SPRING BOOT CUSTOMIZATION

In Spring Boot, "customization" refers to the process of adapting and configuring application to the specific needs of project or business.

CONTENT

DEFINE SPRING BOOT FILTER

- Spring Boot Filter is part of Spring Boot's architecture, allow to perform operations before and after requests are sent to application endpoints. This is a powerful way to perform functions such as authentication, logging...

- Here's how to define a Spring Boot Filter:

1) <u>Create a Filter class</u>:

We need to create a new class that implements Filter interface throught “import jakarta.servlet.Filter”. In this class, we will override doFilter() method to handle the Filter request and response.

DEFINE SPRING BOOT FILTER

1) <u>Create a Filter class:</u>

We need to create a new class that implements Filter interface throught “import jakarta.servlet.Filter”. In this class, we will override doFilter() method to handle the Filter request and response.

```
public class MyFilter implements Filter {
    new *
    @Override
    public void init(FilterConfig filterConfig) {}
    no usages   new *
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) t
        System.out.println("Before sending request to API!"); //Processing before forwarding the request
        System.out.println("-------------------------------");
        filterChain.doFilter(servletRequest,servletResponse); //Forward the request to another endpoint or Servlet
        System.out.println("After receiving response from API!"); //Processing after forwarding the request
        System.out.println("Finish!");
    }
    new *
    @Override
    public void destroy() {}
```

DEFINE SPRING BOOT FILTER

1) Create a Filter class:

```java
public class MyFilter implements Filter {
    new *
    @Override
    public void init(FilterConfig filterConfig) {}
    no usages  new *
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) {
        System.out.println("Before sending request to API!"); //Processing before forwarding the request
        System.out.println("-------------------------------");
        filterChain.doFilter(servletRequest,servletResponse); //Forward the request to another endpoint or Servlet
        System.out.println("After receiving response from API!"); //Processing after forwarding the request
        System.out.println("Finish!");
    }
    new *
    @Override
    public void destroy() {}
```

Method doFilter() is called every time a request comes from the client and before it is forwarded to another endpoint or Servlet.

DEFINE SPRING BOOT FILTER

1) <u>Create a Filter class:</u>

We need to create a new class that implements the “import jakarta.servlet.Filter;” interface. In this class, we will implement the doFilter() method to handle the Filter request and response.

2) <u>Mark the Filter class with @Component or @Configuration</u>:

We mark the Filter class with the @Component annotation to have Spring Boot automatically manage and configure it, or we can mark it with @Configuration to configure it manually.

DEFINE SPRING BOOT FILTER

1) <u>Create a Filter class:</u>

We need to create a new class that implements the “import jakarta.servlet.Filter;” interface. In this class, we will implement the doFilter() method to handle the Filter request and response.

2) <u>Mark the Filter class with @Component or @Configuration</u>:

- We mark the Filter class with the @Component annotation to have Spring Boot automatically manage and configure it, or we can mark it with @Configuration to configure it manually.

- If not using @Component to manage the Filter class, we need to configure it in Spring Boot using FilterRegistrationBean or FilterRegistration beans in a configuration class.

DEFINE SPRING BOOT FILTER

2) Mark the Filter class with @Component or @Configuration:

Example with @Component:

```
@Component
public class MyFilter implements Filter {
    new *
    @Override
    public void init(FilterConfig filterConfig) {}
    no usages  new *
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) t
        System.out.println("Before sending request to API!"); //Processing before forwarding the request
        System.out.println("-------------------------------");
        filterChain.doFilter(servletRequest,servletResponse); //Forward the request to another endpoint or Servlet
        System.out.println("After receiving response from API!"); //Processing after forwarding the request
        System.out.println("Finish!");
    }
```

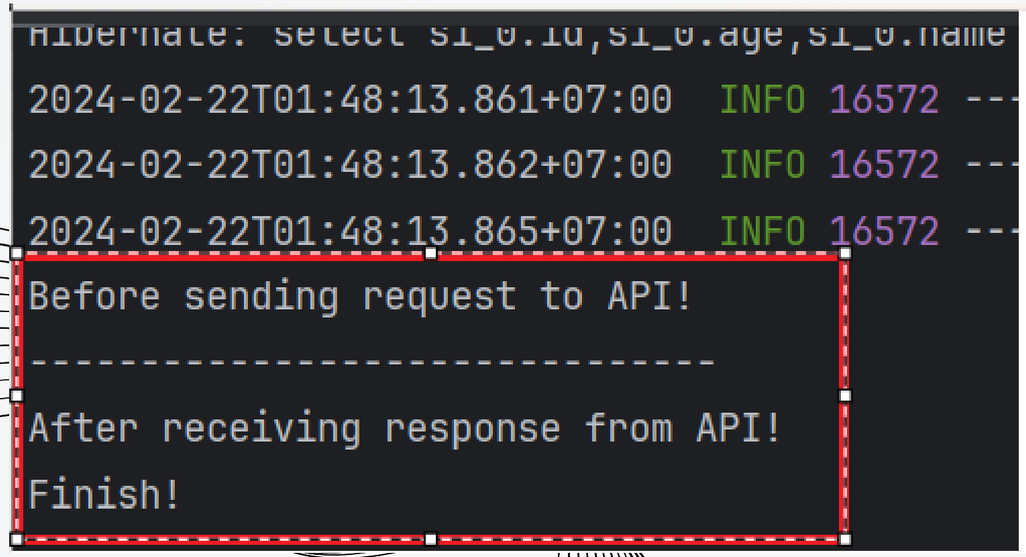
DEFINE SPRING BOOT FILTER

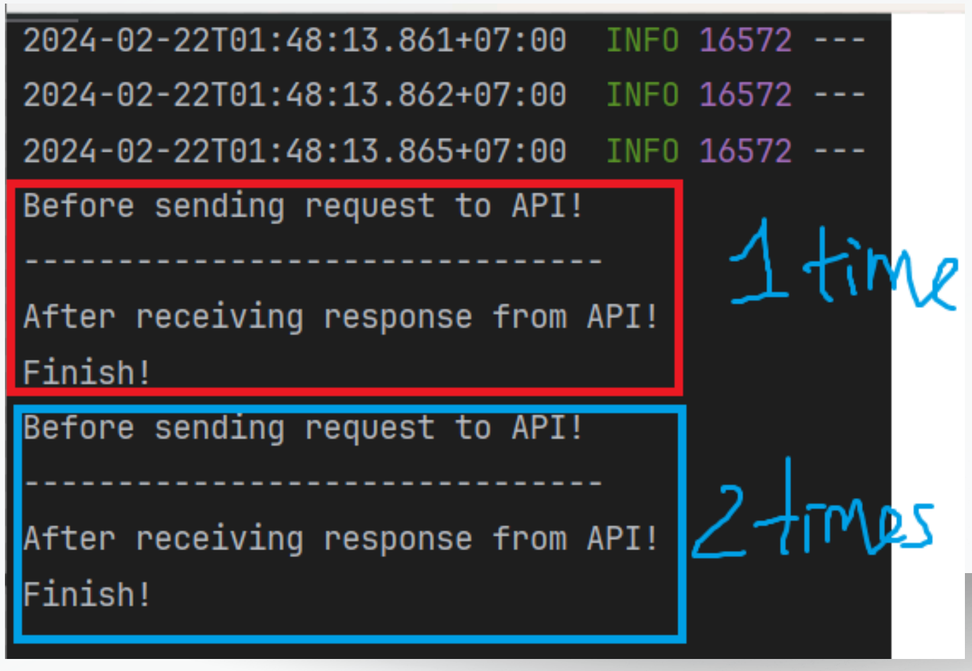
Each time when I run API “api/shoes” in port 8080 on localhost:

localhost:8080/api/shoes


```
{
    "name": "Rosh One",
    "brand": "Nike",
    "date": "2024-02-22"
}
```


Then the result in console:





DEFINE SPRING BOOT FILTER

2) Mark the Filter class with @Component or @Configuration:

Example with @Configuration:

```java
@Configuration
public class FilterConfig {

    new *
    @Bean
    public FilterRegistrationBean<MyFilter> myFilterRegistrationBean() {
        FilterRegistrationBean<MyFilter> registrationBean = new FilterRegistrationBean<>();
        registrationBean.setFilter(new MyFilter());
        registrationBean.addUrlPatterns("/api/*"); //Apply Filter to APIs under /api/* path
        //registrationBean.setOrder(1); // setting the order of filter if necessary
        return registrationBean;
    }
```

```java
@Configuration
public class FilterConfig {
    new *
    @Bean
    public FilterRegistrationBean<MyFilter> myFilterRegistrationBean() {
        FilterRegistrationBean<MyFilter> registrationBean = new FilterRegistrationBean<>();
        registrationBean.setFilter(new MyFilter());
        registrationBean.addUrlPatterns("/api/*"); //Apply Filter to APIs under /api/* path
        //registrationBean.setOrder(1); // setting the order of filter if necessary
        return registrationBean;
    }
```

- public FilterRegistrationBean<MyFilter> myFilterRegistrationBean(): This method is create and configure a FilterRegistrationBean to register a Filter (in this case, MyFilter) throught method setFilter()

- registrationBean.addUrlPatterns("/api/*"): It configures the URLs to which filter MyFilter will be applied.

DEFINE SPRING BOOT FILTER

Each time when I run API “api/getStudentsObject” in port 8080 on localhost:

localhost:8080/api/getStudentsObject


```
[
    {
        "id": 1,
        "name": "Trung Kien",
        "age": 20
    },
    {
        "id": 2,
        "name": "Trung Dung",
        "age": 20
    },
    {
        "id": 3,
        "name": "Trung Hieu",
        "age": 18
    },
    {
        "id": 4,
        "name": "Quoc Toan",
        "age": 19
    }
]
```


Then the result in console:

```
Before sending request to API!
------------------------------
Hibernate: select s1_0.id,s1_0.age,s1_0.name from student s1_0
After receiving response from API!
Finish!
```

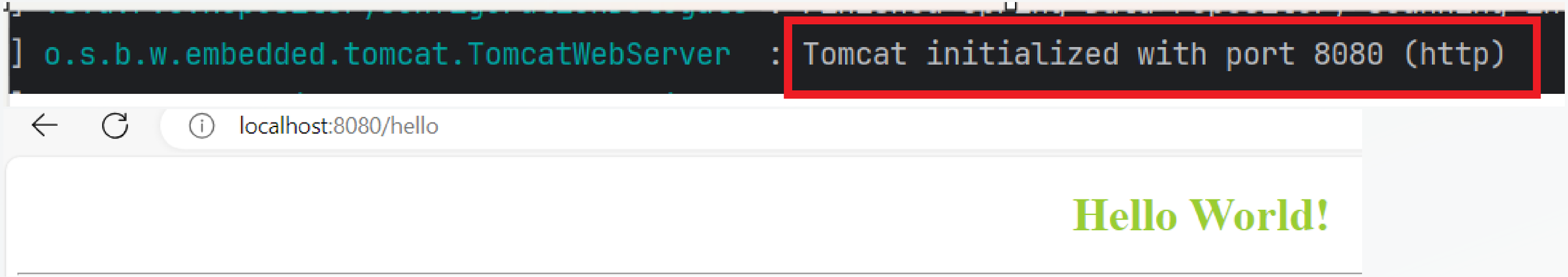
```
Before sending request to API!
------------------------------
Hibernate: select s1_0.id,s1_0.age,s1_0.name from student s1_0
After receiving response from API!
Finish!
```

```
Before sending request to API!
------------------------------
Hibernate: select s1_0.id,s1_0.age,s1_0.name from student s1_0
After receiving response from API!
Finish!
```

```
Before sending request to API!
------------------------------
Hibernate: select s1_0.id,s1_0.age,s1_0.name from student s1_0
After receiving response from API!
Finish!
```

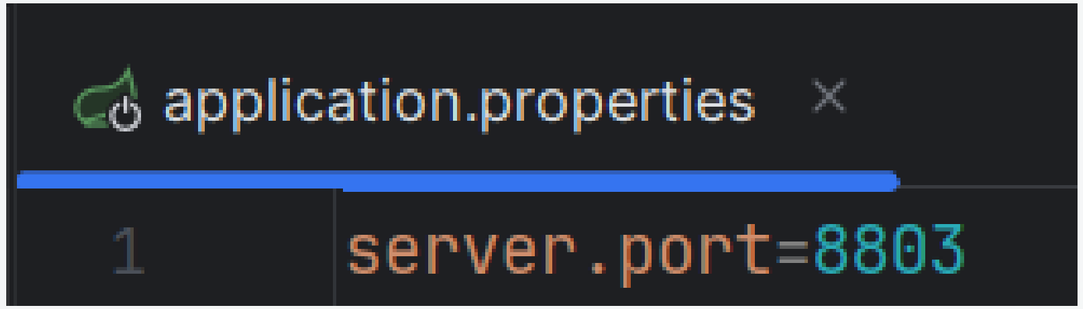
SOME WAYS TO CHANGE DEFAULT PORT IN SPRING BOOT

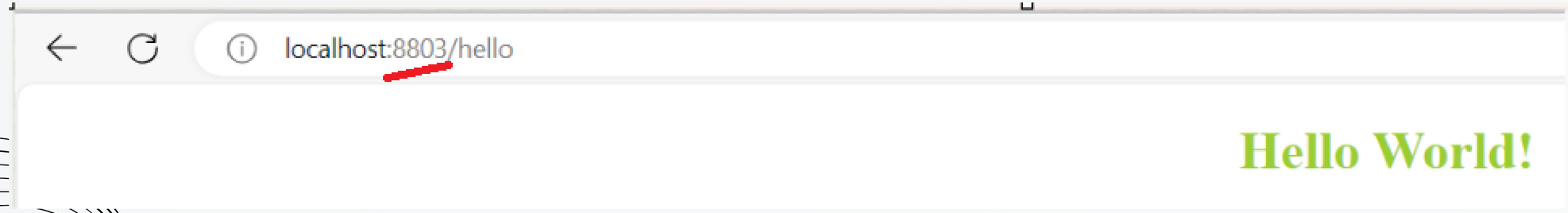
By default, when we run SpringBoot Application, localhost will start with port 8080



SOME WAYS TO CHANGE DEFAULT PORT IN SPRING BOOT

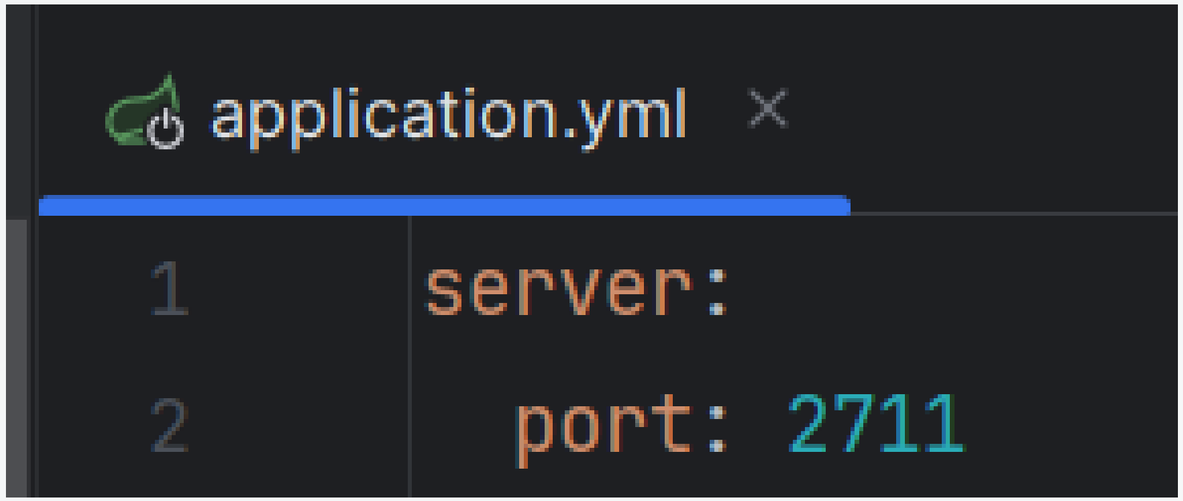
1) To change the default port in Spring Boot, we can setting the “server.port” property in the application.properties or application.yml file(with Maven project). For example:





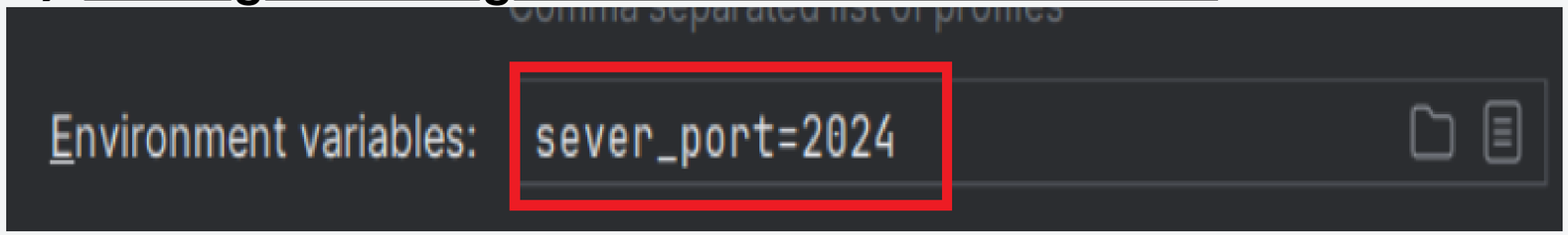
THE WAY TO CHANGE DEFAULT PORT IN SPRING BOOT

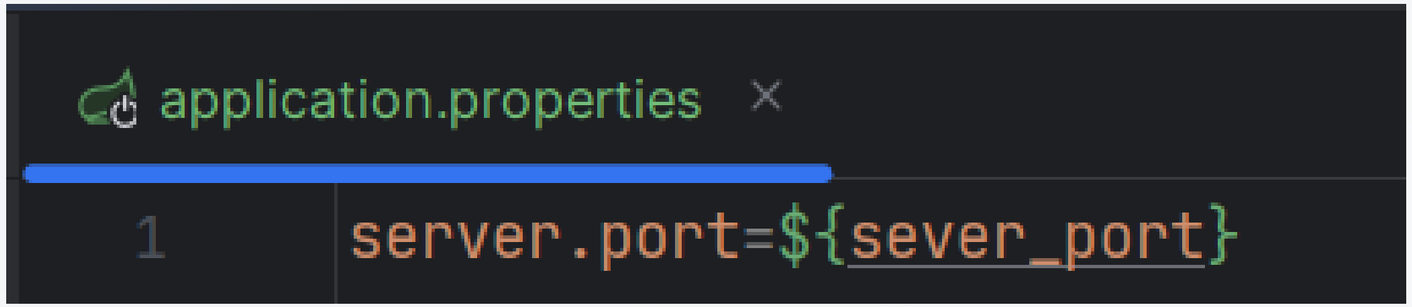
To change the default port in Spring Boot, we can setting the server.port property in the application.properties or application.yml file(with Maven project). For example:



SOME WAYS TO CHANGE DEFAULT PORT IN SPRING BOOT

2) Change throught Variable Enviroment:





SOME WAYS TO CHANGE DEFAULT PORT IN SPRING BOOT

2) Change throught Variable Enviroment:

application.properties

```
1  server.port=${sever_port}
```

Environment variables: sever_port=2024

The result on localhost:

localhost:2024/api/shoes


```
{
    "name": "Rosh One",
    "brand": "Nike",
    "date": "2024-02-22"
}
```

SOME WAYS TO CHANGE DEFAULT PORT IN SPRING BOOT

2) Change throught Variable Enviroment:

application.properties

```
1  server.port=${sever_port}
```

Environment variables: sever_port=2024

The result on localhost:

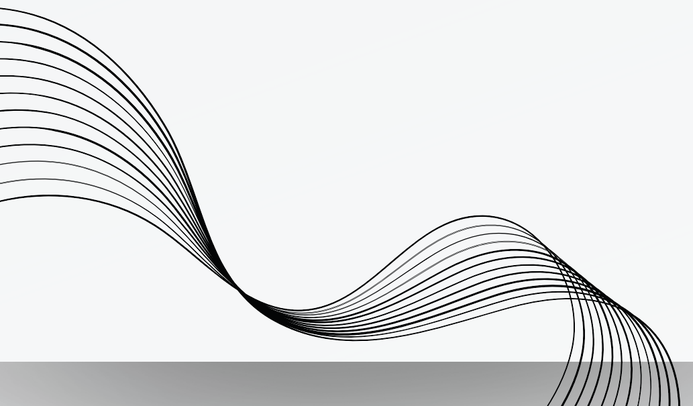
localhost:2024/api/shoes


```
{
    "name": "Rosh One",
    "brand": "Nike",
    "date": "2024-02-22"
}
```

THE WAY TO CHANGE CONTEXT PATH IN SPRING BOOT

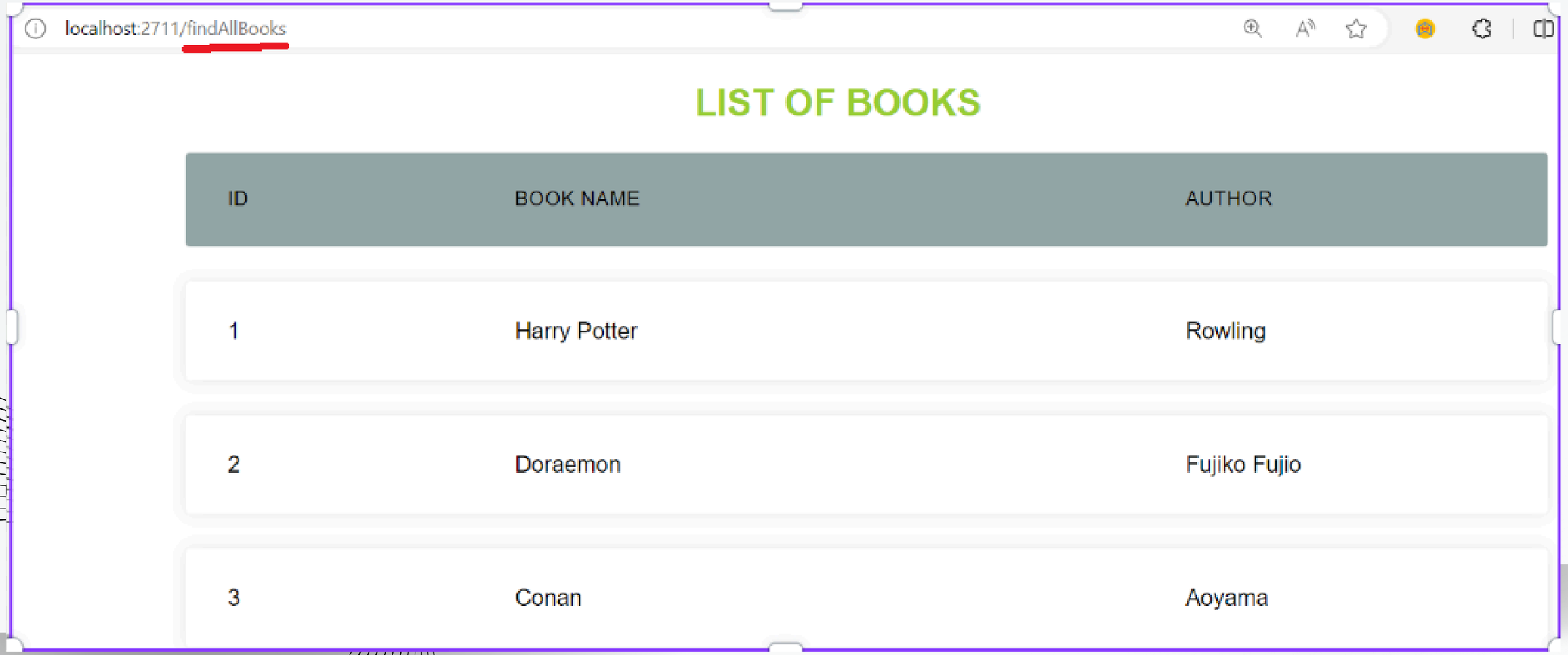
- In Spring Boot, context path is the base path which web application is deployed under on the web server. It is often used to determine how to access resources and application endpoints.

- By default, the context path in Spring Boot is “/”. This means our application will be deployed directly from the root of the web server, and all endpoints and resources will be accessed from there.



THE WAY TO CHANGE CONTEXT PATH IN SPRING BOOT

For example, I have context path “/findAllBooks”



THE WAY TO CHANGE CONTEXT PATH IN SPRING BOOT

1) we can configure the context path as desired by using the “server.servlet.context-path” property in the application.properties or application.yml file (for Maven).

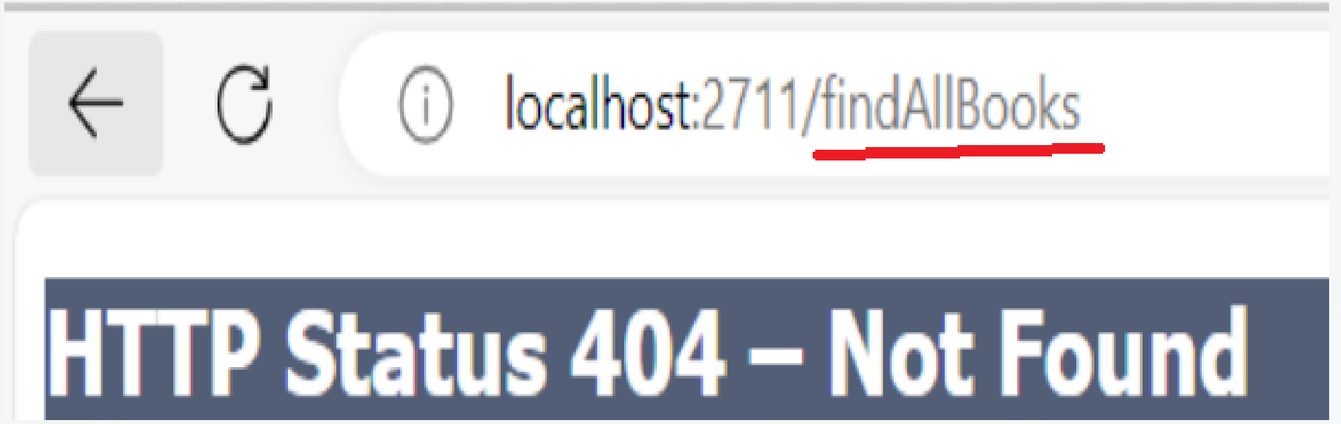
```yaml
server:
    servlet:
        context-path: /apiDay7
```

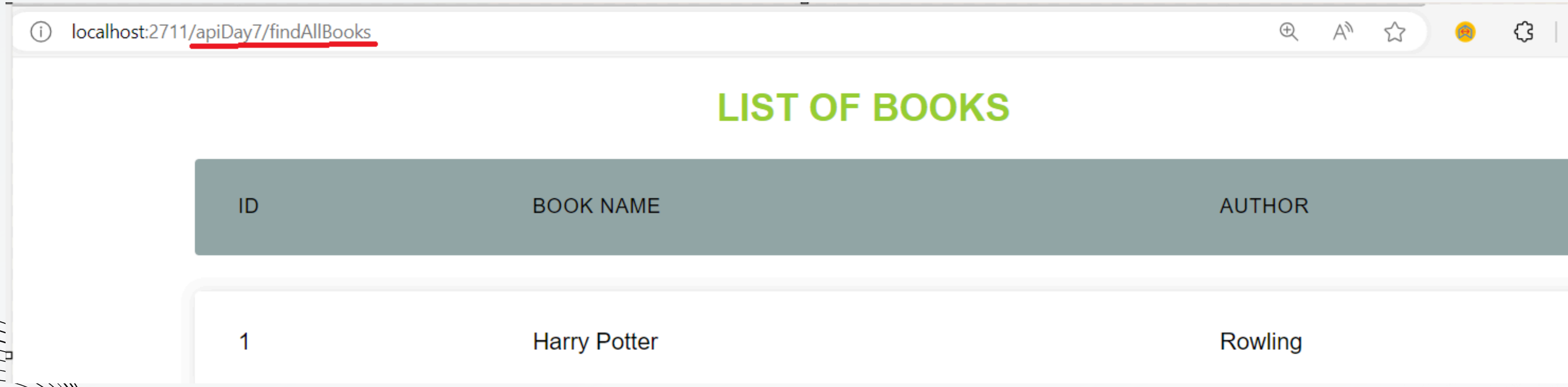
application.properties

```
1  server.servlet.context-path=/apiDay7
```

- All endpoints and resources in application will be prefixed with /apiDay7. For example, the “/home” endpoint will become “/apiDay7/home”

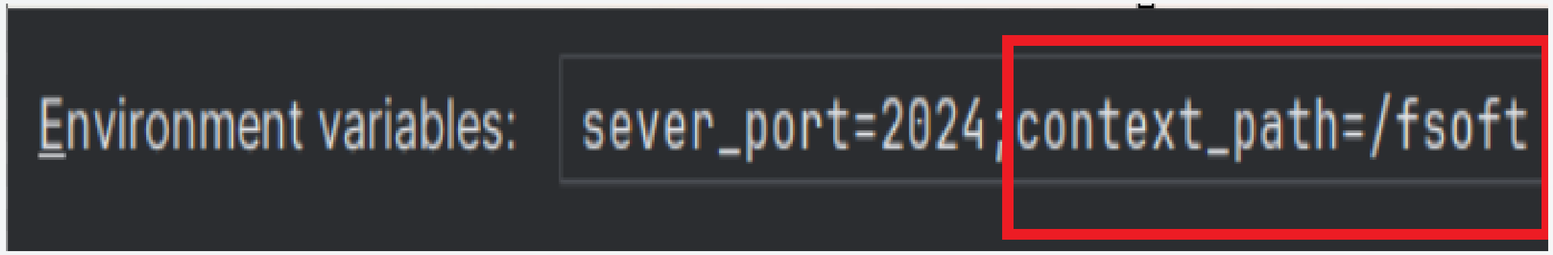
```yaml
server:
    servlet:
        context-path: /apiDay7
```



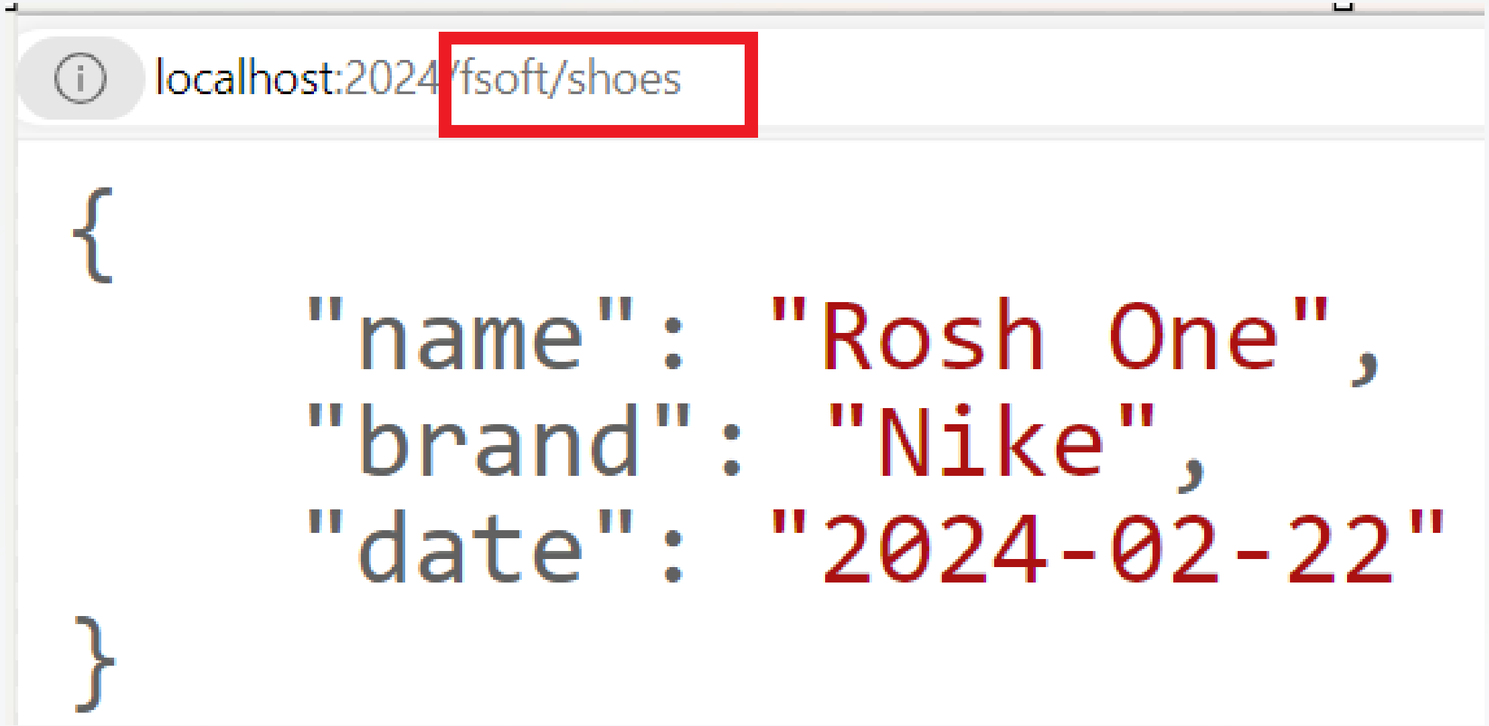


THE WAY TO CHANGE CONTEXT PATH IN SPRING BOOT

2) Change through Variable Environment:

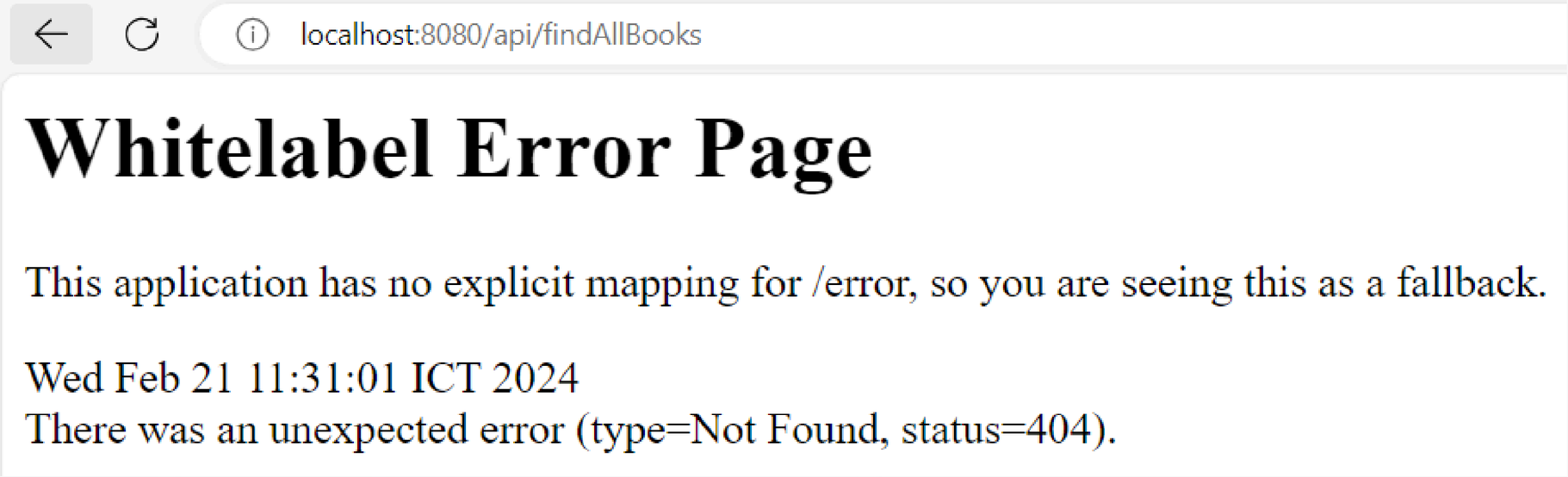


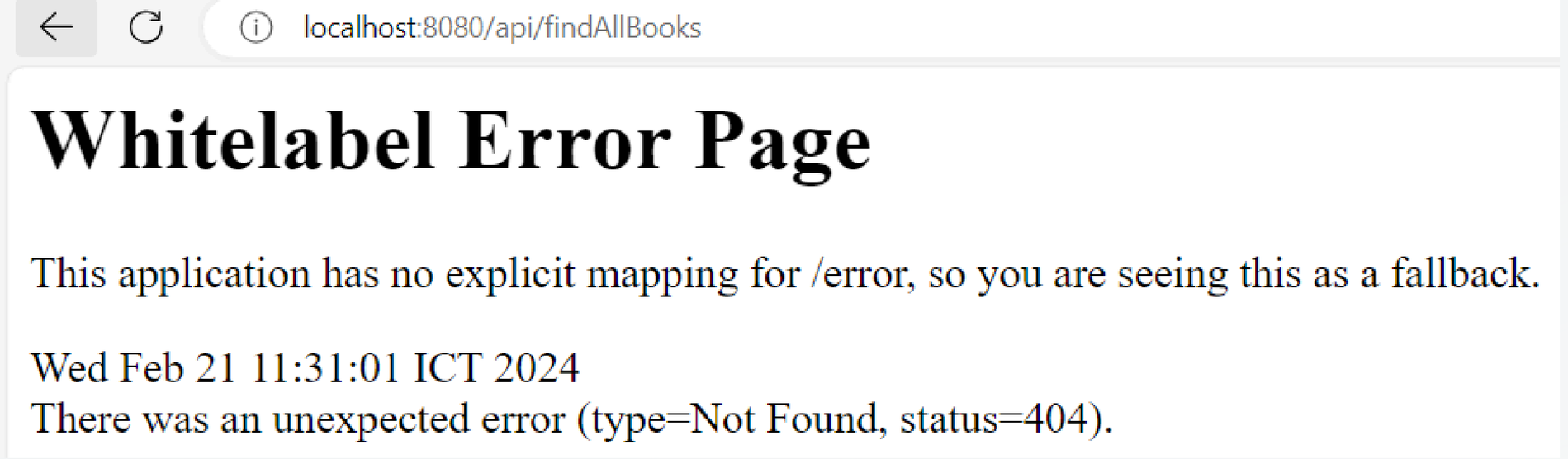
The result on localhost:



CUSTOMIZE WHITELABLE ERROR PAGE

In Spring Boot, when an error occurs in application and no specific handling is defined for it, Spring Boot automatically returns a default error page called "Whitelabel Error Page". This page provides error information including HTTP error codes, error messages, and more. For example:





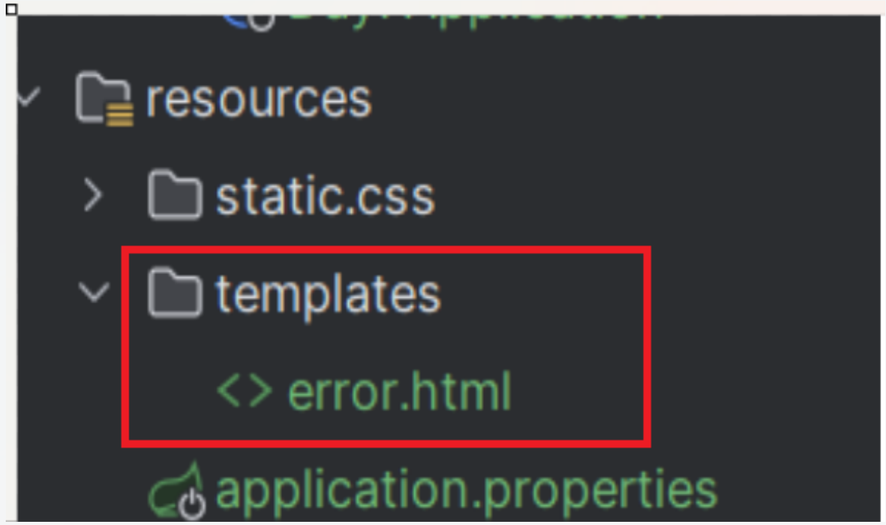
- “This application has no explicit mapping for /error, so you are seeing this as a fallback”: In Spring Boot, /error is often used to handle error cases and display the corresponding error page. However, in this case, no specific mapping is defined, so the Whitelabel Error Page is displayed as a fallback.

- Wed Feb 21 11:31:01 ICT 2024: This is the time the error occurred

- “There was an unexpected error (type=Not Found, status=404)”: The error is "Not Found" with an HTTP status code of 404. This means that the request could not find the requested resource or page on the server.

STEP TO CUSTOMIZE THE WHITELABLE ERROR PAGE:

1) Create a custom error page named “error.html” in folder src/main/java/resources/templates/error.html



Code in file error.html:

```
<body>
<div id="notfound">
    <div class="notfound-bg"></div>
    <div class="notfound">
        <div class="notfound-404">
            <h1>404</h1>
        </div>
        <h2>Oops! Page Not Found</h2>
        <a href="#">Back To Homepage | FPT SOFTWARE</a>
    </div>
</div>
</body>
```

STEP TO CUSTOMIZE THE WHITELABLE ERROR PAGE:

1) Create a custom error page named error.html in folder src/main/java/resources/templates/error.html

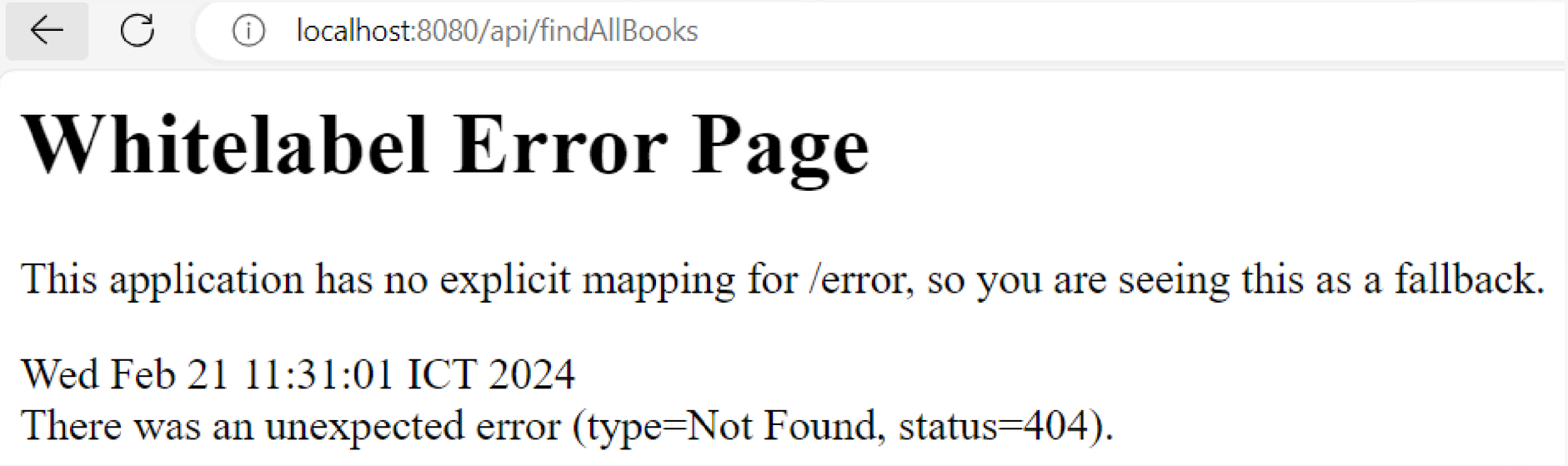
2) Turn off Whitelabel Error Page:

- To prevent Spring Boot from using the default Whitelabel Error Page, we can configure the application to turn it off.
- This can be done by setting the “server.error.whitelabel.enabled” property in the application.properties or application.yml file. After that, we done customize the whitelable error page

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
server.error.whitelabel.enabled=false
```

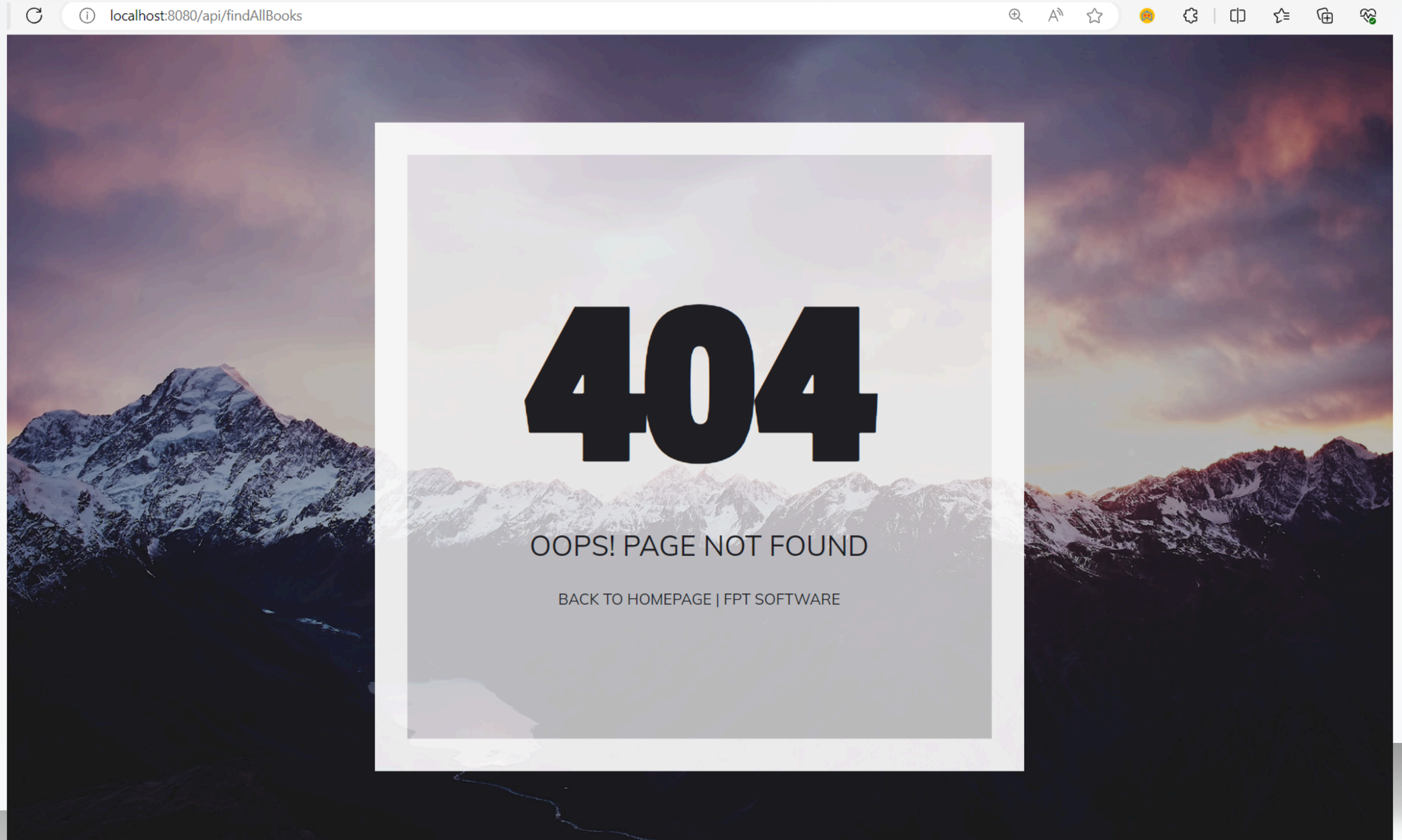
Here is the result:

Before:



Here is the result:

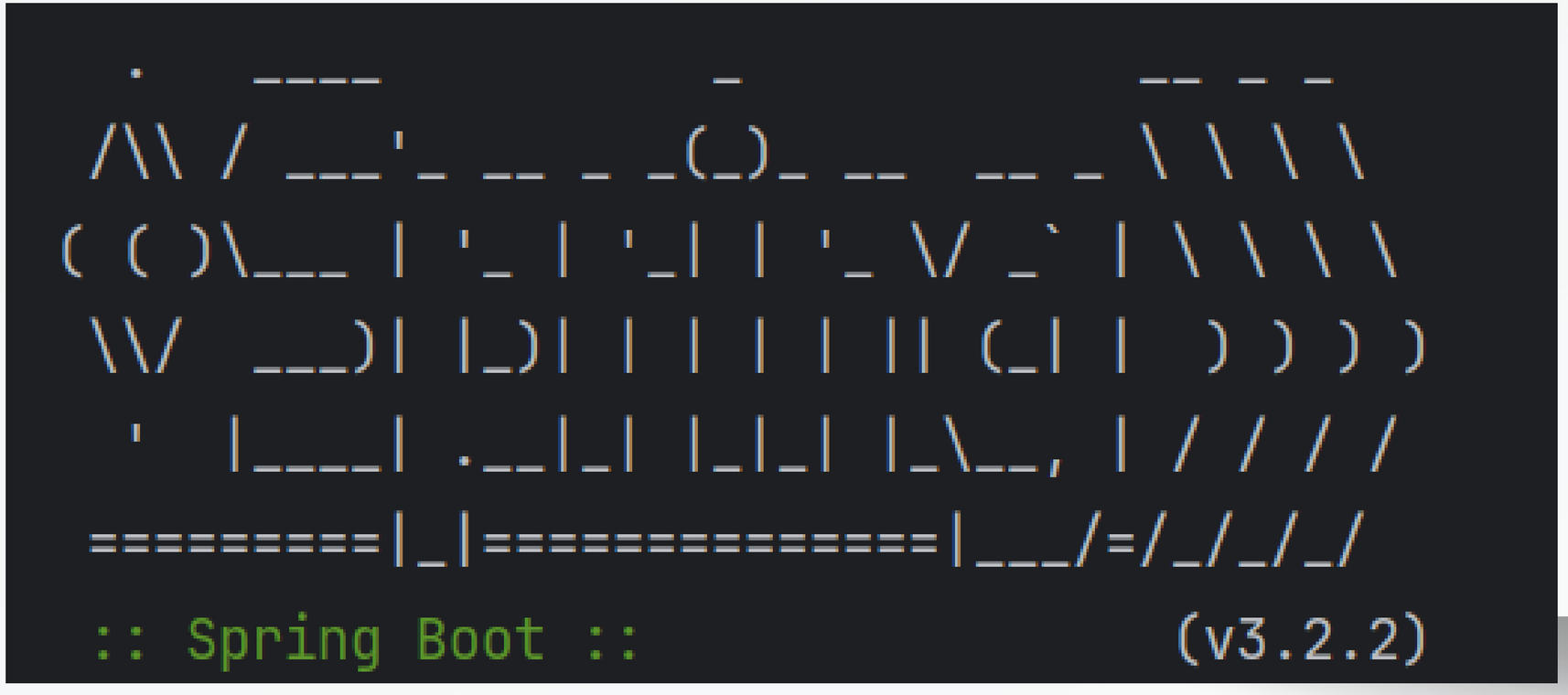
After:



CUSTOM BANNERS IN SPRING BOOT

- Banner is a message displayed on the console when a Spring Boot application starts, helping to customize and create experience for java application.

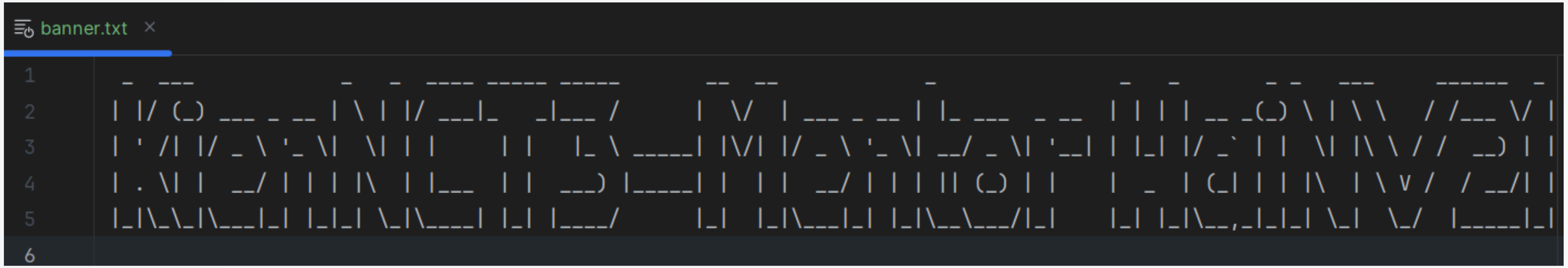
- We can use a custom image or text as a banner for our application when it starts. In Spring Boot, the default banner is:



WAY TO USE CUSTOM BANNES IN SPRING BOOT:

1) Create custom banners:

We can create a custom banner by creating a text or image file with the content we want to display when the application starts. Banner text is usually placed in a file named “banner.txt”, while banner images can be a PNG or JPG image.



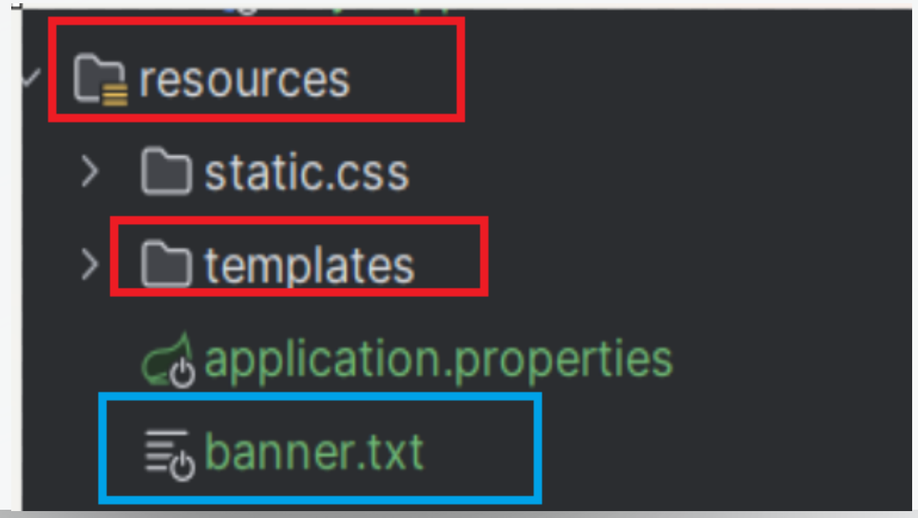
WAY TO USE CUSTOM BANNES IN SPRING BOOT:

1) Create custom banners:

We can create a custom banner by creating a text or image file with the content we want to display when the application starts. Banner text is usually placed in a file named “banner.txt”, while banner images can be a PNG or JPG image.

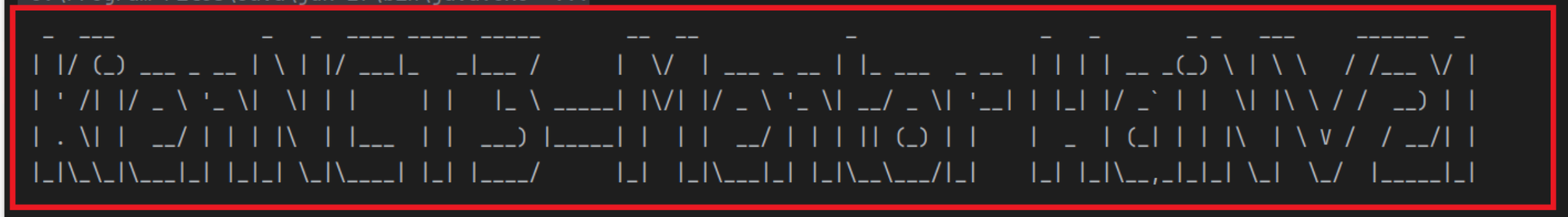
2) Place the banner in the right position:

- In case we create a text banner, place it in the src/main/resources folder. Spring Boot will automatically detect and use it when the application starts.
- For image banners, we can also place it in the src/main/resources folder, and name the file banner.png or banner.jpg.



Here is the result in console:

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
```



```
2024-02-21T14:04:36.408+07:00  INFO 14436 --- [           main] org.kiennguyenfpt.day7.Day7Application   : Starting Day7Application using Java 17.0.9 with PID
2024-02-21T14:04:36.413+07:00  INFO 14436 --- [           main] org.kiennguyenfpt.day7.Day7Application   : No active profile set, falling back to 1 default pr
2024-02-21T14:04:37.126+07:00  INFO 14436 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAU
2024-02-21T14:04:37.155+07:00  INFO 14436 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 14 ms.
2024-02-21T14:04:37.770+07:00  INFO 14436 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
2024-02-21T14:04:37.786+07:00  INFO 14436 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-02-21T14:04:37.787+07:00  INFO 14436 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.18]
2024-02-21T14:04:37.855+07:00  INFO 14436 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2024-02-21T14:04:37.856+07:00  INFO 14436 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization complete
2024-02-21T14:04:38.025+07:00  INFO 14436 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: de
2024-02-21T14:04:38.092+07:00  INFO 14436 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate ORM core version 6.4.1.Final
2024-02-21T14:04:38.126+07:00  INFO 14436 --- [           main] o.h.c.internal.RegionFactoryInitiator    : HHH000026: Second-level cache disabled
2024-02-21T14:04:38.388+07:00  INFO 14436 --- [           main] o.s.o.j.p.SpringPersistenceUnitInfo      : No LoadTimeWeaver setup: ignoring JPA class transfo
2024-02-21T14:04:38.424+07:00  INFO 14436 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2024-02-21T14:04:38.883+07:00  INFO 14436 --- [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection com.mysql.cj.jdbc.C
2024-02-21T14:04:38.888+07:00  INFO 14436 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2024-02-21T14:04:38.971+07:00  WARN 14436 --- [           main] org.hibernate.orm.deprecation            : HHH90000025: MySQL8Dialect does not need to be spec
2024-02-21T14:04:38.974+07:00  WARN 14436 --- [           main] org.hibernate.orm.deprecation            : HHH90000026: MySQL8Dialect has been deprecated; use
```

WAY TO USE CUSTOM BANNES IN SPRING BOOT:

In some cases, we have multiple files banner, and we can force to run specific banner by setting “spring.banner.location” in configure file. For example:

application.yml

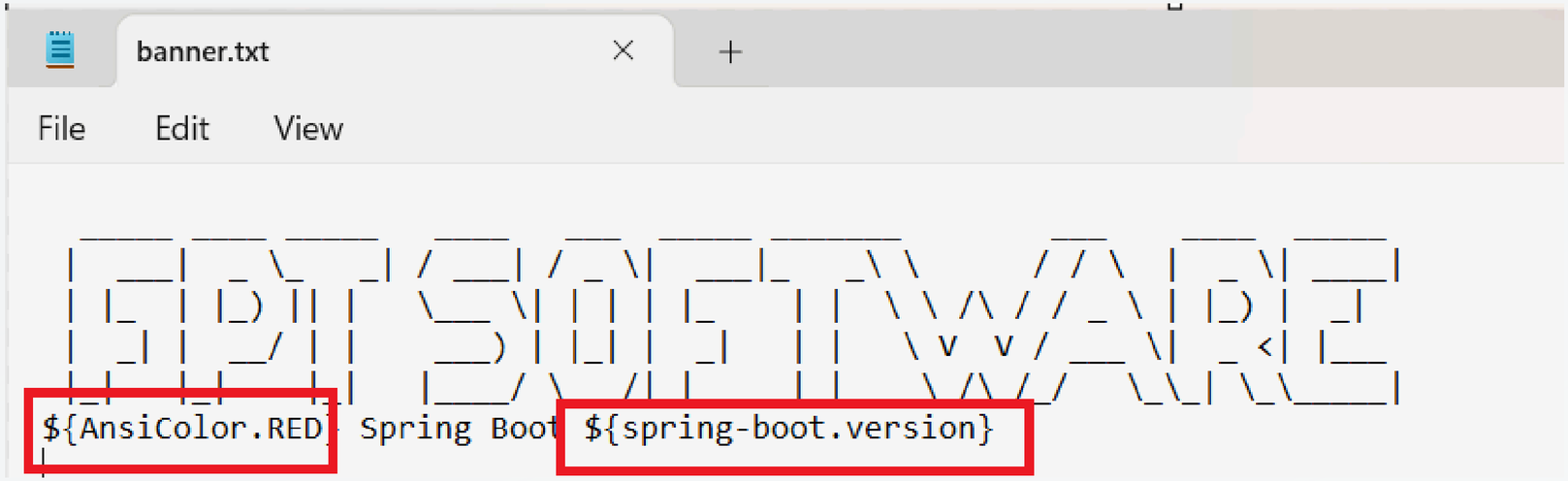
```yaml
spring:
  banner:
    location: banner.txt
```

Result in console:

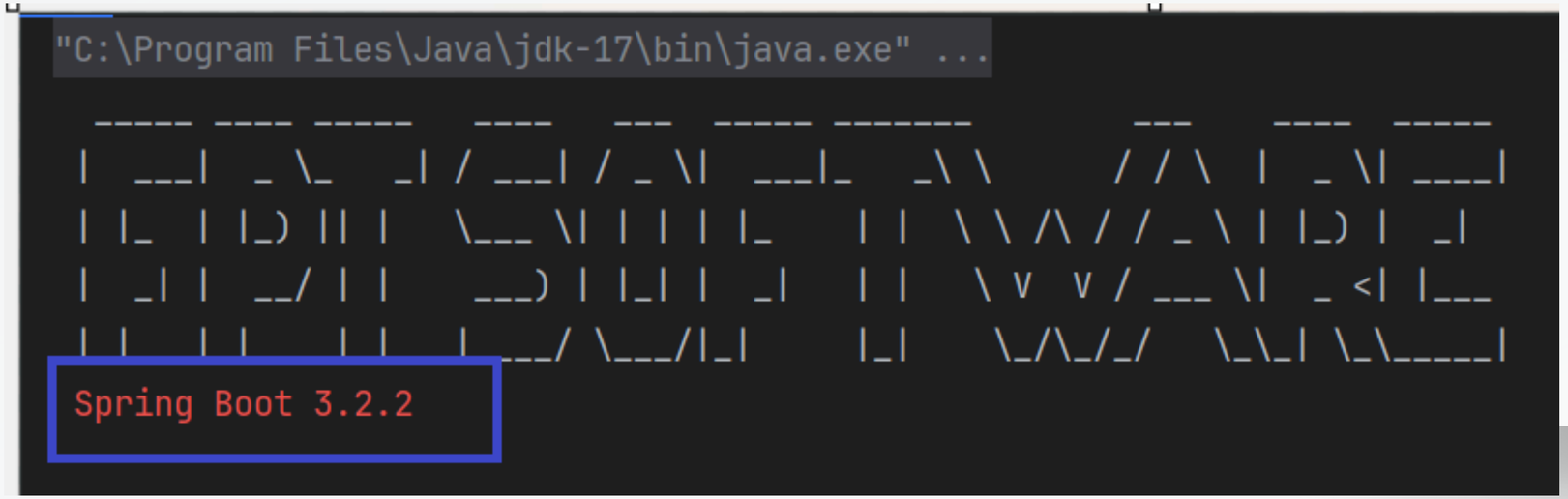
```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
  _____ ____ _____   ____   ___  _____ _______        ___    ____  _____
 |  ___|  _ \_   _| / ___| / _ \|  ___|_   _\ \      / / \  |  _ \| ____|
 | |_  | |_) || |   \___ \| | | | |_    | |  \ \ /\ / / _ \ | |_) |  _|
 |  _| |  __/ | |    ___) | |_| |  _|   | |   \ V  V / ___ \|  _ <| |___
 |_|   |_|    |_|   |____/ \___/|_|     |_|    \_/\_/_/   \_\_| \_\_____|

2024-02-21T14:33:22.823+07:00  INFO 25696 --- [           main] org.kiennguyenfpt.day7.Day7Application   : Starting Day7Application using Java 17.0.9 with PID
2024-02-21T14:33:22.828+07:00  INFO 25696 --- [           main] org.kiennguyenfpt.day7.Day7Application   : No active profile set, falling back to 1 default pro
2024-02-21T14:33:23.508+07:00  INFO 25696 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAUL
2024-02-21T14:33:23.529+07:00  INFO 25696 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 13 ms. F
2024-02-21T14:33:24.041+07:00  INFO 25696 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
2024-02-21T14:33:24.053+07:00  INFO 25696 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-02-21T14:33:24.054+07:00  INFO 25696 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.18]
2024-02-21T14:33:24.130+07:00  INFO 25696 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2024-02-21T14:33:24.131+07:00  INFO 25696 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed
```

USING BANNER VARIABLE: ${spring-boot.version}

This annotation indicates the version of Spring Boot. For example:



Result in console:



Generate banner by SpringApplication.setBanner(...)

We can use SpringApplication.setBanner(…) method to generate a banner in Spring Boot application, like:

```java
@SpringBootApplication
public class Day7Application {
    new *
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(Day7Application.class);
        app.setBanner(((environment, sourceClass, out) -> {
            out.println("BANNER: WELLCOME TO FSOFT!");
        }));
        app.run(args);
    }
```

Generate banner by SpringApplication.setBanner(...)

We can use SpringApplication.setBanner(…) method to programmatically generate a banner in Spring Boot application, like:

```java
@SpringBootApplication
public class Day7Application {
    new *
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(Day7Application.class);
        app.setBanner(((environment, sourceClass, out) -> {
            out.println("BANNER: WELLCOME TO FSOFT!");
        }));
        app.run(args);
    }
```

Result in console:

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
BANNER: WELLCOME TO FSOFT!
2024-02-21T14:51:57.538+07:00  INFO 25300 --- [
2024-02-21T14:51:57.543+07:00  INFO 25300 --- [
2024-02-21T14:51:58.293+07:00  INFO 25300 --- [
2024-02-21T14:51:58.310+07:00  INFO 25300 --- [
2024-02-21T14:51:58.812+07:00  INFO 25300 --- [
2024-02-21T14:51:58.823+07:00  INFO 25300 --- [
2024-02-21T14:51:58.824+07:00  INFO 25300 --- [
2024-02-21T14:51:58.884+07:00  INFO 25300 --- [
```

ABOUT JACKSON OBJECT MAPPER

- In Java, Jackson is a powerful library used to convert between Java objects and JSON format (JSON stands for Javascript Object Notation). Example of JSON format:

```
{
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
}
```

This example defines an employees object containing an array of 3 employee.

ABOUT JACKSON OBJECT MAPPER

- In Java, Jackson is a powerful library used to convert between Java objects and JSON format (JSON stands for Javascript Object Notation).

- ObjectMapper in Jackson is an important component in this process, it is a class of library Jackson. It provides methods to perform “serialize” and “deserialize”:
 + Serialize: It is the process of converting a Java object into JSON format.
 + Deserialize: It is the process of converting JSON into the corresponding Java object.

EXAMPLE OF SERIALIZE:

First, define entity Student

```java
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    1 usage
    private String name;
    1 usage
    private int age;
    //constructor, getter, setter method
```

Second, define Student repository

```java
5 usages   new *
public interface StudentRepository extends JpaRepository<Student, Integer> {}
```

EXAMPLE OF SERIALIZE:

Third, register Bean CommandLineRunner to add students to database:

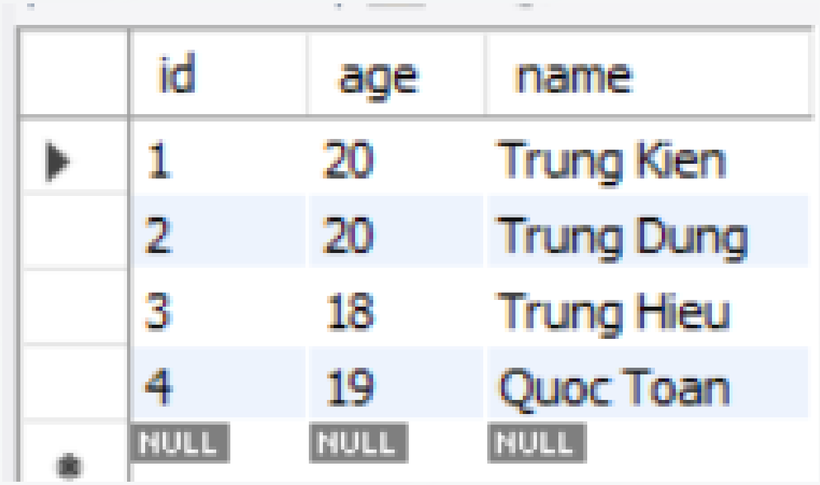
```
@Bean
public CommandLineRunner demo(StudentRepository rep) {
    return(args) -> {
        rep.save(new Student( id: 1,  name: "Trung Kien",  age: 20));
        rep.save(new Student( id: 2,  name: "Trung Dung",  age: 20));
        rep.save(new Student( id: 3,  name: "Trung Hieu",  age: 18));
        rep.save(new Student( id: 4,  name: "Quoc Toan",  age: 19));
    };
}
```

EXAMPLE OF SERIALIZE:

Third, register Bean CommandLineRunner to add students to database:

```java
@Bean
public CommandLineRunner demo(StudentRepository rep) {
    return(args) -> {
        rep.save(new Student( id: 1,  name: "Trung Kien",  age: 20));
        rep.save(new Student( id: 2,  name: "Trung Dung",  age: 20));
        rep.save(new Student( id: 3,  name: "Trung Hieu",  age: 18));
        rep.save(new Student( id: 4,  name: "Quoc Toan",  age: 19));
    };
}
```

Records in MySQL:



	id	age	name
▶	1	20	Trung Kien
	2	20	Trung Dung
	3	18	Trung Hieu
	4	19	Quoc Toan
*	NULL	NULL	NULL

EXAMPLE OF SERIALIZE:

Finally, in @RestController class:

```
@RestController
@RequestMapping("/api")
public class StudentController {
    4 usages
    private final StudentRepository rep;
    new *
    @Autowired
    public StudentController(StudentRepository rep) {
        this.rep = rep;
    }
    new *
    @GetMapping("getStudentsJson")
    public String getStudent() throws Exception {
        ObjectMapper objectMapper = new ObjectMapper();
        Iterable<Student> students = rep.findAll();
        return objectMapper.writeValueAsString(students);
    }
```

EXAMPLE OF SERIALIZE:

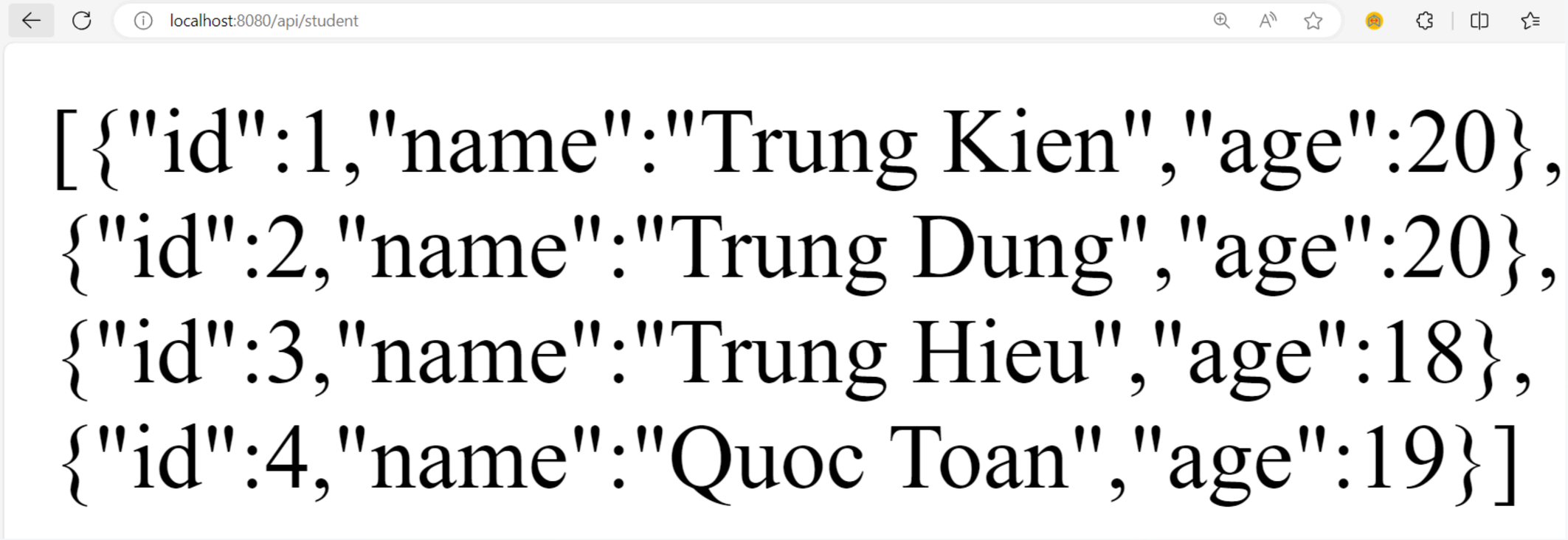
```java
@RestController
@RequestMapping("/api")
public class StudentController {
    4 usages
    private final StudentRepository rep;
    new *
    @Autowired
    public StudentController(StudentRepository rep) {
        this.rep = rep;
    }
    new *
    @GetMapping("getStudentsJson")
    public String getStudent() throws Exception {
        ObjectMapper objectMapper = new ObjectMapper();
        Iterable<Student> students = rep.findAll();
        return objectMapper.writeValueAsString(students);
    }
```

1) Iterable<Student> students = rep.findAll(): Fetch all the students from the database.

2) writeValueAsString() method is used to change students object to JSON string.

EXAMPLE OF SERIALIZE:

The result on localhost:



EXAMPLE OF DESERIALIZE:

Same with serialize example, we have Student class, StudentRepository interface

```
@GetMapping("/getStudentsObject")
public List<Student> jsonToObject() {
    Iterable<Student> iterable = rep.findAll();
    List<Student> listStudent = new ArrayList<>();
    ObjectMapper objectMapper = new ObjectMapper();
    iterable.forEach(student -> {
        try {
            // Convert Student object to JSON string
            String json = objectMapper.writeValueAsString(student);

            // Convert JSON string to Student object
            Student studentObject = objectMapper.readValue(json, Student.class);
            listStudent.add(studentObject);
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
    return listStudent;
}
```

EXAMPLE OF DESERIALIZE:

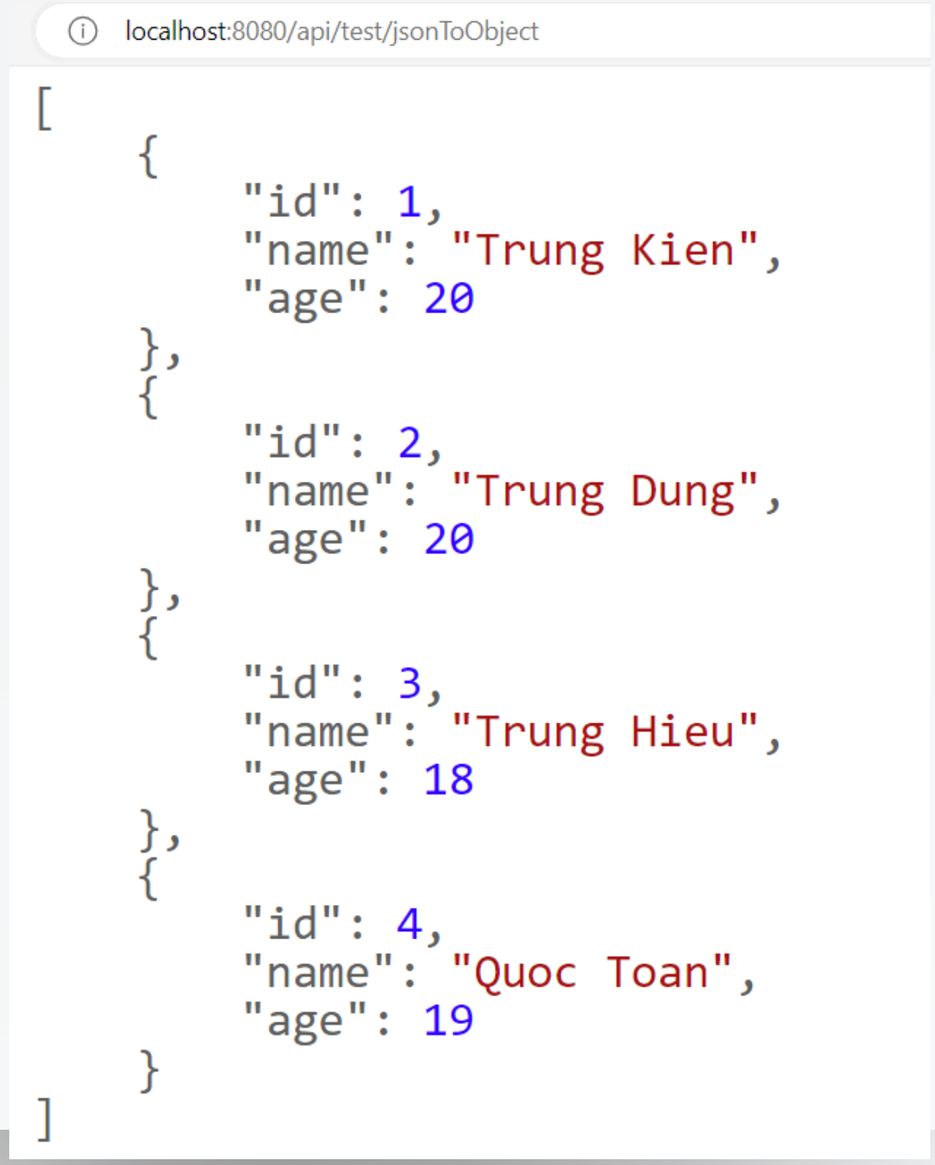
```
// Convert JSON string to Student object
Student studentObject = objectMapper.readValue(json, Student.class);
```

- readValue() is a method of the ObjectMapper class used to convert an object from JSON data to a Java object.

- json is a string containing JSON data that needs to be converted into a Java object. Student.class is the second argument to the readValue() method, indicating that the resulting object will be converted to an object of the Student class.
- Jackson will use the information from the JSON data to match the fields and methods of the Student class.

EXAMPLE OF DESERIALIZE:

The result on localhost:



CUSTOMIZE JACKSON OBJECTMAPPER

Customizing the Jackson ObjectMapper in a Spring Boot application is important to control how data is converted into JSON format and vice versa. Quick example:

```
5 usages   new *
public class Shoes {
    2 usages
    private String name;
    2 usages
    private String brand;
    2 usages
    private LocalDateTime date;
```

```
@GetMapping("/shoes")
public Shoes getShoes()  {
    Shoes s = new Shoes();
    s.setName("Rosh One");
    s.setBrand("Nike");
    s.setDate(LocalDateTime.now());
    return s;
}
```

CUSTOMIZE JACKSON OBJECTMAPPER

The result of example on localhost:

```java
5 usages  new *
public class Shoes {
    2 usages
    private String name;
    2 usages
    private String brand;
    2 usages
    private LocalDateTime date;
```

localhost:8080/api/shoes


```
{
    "name": "Rosh One",
    "brand": "Nike",
    "date": "2024-02-21T23:45:42.0750684"
}
```


Problem: Target is set date follow format: dd/MM/yyyy like: 21/02/2024.
Below are the way to customize Jackson ObjectMapper

Use interface “Jackson2ObjectMapperBuilderCustomizer”

```
5 usages   new *
public class Shoes {
    2 usages
    private String name;
    2 usages
    private String brand;
    2 usages
    private LocalDateTime date;
```

```
@Configuration
public class FilterConfig {
    new *
    @Bean
    public Jackson2ObjectMapperBuilderCustomizer jsonCustomizer() {
        return builder -> {
            builder.serializerByType(LocalDateTime.class, new LocalDateTimeSerializer(
                    DateTimeFormatter.ofPattern("yyyy-MM-dd")));
        };
    }
```

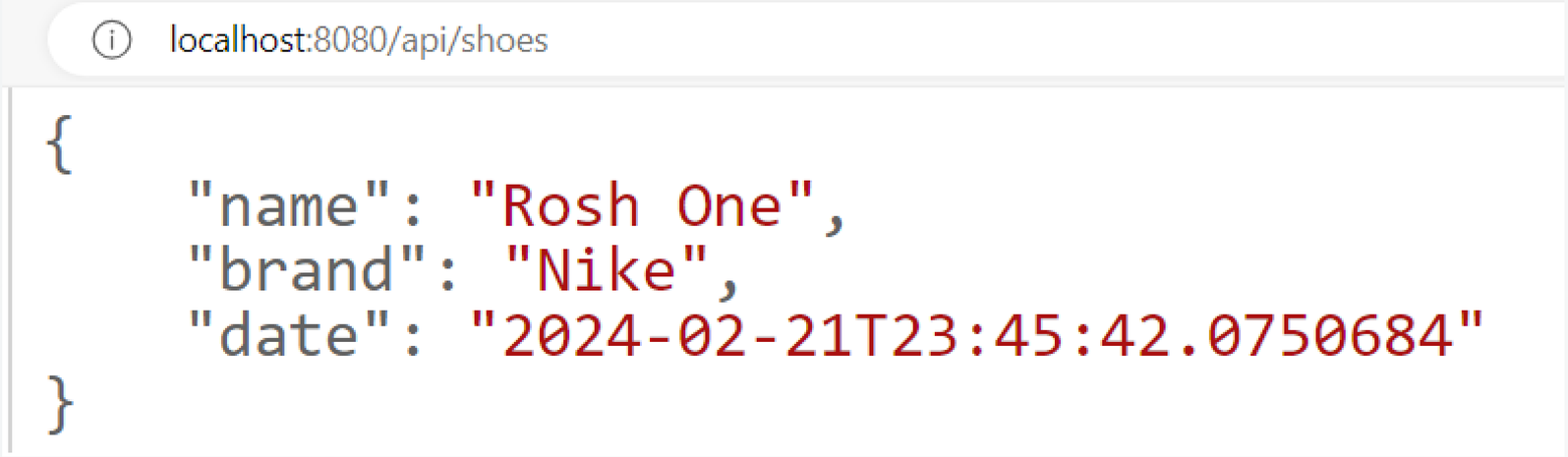
In this case, we use “builde”r to add a custom serializer for the LocalDateTime data type. Custom Serializers are created by LocalDateTimeSerializer, to which can provide configurations such as datetime format ("yyyy-MM-dd") via DateTimeFormatter.

With this configuration, ObjectMapper will use a custom serializer to convert the LocalDateTime object to a string format in the format "yyyy-MM-dd" when serializing data

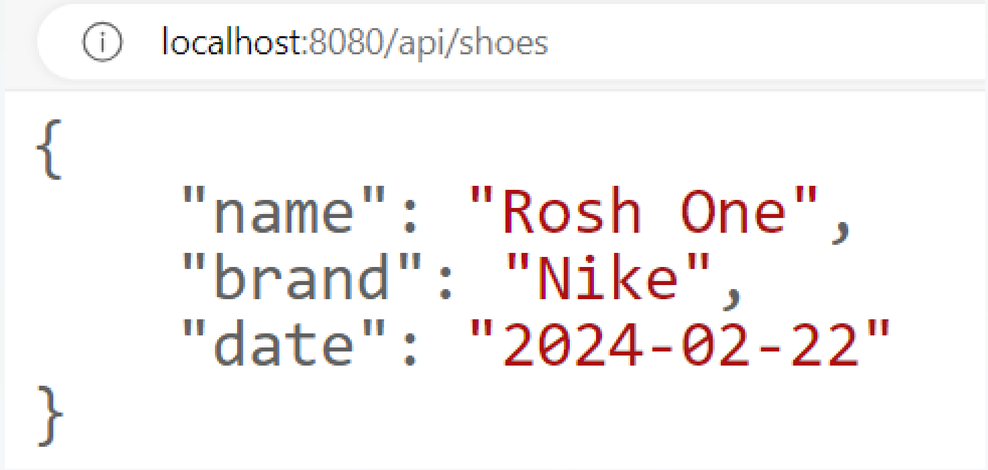
CUSTOMIZE JACKSON OBJECTMAPPER

The result on localhost:

Before:



After:



TESTING CONFIGURATION:

Problem: check if the "/api/shoes" API returns properly formatted data or not

```
@Autowired
private MockMvc mockMvc;
new *
@Test
public void testDateFormat() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/shoes"))
            .andExpect(MockMvcResultMatchers.content().json( jsonContent: "{\n" +
                    "    \"name\": \"Rosh One\",\n" +
                    "    \"brand\": \"Nike\",\n" +
                    "    \"date\": \"2024-02-22\"\n" +
                    "}"));
}
```

-MockMvc is a tool that allows to test Spring MVC APIs

-mockMvc.perform(MockMvcRequestBuilders.get("/api/shoes")): Here, we use mockMvc to make an HTTP GET request to the address “/api/shoes”

- .andExpect(MockMvcResultMatchers.content().json("{...}")): With .andExpect(), we check that the content of the response is a JSON fragment with the specified properties in the JSON string.

TESTING CONFIGURATION:

```java
@Test
public void testDateFormat() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/shoes"))
            .andExpect(MockMvcResultMatchers.content().json( jsonContent: "{\n" +
                    "    \"name\": \"Rosh One\",\n" +
                    "    \"brand\": \"Nike\",\n" +
                    "    \"date\": \"2024-02-22\"\n" +
                    "}"));
}
```

localhost:8080/api/shoes


```
{
    "name": "Rosh One",
    "brand": "Nike",
    "date": "2024-02-22"
}
```


The result:

```
✔ Tests passed: 1 of 1 test – 326 ms
"C:\Program Files\Java\jdk-
01:04:06.351 [main] INFO or
01:04:06.483 [main] INFO or
```

GREATFUL THANK YOU FOR YOUR LISTENING!