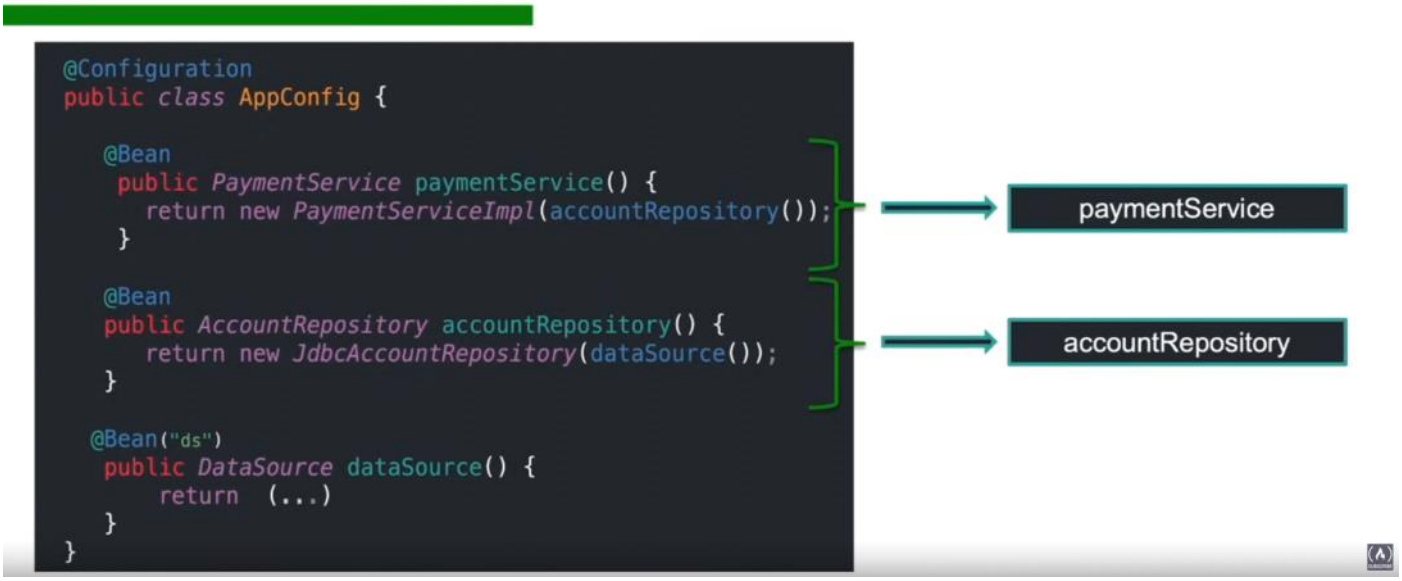
There is the @Repositorry, @Service and @Controller to further refine containers behaviour.

@Autowired creates instances of required services when needed in the IOC container.


## Naming:

By default beans are named same as method, but if name is given it will be used

# BEAN NAMING


@Configuration
public class AppConfig {
@Bean
public PaymentService paymentService() {
return new PaymentServiceImpl(accountRepository());
}
@Bean
public AccountRepository accountRepository() {
return new JdbcAccountRepository(dataSource());
}
@Bean("ds")
public DataSource dataSource() {
return (...)
}
}
paymentService
accountRepository

# Dependency Injection

05 May 2024 16:48

## Constructor Injection:


Bean handles lifecycle with IOC container using constructor

@Autowired isn't included here, so dependences aren't instantiated by in practice, should be.

**THIS IS THE INJECTION SHOULD YOU USE TO CREATE BEANS**

Suppose you have two implementations of a `MessageService` interface:

java Copy code

```java
public interface MessageService {
    String getMessage();
}

@Component("emailService")
public class EmailService implements MessageService {
    public String getMessage() {
        return "Email message";
    }
}

@Component("smsService")
public class SmsService implements MessageService {
    public String getMessage() {
        return "SMS message";
    }
}
```

Now, if you have a class that depends on `MessageService`, and you have both `EmailService` and `SmsService` beans defined in your Spring application context, you can use `@Qualifier` to specify which one to inject:

java Copy code

which one to inject:

java Copy code

```java
@Component
public class MyService {
    private final MessageService messageService;

    @Autowired
    public MyService(@Qualifier("emailService") MessageService messageService) {
        this.messageService = messageService;
    }

    // Other methods
}
```

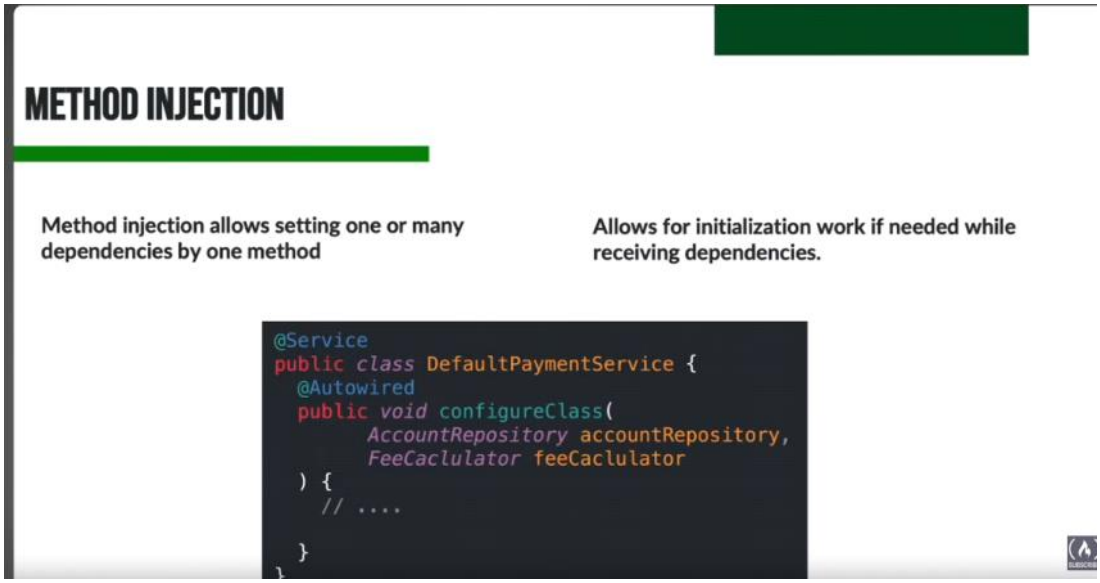
Remember @Autowired handles dependency injection

Use @Qualifer if you have multiple beans in a service so that dependent classes to know what bean is what

If you only have two, you can just use @primary

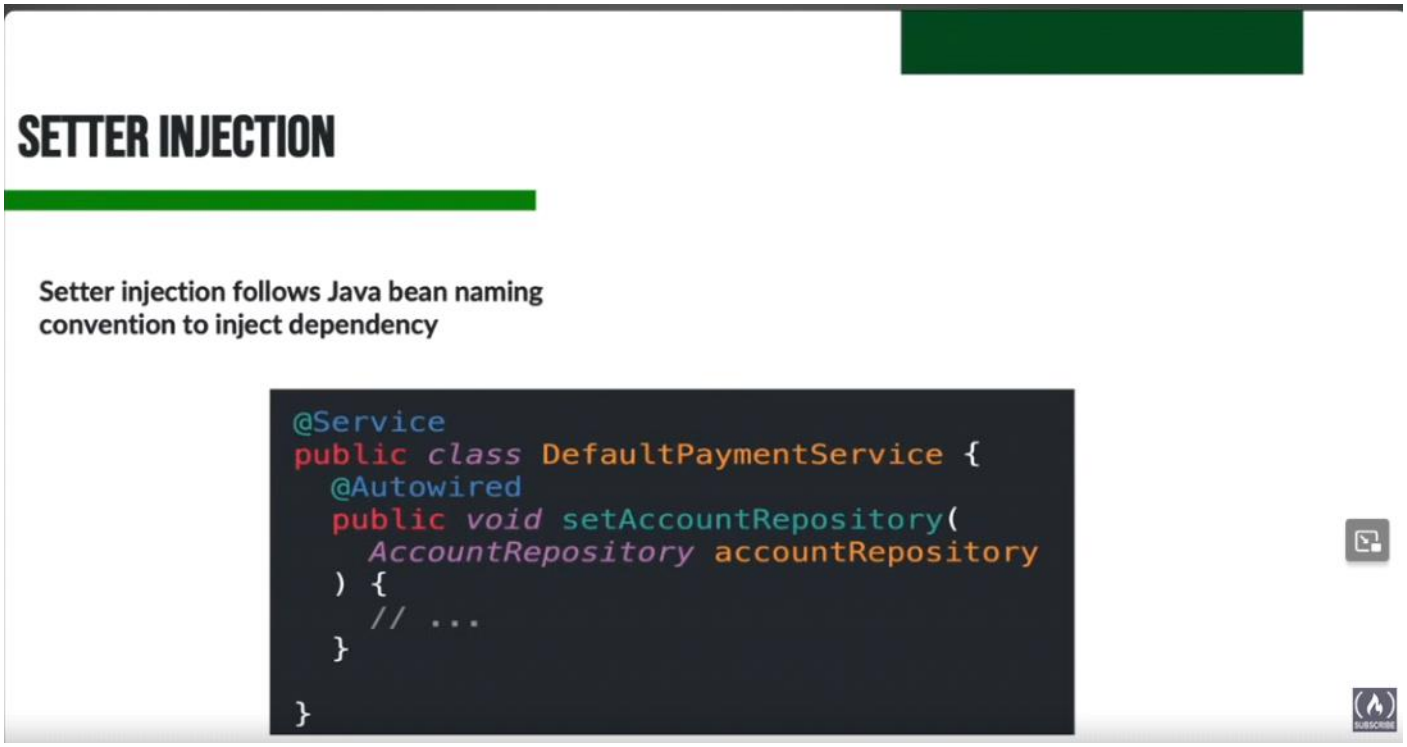
**You should use constructor injection because you can test it much easier, not field injection**

## Method Injection:




Using a method to create beans and do dependency injection allows us to do 'pre' injection work

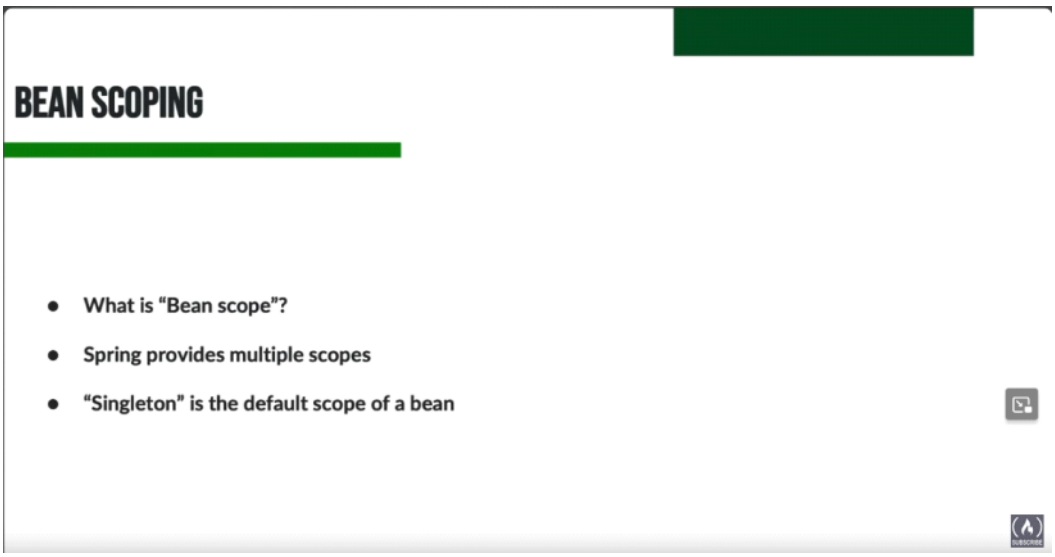
## Setter Injection:




Use setter injection for optional dependencies.

# Bean Scoping

05 May 2024 19:09




All beans are by default singleton

There is a selection of scopes:

Singleton - default (doesn't carry state)

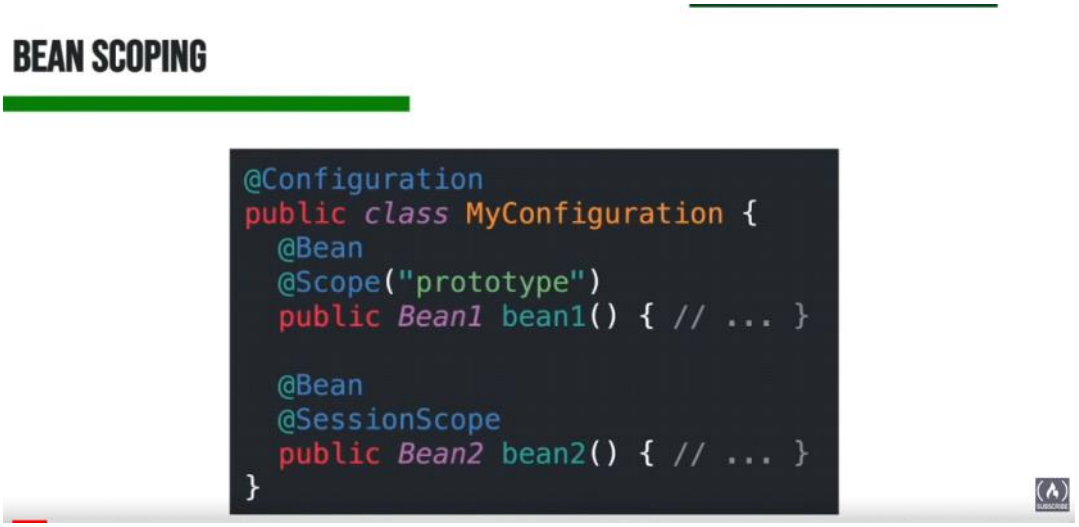
Prototype - creates new instance with each request useful for user specific state

Request - a new bean is created and destroyed with each HTTP request

Session - only kept alive and created during a HTTP session

Application - kept alive with application

Websocket - only kept alive during websocket

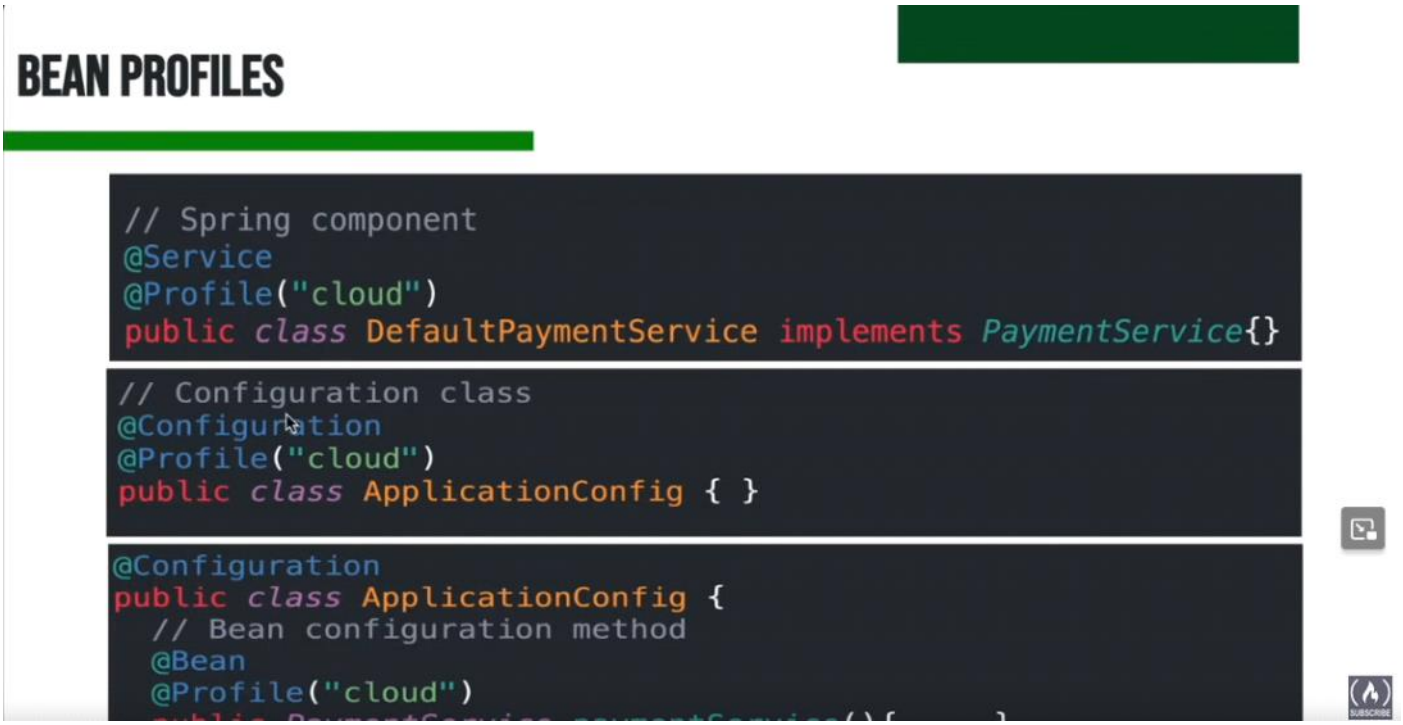


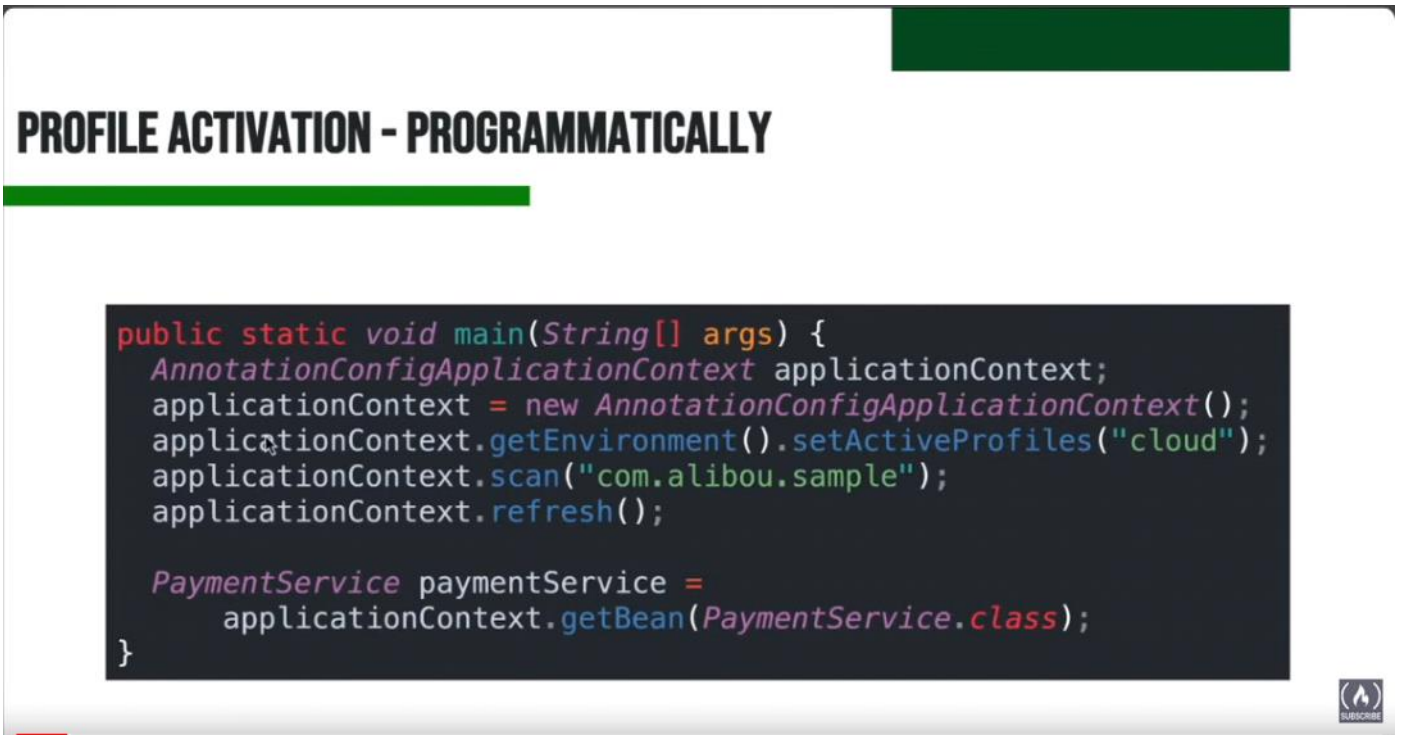
# Bean profile

05 May 2024 19:42

You use bean profiles to trigger beans when conditions are met (development, testing, production)

You can also use this for configurations (that create many beans u know)




Because of we have set profile and we can now use the beans with this profile.

We can activate it using the prop file as well

# PROFILE ACTIVATION – PROPERTIES FILE

application.yaml

```yaml
spring:
  profiles:
    active: cloud
```

application.properties

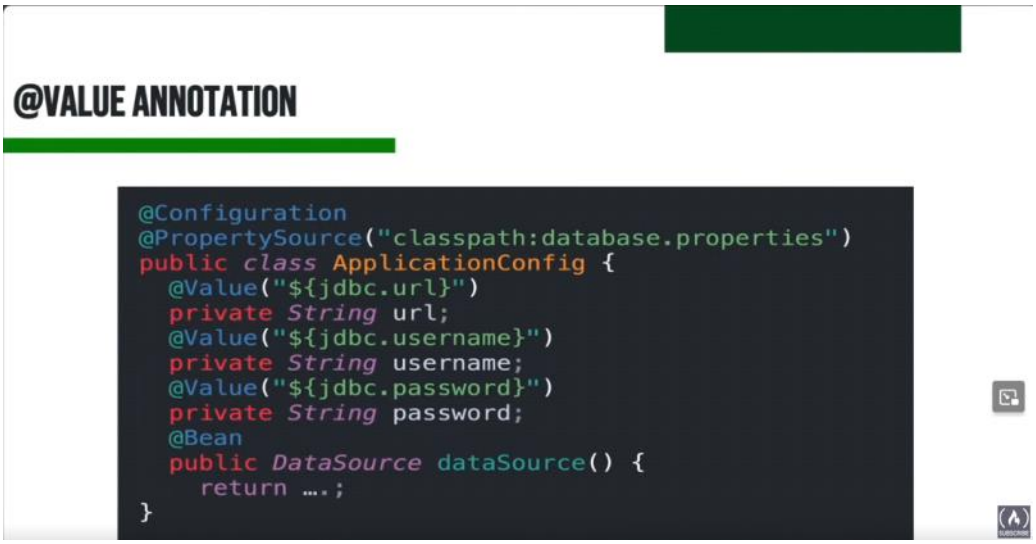
```properties
spring.profiles.active= cloud
```

# Value Annotation

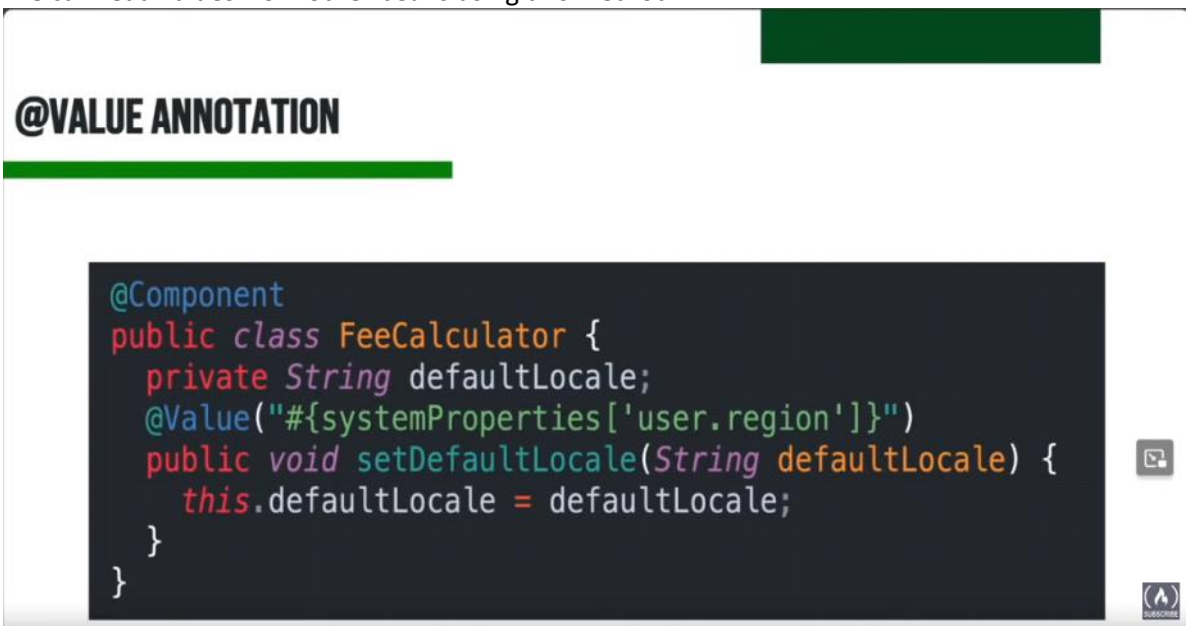
06 May 2024 17:11

@Value Annotation allows us to inject values from a file

@PropertySource allows us to set the location of this file




We can read values from other beans using this method

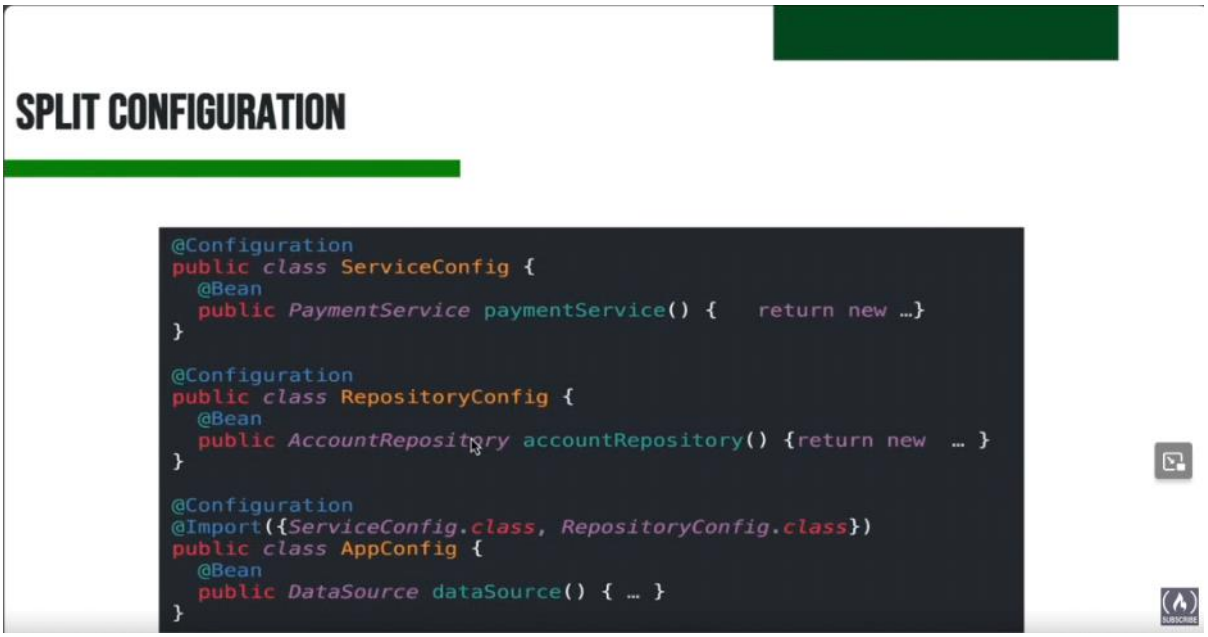


# Split Configuration

06 May 2024 17:16

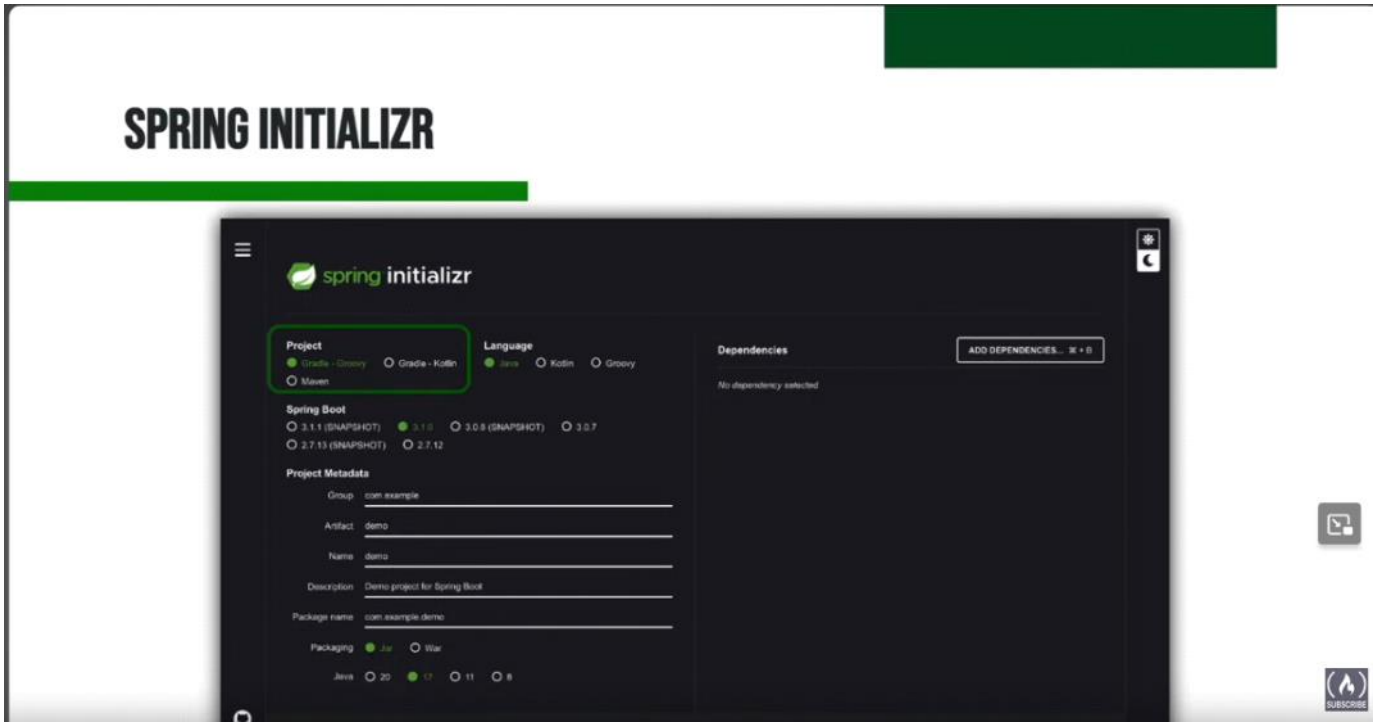
Ensure that all your spring bean configs are split into different classes

Then, merge them using @Import



# Spring Initializer & setup

06 May 2024 17:21




You need Spring web dependency to use it with REST

Port: 8080

Group: com.Company

Artifact: Product Name

In intelj use Project Structure to change Java Version

# Why Spring?

06 May 2024 17:24

Standalone apps

Embeded server (tomcat) etc

Starters - Porm or Gradle (Use groovy)

A lot of Auto Config

A lot of Production ready features

No XML Configs

# File Structure

07 May 2024 05:26

Resources folder:

Static is for static HTML etc (don't use this, use REST)

Application.properties - stores custom spring configuration and/or our own custom configuration values

Test folder:

All tests go in here

Pom.xml:

Parent - means everything extends from this parent (in our case spring-boot-starter-parent)

Dependencies -> dependency - Is where you can view, remove and install new dependency. You can also set there 'scope' i.e in what profile they are available

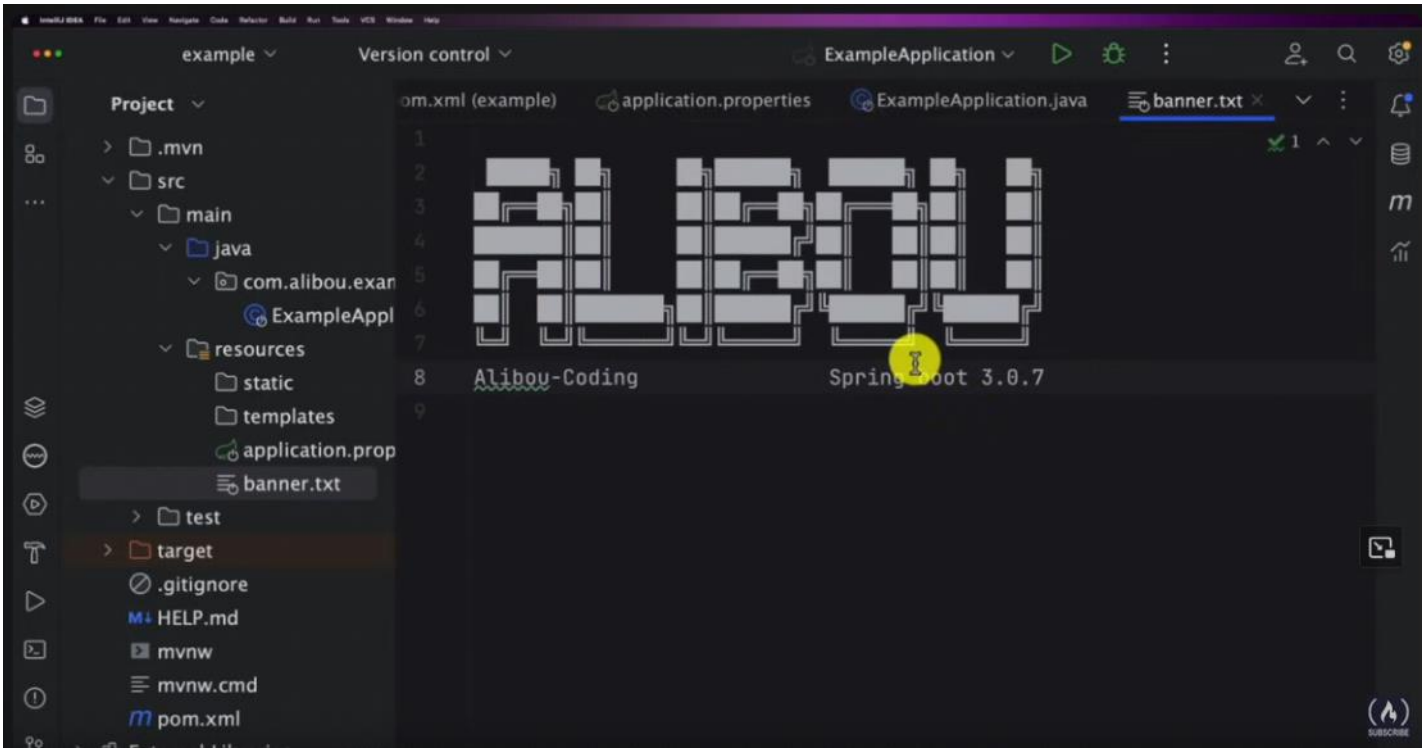
For all dependency - parent version is extended

# Changing the banner

07 May 2024 05:38

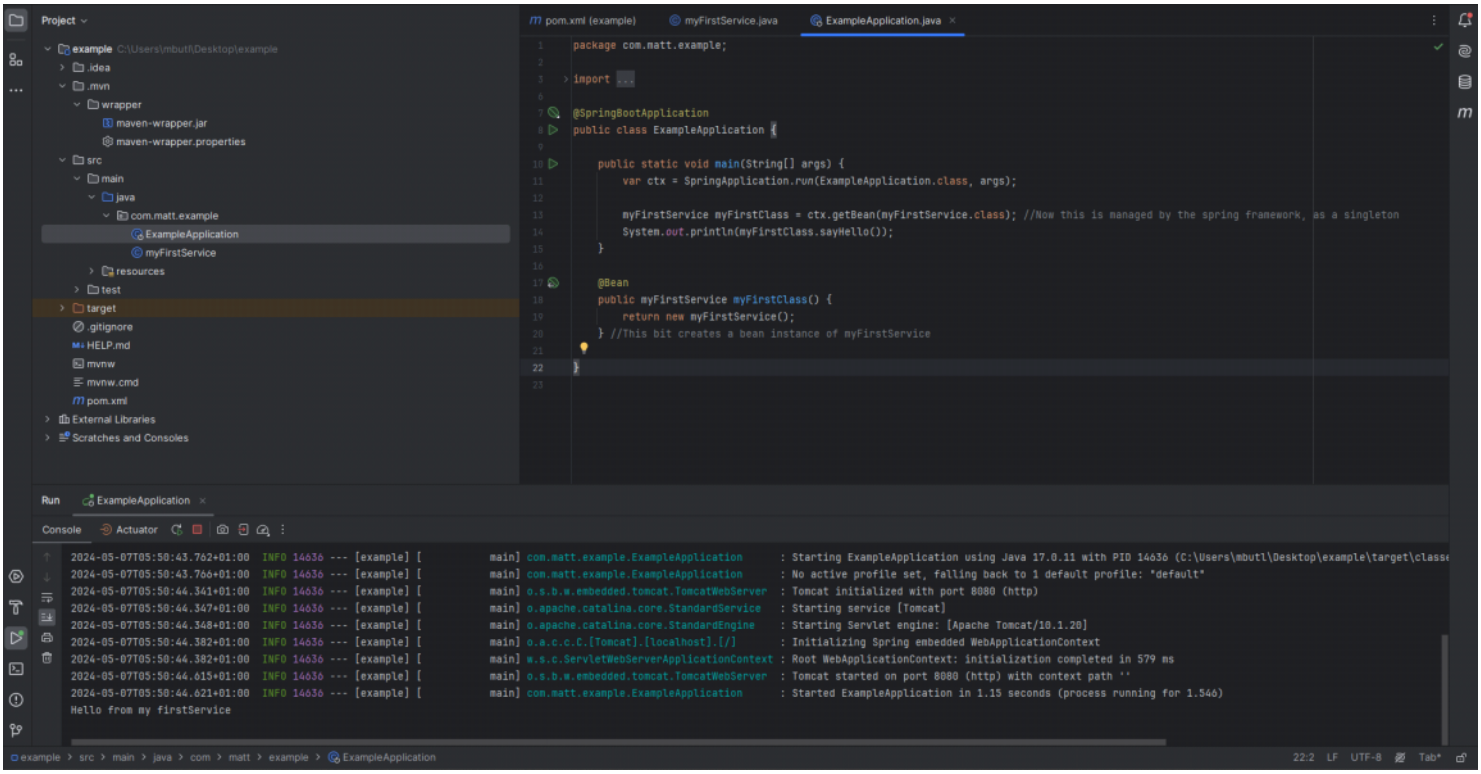
You can change the banner from Spring to whatever

Create a file in resources called banner.txt

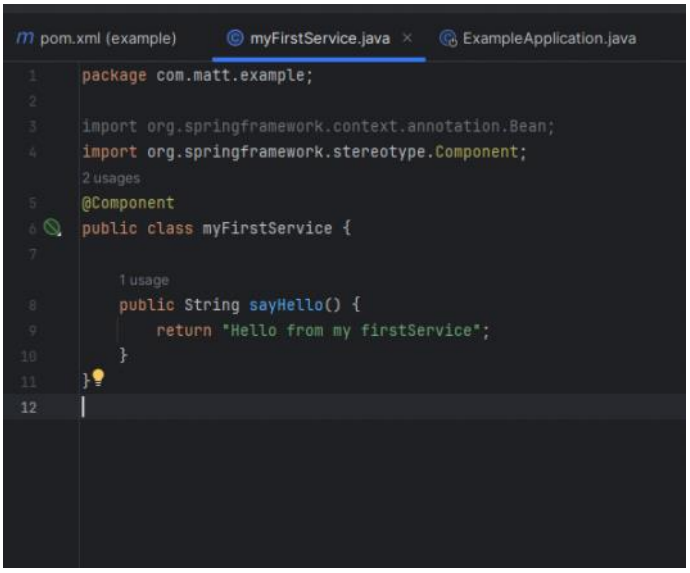


# Beans in action + BP

07 May 2024 05:44




**A BETTER WAY BELOW:**



```java
package com.matt.example;

import ...

@SpringBootApplication
public class ExampleApplication {

    public static void main(String[] args) {
        var ctx = SpringApplication.run(ExampleApplication.class, args);

        myFirstService myFirstClass = ctx.getBean(myFirstService.class); //Now this is managed by the spring framework, as a singleton
        System.out.println(myFirstClass.sayHello());
    }


}

```

**We don't need a bean instance creation if we use Component annotation, same with Service annotation .**

**@Service, @Repository** extends **@Component**

Best practice:

Split your configuration into a ApplicationConfig to handle bean instance creation

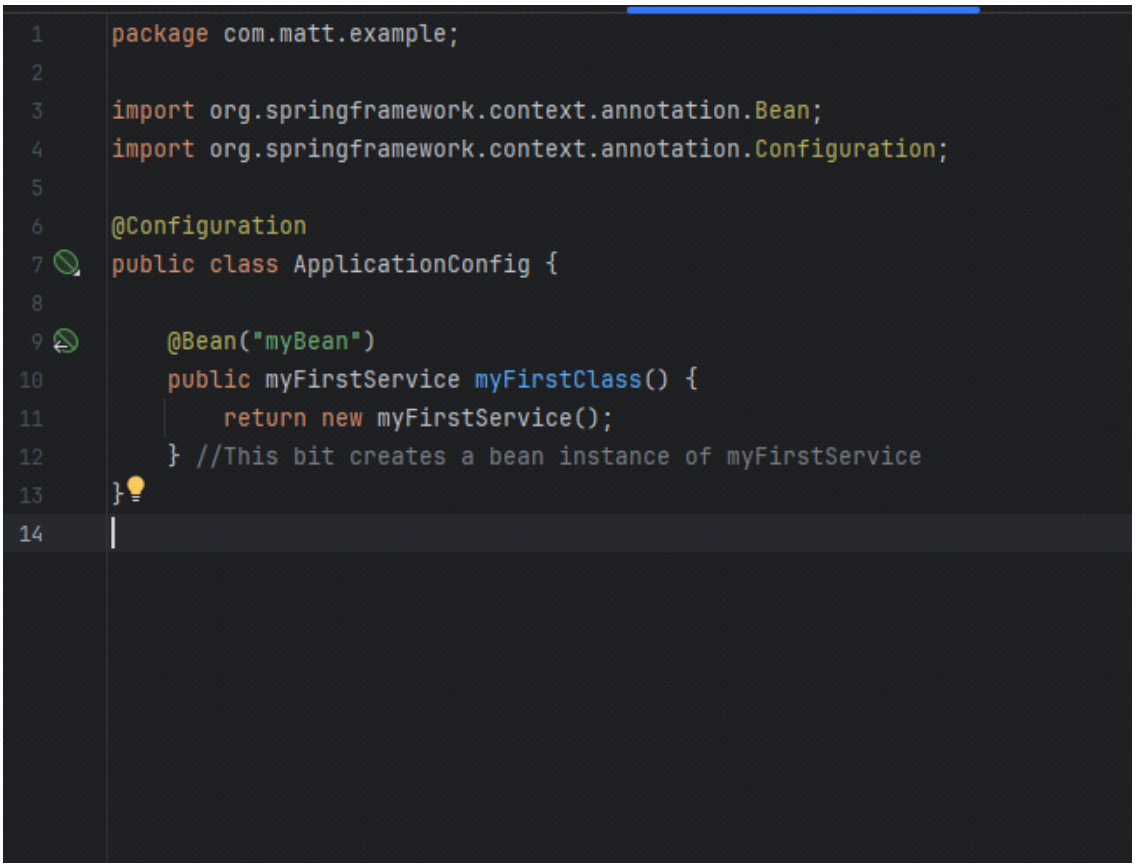
pom.xml (example) | myFirstService.java | ApplicationConfig.java | ExampleApplication.java

```java
package com.matt.example;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ApplicationConfig {

    @Bean
    public myFirstService myFirstClass() {
        return new myFirstService();
    } //This bit creates a bean instance of myFirstService
}

```

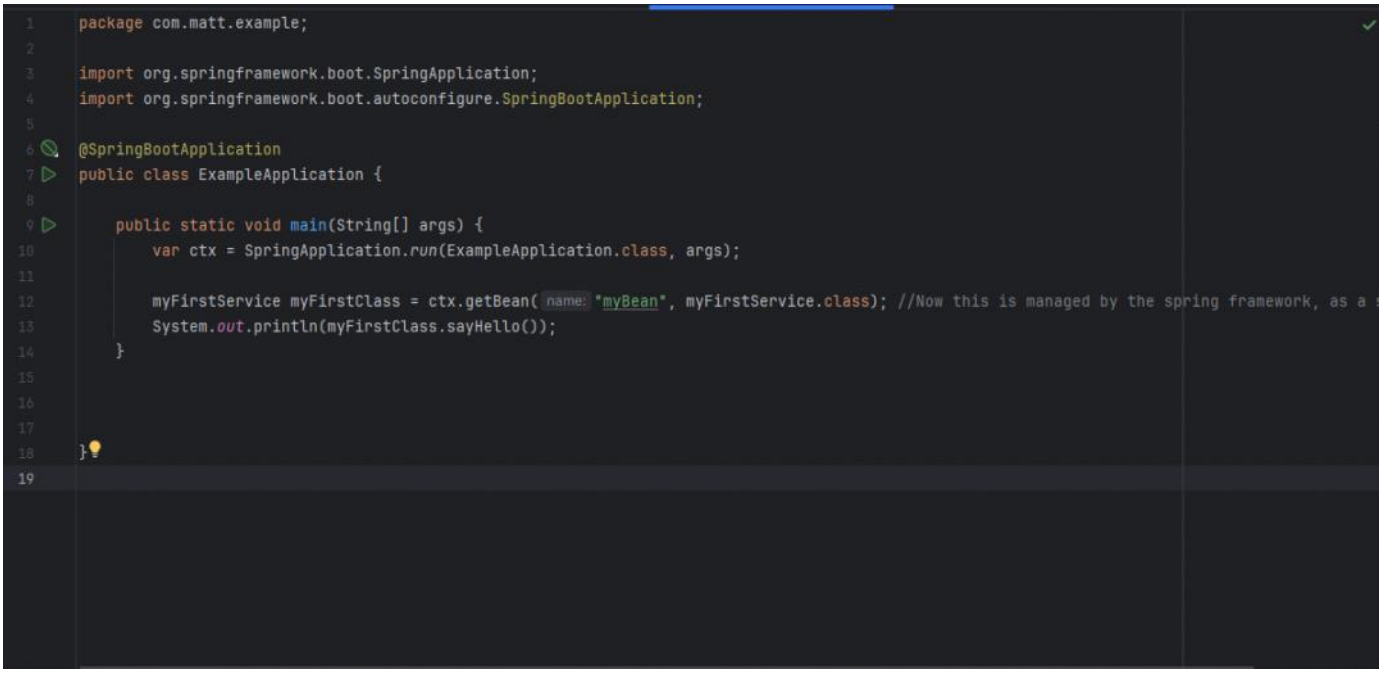
Now no need for @Service, @Component annotation for myFirstService class because its all handled here

# Bean naming in action

07 May 2024 06:02



```java
package com.matt.example;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ApplicationConfig {

    @Bean("myBean")
    public myFirstService myFirstClass() {
        return new myFirstService();
    } //This bit creates a bean instance of myFirstService
}

```



```java
package com.matt.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ExampleApplication {

    public static void main(String[] args) {
        var ctx = SpringApplication.run(ExampleApplication.class, args);

        myFirstService myFirstClass = ctx.getBean( name: "myBean", myFirstService.class); //Now this is managed by the spring framework, as a
        System.out.println(myFirstClass.sayHello());
    }


}

```

# Beans & Constructor Dependency Injection

07 May 2024 06:22

## Autowired handles Dependency injection

Autowired ensures that all bean dependencies are initialized when needed

Without autowired in below code there wouldn't be an instance of myFirstClass bean for constructor to use

**KEEP IN MIND: MyFirstCLass is the BEAN here (as per our ApplicationConfig) not myFirstService**

**MyFirstService is where the MyFirstClass bean is injected**

```java
package com.matt.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

2 usages
@Service
public class myFirstService {

    2 usages
    private final myFirstClass myFirstClass;


    @Autowired
    public myFirstService(myFirstClass myFirstClass) {
        this.myFirstClass = myFirstClass;
    } //Dependency beans are initialized using Autowired

    1 usage
    public String TellAStory() {
        return "the dependency is saying:  " + myFirstClass.sayHello();
    };
}
```

In modern spring, we don't need autowired, spring will search for a bean called "myFirstClass" and inject it

# Lots of beans...

07 May 2024 06:55

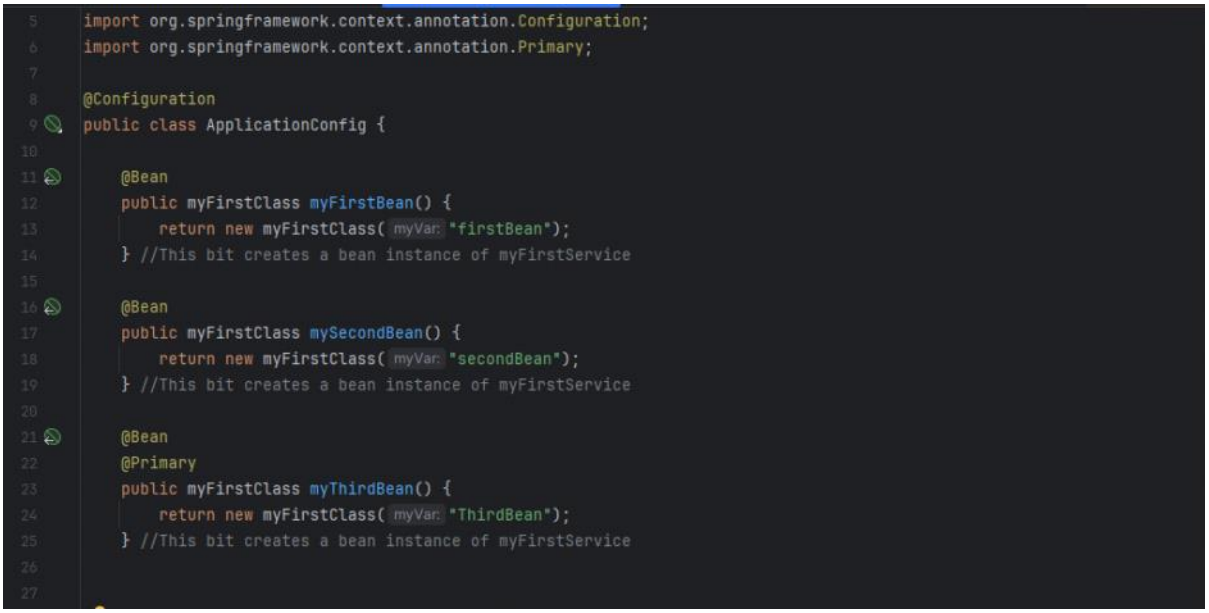
**If you have serval beans of the same type use qualifiers**

```
2 usages
@Service
public class myFirstService {

    2 usages
    private final myFirstClass myFirstClass;

    @Autowired
    public myFirstService(@Qualifier("bean1") myFirstClass myFirstClass) {
        this.myFirstClass = myFirstClass;
    } //Dependency beans are initialized using Autowired
```

```
6
7   @Configuration
8   public class ApplicationConfig {
9
10      @Bean("myBean")
11      @Qualifier("bean1")
12      public myFirstClass myFirstBean() {
13          return new myFirstClass( myVar: "firstBean");
14      } //This bit creates a bean instance of myFirstService
15
16      @Bean("myBean")
17      @Qualifier("bean2")
18      public myFirstClass mySecondBean() {
19          return new myFirstClass( myVar: "secondBean");
20      } //This bit creates a bean instance of myFirstService
21  }
22
```

**This also works with different types (int strings)**

You can tell Spring in what the 'primary'  bean is instead of using qualifers



```java
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration
public class ApplicationConfig {

    @Bean
    public myFirstClass myFirstBean() {
        return new myFirstClass( myVar: "firstBean");
    } //This bit creates a bean instance of myFirstService

    @Bean
    public myFirstClass mySecondBean() {
        return new myFirstClass( myVar: "secondBean");
    } //This bit creates a bean instance of myFirstService

    @Bean
    @Primary
    public myFirstClass myThirdBean() {
        return new myFirstClass( myVar: "ThirdBean");
    } //This bit creates a bean instance of myFirstService


```

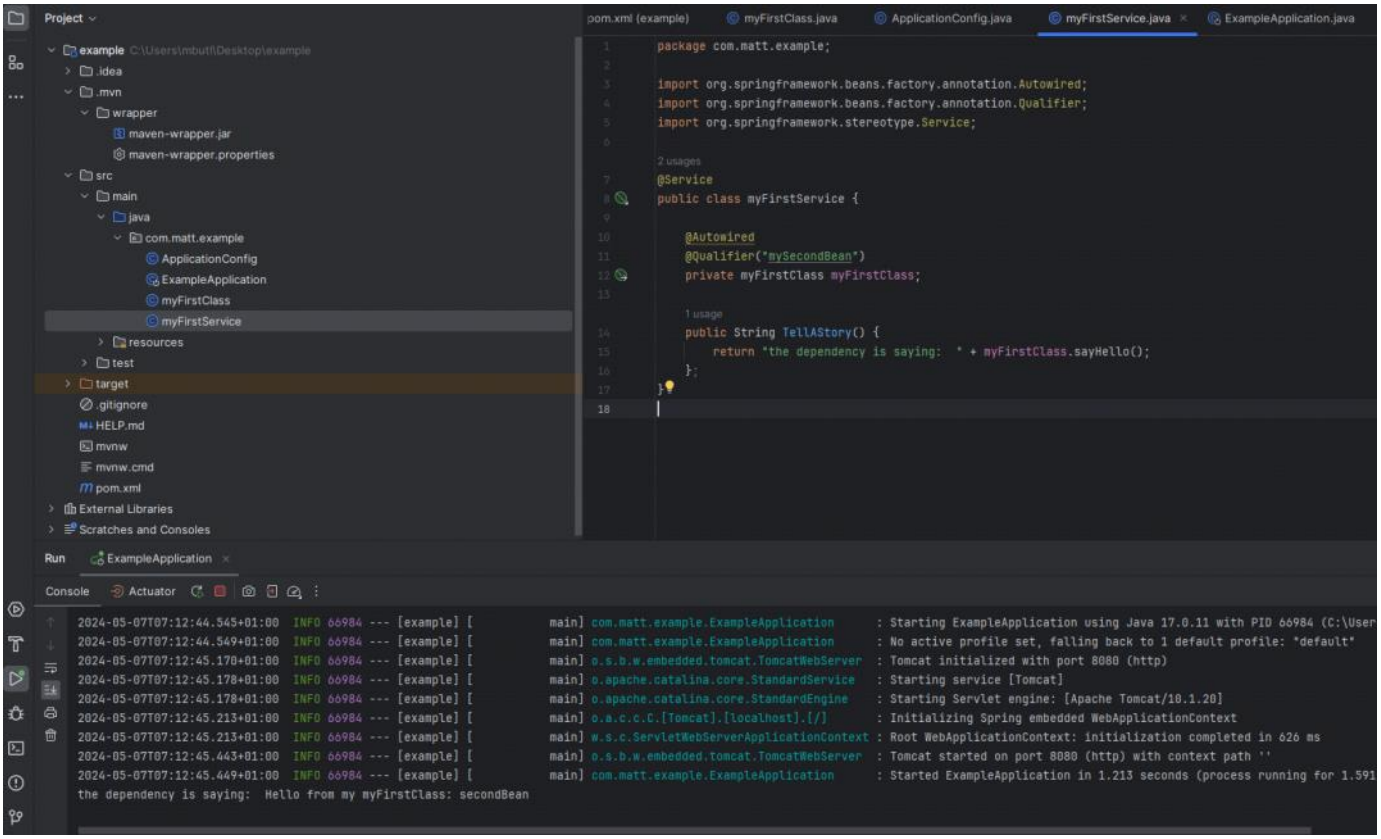
# Naming & Field injection

07 May 2024 07:07

```java
package com.matt.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

2 usages
@Service
public class myFirstService {

    @Autowired
    private myFirstClass myFirstClass;

    1 usage
    public String TellAStory() {
        return "the dependency is saying:  " + myFirstClass.sayHello();
    };
}
```

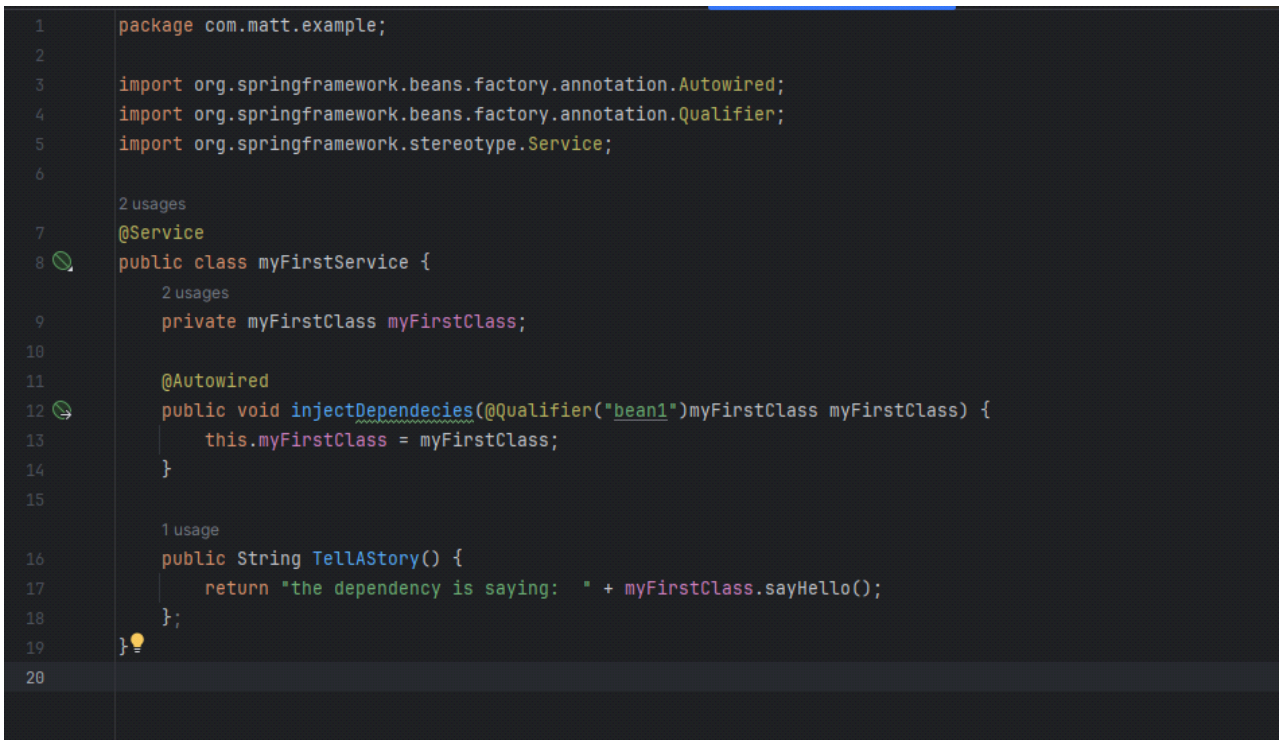
DON’T USE THIS

The default name of beans are there method names so qualifiers can be used like this:


Project
example C:\Users\mbutl\Desktop\example
.idea
.mvn
wrapper
maven-wrapper.jar
maven-wrapper.properties
src
main
java
com.matt.example
ApplicationConfig
ExampleApplication
myFirstClass
myFirstService
resources
test
target
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
External Libraries
Scratches and Consoles
pom.xml (example)
myFirstClass.java
ApplicationConfig.java
myFirstService.java
ExampleApplication.java
package com.matt.example;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;
2 usages
@Service
public class myFirstService {
@Autowired
@Qualifier("mySecondBean")
private myFirstClass myFirstClass;
1 usage
public String TellAStory() {
return "the dependency is saying: " + myFirstClass.sayHello();
};
}
Run
ExampleApplication
Console
Actuator
2024-05-07T07:12:44.545+01:00 INFO 66984 --- [example] [ main] com.matt.example.ExampleApplication : Starting ExampleApplication using Java 17.0.11 with PID 66984 (C:\User
2024-05-07T07:12:44.549+01:00 INFO 66984 --- [example] [ main] com.matt.example.ExampleApplication : No active profile set, falling back to 1 default profile: "default"
2024-05-07T07:12:45.170+01:00 INFO 66984 --- [example] [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-05-07T07:12:45.178+01:00 INFO 66984 --- [example] [ main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-05-07T07:12:45.178+01:00 INFO 66984 --- [example] [ main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.20]
2024-05-07T07:12:45.213+01:00 INFO 66984 --- [example] [ main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-05-07T07:12:45.213+01:00 INFO 66984 --- [example] [ main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 626 ms
2024-05-07T07:12:45.443+01:00 INFO 66984 --- [example] [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-05-07T07:12:45.449+01:00 INFO 66984 --- [example] [ main] com.matt.example.ExampleApplication : Started ExampleApplication in 1.213 seconds (process running for 1.591
the dependency is saying: Hello from my myFirstClass: secondBean

# Method injection

07 May 2024 07:15



```java
package com.matt.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

2 usages
@Service
public class myFirstService {
    2 usages
    private myFirstClass myFirstClass;

    @Autowired
    public void injectDependecies(@Qualifier("bean1")myFirstClass myFirstClass) {
        this.myFirstClass = myFirstClass;
    }

    1 usage
    public String TellAStory() {
        return "the dependency is saying:  " + myFirstClass.sayHello();
    };
}
```

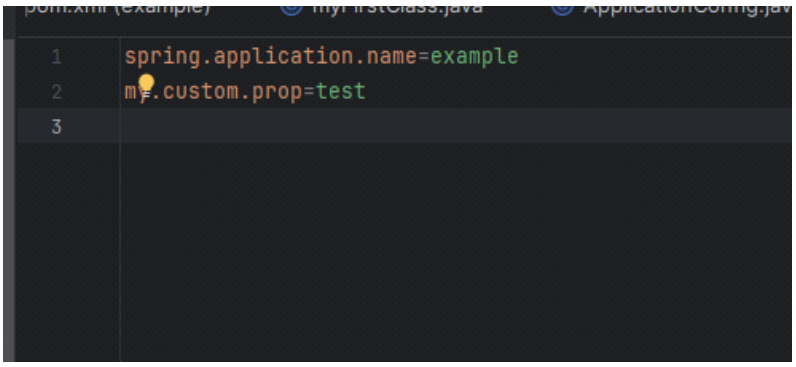
# Environment bean

07 May 2024 07:29

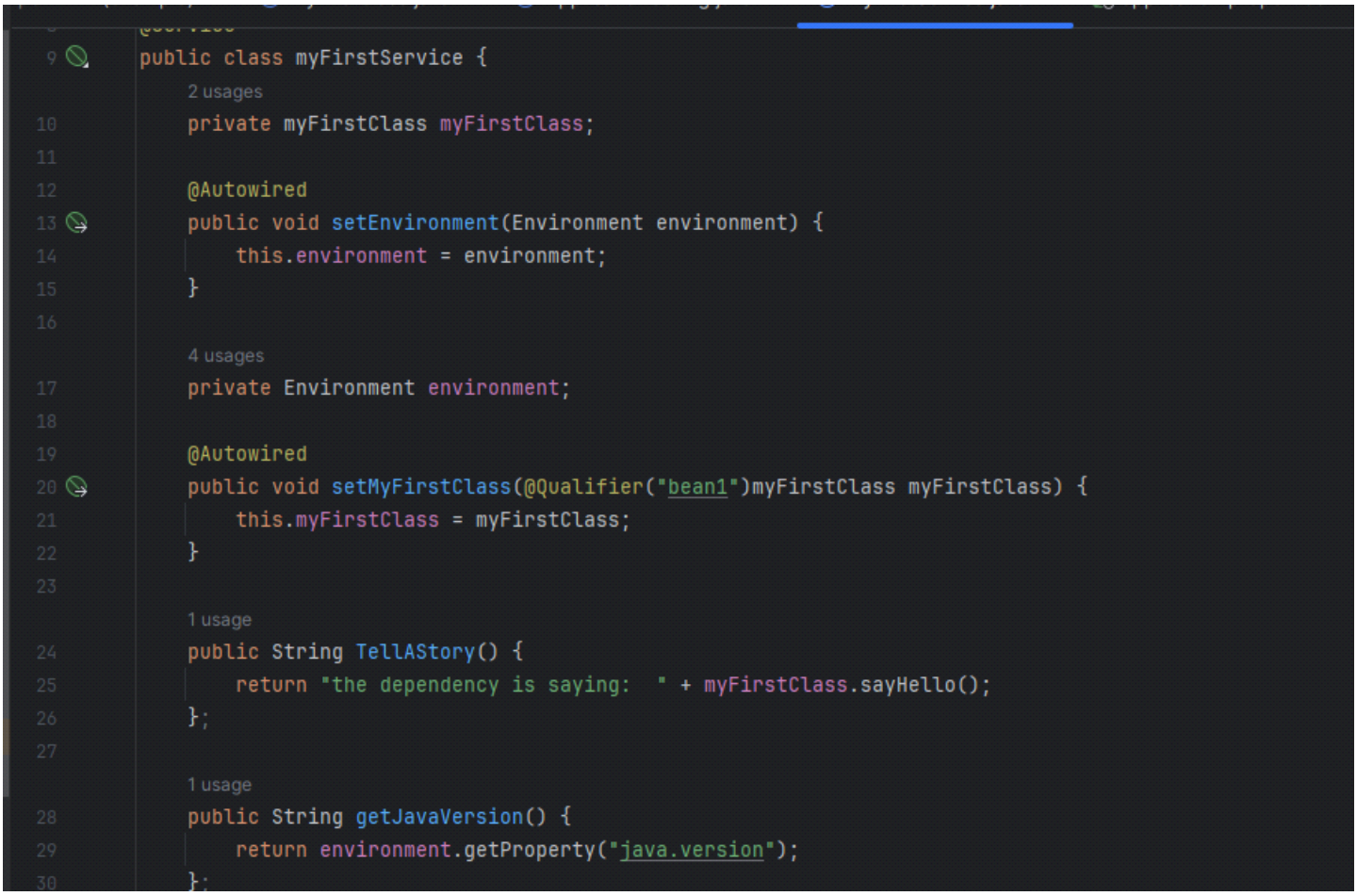
Beans can be injected in any way (ideally constructor) in below examples we use setter injection

The environment bean has lots of built in functionality

We can add custom props in application.properties file



```
spring.application.name=example
my.custom.prop=test
```

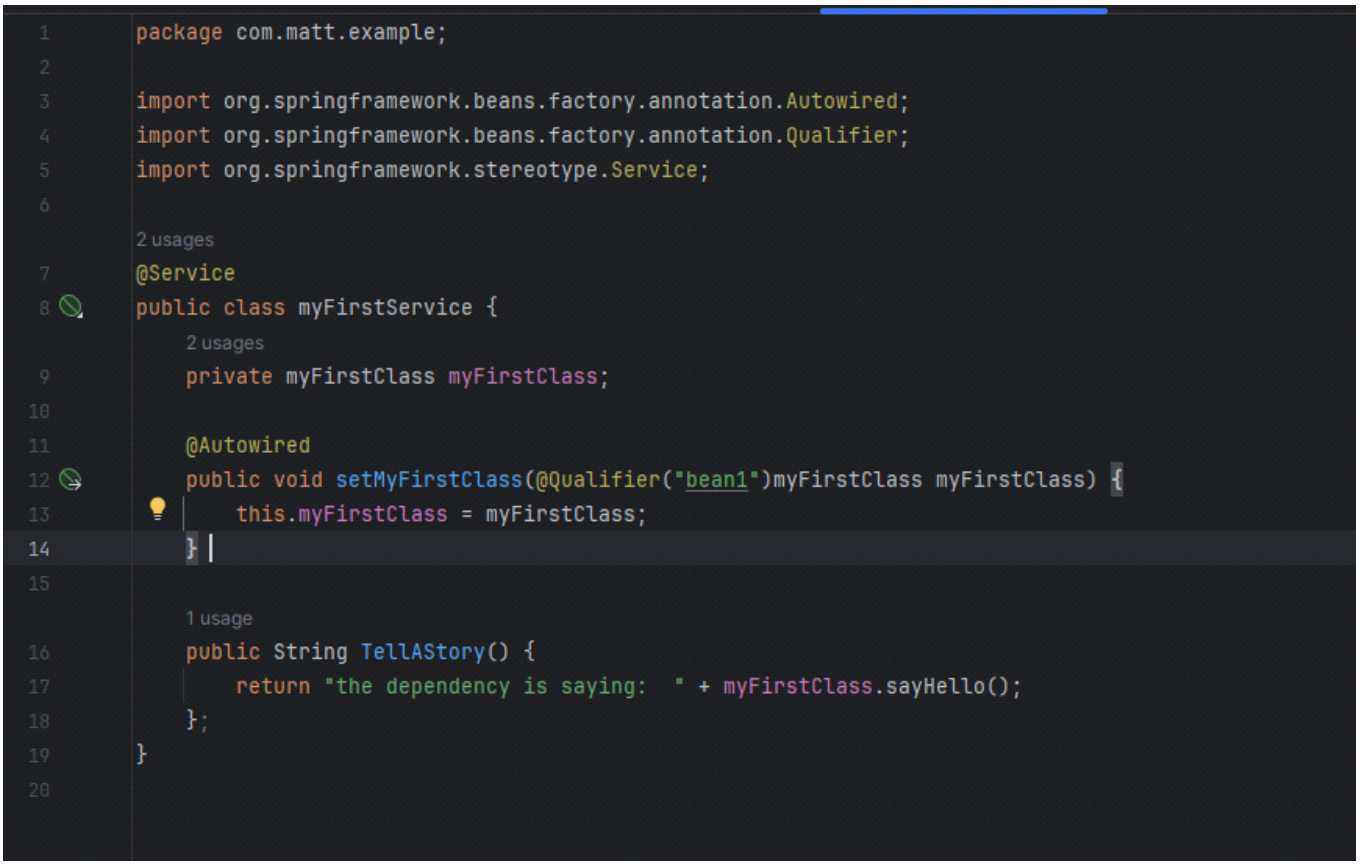


```
public class myFirstService {
    2 usages
    private myFirstClass myFirstClass;

    @Autowired
    public void setEnvironment(Environment environment) {
        this.environment = environment;
    }

    4 usages
    private Environment environment;

    @Autowired
    public void setMyFirstClass(@Qualifier("bean1")myFirstClass myFirstClass) {
        this.myFirstClass = myFirstClass;
    }

    1 usage
    public String TellAStory() {
        return "the dependency is saying:  " + myFirstClass.sayHello();
    };

    1 usage
    public String getJavaVersion() {
        return environment.getProperty("java.version");
    };
```

# Setter injection

07 May 2024 07:25

Create a setter for each bean to handle injection



```java
package com.matt.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

2 usages
@Service
public class myFirstService {
    2 usages
    private myFirstClass myFirstClass;

    @Autowired
    public void setMyFirstClass(@Qualifier("bean1")myFirstClass myFirstClass) {
        this.myFirstClass = myFirstClass;
    }

    1 usage
    public String TellAStory() {
        return "the dependency is saying:  " + myFirstClass.sayHello();
    };
}
```

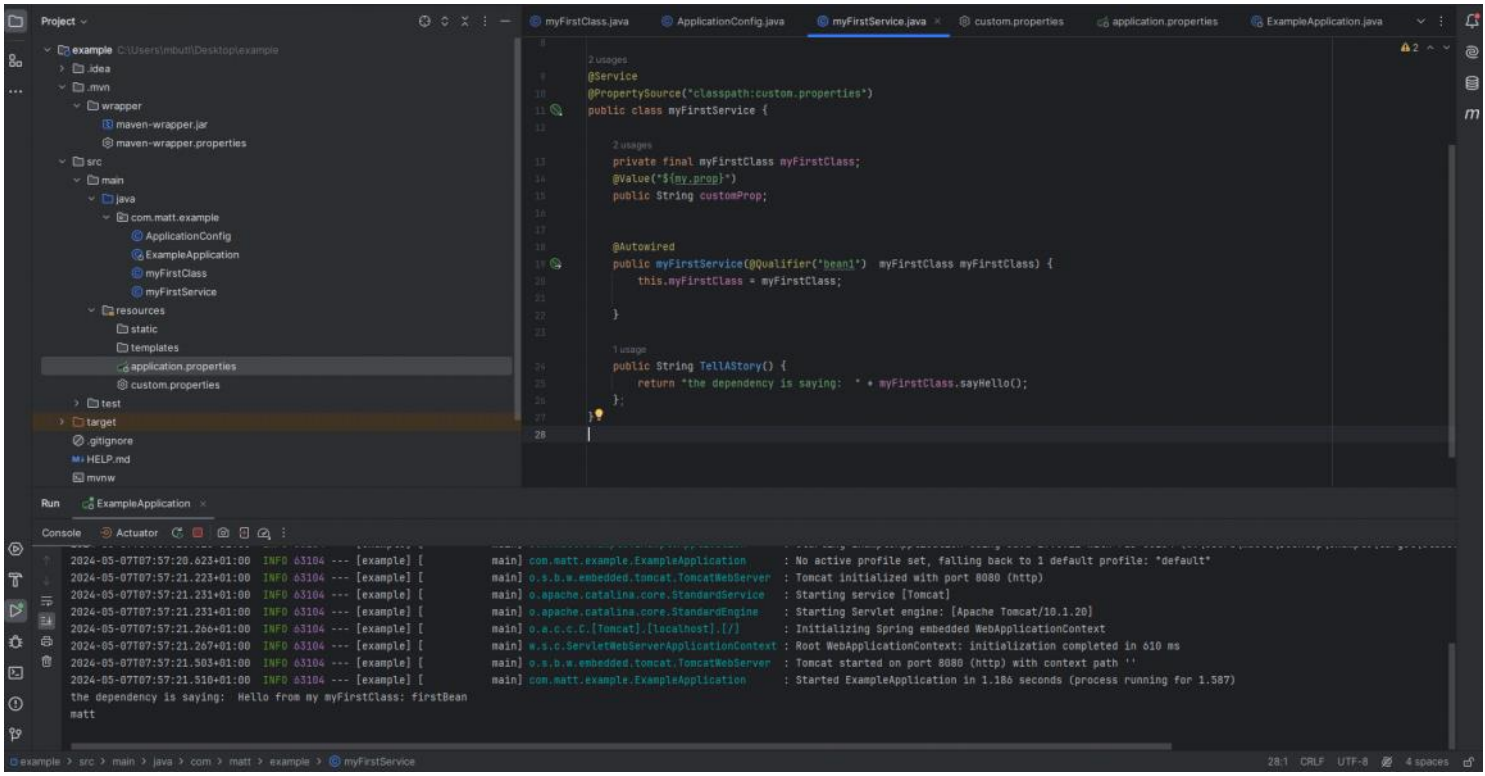
ALWAYS USE CONSTRUCTOR INJECTOR

# $Value

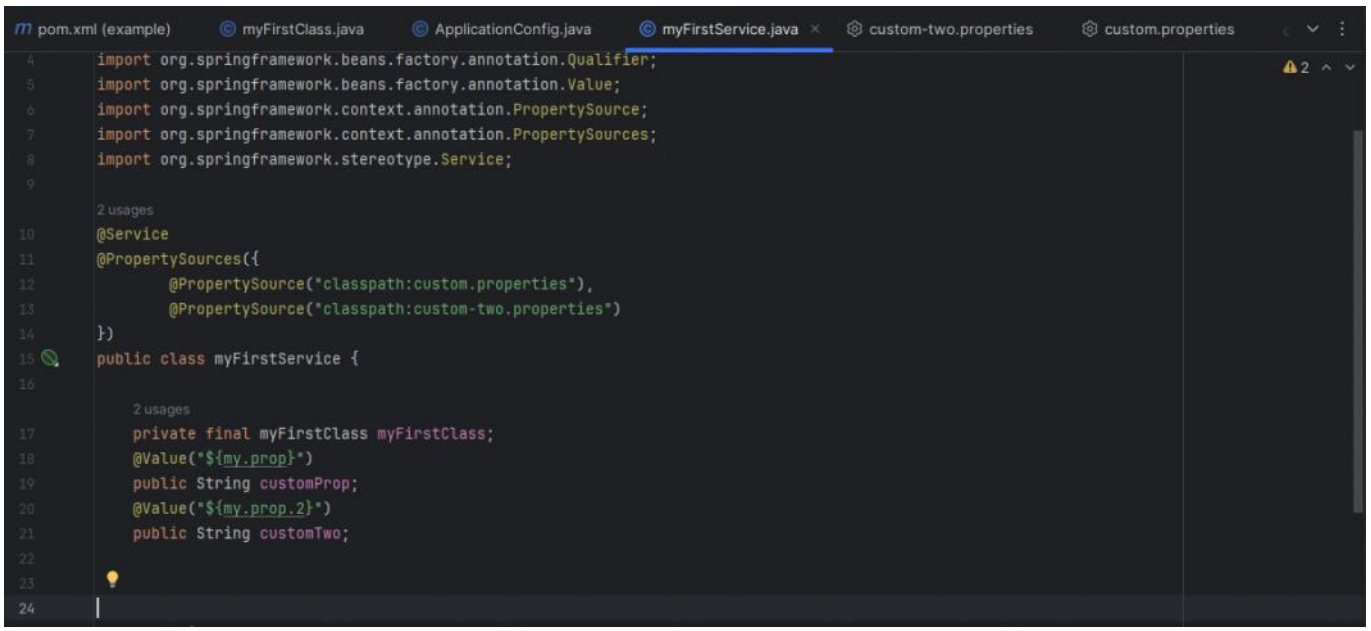
07 May 2024 07:51

$Value Is used to read custom values stored in external files

Set a custom location with PropertySource



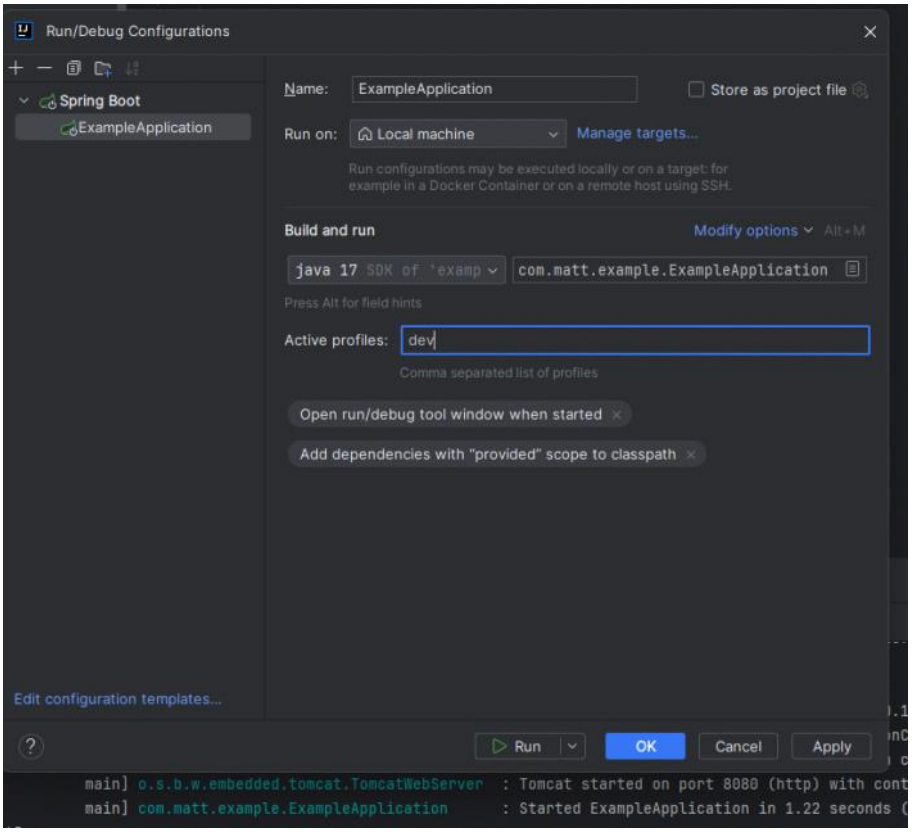

Using two files:



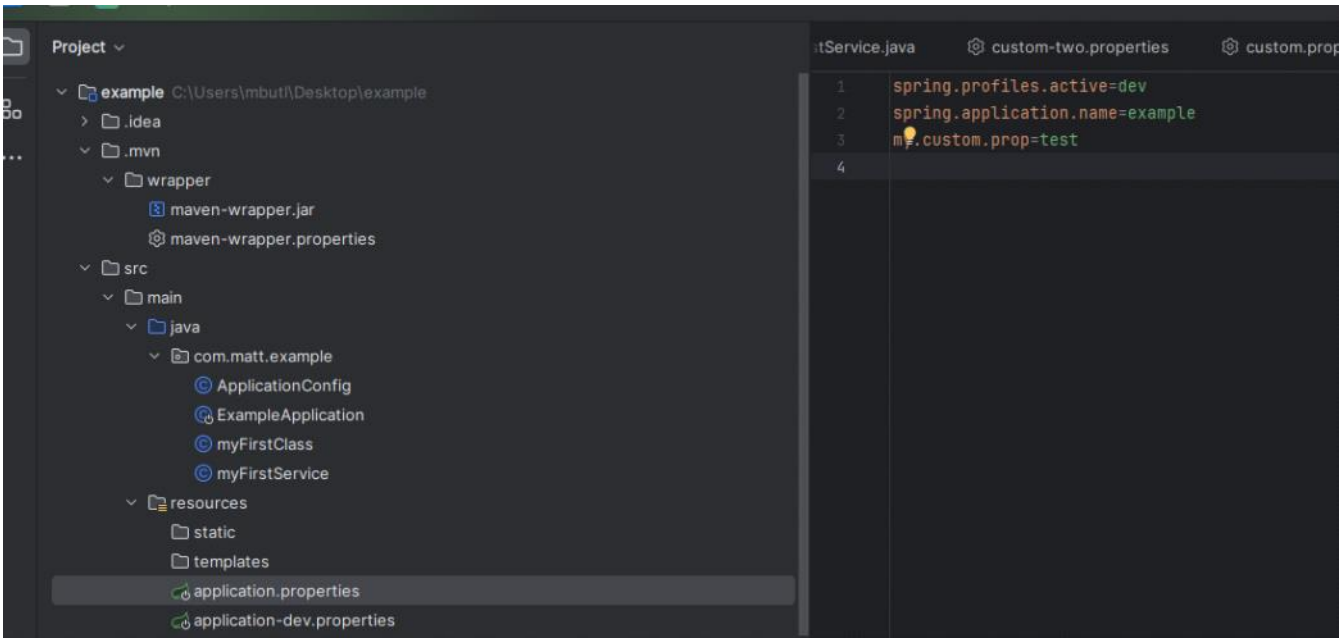
# Spring Profiles

07 May 2024 08:16

Spring profiles allow you to activate and de-active beans depending on if it's a production environment or not




OR:

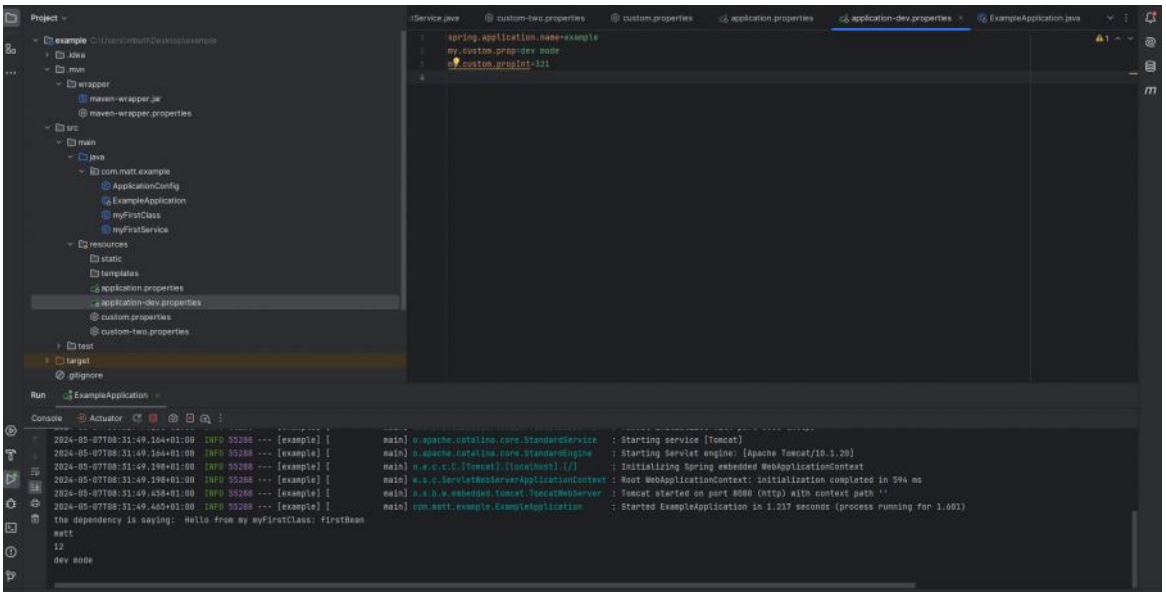


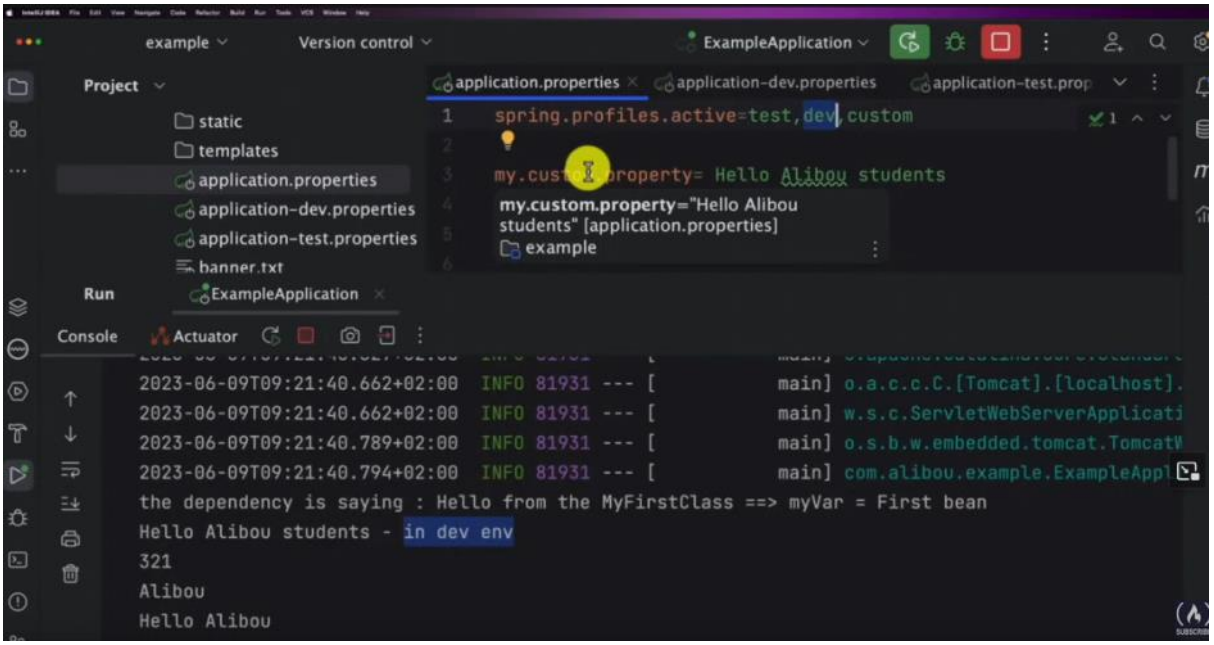

OR:

roperties custom.properties application.properties application-test.properties application-dev

```java

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.Collections;

@SpringBootApplication
public class ExampleApplication {

    public static void main(String[] args) {
        var app = new SpringApplication(ExampleApplication.class);
        app.setDefaultProperties(Collections.singletonMap("spring.profiles.active", "dev"));
        var ctx = app.run(args);
```

Will use Dev named properties file (check screenshot)






The final profile in the list is the 'overriding' one

# Profile available beans

07 May 2024 10:37

```java

@Configuration
public class ApplicationConfig {

    @Bean
    @Qualifier("bean1")
    @Profile("dev")
    public myFirstClass myFirstBean() {
        return new myFirstClass( myVar: "firstBean");
    } //This bit creates a bean instance of myFirstService

    @Bean
    @Profile("test")
    public myFirstClass mySecondBean() {
        return new myFirstClass( myVar: "secondBean");
    } //This bit creates a bean instance of myFirstService

    @Bean
    @Primary
    public myFirstClass myThirdBean() {
        return new myFirstClass( myVar: "ThirdBean");
    } //This bit creates a bean instance of myFirstService

```

**These beans are now only available on dev and test profiles respectively**

Make a configuration profile dependent:

myFirstClass.java | ApplicationConfig.java | myFirstService.java | custom-two.properties | custom.properties | application.properties

Change profiles...

```java
@Configuration
@Profile("dev")
public class ApplicationConfig {

    @Bean
    @Qualifier("bean1")
    public myFirstClass myFirstBean() {
        return new myFirstClass( myVar: "firstBean");
    } //This bit creates a bean instance of myFirstService

    @Bean
    @Profile("test")
    public myFirstClass mySecondBean() {
        return new myFirstClass( myVar: "secondBean");
    } //This bit creates a bean instance of myFirstService

    @Bean
    @Primary
    public myFirstClass myThirdBean() {
        return new myFirstClass( myVar: "ThirdBean");
    } //This bit creates a bean instance of myFirstService

```

# REST overview

07 May 2024 11:12

## SPRING REST - OVERVIEW

- REST = Representational State Transfer
- REST was first defined by Roy Fielding in his 2000 Doctoral dissertation
- Client-Server Architecture
- Stateless
- Cacheable
- Layered System
- Code on Demand (optional)
- Uniform Interface
  - Identification of resources
  - Manipulation of resources through these representations
  - Self-descriptive messages
  - Hypermedia As The Engine Of Application State (HATEOS)

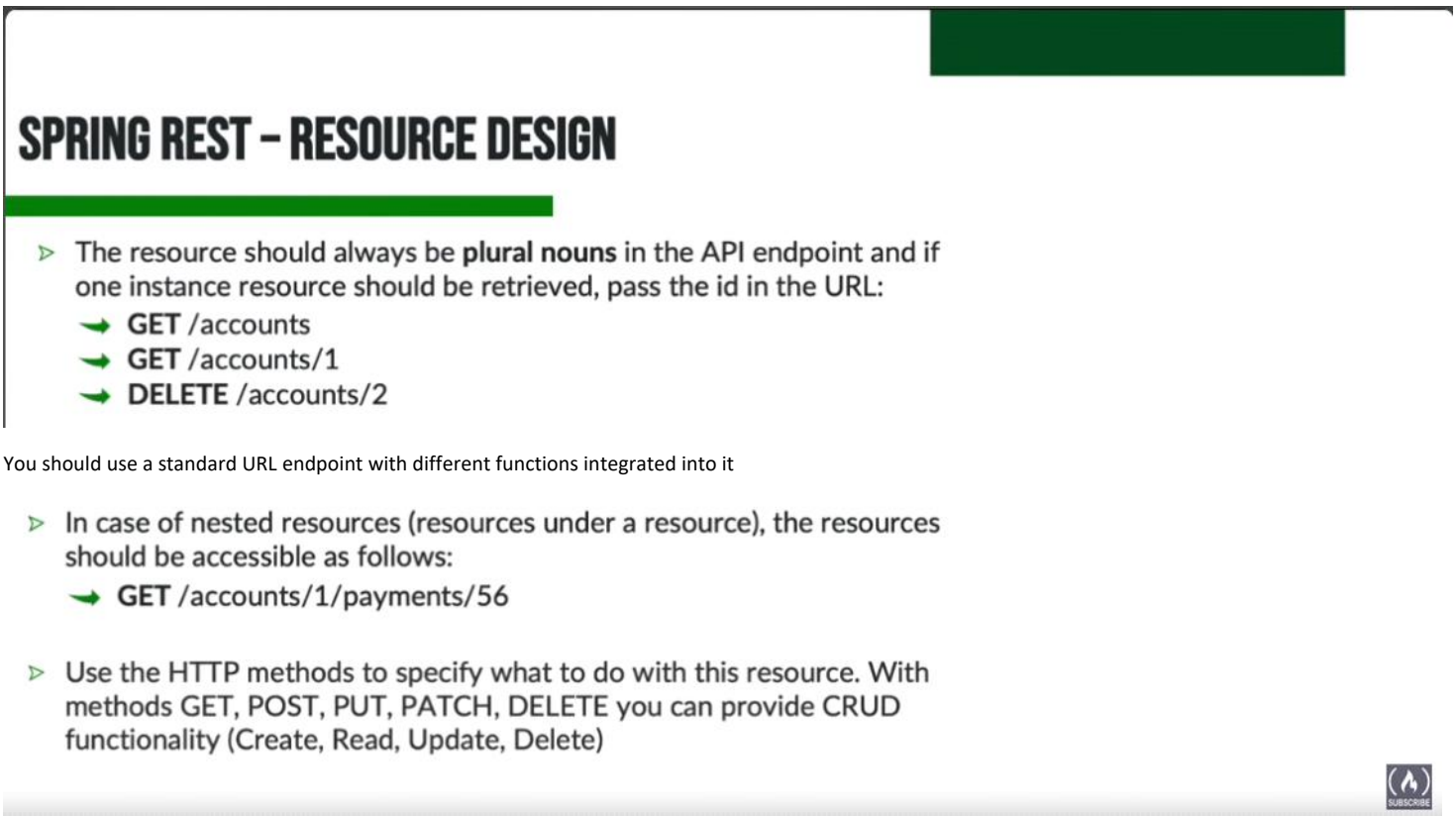
## SPRING REST - OVERVIEW

- The Web's architectural principles
- Unique Identification of resources (URI)
- Different resource representation
- Hypermedia / Linking of resources
- Stateless communication
- Standard methods (GET, POST, PUT, DELETE) and responses (200 OK, 404 Not Found)

# REST Resource Design

07 May 2024 11:20

You should always use Plural nouns for API endpoints



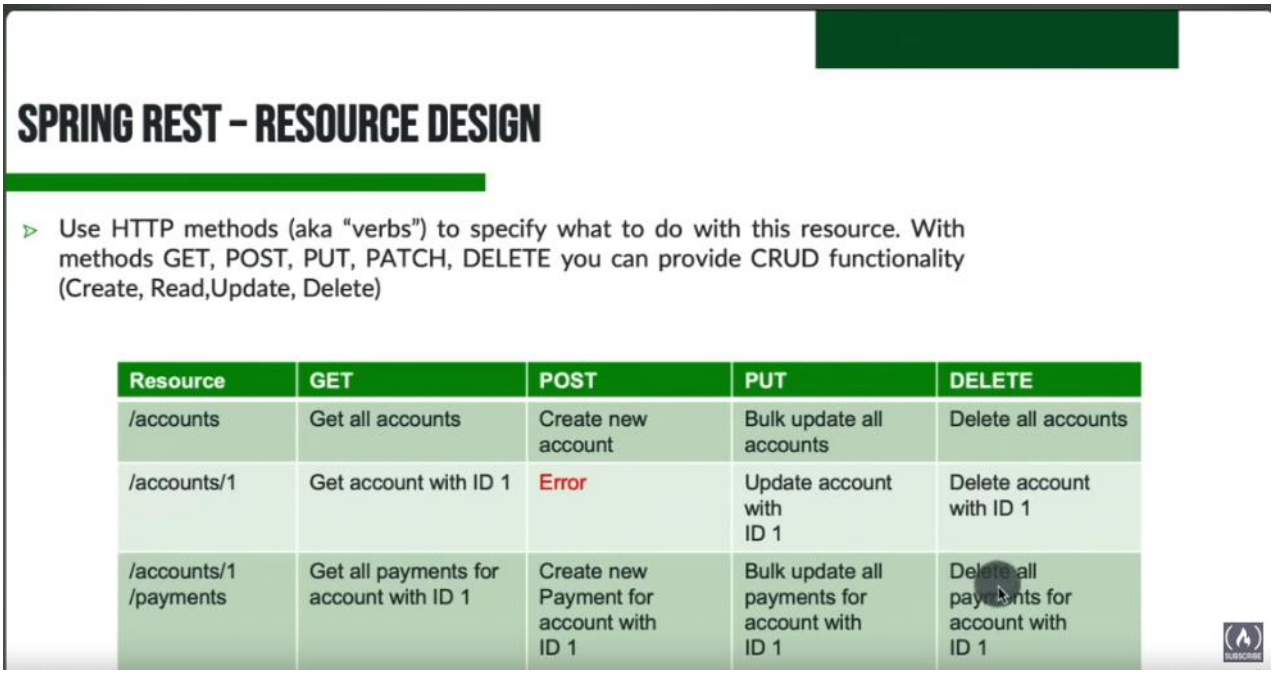

You should use a standard URL endpoint with different functions integrated into it




To access a resource with nested context do as above

Always use correct HTTP methods for CRUD

**Table example:**



# HTTP Methods

07 May 2024 11:28

- **GET**:
  - Retrieve data from the server.
  - Used for reading resources.
  - Idempotent (multiple identical requests will produce the same result).
- **POST**:
  - Submit data to the server.
  - Create a new resource or perform a custom action.
  - Not idempotent (multiple identical requests may result in different outcomes).
- **PUT**:
  - Update data on the server.
  - Typically used to update an existing resource with the provided data.
  - Idempotent (multiple identical requests will produce the same result).
- **PATCH**:
  - Partially update data on the server.
  - Used when you want to update only a part of the resource.
  - Typically used when a full resource update via PUT is not practical.
  - Not always guaranteed to be idempotent.
- **DELETE**:
  - Remove data from the server.
  - Used to delete a resource.
  - Idempotent (multiple identical requests will produce the same result).
- **OPTIONS**:
  - Get information about the communication options available for the target resource.
  - Used to describe the communication options for the target resource.
- **HEAD**:
  - Retrieve the headers for a resource.
  - Similar to GET but only retrieves the headers, not the actual content.
  - Useful for checking resource metadata without downloading the entire resource.

These methods are fundamental building blocks for interacting with RESTful APIs, each serving a specific purpose in the CRUD (Create, Read, Update, Delete) operations.

From <https://chatgpt.com/c/c20759ce-54e7-403e-acd3-a7ed55823995>

# HTTP status codes

07 May 2024 11:30

## Codes:

100 (Info):

200 (success):

200 - ok went as planned, data returned if needed

201 - Created, resource created

204 - ok went as planned, no data (delete, put, post)

300 (redirection):

304 - not modified, is cached version from client up to date, saves a lot bandwidth

400 (client error):

400 - bad request, bad client request form

401 - unauthorized, invalid creds or bad permissions etc

403 - bad permissions, can we used to hide details

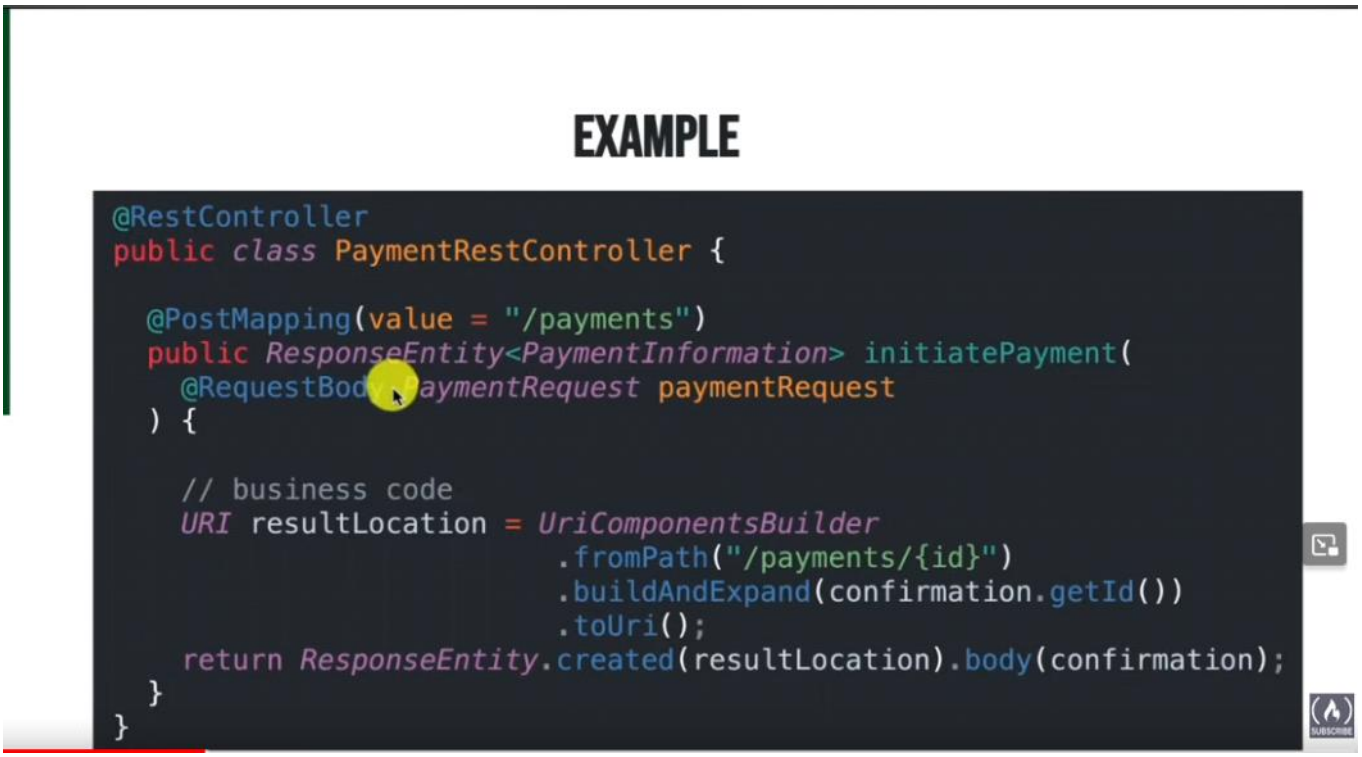
500 (server error):

500 - internal server error (servers fault, no more info)

503 - Server unavailable, not available for a short amount of time, header includes how long to wait before retry

# Rest example

07 May 2024 14:14




**PostMapping - HTTP verb**

**ResponseEntity<PaymentInfomation> - Will return a ResponseEntity containing a PaymentInfomation instance**

**RequestBody PaymentRequest paymentRequest - A payment request body is expected with request**

**Ingore resultLocation**

**Return ResponseEnitity.created() - is what sorts our response, created being HTTP code**

# ResponoseStatus

07 May 2024 14:21

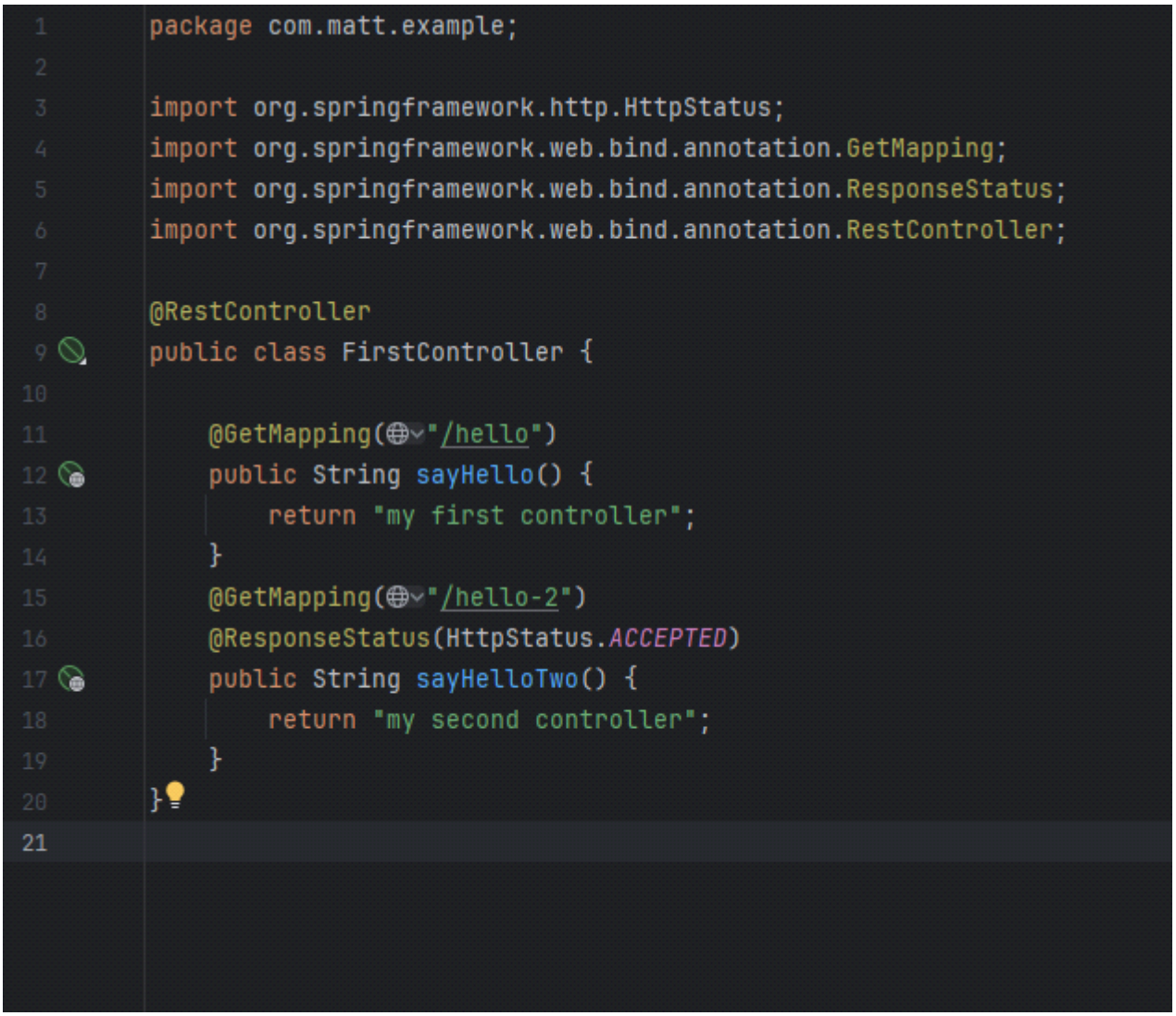
Using @ResponseStatus you can set your response code

**This is only used if no expectations are made**

# RestController

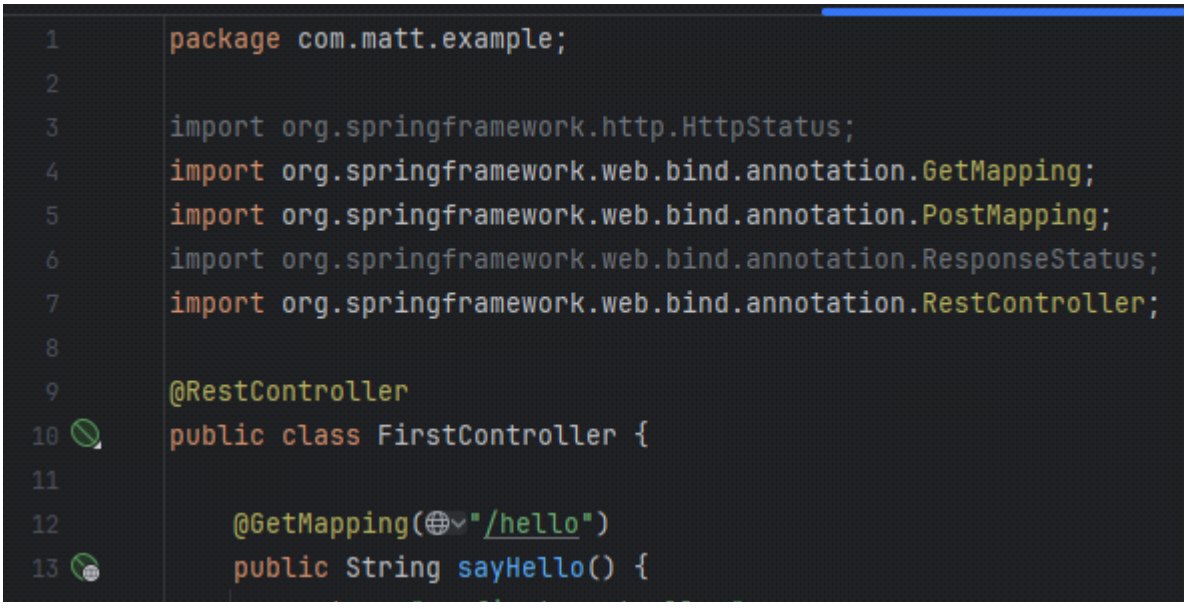
07 May 2024 14:36

Look at screenshot for details

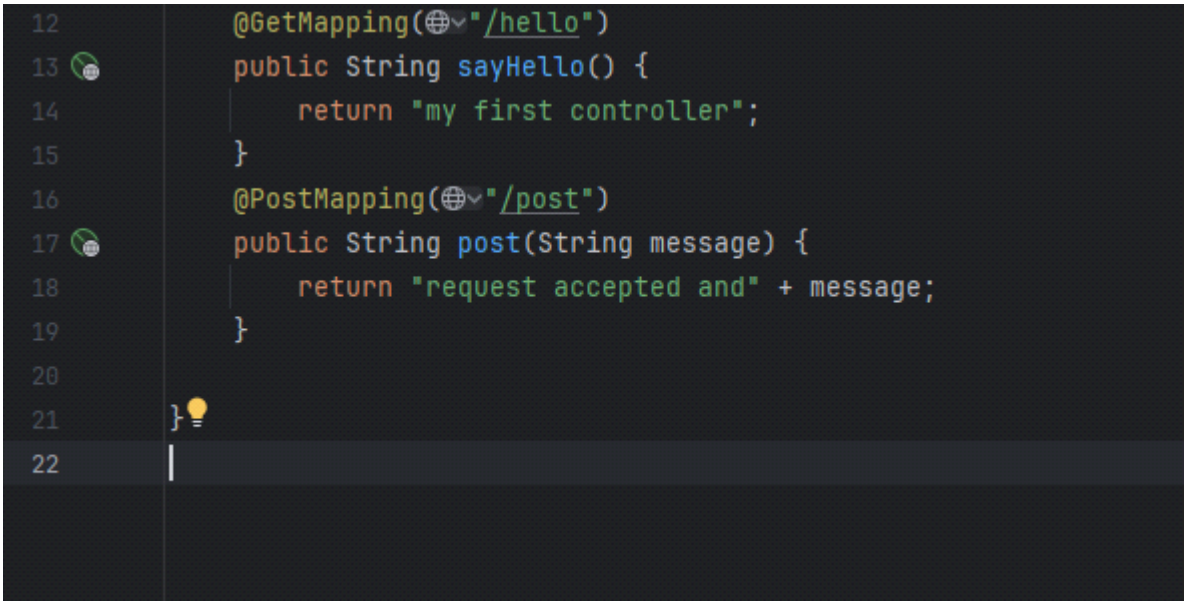


```java
package com.matt.example;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FirstController {

    @GetMapping("/hello")
    public String sayHello() {
        return "my first controller";
    }
    @GetMapping("/hello-2")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public String sayHelloTwo() {
        return "my second controller";
    }
}

```

ResponseStatus only used with no exceptions



```java
package com.matt.example;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FirstController {

    @GetMapping("/hello")
    public String sayHello() {
```




**Note request body used to specify input body requirements**

Obviously we will normally use model but he's how to do it with variables

Because its just a variable not a model we can use raw

'input' assigned as message

# Rest Controller with JSON input

07 May 2024 14:54

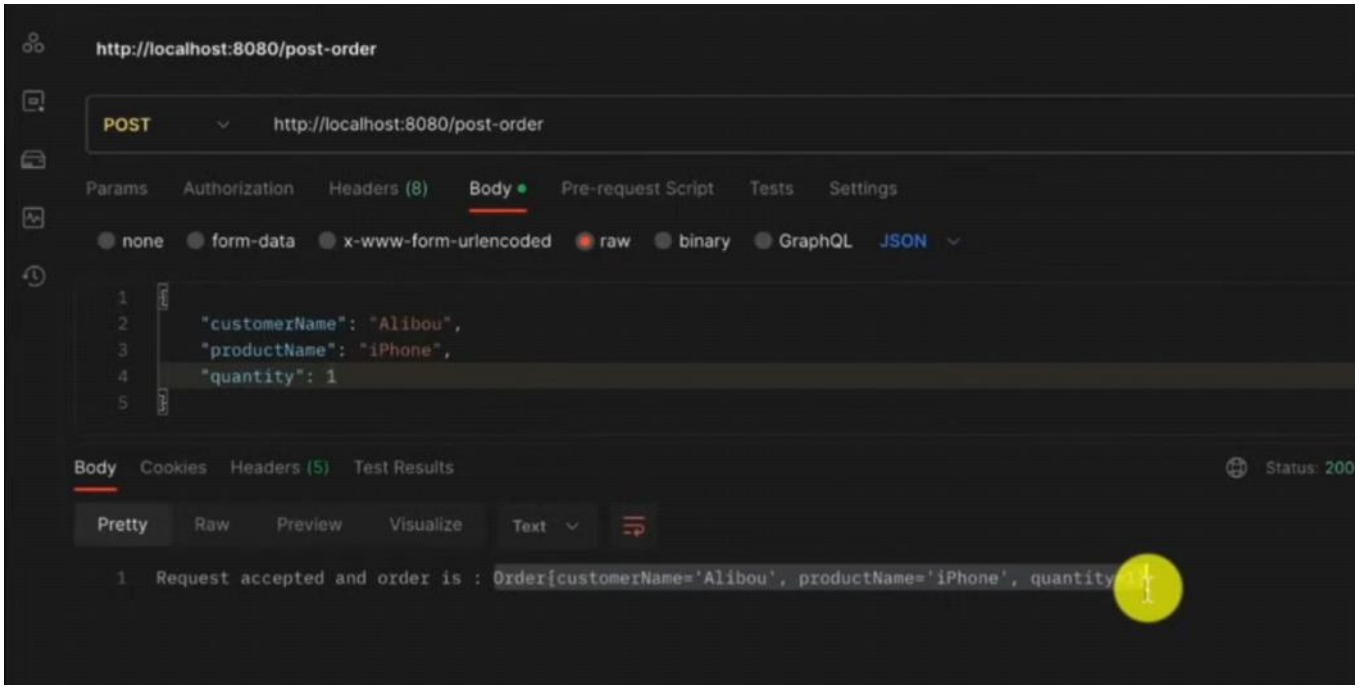
```java
package com.matt.example;

1 usage
public class Order {

    3 usages
    private String customerName;
    3 usages
    private String ProductName;
    3 usages
    private int quantity;

    @Override
    public String toString() {
        return "Order{" +
                "customerName='" + customerName + '\'' +
                ", ProductName='" + ProductName + '\'' +
                ", quantity=" + quantity +
                '}';
    }

    no usages
    public String getCustomerName() {
        return customerName;
    }

}
```

**Custom To String method generated from intelj**

**@overide is overidding the default Object parent class toString method**

You must have accessors (getters and setters) to serialize and deserialize the data

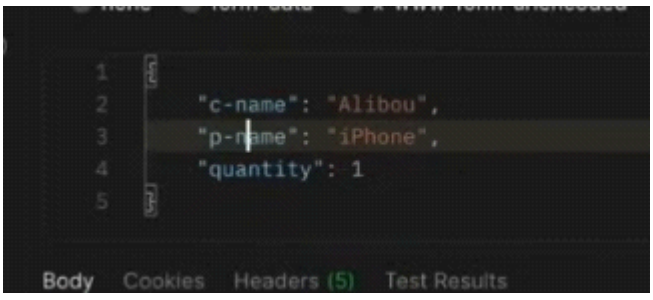
```java
        return "request accepted and" + message;
    }
    @PostMapping("/post-order")
    public String postOrder(@RequestBody Order order) {
        return "request accepted and" + order.toString();
    }

}

```


http://localhost:8080/post-order
POST
http://localhost:8080/post-order
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"customerName": "Alibou",
"productName": "iPhone",
"quantity": 1
}
Body
Cookies
Headers (5)
Test Results
Status: 200
Pretty
Raw
Preview
Visualize
Text
Request accepted and order is : Order{customerName='Alibou', productName='iPhone', quantity=1}

# Renaming request Json field compared to class

08 May 2024 05:34

```java
import com.fasterxml.jackson.annotation.JsonProperty;

1 usage
public class Order {


    3 usages
    @JsonProperty("c-name")
    private String customerName;


    3 usages
    @JsonProperty("p-name")
    private String ProductName;

    3 usages
    @JsonProperty("quantity")
    private int quantity;

    @Override
    public String toString() {
        return "Order{" +
                "customerName='" + customerName + '\'' +
                ", ProductName='" + ProductName + '\'' +
                ", quantity=" + quantity +
                '}';
    }
```




JsonProperty allows you to rename fields

# Java Records

08 May 2024 05:44

POJO classes are normally used for data transfer however, from version 14 records can be used

You can use java records as request body's in Spring

```
1  package com.matt.example;
2
3  import com.fasterxml.jackson.annotation.JsonProperty;
4
   1 usage
5  public record orderRecord(String customerName,
                             no usages
6                            String productName,
                             no usages
7                            int Quality) {
8
9  }
10
```

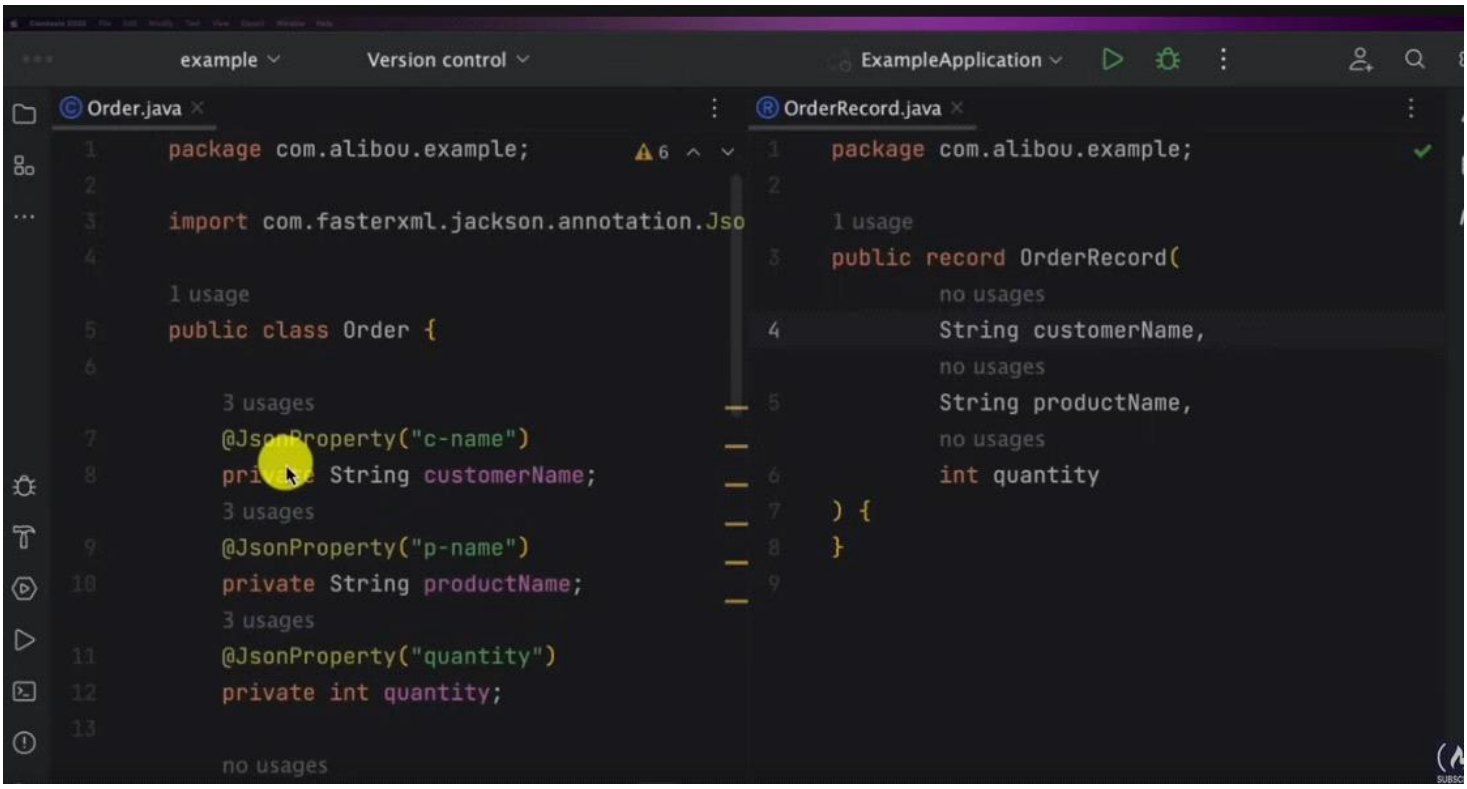
Records do not require you to make getters and setters to serialize and deserialize

The toString function formats the output correctly without needing an override

```
20     }
21     @PostMapping("/post-record")
22     public String postRecord(@RequestBody orderRecord record) {
23         return "request accepted and" + record.toString();
24     }
```

# Records vs POJO Classes

08 May 2024 05:55




Classes:

Requires ToString override

Requires setters and getters to serialize and deserialize

Can have logic within the class

Records:

Records are final so they can't be used with mutable requirement library

A better choice for simple data transfer with no logic

# Path variables & request params

08 May 2024 05:58

Path variables allows us to do things like

**/hello/Matt (username)**

If URL path name is the same as variable name:

```
25
26      @GetMapping("/hello/{username}")
27      public String pathVar(@PathVariable String username) {
28
29          return "my value = " + username;
30      }
```

Use parameter if not

```
25
26      @GetMapping("/hello/{user-name}")
27      public String pathVar(@PathVariable("user-name") String username) {
28
29          return "my value = " + username;
30      }
```

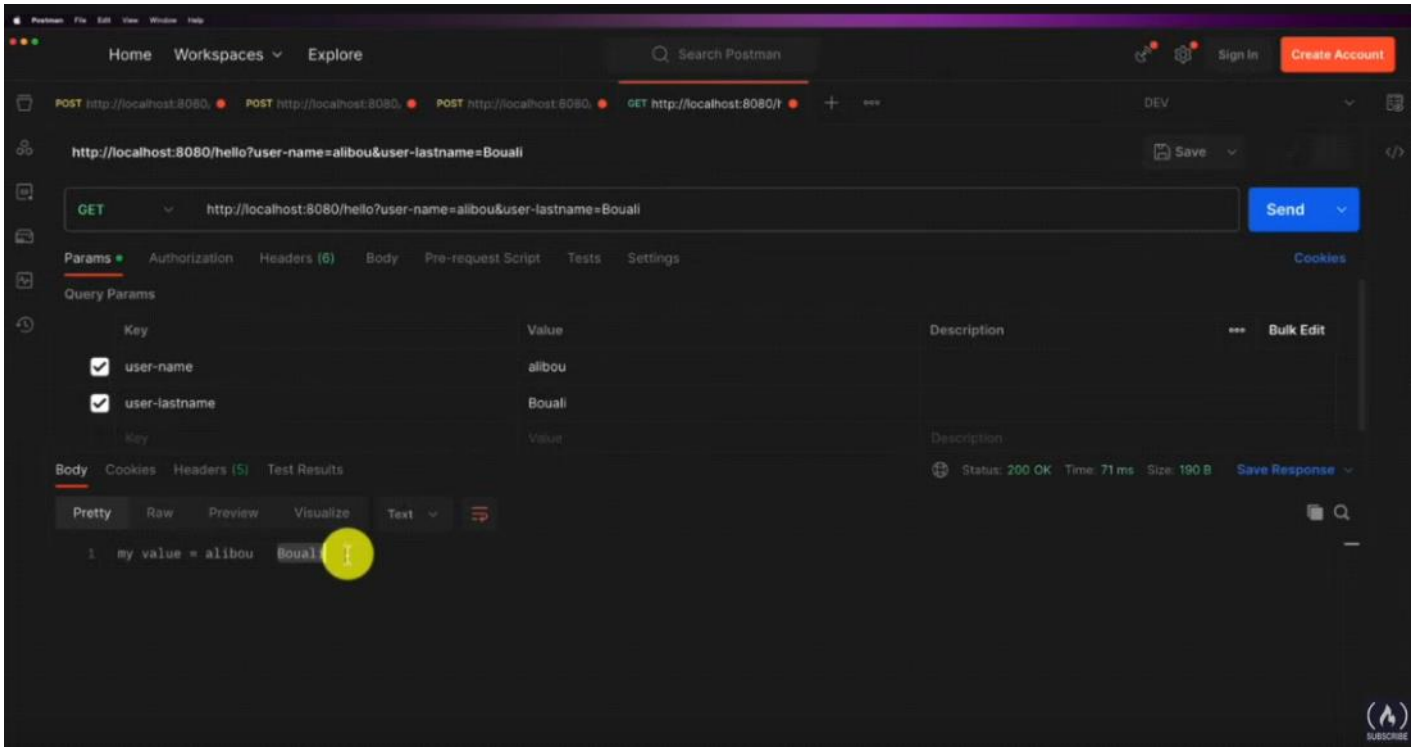
**This way is recommended for dash readability**

## Request Parms:

Request param:

**/hello/?paramNamepvaram_value&param_value_2=value_2**

```
25
26      @GetMapping("/helloParams")
27      public String pathVar(@RequestParam("param_value") String ParamValue, @RequestParam("param_value_2") String ParamValue2) {
28
29          return "my value = " + ParamValue + ParamValue2;
30      }
31
32  }
33
```


Home
Workspaces
Explore
Search Postman
Sign In
Create Account
DEV
http://localhost:8080/hello?user-name=alibou&user-lastname=Bouali
Save
GET
http://localhost:8080/hello?user-name=alibou&user-lastname=Bouali
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
user-name
alibou
user-lastname
Bouali
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 71 ms
Size: 190 B
Save Response
Pretty
Raw
Preview
Visualize
Text
1 my value = alibou Boual

# Path Variables vs Request Params

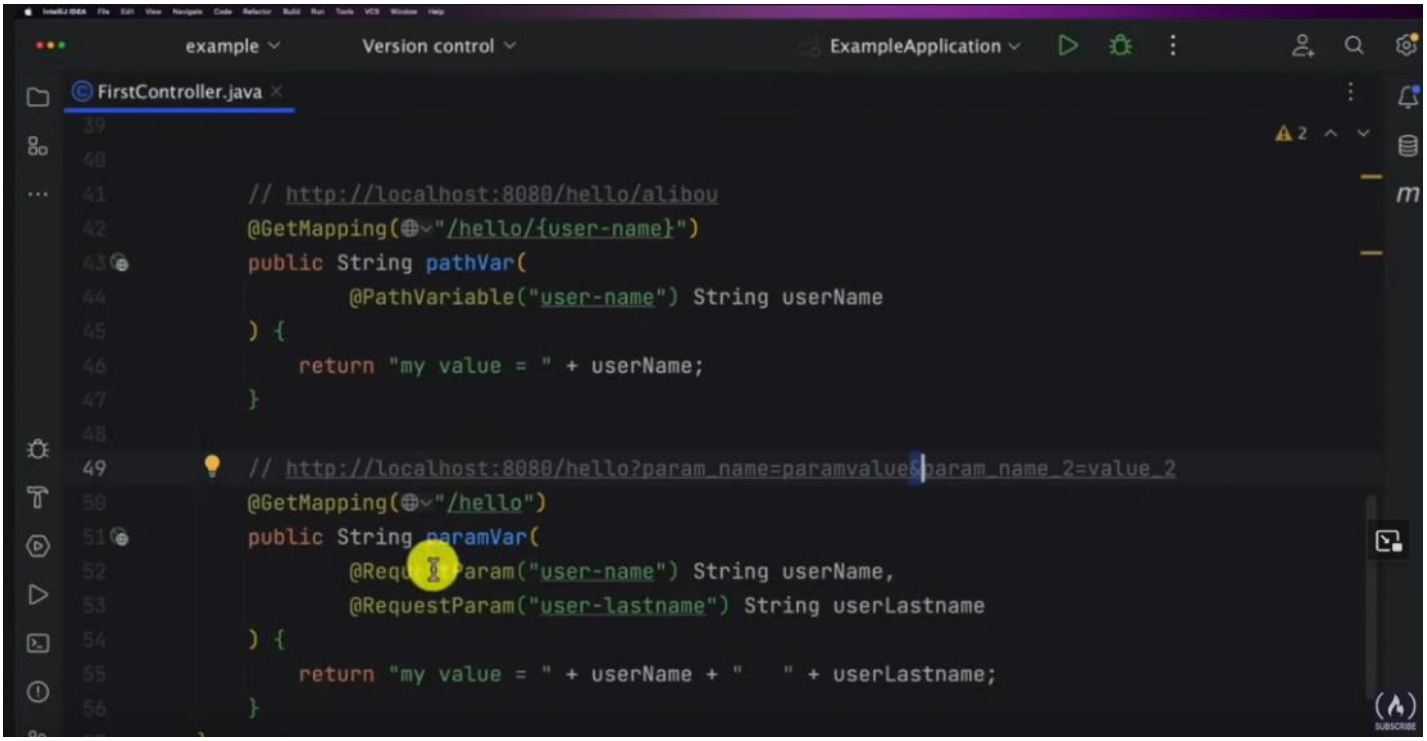
08 May 2024 06:16

Path Variables:

Used to identify resources as part of the URL

Request Params:

Used to extract query params



# @RequestController, @PostMapping, @GetMapping

08 May 2024 06:20

```java
package com.matt.example;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

@RestController
public class FirstController {

    @GetMapping("/hello")
    public String sayHello() {
        return "my first controller";
    }
    @PostMapping("/post")
    public String post(@RequestBody String message) {
        return "request accepted and" + message;
    }
    @PostMapping("/post-order")
    public String postOrder(@RequestBody Order order) {
        return "request accepted and" + order.toString();
    }
    @PostMapping("/post-record")
    public String postRecord(@RequestBody orderRecord record) {
        return "request accepted and" + record.toString();
    }
```

**Spring will scan for Rest Controllers components and create beans of them**

```java
    @GetMapping("/hello")
    public String sayHello() {
        return "my first controller";
    }
    @PostMapping("/post")
    public String post(@RequestBody String message) {
        return "request accepted and" + message;
    }
    @PostMapping("/post-order")
    public String postOrder(@RequestBody Order order) {
        return "request accepted and" + order.toString();
    }
    @PostMapping("/post-record")
    public String postRecord(@RequestBody orderRecord record) {
        return "request accepted and" + record.toString();
    }
```

**Various mappings are created by Spring because of Post, Get mapping annoations**

done by RequestMappingHandlerMapping which builds a registry of all your URL's

# Diagram explainer

08 May 2024 06:39

Dispatcher Servlet - dispatches requests and forwards them to Handler Mapping

Handler Mapping - Consults the Mapping Registry to find correct controller, returns found controller (all handles Methods, pathVariables, Params etc)

Controller - Handles request and contacts required business logic

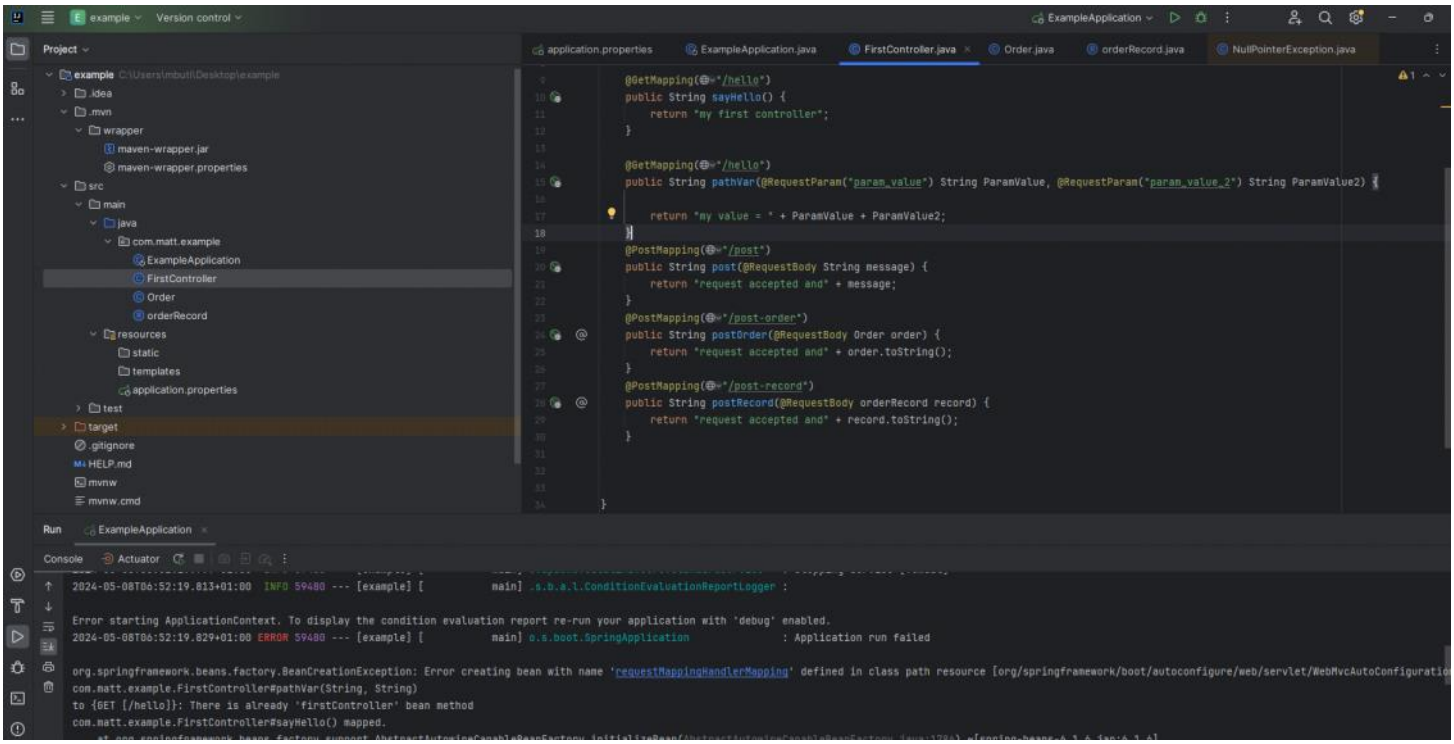
Business logic - Does "service" then returns response to controller

Controller sends responses to servlet dispatcher

Servlet dispatcher - sends response back to client

# URL unqiueness

08 May 2024 06:52



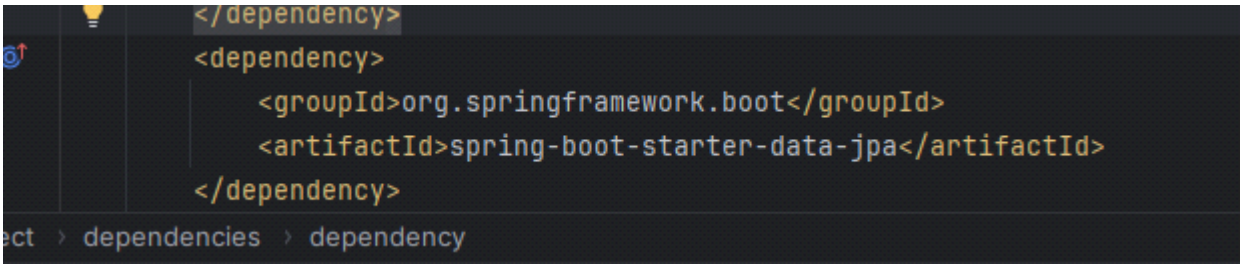

Only URL variables make a controller unique not parameters (check screenshot)

The path not the method name makes a controller unique

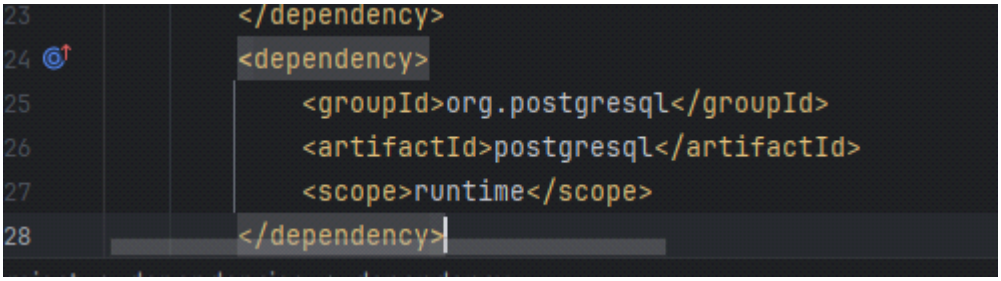
# Spring data JPA dependencies

08 May 2024 06:55

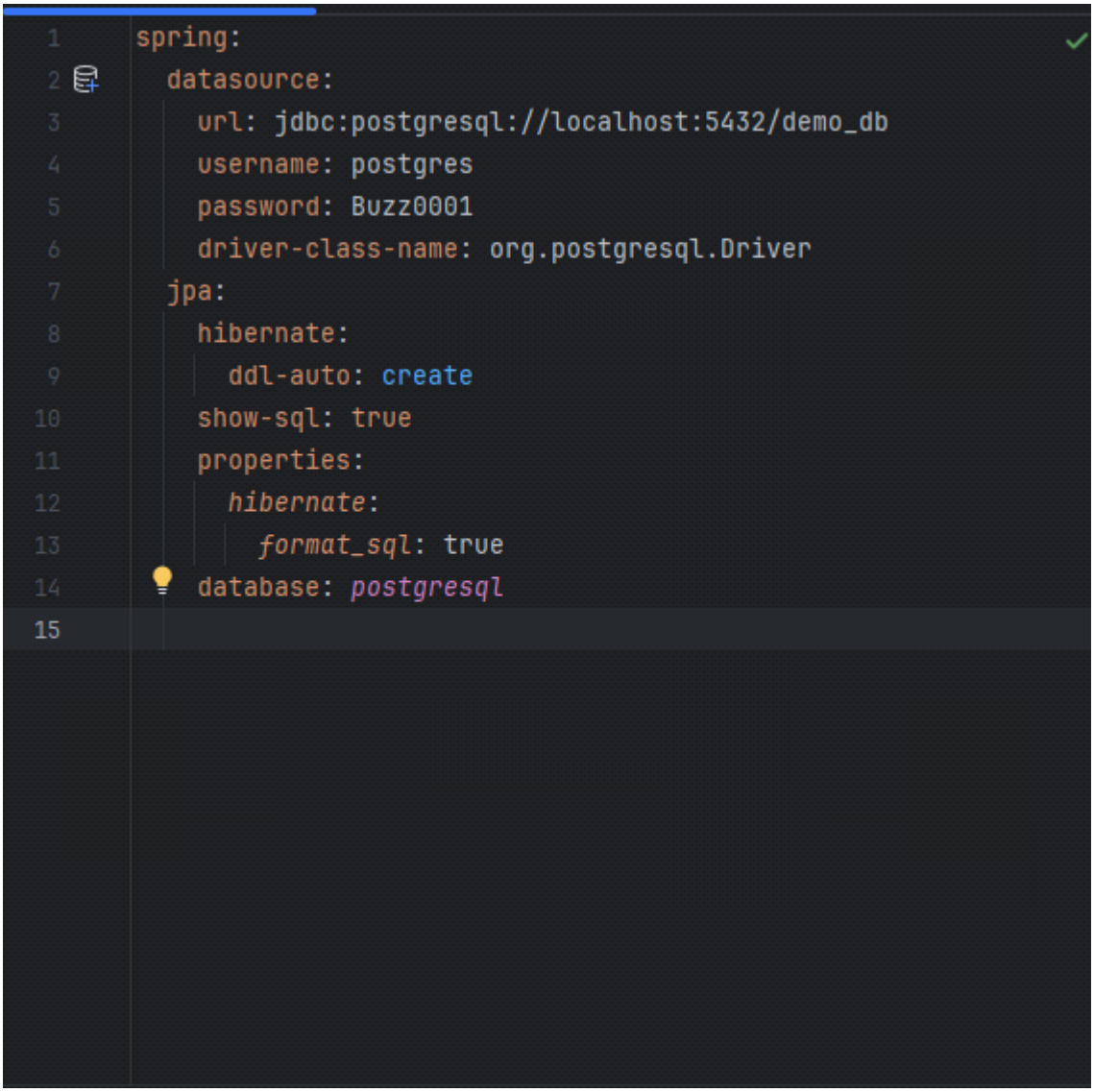
Spring Data JPA




PostgresSQL




Application.yaml (formally application.Properties):




Demo_db being our database

Ignore jpa config, the rest is good though and is self explanatory

# Data storage Diagram

08 May 2024 07:53

Entity - A java object that is designed to be saved in a database

**All entities must have an empty constructor**

Each entity is a table in a database

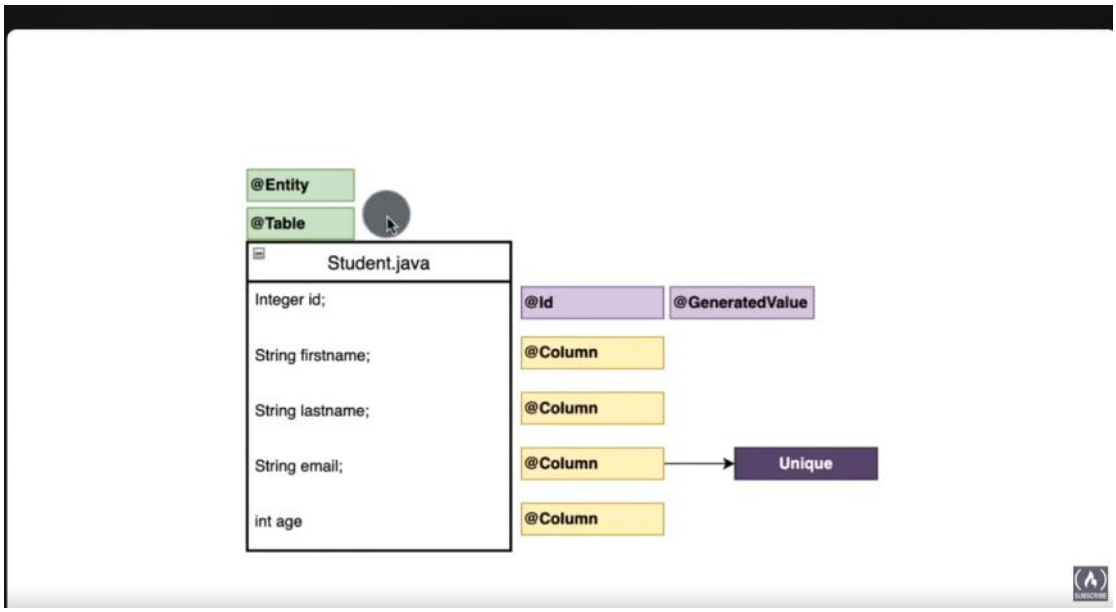
Columns - Each field in your model is a field

Row - the data

Every Entity must contain an ID that is a unique value

The table name is by default the class name but we set it with @table("NAME")

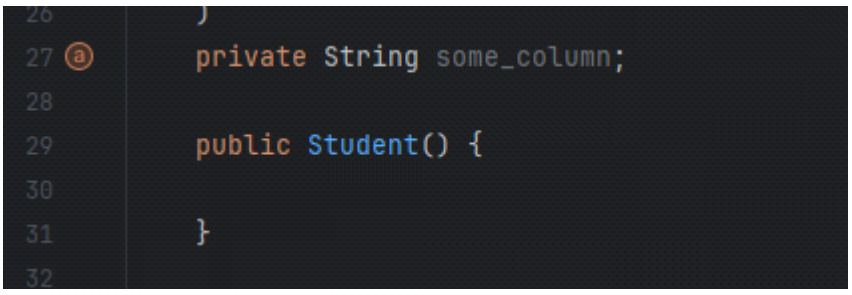
Spring Data then generates SQL queries on these entities



# Spring Data Entity

08 May 2024 08:04

```
@Entity
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue
    private Integer id;

    2 usages
    @Column(
            name = "c_fname",
            length = 20
    )
    private String firstname;
    2 usages
    private String lastname;

    2 usages
    @Column(unique = true)
    private String email;
    2 usages
    private int age;
```



```
    private String some_column;

    public Student() {

    }
```

Must have an empty constructor

@Entity - A java object that is designed to be saved in a database

@Table - sets a table  name, by default class name

@Column - lots of different features (use lookup) examples in screenshot. Remember each field is a different column

@Column(unique=true) - forces this field to be unique per user

@Column(updatable=false) - means field cannot be modified

@Column(insertable=false) - means field cannot be inserted

@Id ensure a unique primary key is made (required)

@Id
@GeneratedValue - will generate an id for our object that is completely unique

For all of these, use lookup for more info

# Spring Data Repository

08 May 2024 10:46

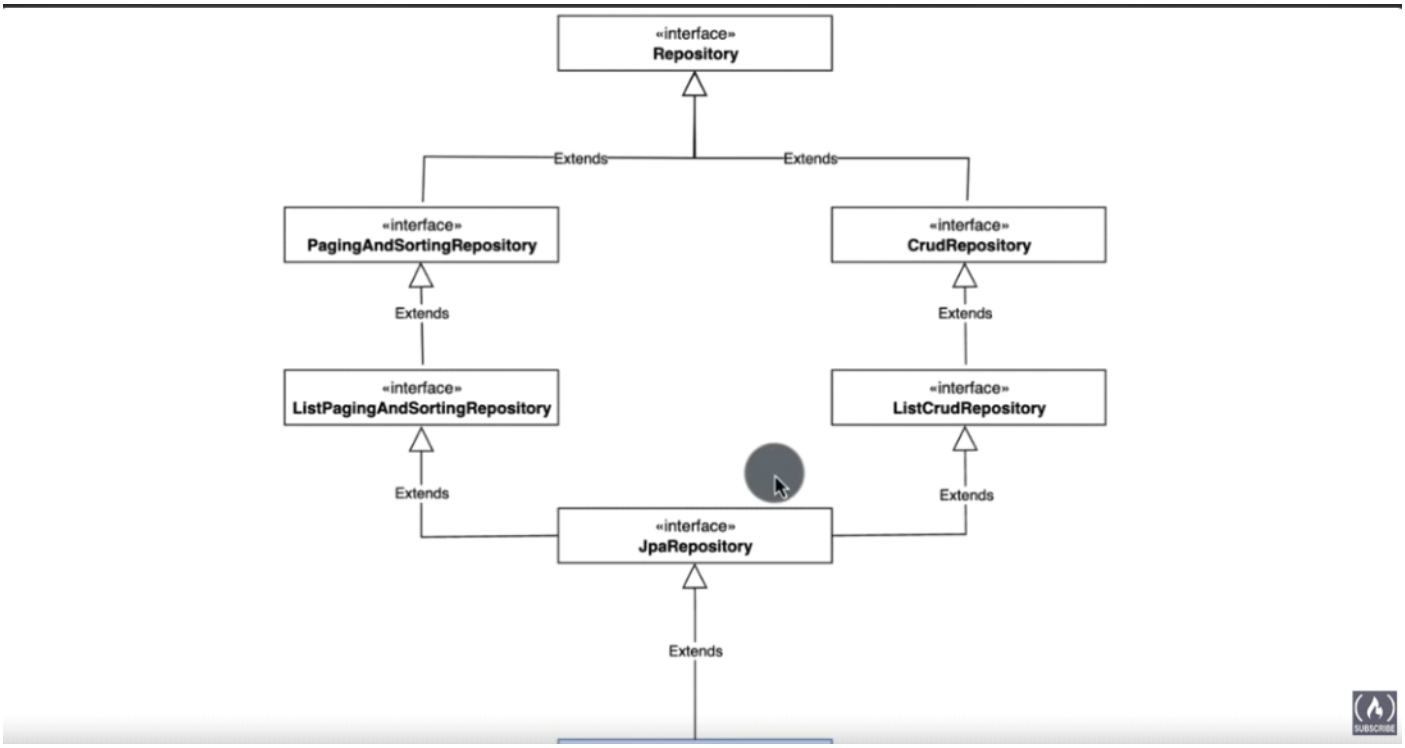
The Spring data JPA repository extends various different interfaces creating a lot of different functionality

These various interfaces, that you can choose, have functions like data save, data find, data delete. Allowing you to interact with your database without SQL

The spring JPA Repositories interact contains a lot of different method signatures with SQL functionality (findOneByID)

Spring JPA repositories also allow you to join together Method signatures for queries

Spring JPA Repositories also allow you to create custom queries using @Query annotation in JPQL or SQL to a custom DTO



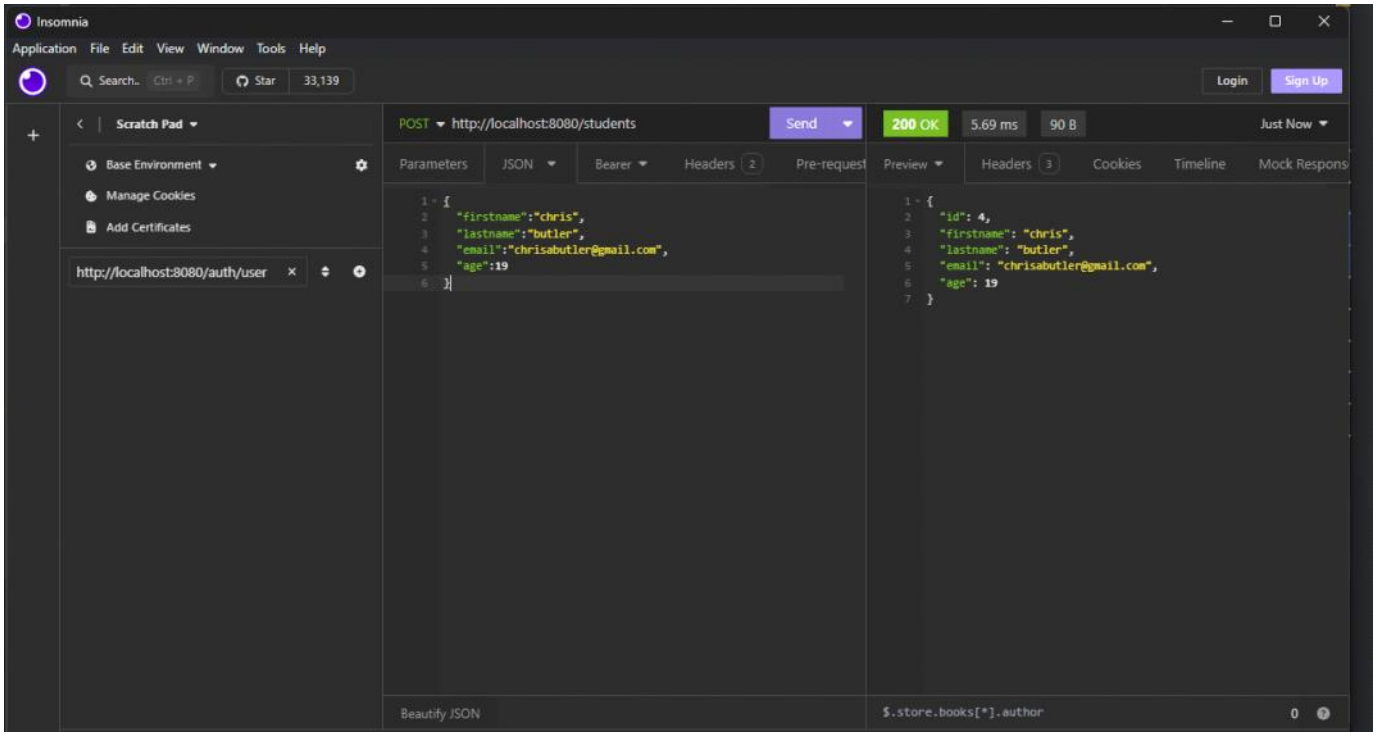
# Student Repository

08 May 2024 10:56

```java
2 usages
@Repository
public interface StudentRepository extends JpaRepository<Student, Integer> {

}
```

JPA Repositories takes in T (Object Type) and the type of the objects ID

```java
@RestController
public class FirstController {
    2 usages
    private final StudentRepository repository;

    @Autowired
    public FirstController(StudentRepository repository) {
        this.repository = repository;
    }

    @PostMapping("/students")
    public Student post(@RequestBody Student student) {
        return repository.save(student);
    }

}
```


Insomnia
Application File Edit View Window Tools Help
Search.. Ctrl + P
Star 33,139
Login
Sign Up
Scratch Pad
Base Environment
Manage Cookies
Add Certificates
http://localhost:8080/auth/user
POST http://localhost:8080/students
Send
Parameters
JSON
Bearer
Headers 2
Pre-request
{
"firstname":"chris",
"lastname":"butler",
"email":"chrisabutler@gmail.com",
"age":19
}
Beautify JSON
200 OK
5.69 ms
90 B
Just Now
Preview
Headers 3
Cookies
Timeline
Mock Respons
{
"id": 4,
"firstname": "chris",
"lastname": "butler",
"email": "chrisabutler@gmail.com",
"age": 19
}
$.store.books[*].author
0

# REST crud functions + OrElse

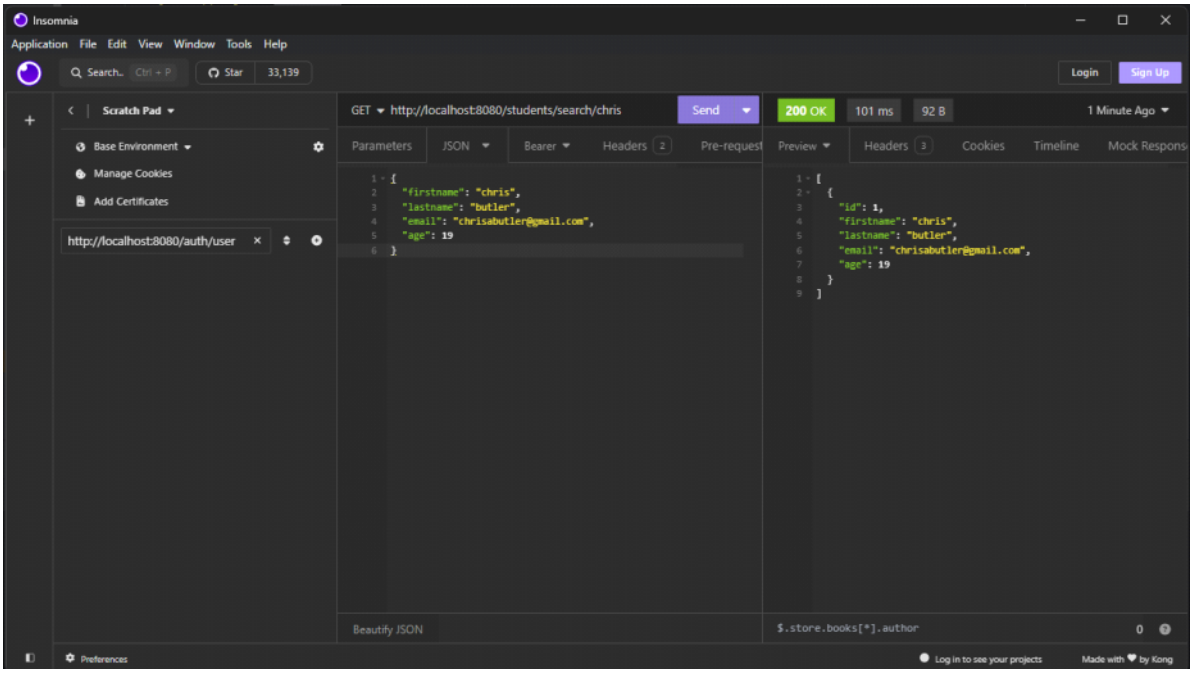
08 May 2024 11:28

```java
package com.matt.example;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

2 usages
@Repository
public interface StudentRepository extends JpaRepository<Student, Integer> {

    1 usage
    List<Student> findAllByFirstname(String name);

}
```

Build queries with a method signatures like above

```java
@RestController
public class FirstController {
    6 usages
    private final StudentRepository repository;

    @Autowired
    public FirstController(StudentRepository repository) {
        this.repository = repository;
    }

    @PostMapping("/students")
    public Student post(@RequestBody Student student) {
        return repository.save(student);
    }

    @GetMapping("/students")
    public List<Student> findAllStudent() {
        return repository.findAll();
    }
    @GetMapping("/students/{student-id}")
    public Student findStudentById(@PathVariable("student-id") int studentId) {
        return repository.findById(studentId).orElse(new Student());
    }
    @GetMapping("/students/search/{student-name}")
    public List<Student> findStudentByName(@PathVariable("student-name") String studentName) {
        //Student[] students = {};
        return repository.findAllByFirstname(studentName);
    }
```

```
        return repository.findAllByFirstname(studentName);
    }

    @DeleteMapping("/students/{student-id}")
    @ResponseStatus(HttpStatus.OK) //Sets response code
    public void delete(@PathVariable("student-id") Integer studentId) {
        repository.deleteById(studentId);
```




```
return repository.findById(studentId).orElse( other: null);
```

**.OrElse function - can be used on any optional object**

Repository.save() returns saved values

```
    @DeleteMapping("/students/{student-id}")
    @ResponseStatus(HttpStatus.OK) //Sets response code
    public void delete(@PathVariable("student-id") String studentId) {
        repository.deleteStudentById(studentId);
    }


}
```

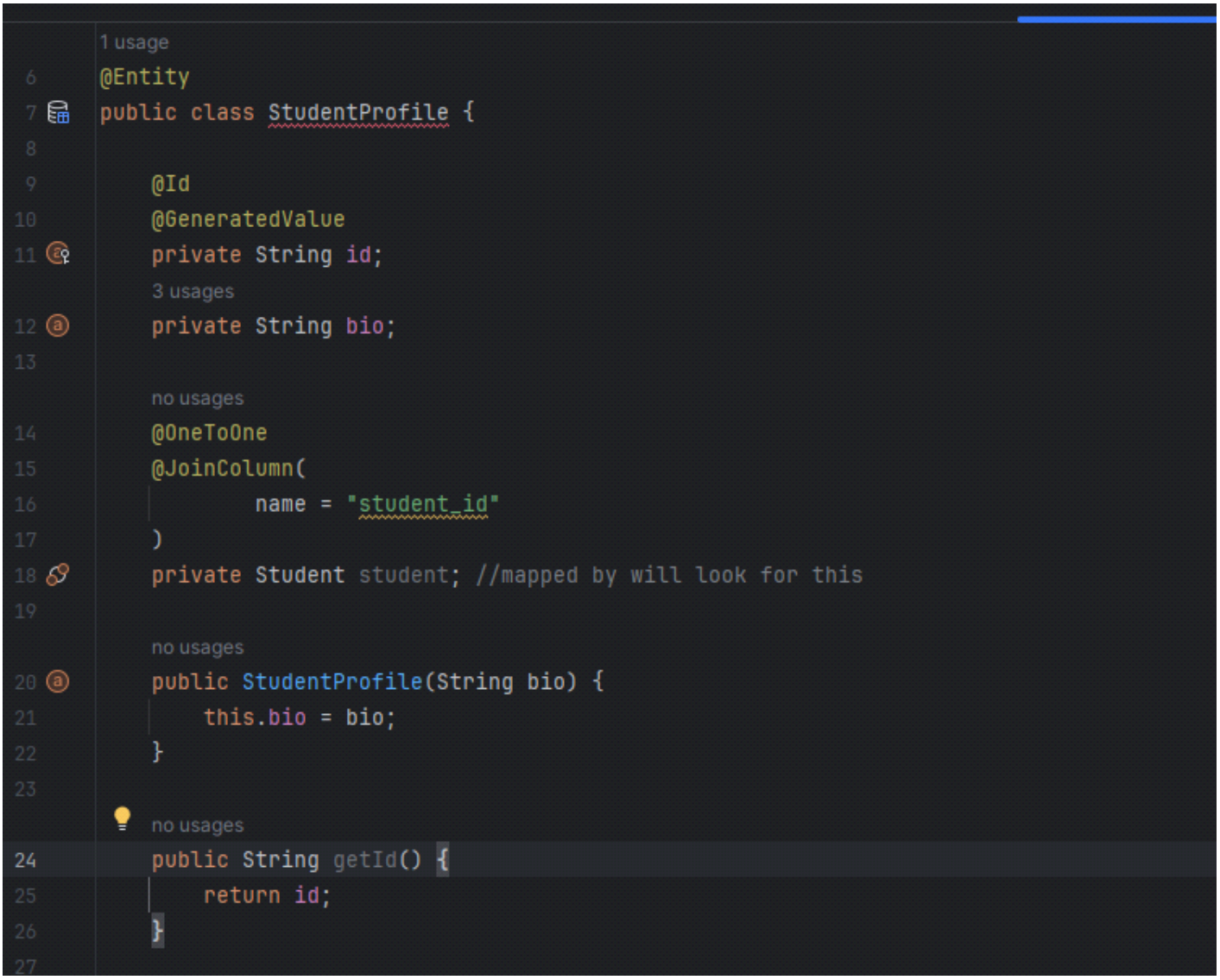
**ResponseStatus set given no exceptions**

# Spring Relationships

08 May 2024 12:18

Two Relationship Example:

## One to one Student to StudentProfile



```
1 usage
@Entity
public class StudentProfile {

    @Id
    @GeneratedValue
    private String id;
    3 usages
    private String bio;

    no usages
    @OneToOne
    @JoinColumn(
            name = "student_id"
    )
    private Student student; //mapped by will look for this

    no usages
    public StudentProfile(String bio) {
        this.bio = bio;
    }

    no usages
    public String getId() {
        return id;
    }
```

```java
9 usages
@Entity
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue
    private Integer id;

    2 usages
    @Column(
            name = "c_fname",
            length = 20
    )
    private String firstname;
    2 usages
    private String lastname;

    2 usages
    @Column(unique = true)
    private String email;
    2 usages
    private int age;

    no usages
    @OneToOne(mappedBy = "student",
    cascade = CascadeType.ALL) //Cascade ='s what happens on delete
    private StudentProfile studentProfile; //One to one relationship

    public Student() {

```

@OneToOne - create a one to one relationship:

mappedBy = "student" - looks for the correct corresponding field in another object (in this case school) - MUST BE SPELT SAME AS FIELD NAME

Cascade = CascadeType.ALL

## One to Many Student to School

```java
3 usages
@Entity
public class School {

    @Id
    @GeneratedValue
    private String id;

    2 usages
    private String name;

    2 usages
    @OneToMany(
            mappedBy = "school"
    )
    private List<Student> students;
```

```java
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue
    private Integer id;

    2 usages
    @Column(
            name = "c_fname",
            length = 20
    )
    private String firstname;
    2 usages
    private String lastname;
    2 usages
    @ManyToOne
    @JoinColumn(
            name = "school_id"
    )
    private School school;
```

@JoinColumn - creates a column to store the ID of the relationed object

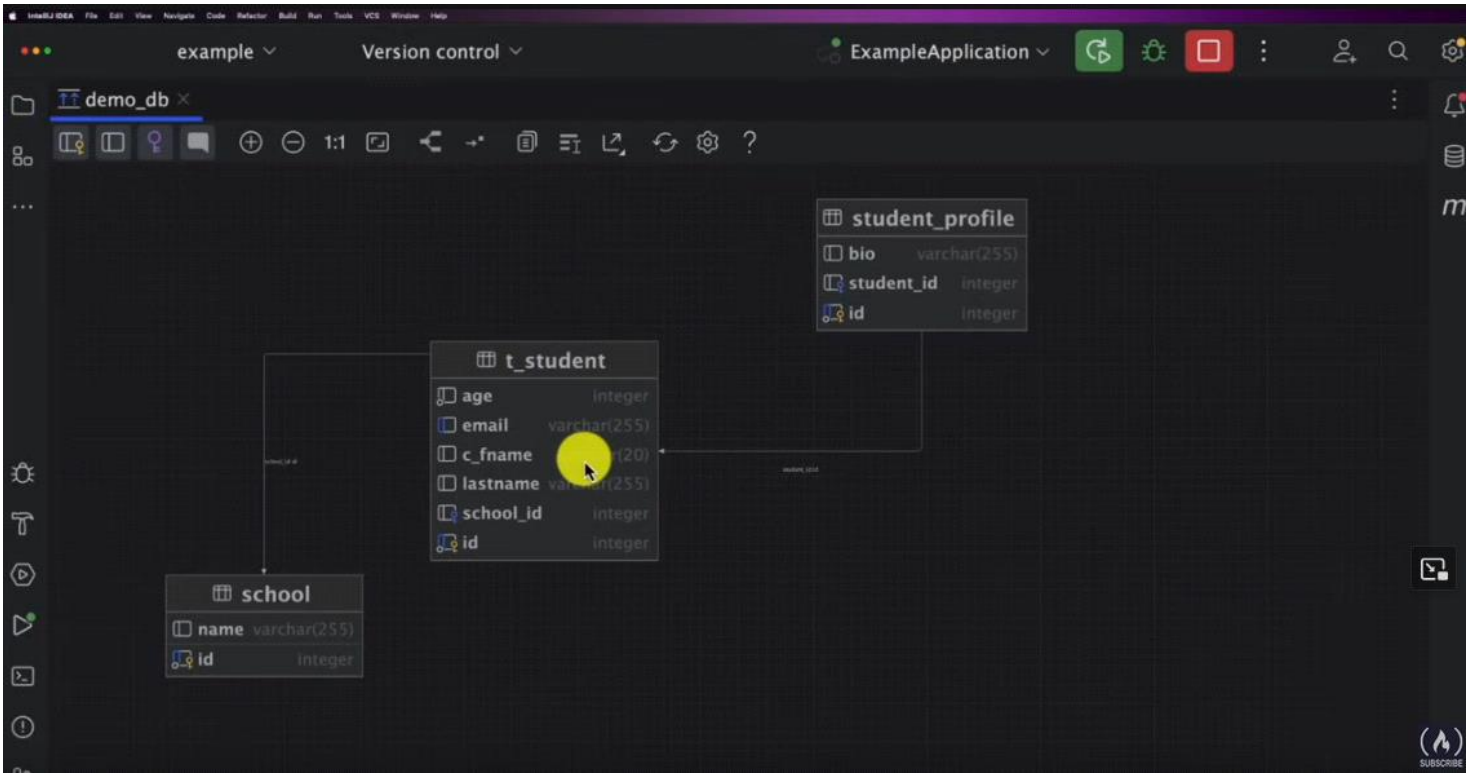
Join Column should be given a name and added to the secondary object (child)

So Student has it for Schools

And StudentProfile has it for Student

# Checking relationships

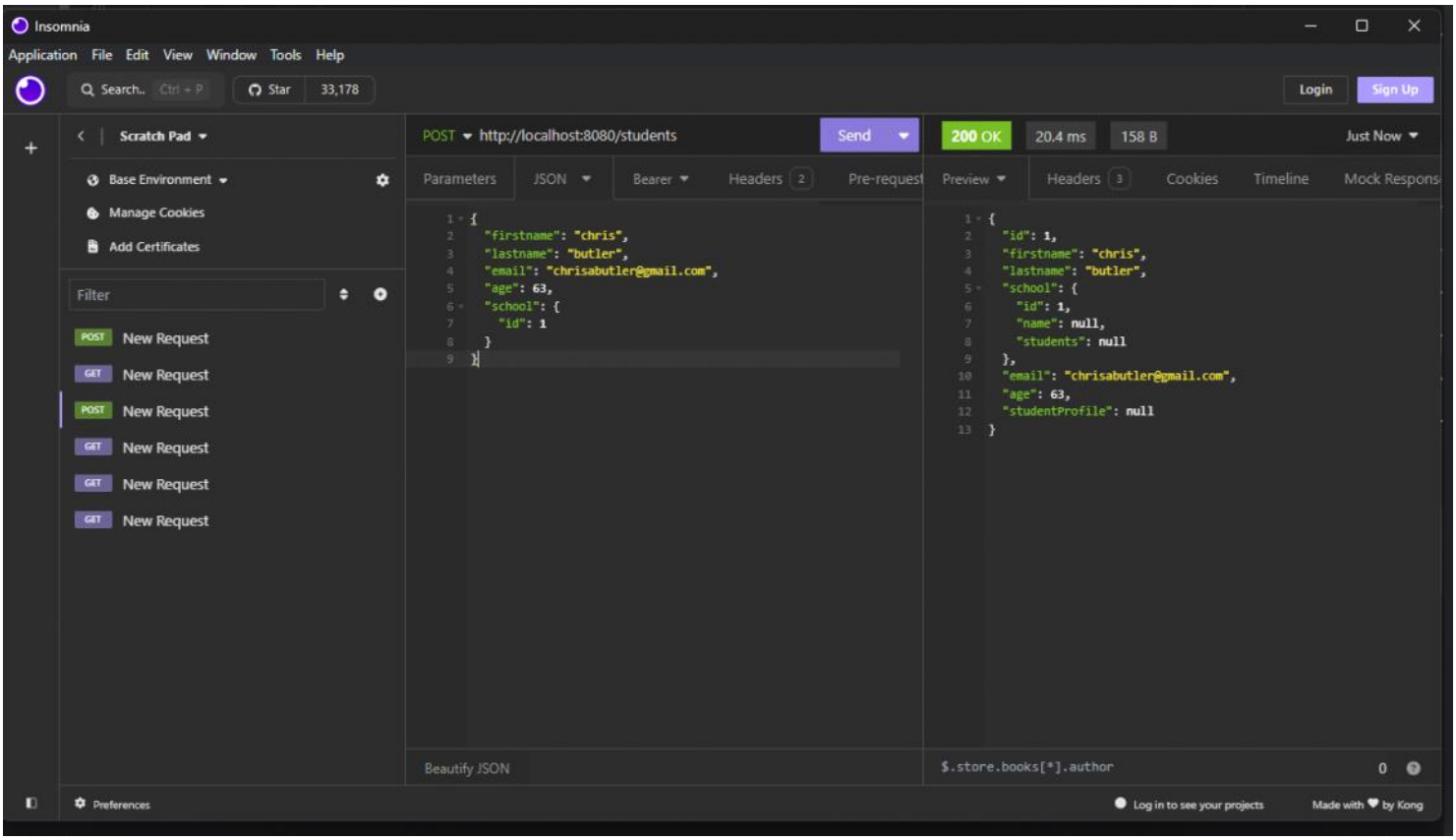
08 May 2024 13:39




Inteji Show Diagrams

# Relationships working Screenshots

08 May 2024 14:00




```
3 usages
@Entity
public class StudentProfile {

    @Id
    @GeneratedValue
    private String id;
    3 usages
    private String bio;

    no usages
    @OneToOne
    @JoinColumn(
            name = "student_id"
    )
    private Student student; //mapped by will look for this

```

```
8 usages
@Entity
public class School {

    @Id
    @GeneratedValue
    private Integer id;

    2 usages
    private String name;

    2 usages
    @OneToMany(
            mappedBy = "school"
    )
    private List<Student> students;


    public School() {

```

```java
12 usages
@Entity
@Table(name = "t_student")
public class Student {

    @Id
    @GeneratedValue
    private Integer id;


    2 usages
    private String firstname;
    2 usages
    private String lastname;
    2 usages
    @ManyToOne
    @JoinColumn(
            name = "school_id"
    )
    private School school;
```

```java
    2 usages
    @OneToOne(mappedBy = "student",
    cascade = CascadeType.ALL) //Cascade ='s what happens on delete
    private StudentProfile studentProfile; //One to one relationship

    public Student() {
```

**StudentProfile bond Student**

**Student bond School**

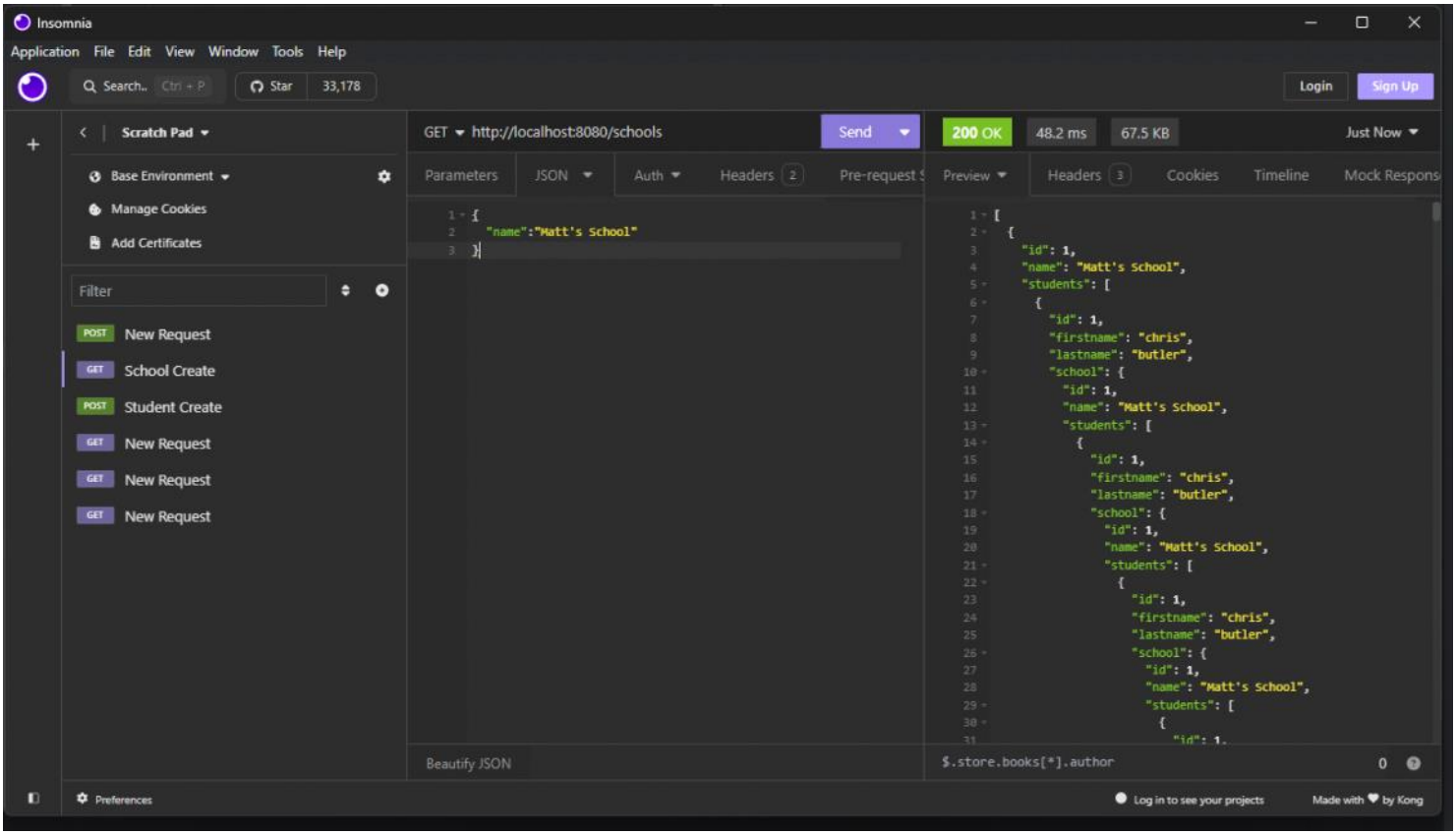
**School (One to many)**

**Student to school (Many to one)**

**StudentProfile to student (One to One)**

# Relationship looping @JsonManagedReference

09 May 2024 05:38




**When request objects that are in a relationship with each other you can get stuck in a loop**

@JsonManagedreference (applied to parent field)

```
7 usages                                                    5
@Entity
public class School {

    @Id
    @GeneratedValue
    private Integer id;

    2 usages
    private String name;

    2 usages
    @OneToMany(
            mappedBy = "school"
    )
    @JsonManagedReference //Basically tells School to manage s
    private List<Student> students;


    public School() {

    }



    no usages
```

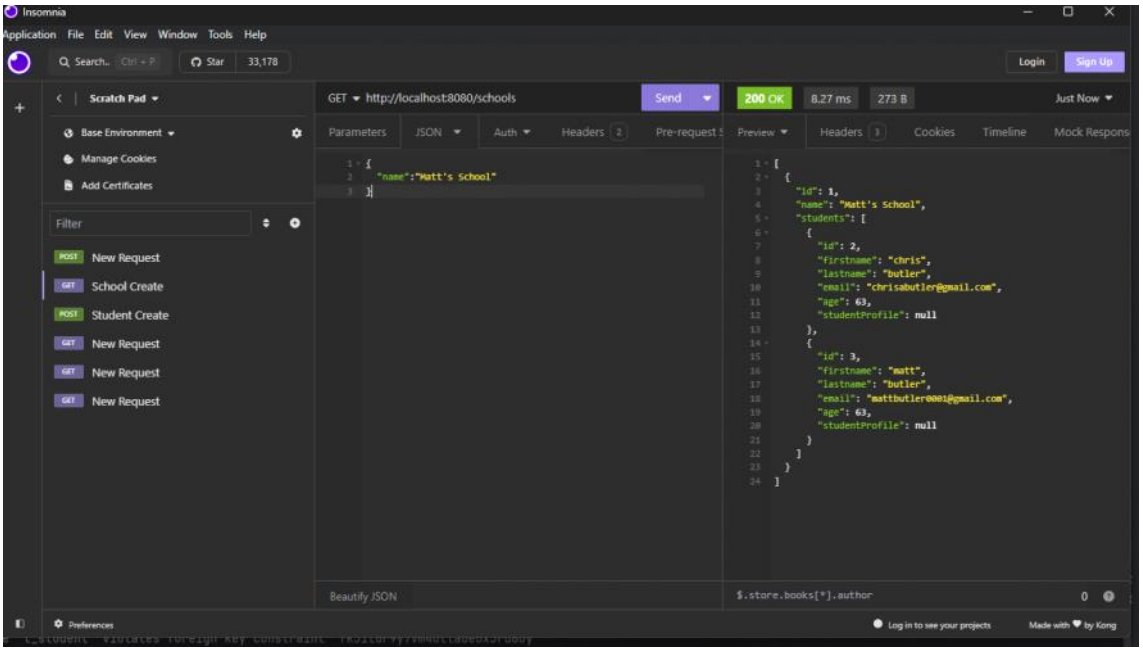
Basically tells School to manage serializing the child and not to allow child to serialize parent

```
                                                        16  1
import com.fasterxml.jackson.annotation.JsonBackReference;
import jakarta.persistence.*;

12 usages
@Entity
@Table(name = "t_student")
public class Student {
```

```java
import com.fasterxml.jackson.annotation.JsonBackReference;
import jakarta.persistence.*;

12 usages
@Entity
@Table(name = "t_student")
public class Student {

    @Id
    @GeneratedValue
    private Integer id;


    2 usages
    private String firstname;
    2 usages
    private String lastname;
    2 usages
    @ManyToOne
    @JoinColumn(
            name = "school_id"
    )
    @JsonBackReference
    private School school;

    2 usages
    @Column(unique = true)
    private String email;
    2 usages
```

@JsonBackReference (applied to child field)

Basically tells Student to not attempt to serialize School



# DTO Pattern, Representations

09 May 2024 05:53

The goal of a DTO is to encapsulate and transfer data between systems

DTO only includes simple data fields, doesn't have behaviours of model it represents

We use a mapper to prevent sensitive information from being dispatched when it shouldn't be.

Mappers have multiple representations, only expose X fields

We can also use different representations for read and write

This ensures:

Data separation - only gives required data

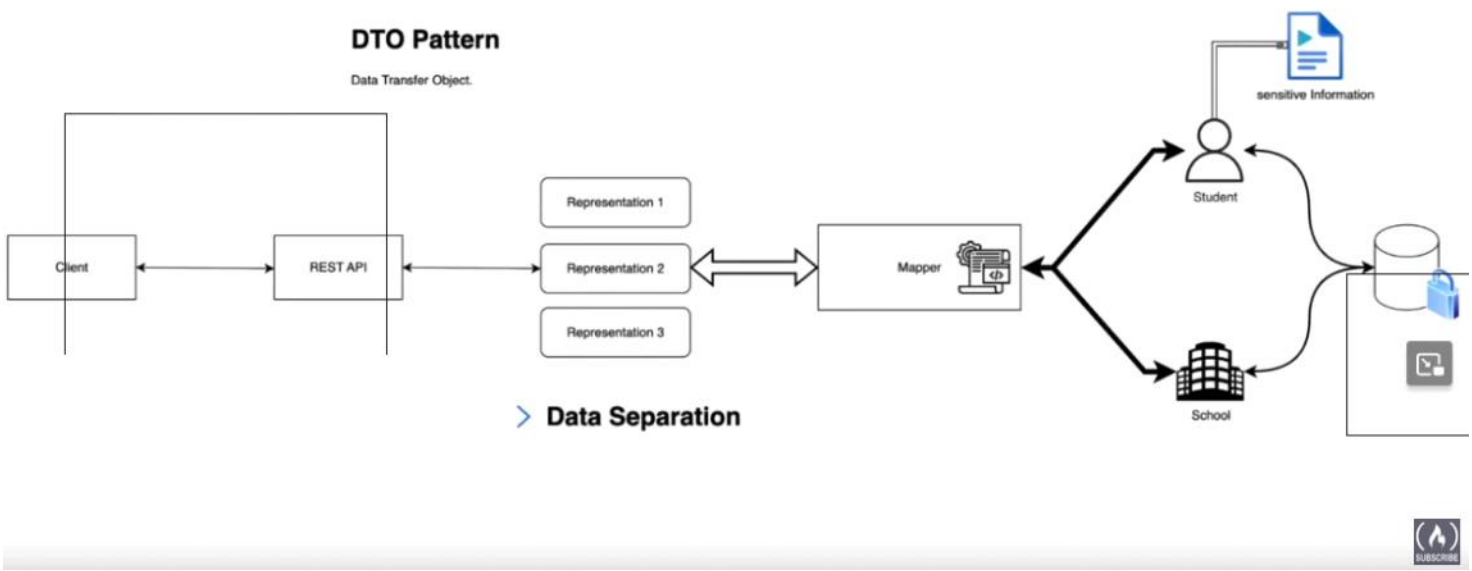
Abstractions layer - gives you a clear representation of what will be sent

Performance - instead of sending lots of data, we only send what is required over network

Flexibility - allows you to only give the client what your client needs

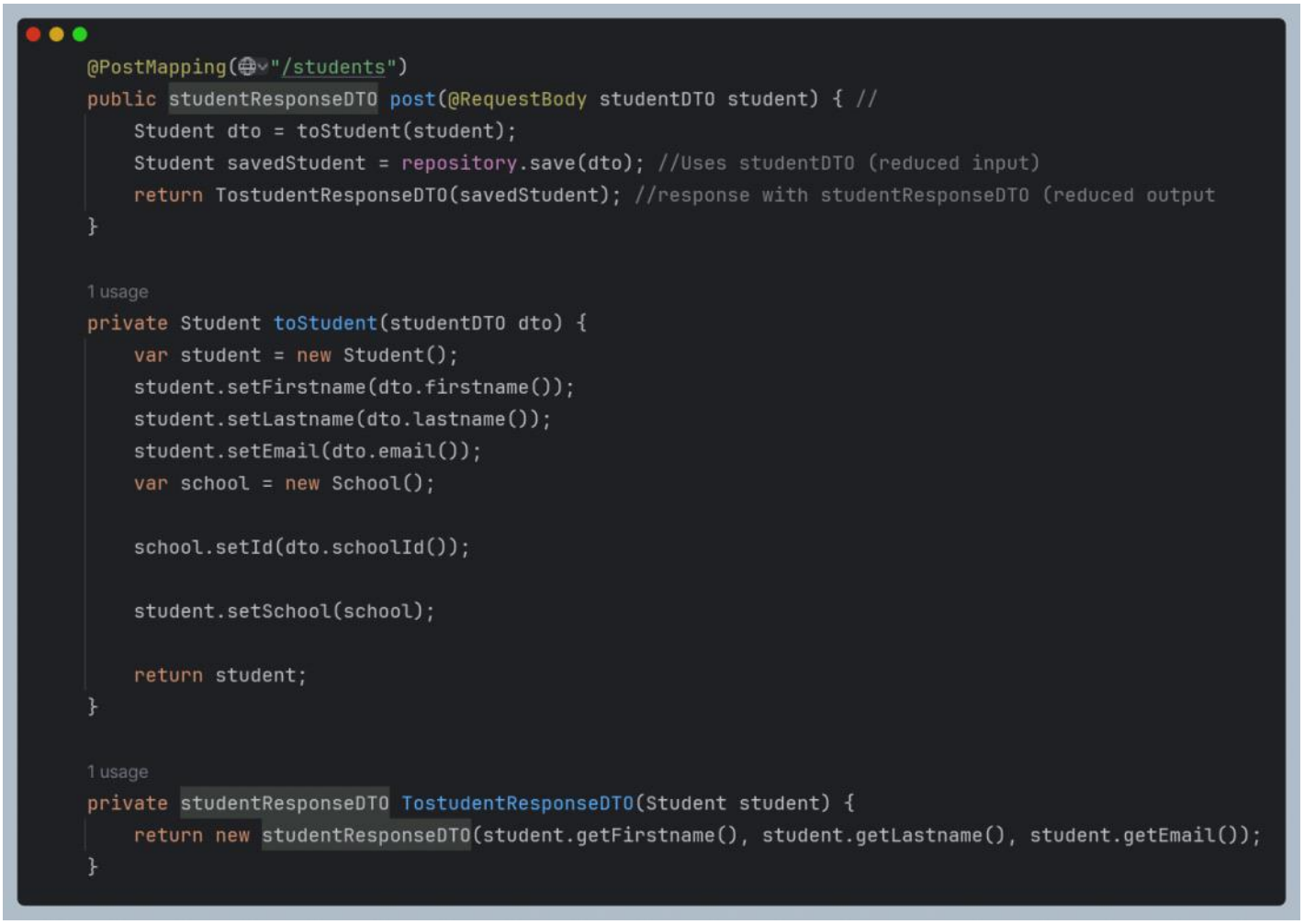
Versioning - allows you to support multiple versions of the same API

---



# DTO in action

09 May 2024 06:13



```java
@PostMapping("/students")
public studentResponseDTO post(@RequestBody studentDTO student) { //
    Student dto = toStudent(student);
    Student savedStudent = repository.save(dto); //Uses studentDTO (reduced input)
    return TostudentResponseDTO(savedStudent); //response with studentResponseDTO (reduced output
}

1 usage
private Student toStudent(studentDTO dto) {
    var student = new Student();
    student.setFirstname(dto.firstname());
    student.setLastname(dto.lastname());
    student.setEmail(dto.email());
    var school = new School();

    school.setId(dto.schoolId());

    student.setSchool(school);

    return student;
}

1 usage
private studentResponseDTO TostudentResponseDTO(Student student) {
    return new studentResponseDTO(student.getFirstname(), student.getLastname(), student.getEmail());
}
```

Mapper functions basically encode and decode to and from DTO objects to the 'real data' model

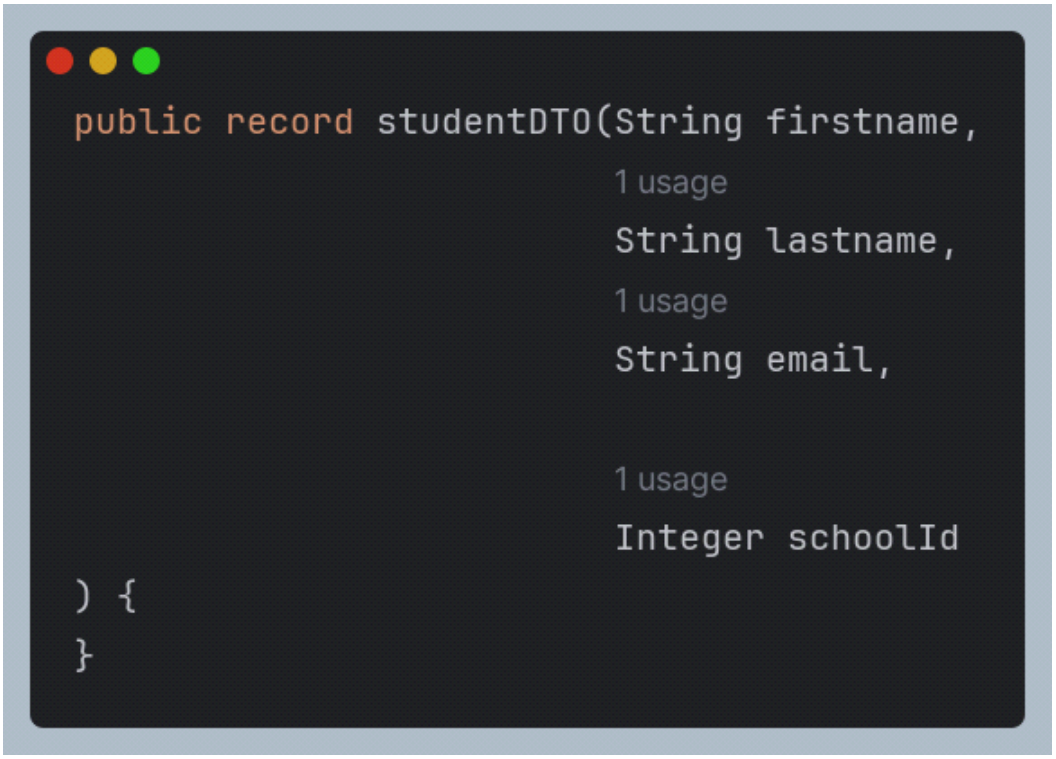
ToStudentResponseDTO is a mapping function

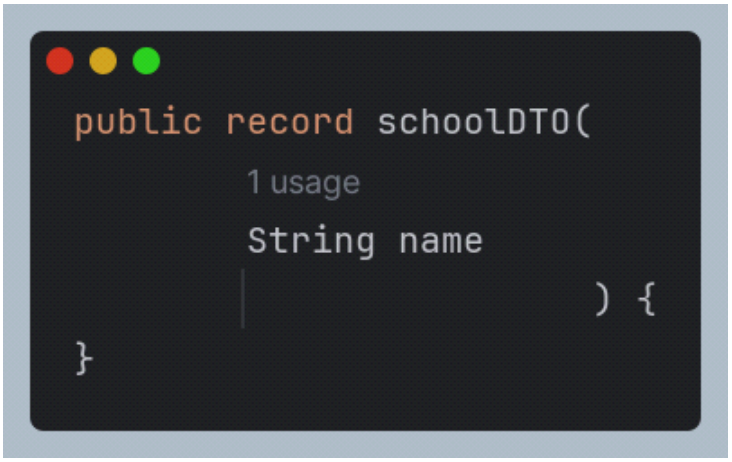
ToStudent is a mapping function

```java
public class SchoolController {

    3 usages
    private final SchoolRepository schoolRepository;

    @Autowired
    public SchoolController(SchoolRepository schoolRepository) { this.schoolRepository = schoolRepository; }

    @PostMapping("/schools")
    public schoolDTO create (@RequestBody schoolDTO school) {
        School schoolfromDTO = toSchool(school);
        School savedSChool = schoolRepository.save(schoolfromDTO);
        return school;

    }

    1 usage
    public School toSchool(schoolDTO dto) {
        School school = new School();
        school.setName(dto.name());
        return school;
    }
```

ToSchool is a mapping function

```java
public record studentResponseDTO(
        no usages
        String firstname,
        no usages
        String lastname,
        no usages
        String email
) {
}
```




Above is various representations

# Java Streams

09 May 2024 07:18

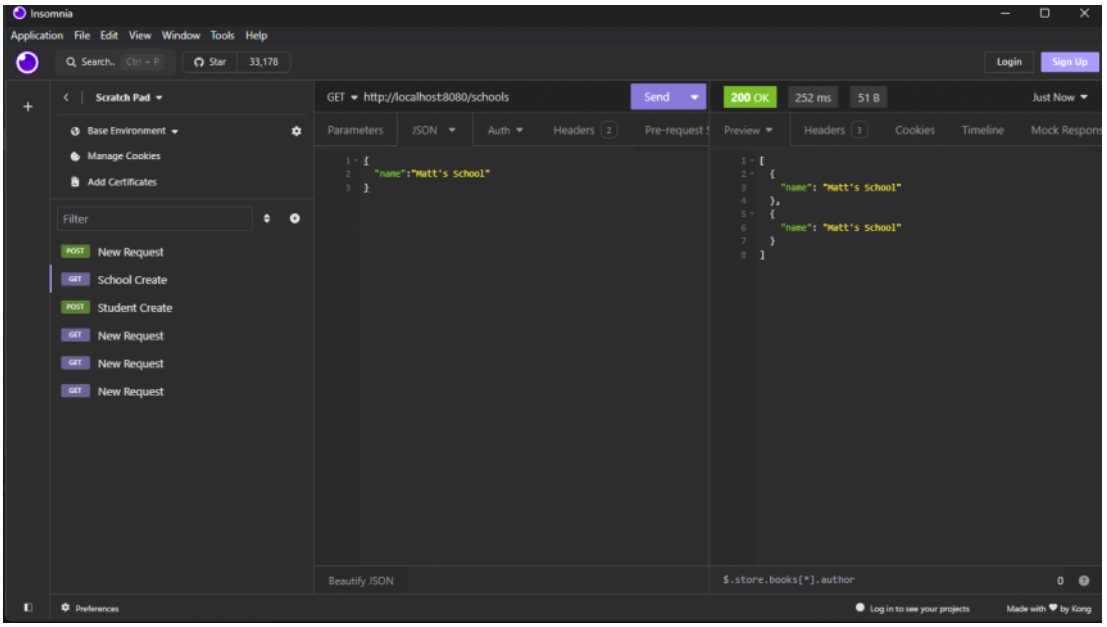
Java Streams:

```java
private schoolDTO toSchoolDTO(School school) {
    return new schoolDTO(school.getName());
}

@GetMapping("/schools")
public List<schoolDTO> findAll () {
    return schoolRepository.findAll().stream()
            .map(this::toSchoolDTO).collect(Collectors.toList());


}
```

Steam - in java converts a list of items to be processed one at a time

.Map(this::toSchoolDTO) - applies schoolDTO to each of the items in the map

.Collect puts the results of this back into a list



# Service Layer

09 May 2024 07:31

Does all complex logic and business functions

Putting our code in a service layer makes our code more reusable and maintainable

Controllers - handles incoming requests and responses, nothing more

Repositories - Handle data persistence and retrieval

Services - handle all in-between logic and promotes separation

Service layer allows us to unit test our business logic independently

# Services in action

09 May 2024 07:36

Check codebase for overview

Inject service with controller constructor

```java
@RestController
public class StudentController {
    6 usages
    private final StudentService studentService;

    @Autowired
    public StudentController(StudentService studentService) {

        this.studentService = studentService;
    }

    @PostMapping("/students")
    public studentResponseDTO saveStudent(@RequestBody studentDTO student) { //
        return this.studentService.saveStudent(student);
    }


    @GetMapping("/students")
    public List<Student> findAllStudent() {
        return this.studentService.findAllStudent();
    }
    @GetMapping("/students/{student-id}")
    public Student findStudentById(@PathVariable("student-id") int studentId) {
        return this.studentService.findStudentById(studentId);
    }
    @GetMapping("/students/search/{student-name}")
    public List<Student> findStudentByName(@PathVariable("student-name") String studentName) {
        //Student[] students = {};
        return this.studentService.findStudentByName(studentName);
    }

    @DeleteMapping("/students/{student-id}")
    @ResponseStatus(HttpStatus.OK) //Sets response code
    public void delete(@PathVariable("student-id") Integer studentId) {
        this.studentService.delete(studentId);
        return;
    }


}
```

As you can see the Student Service has been injected

```java
@Service
public class StudentService {

    6 usages
    private final StudentRepository repository;

    6 usages
    private final StudentMapper studentMapper;

    @Autowired
    public StudentService(StudentRepository repository, StudentMapper studentMapper) {
        this.repository = repository;
        this.studentMapper = studentMapper;
    }
    1 usage
    public studentResponseDTO saveStudent(studentDTO student) {
        Student dto = studentMapper.toStudent(student);
        Student savedStudent = repository.save(dto); //Uses studentDTO (reduced input)
        return studentMapper.TostudentResponseDTO(savedStudent); //response with studentResponseDTO (reduced output
    }

    1 usage
    public List<studentResponseDTO> findAllStudent() {
        return this.repository.findAll().stream().map(studentMapper::TostudentResponseDTO).collect(Collectors.toList());
    }
    1 usage
    public studentResponseDTO findStudentById(int id) {
        return repository.findById(id).map(studentMapper::TostudentResponseDTO).orElse( other: null);

        //Any Optional Type in java can be transformed into a map and sent to a method for transforming.
    }
    1 usage
    public List<studentResponseDTO> findStudentByName(String name) {
       return  this.repository.findAll().stream().map(studentMapper::TostudentResponseDTO).collect(Collectors.toList());
    }
    1 usage
    public void delete(int studentId) {
        repository.deleteById(studentId);
        return;
    }
}
```

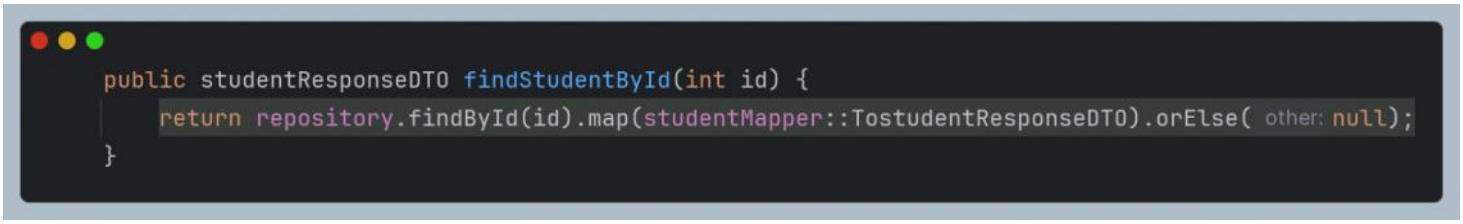
The service injects the repository beans

# Java Optional

09 May 2024 08:13

Any Optional Type in java can be transformed into a mapped to a method and transformed, :: being a method reference.

Spring repositories return Optional Objects



# Organisation of codebase

09 May 2024 08:38

Look at codebase

**By feature (preferred personally) :**

Split your codebase by features

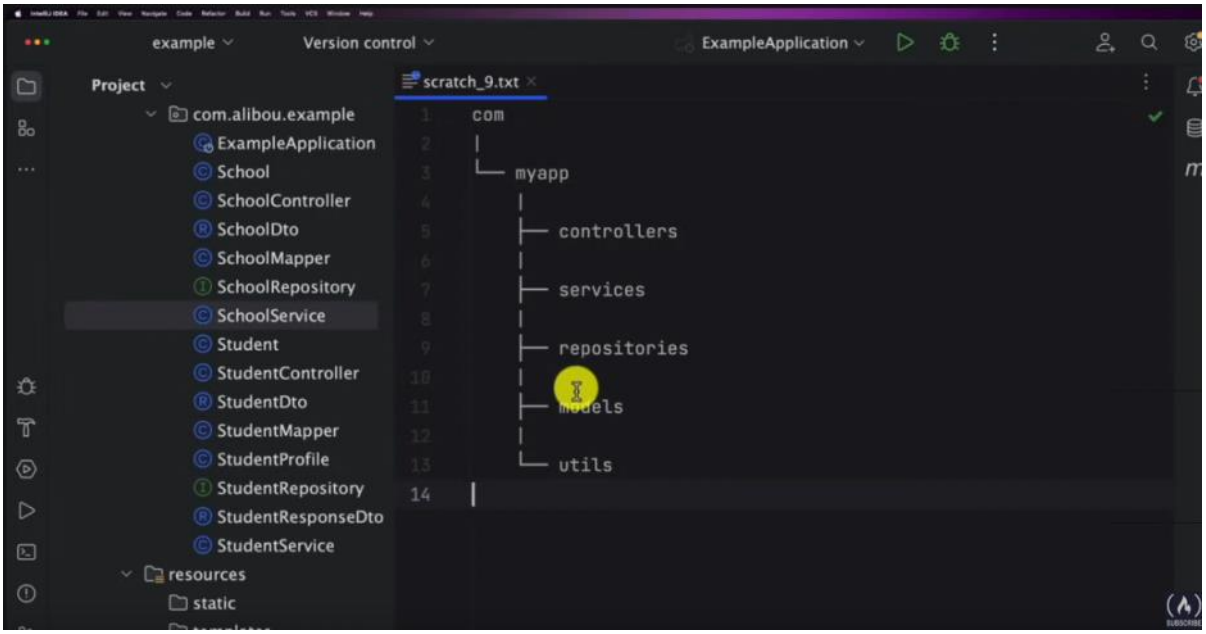
Each with controller, repositories, services

Good for big teams


**Layer approach:**

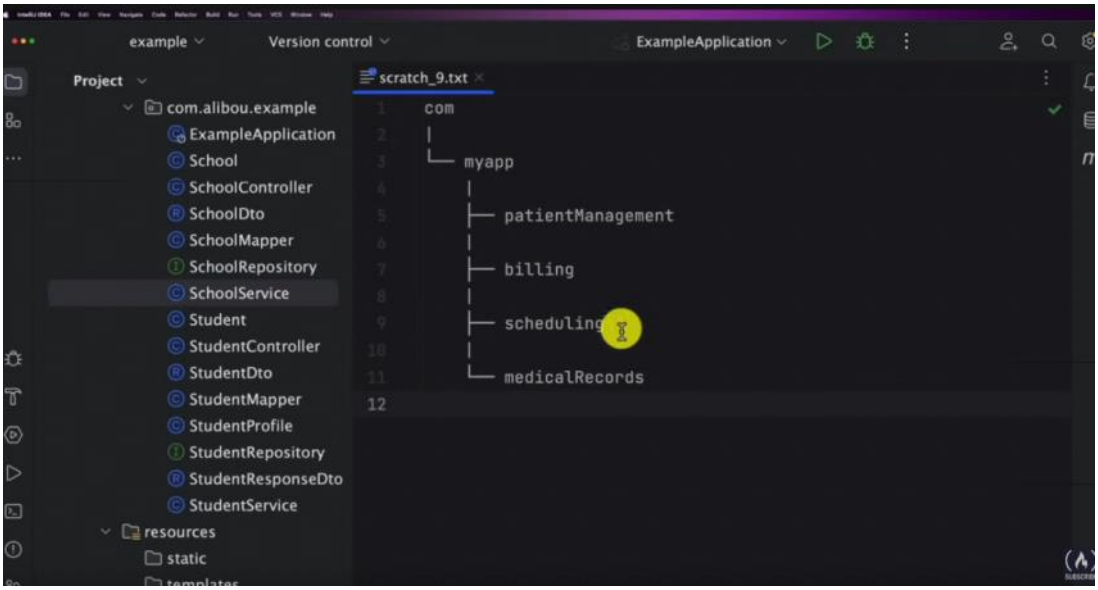
All tech layers are stored in corresponding class

However a change to one feature might break more features within the package




**By domain:**

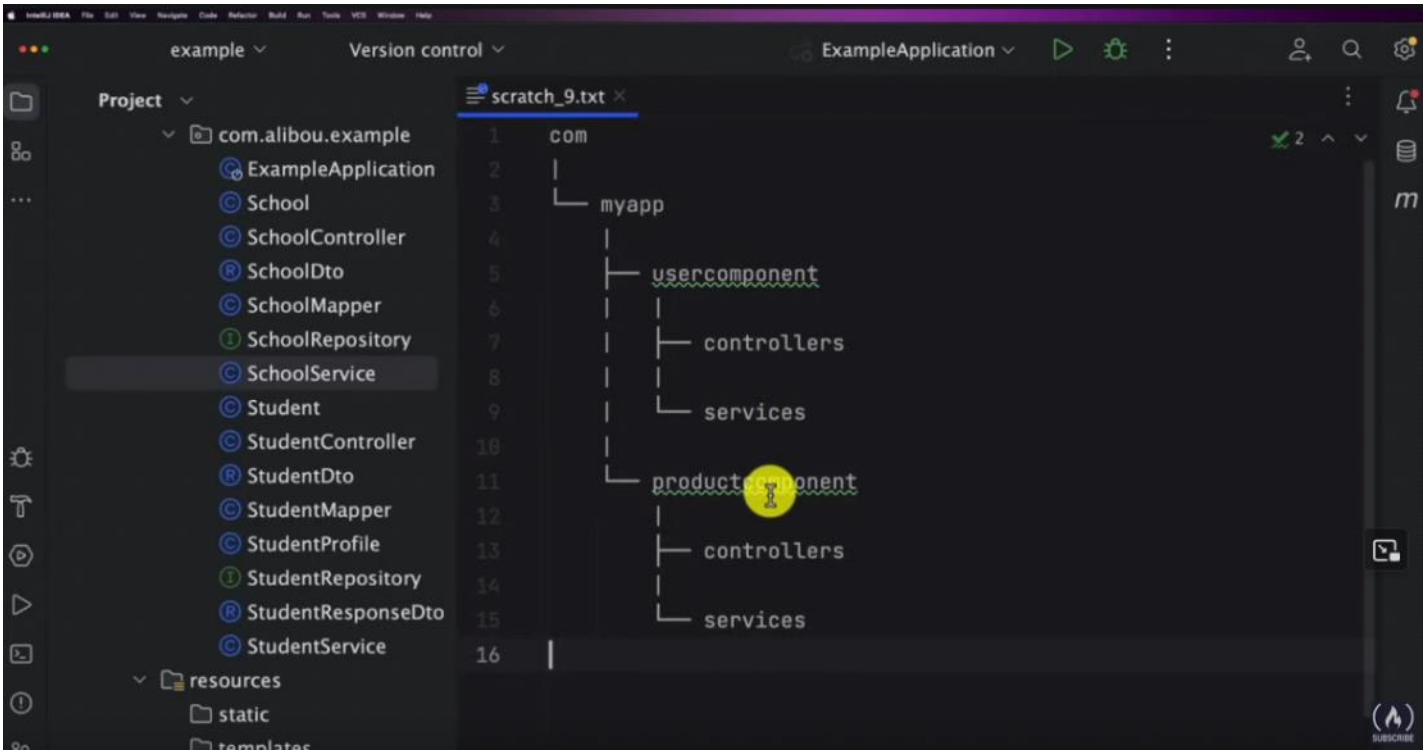
Basically by department of the software




By Component:

Break it down into components of your choosing

then further into controllers, services etc


example
Version control
ExampleApplication
Project
com.alibou.example
ExampleApplication
School
SchoolController
SchoolDto
SchoolMapper
SchoolRepository
SchoolService
Student
StudentController
StudentDto
StudentMapper
StudentProfile
StudentRepository
StudentResponseDto
StudentService
resources
static
scratch_9.txt
com
myapp
usercomponent
controllers
services
productcomponent
controllers
services

# Data validation

09 May 2024 11:10

## Data validation is important because:

Data integrity - make sure all your data is valid (like emails)

Prevent Attacks - prevent users from sending incorrect data inputs to attack your product

Error Prevention - insures your data is vaild for the rest of your code

User experience - ensuring all data appears correctly and errors are well formatted

Performance - receiving large amounts of incorrect data can affect your applications performance

Business Logic compliance - business services will require certain types and formats of data

# Spring validation

09 May 2024 11:18

We need to ensure the user is entering the correct data inputs to our services and DTO's

Spring boot validation dependency:

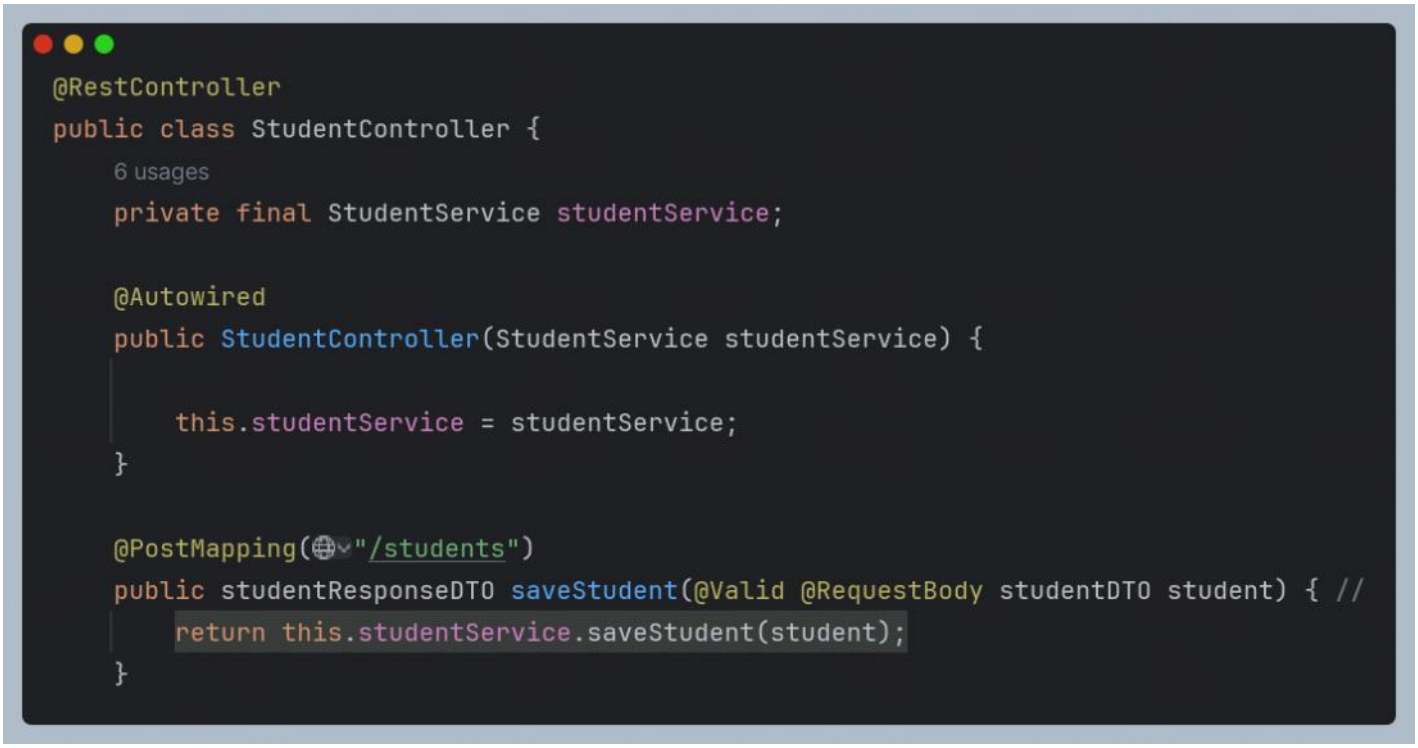
```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
    </dependencies>
```

Validation should be added to the first point of user interaction (the DTO normally)

All the below annotations work with POJO's and Records:

```
public record studentDTO(
        1 usage
        @NotEmpty
        String firstname,

        1 usage
        @NotEmpty
        String lastname,
        1 usage
        String email,

        1 usage
        Integer schoolId
) {
}
```

@NotEmpty - this value cannot be empty



```java
@RestController
public class StudentController {
    6 usages
    private final StudentService studentService;

    @Autowired
    public StudentController(StudentService studentService) {

        this.studentService = studentService;
    }

    @PostMapping("/students")
    public studentResponseDTO saveStudent(@Valid @RequestBody studentDTO student) { //
        return this.studentService.saveStudent(student);
    }
```

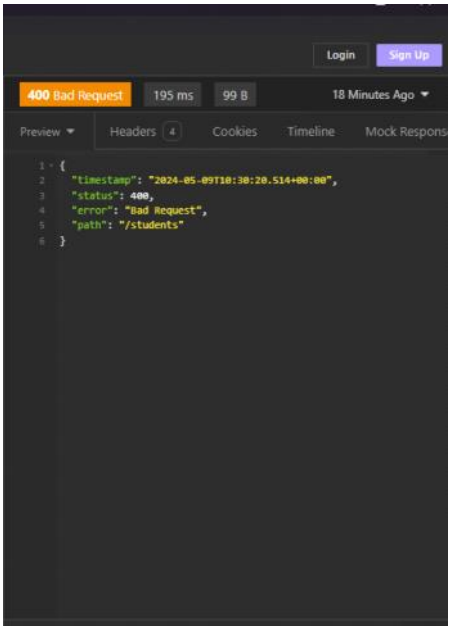
@Valid - enables validation for this controller

# Raising validation Exceptions and General Exceptions + ForEach

09 May 2024 11:35

Creating a method to handle our exception.

By default if validation fails user gets:




Produced this exception:

```
00  WARN 72440 --- [nio-8080-exec-1] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.web.bind.MethodArgumentNotValidException:
```

Exception handler:

```java
@ExceptionHandler
//Every time this exception is thrown, this code will be called
public ResponseEntity<?> MethodArgumentNotValidException( //Question Mark means of return a RespnosEntity of any type
                                                          MethodArgumentNotValidException exp
) {
    HashMap<String, String> errors = new HashMap<~>();
    exp.getBindingResult().getAllErrors().forEach(error -> {
        var fieldName = ((FieldError) error).getField();
        var errorMessage = error.getDefaultMessage();
        errors.put(fieldName, errorMessage);
    });
    return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
}
}
```

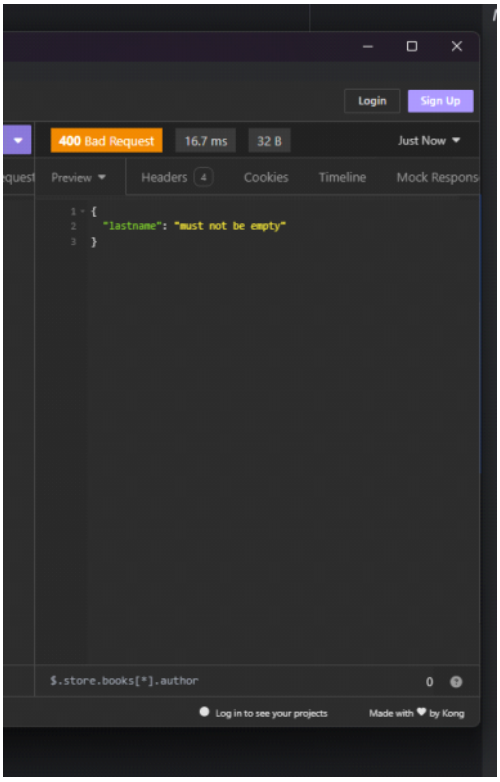
ResponseEntity<?> - Means its gonna return a responseEntity of some type

Exp.getBindingResult().getAllErrors() - get the exception results then get all the errors in a list

.forEach - for each one in the list do this lambda

Return HashMap of errors

Makes:




To set a custom message:

```
package com.matt.example.Student;

import jakarta.validation.constraints.NotEmpty;

3 usages
public record studentDTO(
        1 usage
        @NotEmpty(message = "Firstname should not be empty")
        String firstname,

        1 usage
        @NotEmpty (message = "last name shouldn't be empty")
        String lastname,
        1 usage
        String email,

        1 usage
        Integer schoolId
) {
}
```

# Validation Annotations List

09 May 2024 12:10

- **@NotNull:** Ensures a field is not null.
- **@NotBlank:** Enforces non-nullity and requires at least one non-whitespace character.
- **@NotEmpty:** Guarantees that collections or arrays are not empty.
- **@Min(value):** Checks if a numeric field is greater than or equal to the specified minimum value.
- **@Max(value):** Checks if a numeric field is less than or equal to the specified maximum value.
- **@Size(min, max):** Validates if a string or collection size is within a specific range.
- **@Pattern(regex):** Verifies if a field matches the provided regular expression.
- **@Email:** Ensures a field contains a valid email address format.
- **@Digits(integer, fraction):** Validates that a numeric field has a specified number of integer and fraction digits.
- **@Past** and **@Future** : Checks that a date or time field is in the past and future respectively.
- **@AssertTrue** and **@AssertFalse**: Ensures that a boolean field is true. and false respectively.
- **@CreditCardNumber**: Validates that a field contains a valid credit card number.

From <https://medium.com/@himani.prasad016/validations-in-spring-boot-e9948aa6286b>

To find a bigger list go to External libs and look at constraints

# Why Test

09 May 2024 13:01

Quality assurance - Tests help ensure your application works as expected

Regression Testing - These tests makes sure existing functions work as excepted

Documentation - Helps show your code works as excepted

Code Maintainability - Encourage, more modular, better quality code

Refactoring Confidence - show's you if something you have done is working or not

Collaboration - Tests make sure the work of one doesn't impact another

CI/ID pipelines - automatic testing when you push

Reduced Debugging time - reduces the time spent debugging

Scalability - allows you to confidentially scale up

Security - Some test help find security problems

# Spring testing modules

09 May 2024 13:18

## SPRING TEST - INTRODUCTION

- Spring Boot provides utilities and annotations for testing applications
- Test support is provided by two modules
  - spring-boot-test contains core items
  - spring-boot-test-autoconfigure supports auto-configuration for tests
  - Spring-boot-starter-test "Starter", imports:
    - Spring Boot test modules
    - JUnit, AssertJ, Hamcrest, and a number of other useful libraries

# Spring test

09 May 2024 13:21

A spring boot application is a Spring ApplicationContext

To declare a test with SpringBootTest annotation when you need spring features like ApplicaationContext

Junit 4 support requires @RunWith(SpringRunner.class) to be added to your test

JUnit5 doesn't require this

Spring has lots of autoConfigrations, its often recommended you only load 'slices' of the auto-configs though, as they are too much for tests often.

These auto configs give you a number of annotations and features you can use

Each slice restricts the amount of these you can access

# What to test

09 May 2024 13:29

You only test things with logic within:

Repositories - don’t require tests

Mappers - do require tests

Services - Do require tests

Controllers - do require tests

# Test function explainer

09 May 2024 13:32

Always keep the same package naming between 'main' and tests

Tests use JUnit5 or JUnit4

Use CTRL+SHIFT+T to generate a TEST

Screenshot explains different functions

```java
class StudentMapperTest {

    @BeforeAll
    static void beforeAll() {
        System.out.println("before class, before class  initialize something");
    }

    @AfterAll
    static void afterAll() {
        System.out.println("after we stopped using class, clean up database example");
    }

    @BeforeEach
    void setUp() {
        //Intialize anything before tests here
        System.out.println("before each method, initialize stuff per test");
    }

    @AfterEach
    void tearDown() {
        //Runs after every test
        System.out.println("after each method");
    }


    @Test
    public void testMethod1() {
        System.out.println("my first test");
    }
    @Test
    public void testMethod2() {
        System.out.println(" my second test");
    }


}
```

# Testing the mappers (coverage tests)

09 May 2024 14:00

Remember, we are using Junit5

Assertions generate errors based on conditions, used during tests

assertEquals will error if two things are not the same
assertNotNull will error something is null
AssertThrows allows us to check what exception was thrown

Screenshot demo's everything

```
class StudentMapperTest {

    @BeforeEach
    void setUp() {

        //Before each method is run, do this
        studentMapper = new StudentMapper();
    }

    4 usages
    private StudentMapper studentMapper;

    @Test
    public void shouldMapStudentDtoToStudentClass() {
        studentDTO dto = new studentDTO( firstname: "john", lastname: "doe", email: "john@mail.com", schoolId: 1);

        Student student = studentMapper.toStudent(dto);

        //Test "mark scheme"
        Assertions.assertEquals(dto.firstname(), student.getFirstname());
        Assertions.assertEquals(dto.lastname(), student.getLastname());
        Assertions.assertEquals(dto.email(), student.getEmail());
        Assertions.assertNotNull(student.getSchool());
        Assertions.assertEquals(dto.schoolId(), student.getSchool().getId());


    }


    @Test
    public void TostudentResponseDTOTest() {
        Student student = new Student( firstname: "matt", lastname: "butler", email: "mattbutler0001@gmail.com", age: 19);
        studentResponseDTO response = studentMapper.TostudentResponseDTO(student);


        //Test "mark scheme"
        Assertions.assertEquals(response.firstname(), student.getFirstname());
        Assertions.assertEquals(response.lastname(), student.getLastname());
        Assertions.assertEquals(response.email(), student.getEmail());

    }

    @Test
    //Testing for exception handling
    public void should_throw_null_pointer_exception_when_student_dto_is_null() {
       var exp =  Assertions.assertThrows(NullPointerException.class, () -> studentMapper.toStudent( dto: null)); //this exception should be thrown
        Assertions.assertEquals( expected: "The Student DTO should not be null", exp.getMessage());

    }

}
```

# Testing the student service

11 May 2024 06:05

**Check screenshots for this one** (trust)

We need to use test isolation where possible

This often includes mocking a database

"Mock" means "faking an objects and/or methods"

**In this case, we will be mocking our dependences using Mockito**

injectMocks - will find any constructor compatible dependencies with @mock and inject them into annotated service

Mock - mock this method / object

Then just open the mocks / enable them for current class

## Mocking our dependency calls:

To find what calls to mock go through method logic

We must mock our dependency calls used in our test

Use Mockito's logic to create them with the same logic as real function

**aguramentMatcher.any(Class) basically, if the input to the call is dynamic, could be any object of a certain type (like many different students). use any(CLASS OBJECT TYPE ).**

## Test layout:

Given (mock calls as well),

When,

Then (mark scheme),

## performance:

Verify - allows us to see how many times a method is called during the execution of the test (stops database dupes etc) **This includes nested calls with our methods (the entire point)**

```java
class StudentServiceTest {

    4 usages
    @InjectMocks //Inject our mocked objects into student service
    private StudentService studentService;

    //Declare dependencies

    8 usages
    @Mock //Mock these objects
    private StudentRepository repository;  //these need to be mocked

    7 usages
    @Mock //Mock these objects
    private StudentMapper studentMapper; //these need to be mocked

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks( testClass: this);
    }

    @Test
    public void should_successfully_save_a_student() {
        //Given
        studentDTO dto = new studentDTO( firstname: "john", lastname: "doe", email: "john@mail.com", schoolId: 1);

        Student student = new Student( firstname: "john", lastname: "doe", email: "john@mail.com", age: 20);

        Student savedStudent = new Student( firstname: "john", lastname: "doe", email: "john@mail.com", age: 20);

        savedStudent.setId(1);


        //Mock the calls used by this test
        Mockito.when(studentMapper.toStudent(dto))
                .thenReturn(student);
        Mockito.when(repository.save(student)).thenReturn(savedStudent);

        Mockito.when(studentMapper.TostudentResponseDTO(savedStudent)).thenReturn(new studentResponseDTO( firstname: "john", lastname: "doe", email: "john@mail.com"));



        //When
        studentResponseDTO response = studentService.saveStudent(dto);

        //Then
        assertEquals((dto.firstname()), response.firstname());
        assertEquals(dto.lastname(), response.lastname());
        assertEquals(dto.email(), response.email());

        Mockito.verify(studentMapper, Mockito.times( wantedNumberOfInvocations: 1)).toStudent(dto); //Check toStudent has only been called once during the execution of the test
        Mockito.verify(repository, Mockito.times( wantedNumberOfInvocations: 1)).save(student); //Check save has only been called once during the execution of the test
        Mockito.verify(studentMapper, Mockito.times( wantedNumberOfInvocations: 1)).TostudentResponseDTO(savedStudent); //Check ToStudentResponseDTO has only been called once during the execution of the test



    }

    @Test
    public void should_return_all_students() {
        //Given
        List<Student> students = new ArrayList<>();
         students.add(new Student( firstname: "john", lastname: "doe", email: "john@mail.com", age: 20));
         //Mock Calls
         Mockito.when(repository.findAll()).thenReturn(students);
         Mockito.when(studentMapper.TostudentResponseDTO(ArgumentMatchers.any(Student.class))).thenReturn(new studentResponseDTO( firstname: "john", lastname: "doe", email: "john@mail.com")); //Because we could pass any student use any()
```

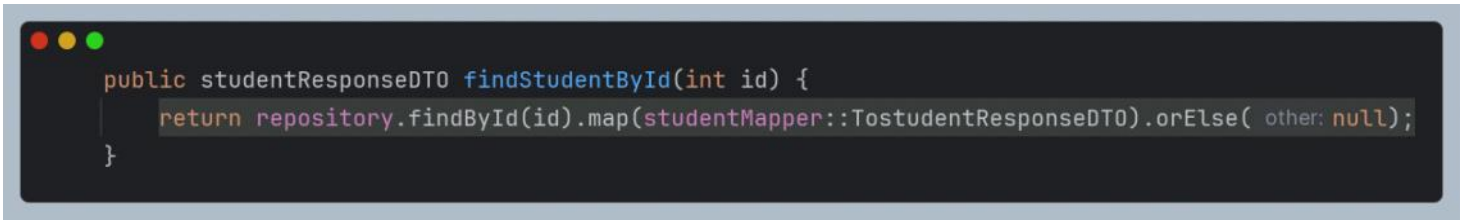
```java
        //Given
        List<Student> students = new ArrayList<>();
         students.add(new Student( firstname: "john", lastname: "doe", email: "john@mail.com", age: 20));
         //Mock Calls
         Mockito.when(repository.findAll()).thenReturn(students);
         Mockito.when(studentMapper.TostudentResponseDTO(ArgumentMatchers.any(Student.class))).thenReturn(new studentResponseDTO( firstname: "john", lastname: "doe", email: "john@mail.com")); //Because we could pass any student use any()

         //When
        List<studentResponseDTO> findAllStudent = studentService.findAllStudent();

        //Then
        assertEquals(students.size(), findAllStudent.size());
        Mockito.verify(repository, Mockito.times( wantedNumberOfInvocations: 1)).findAll();

    }

    @Test
    public void should_return_student_with_id() {
        //Given
        int studentID = 1;
        Student student = new Student( firstname: "john", lastname: "doe", email: "john@mail.com", age: 20);
        studentResponseDTO dto = new studentResponseDTO( firstname: "john", lastname: "doe", email: "john@mail.com");

        //Mock calls
        Mockito.when(repository.findById(studentID)).thenReturn(Optional.of(student));
        Mockito.when(studentMapper.TostudentResponseDTO(student)).thenReturn(dto);

        //When
       studentResponseDTO studentFound =  studentService.findStudentById(1);

       //then
        assertEquals(dto.firstname(), studentFound.firstname());
        assertEquals(dto.lastname(),studentFound.lastname());
        assertEquals(dto.email(),student.getEmail());
        Mockito.verify(repository, Mockito.times( wantedNumberOfInvocations: 1)).findById(studentID);
    }

    @Test
    public void findStudentByNameTest() {

        //Given
        String name = "john";
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student( firstname: "john", lastname: "doe", email: "john@mail.com", age: 20));
        studentResponseDTO dto = new studentResponseDTO( firstname: "john", lastname: "doe", email: "john@mail.com");

        //Mock calls

        Mockito.when(repository.findAllByFirstname(name)).thenReturn(studentList);
        Mockito.when(studentMapper.TostudentResponseDTO(Mockito.any(Student.class))).thenReturn(dto);

        //When

        List<studentResponseDTO> studentDTOs = studentService.findStudentByName("john");

        //Then
        System.out.println(studentDTOs.size());
        System.out.println(studentList.size());
        assertEquals(studentList.size(), studentDTOs.size());
        Mockito.verify(repository, Mockito.times( wantedNumberOfInvocations: 1)).findAllByFirstname(name);

    }
}
```

# Java Optional

09 May 2024 08:13

Any Optional Type in java can be transformed into a mapped to a method and transformed, :: being a method reference.

Spring repositories return Optional Objects



# Java Streams

09 May 2024 07:18

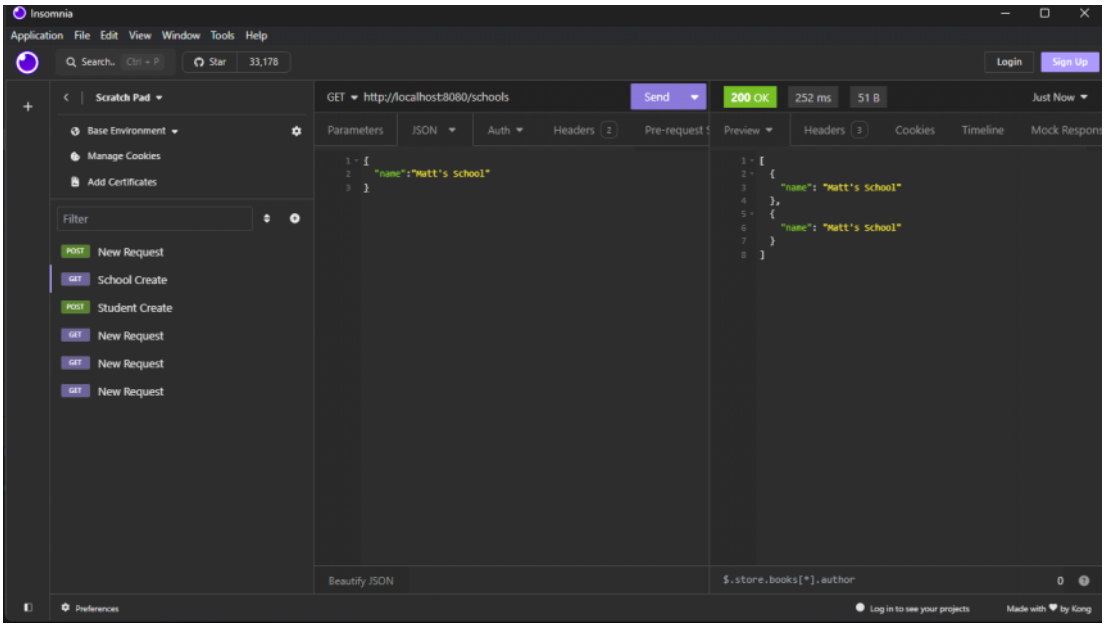
Java Streams:

```java
private schoolDTO toSchoolDTO(School school) {
    return new schoolDTO(school.getName());
}

@GetMapping("/schools")
public List<schoolDTO> findAll () {
    return schoolRepository.findAll().stream()
            .map(this::toSchoolDTO).collect(Collectors.toList());


}
```

Steam - in java converts a list of items to be processed one at a time

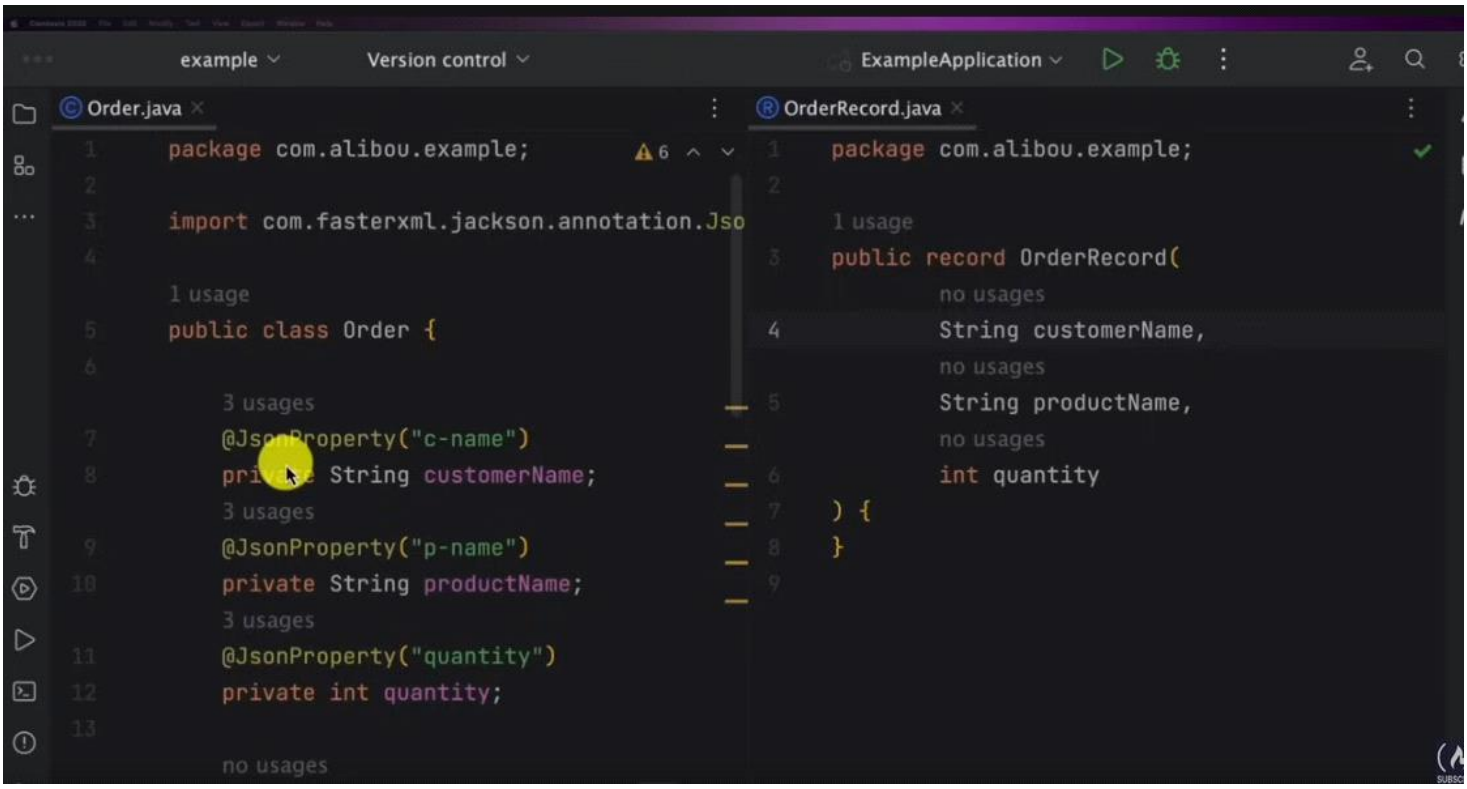
.Map(this::toSchoolDTO) - applies schoolDTO to each of the items in the map

.Collect puts the results of this back into a list



# Records vs POJO Classes

08 May 2024 05:55




Classes:

Requires ToString override

Requires setters and getters to serialize and deserialize

Can have logic within the class

Records:

Records are final so they can't be used with mutable requirement library

A better choice for simple data transfer with no logic

# REST crud functions + OrElse

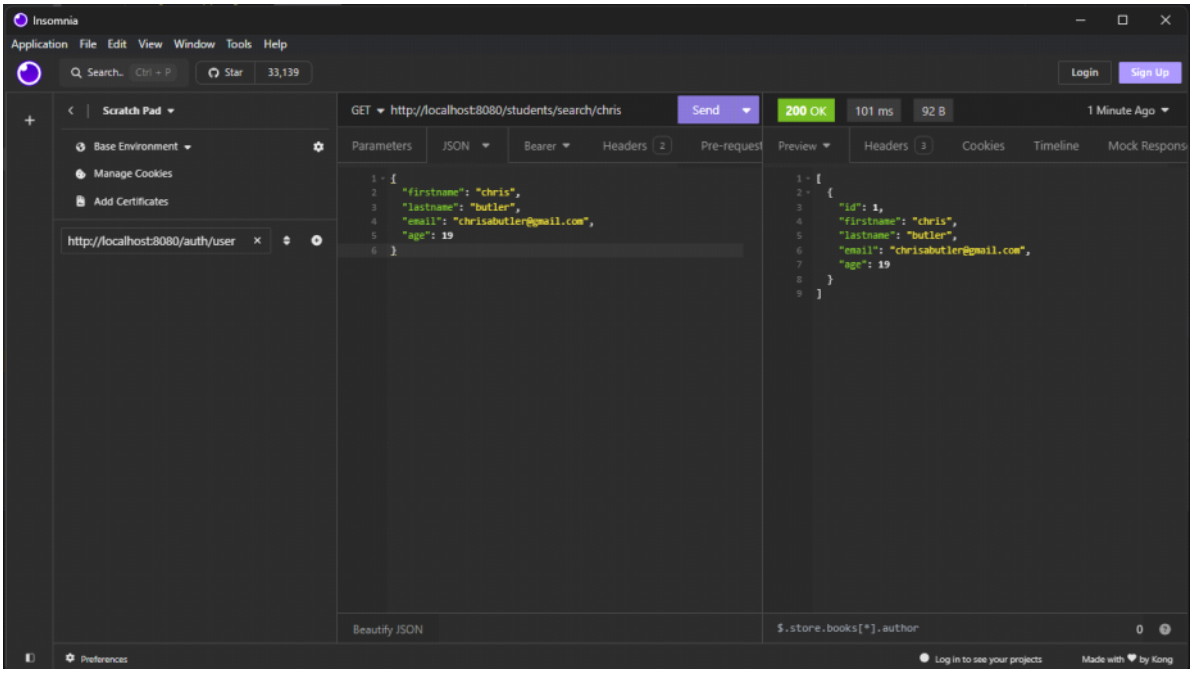
08 May 2024 11:28

```java
package com.matt.example;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

2 usages
@Repository
public interface StudentRepository extends JpaRepository<Student, Integer> {

    1 usage
    List<Student> findAllByFirstname(String name);

}
```

Build queries with a method signatures like above

```java
@RestController
public class FirstController {
    6 usages
    private final StudentRepository repository;

    @Autowired
    public FirstController(StudentRepository repository) {
        this.repository = repository;
    }

    @PostMapping("/students")
    public Student post(@RequestBody Student student) {
        return repository.save(student);
    }

    @GetMapping("/students")
    public List<Student> findAllStudent() {
        return repository.findAll();
    }
    @GetMapping("/students/{student-id}")
    public Student findStudentById(@PathVariable("student-id") int studentId) {
        return repository.findById(studentId).orElse(new Student());
    }
    @GetMapping("/students/search/{student-name}")
    public List<Student> findStudentByName(@PathVariable("student-name") String studentName) {
        //Student[] students = {};
        return repository.findAllByFirstname(studentName);
    }
```

```
        return repository.findAllByFirstname(studentName);
    }

    @DeleteMapping("/students/{student-id}")
    @ResponseStatus(HttpStatus.OK) //Sets response code
    public void delete(@PathVariable("student-id") Integer studentId) {
        repository.deleteById(studentId);
```




```
return repository.findById(studentId).orElse( other: null);
```

**.OrElse function - can be used on any optional object**

Repository.save() returns saved values

```
    @DeleteMapping("/students/{student-id}")
    @ResponseStatus(HttpStatus.OK) //Sets response code
    public void delete(@PathVariable("student-id") String studentId) {
        repository.deleteStudentById(studentId);
    }


}
```

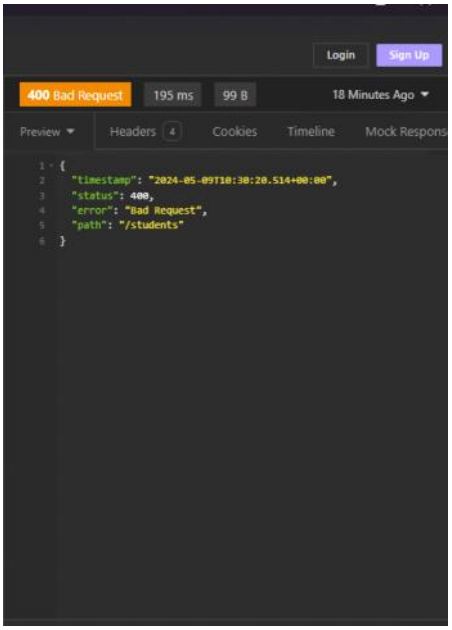
**ResponseStatus set given no exceptions**

# Raising validation Exceptions and General Exceptions + ForEach

09 May 2024 11:35

Creating a method to handle our exception.

By default if validation fails user gets:




Produced this exception:

```
00  WARN 72440 --- [nio-8080-exec-1] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.web.bind.MethodArgumentNotValidException:
```

Exception handler:

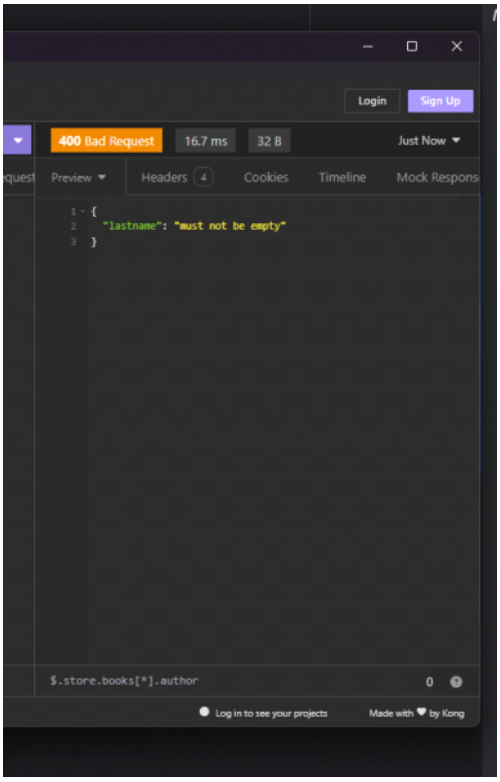
```java
@ExceptionHandler
//Every time this exception is thrown, this code will be called
public ResponseEntity<?> MethodArgumentNotValidException( //Question Mark means of return a RespnosEntity of any type
                                                          MethodArgumentNotValidException exp
) {
    HashMap<String, String> errors = new HashMap<~>();
    exp.getBindingResult().getAllErrors().forEach(error -> {
        var fieldName = ((FieldError) error).getField();
        var errorMessage = error.getDefaultMessage();
        errors.put(fieldName, errorMessage);
    });
    return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
}
}
```

ResponseEntity<?> - Means its gonna return a responseEntity of some type

Exp.getBindingResult().getAllErrors() - get the exception results then get all the errors in a list

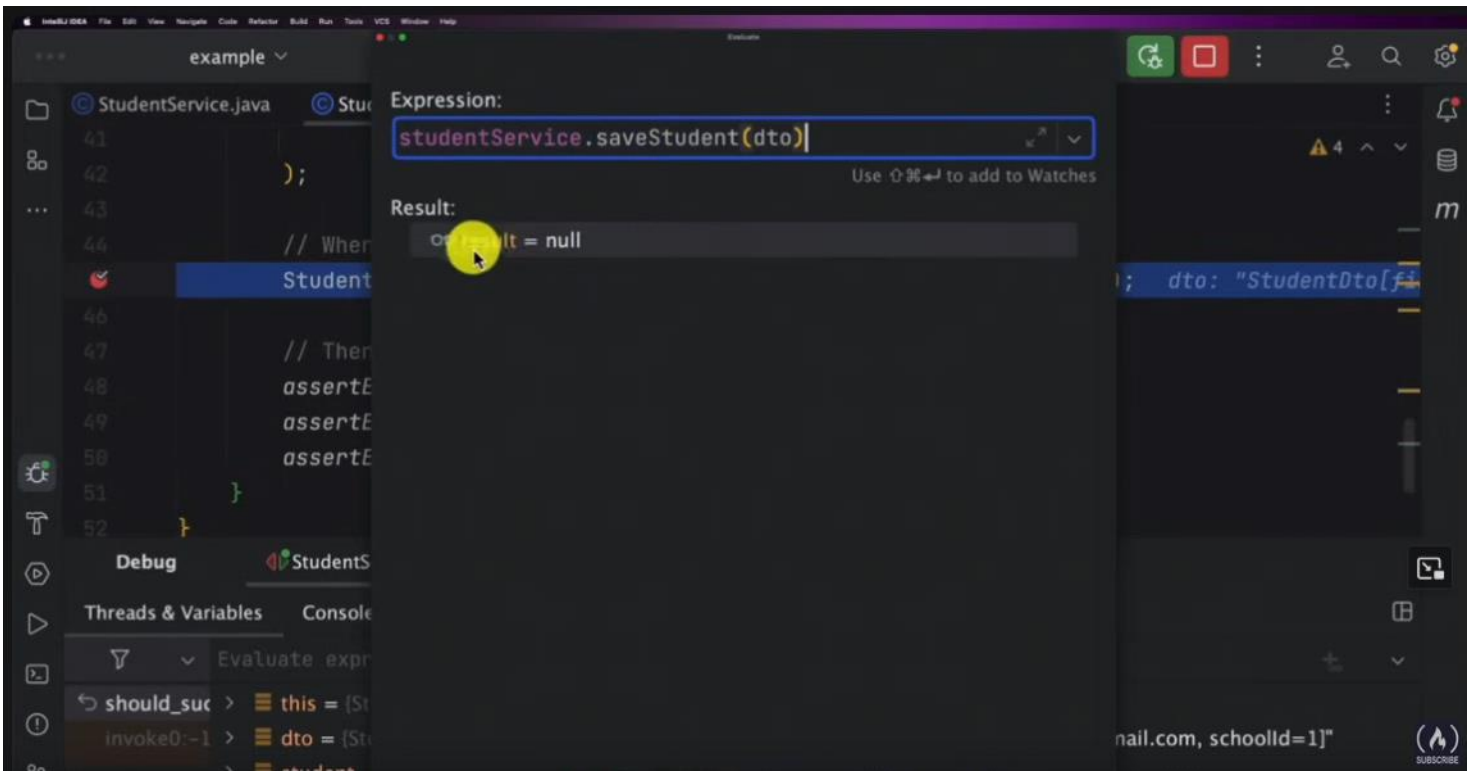
.forEach - for each one in the list do this lambda

Makes:


Login
Sign Up
400 Bad Request
16.7 ms
32 B
Just Now
Preview
Headers 4
Cookies
Timeline
{
"lastname": "must not be empty"
}
$.store.books[*].author
0
Log in to see your projects
Made with by Kong

# Evaluations in intelJ

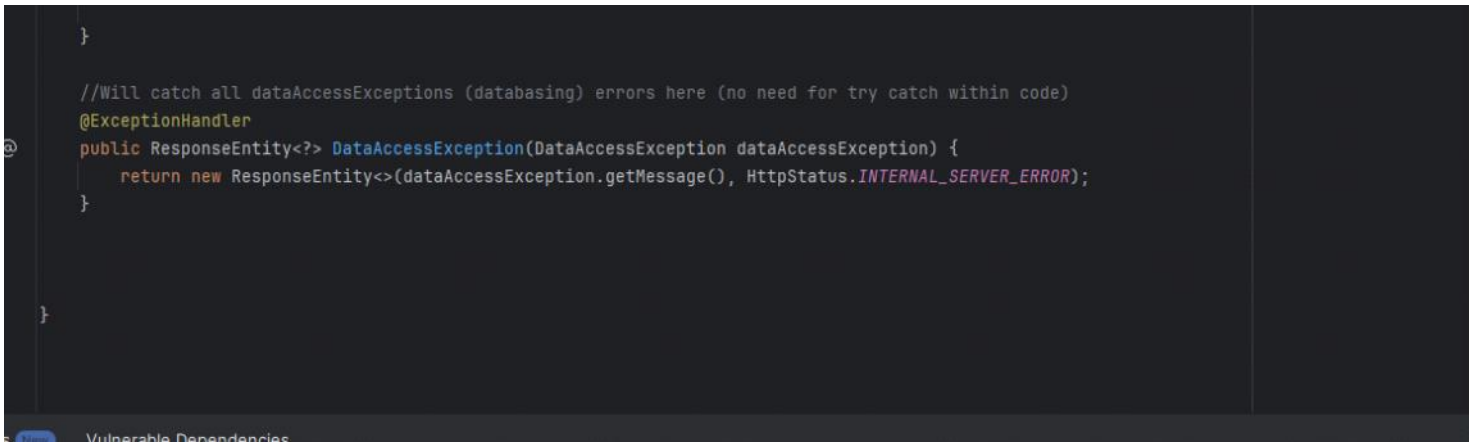
11 May 2024 06:32



# Exceptions

11 May 2024 15:20

YOU ONLY NEED TO BUILD HANDLERS IN YOUR CONTROLLERS TO CATCH EXCEPTIONS NO NEED FOR TRY CATCHES



# .forEach(DO THIS REFERENCE)

15 May 2024 11:56

.Foreach() can be run on collections (LISTS etc)

Or on streams

# shortcuts

08 May 2024 08:23

CTRL+B for lookup on basically anything, just download the sources