

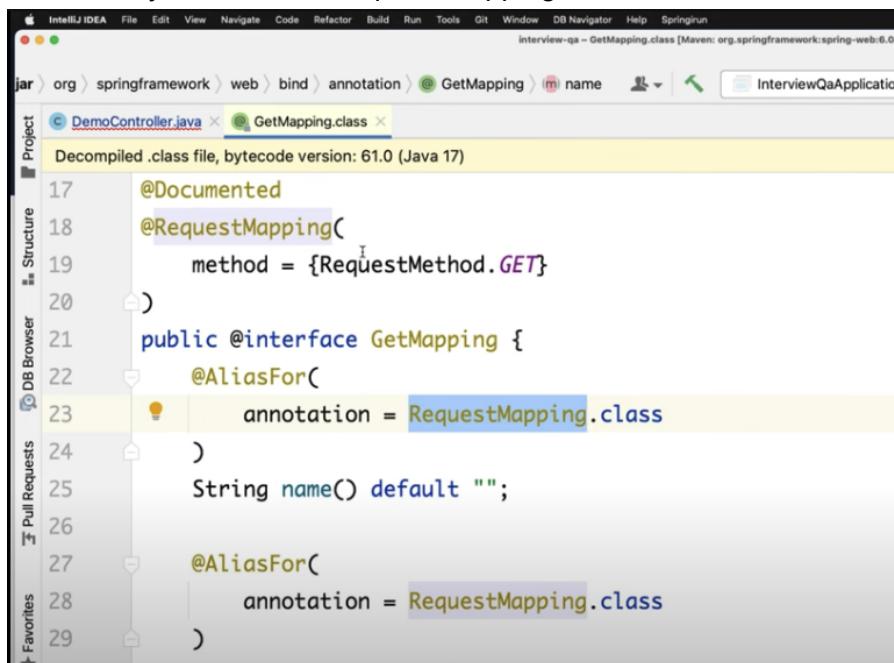
31. Have you worked on Restful webservices ? If yes What all HTTP methods have you used in your project ?

- POST CREATE
- PUT UPDATE
- GET RETRIEVE
- PATCH PARTIAL UPDATE
- DELETE REMOVE

32. How can you specify the HTTP method type for your REST endpoint?

```
@GetMapping("/users")
@PostMapping
@PutMapping
@PatchMapping
@DeleteMapping
@RequestMapping(value = "/users",method = RequestMethod.POST)
public ResponseEntity<?> doSomeOperation(@RequestBody Object input){  
}
```

RequestMapping is the parent annotation of all HTTP method annotations. Alternatively we can use Request mapping as shown above.



```
17     @Documented
18     @RequestMapping(
19         method = {RequestMethod.GET}
20     )
21     public @interface GetMapping {
22         @AliasFor(
23             annotation = RequestMapping.class
24         )
25         String name() default "";
26
27         @AliasFor(
28             annotation = RequestMapping.class
29         )
```

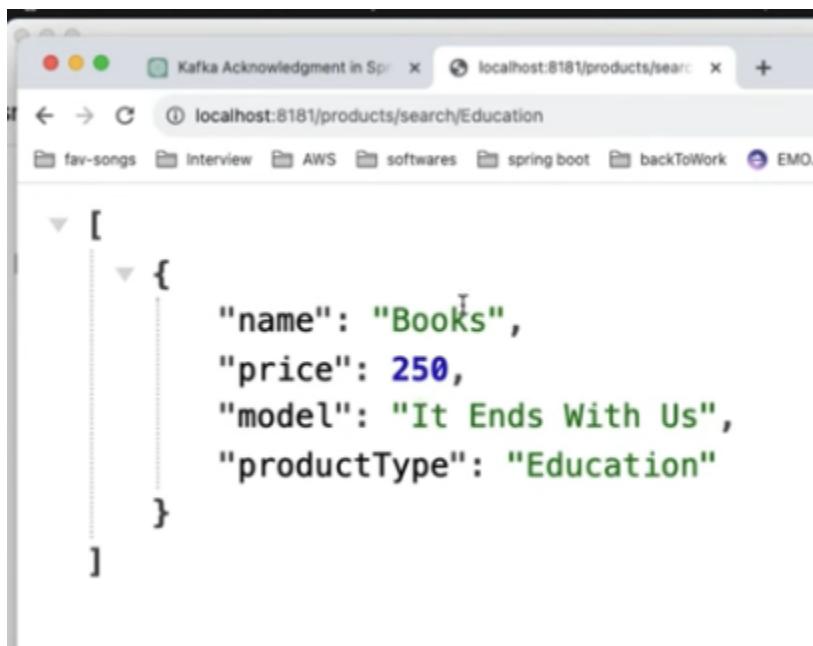
33. Scenario :

Can you design a rest endpoint , Assume that you have a Product database, and your task is to create an API to filter a list of products by productType.



The screenshot shows a Java code editor with several tabs at the top: DemoController.java, ProductController.java (selected), ProductService.java, and GetMapping.class. The code in ProductController.java is as follows:

```
14  @RestController
15  @RequestMapping("/products")
16  public class ProductController {
17
18      @Autowired
19      private ProductService service;
20
21      @GetMapping("/search/{productType}")
22      public ResponseEntity<?> getProducts(@PathVariable String productType){
23          List<Product> products = service.getProductByType(productType);
24          return ResponseEntity.ok(products);
25      }
26  }
```



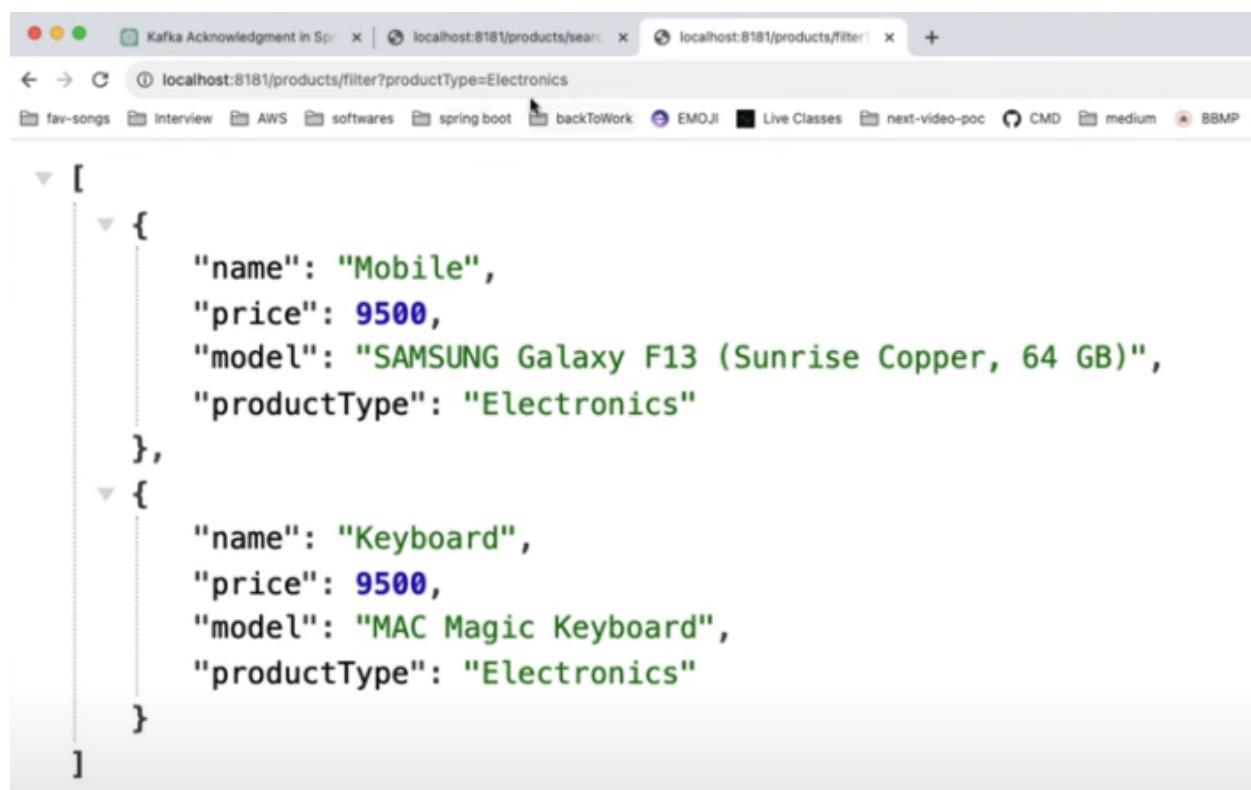
The screenshot shows a browser window with the URL `localhost:8181/products/search/Education`. The response is a JSON object:

```
[{"name": "Books", "price": 250, "model": "It Ends With Us", "productType": "Education"}]
```

34. Scenario

Design endpoints in a way that takes "productType" as input. If the user provides this input, the endpoint should filter products based on the specified condition. If "productType" is not provided, the endpoint should return all the products.

```
23
24     @GetMapping("/filter")
25     public ResponseEntity<?> findProducts(@RequestParam(value = "productType", required = false) String productType)
26         List<Product> productList = productType != null
27             ? service.getProductByType(productType)
28             : service.getProducts();
29         return ResponseEntity.ok(productList);
30     }
31 }
```



The screenshot shows a browser window with three tabs open:

- Kafka Acknowledgment in Spring Boot
- localhost:8181/products/search
- localhost:8181/products/filter?productType=Electronics

The third tab displays the JSON response:

```
[{"name": "Mobile", "price": 9500, "model": "SAMSUNG Galaxy F13 (Sunrise Copper, 64 GB)", "productType": "Electronics"}, {"name": "Keyboard", "price": 9500, "model": "MAC Magic Keyboard", "productType": "Electronics"}]
```

```

[{"name": "Mobile", "price": 9500, "model": "SAMSUNG Galaxy F13 (Sunrise Copper, 64 GB)", "productType": "Electronics"}, {"name": "Keyboard", "price": 9500, "model": "MAC Magic Keyboard", "productType": "Electronics"}, {"name": "Books", "price": 250, "model": "It Ends With Us", "productType": "Education"}, {"name": "Remote Control Toys", "price": 699, "model": "Wembley High Speed Mini 1:24 Scale Rechargeable Remote Control car with Lithium Battery", "productType": "Baby&Kids"}]

```

There is some option like defaultValue if not provide anything it will fetch this product type only.

```

    . findProducts(@RequestParam(value = "productType", required = false, defaultValue = "Electronics"))
    :tList = productType != null

```

35. What is the difference between @PathVariable & @RequestParam

If the path variable is missing we get - 404. Incase of request param it is optional.

```

18  @GetMapping("/search/{productType}")//404
19  public ResponseEntity<?> getProducts(@PathVariable String productType){
20      List<Product> products = service.getProductByType(productType);
21      return ResponseEntity.ok(products);
22  }
23
24  @GetMapping("/filter")
25  public ResponseEntity<?> findProducts(@RequestParam(value = "productType", required = false) String product
26      List<Product> productList = productType != null

```

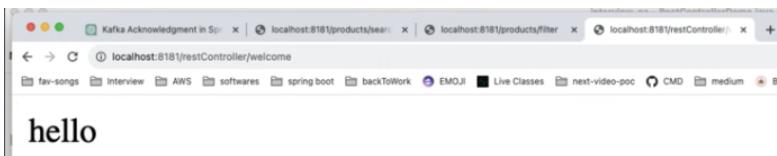
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

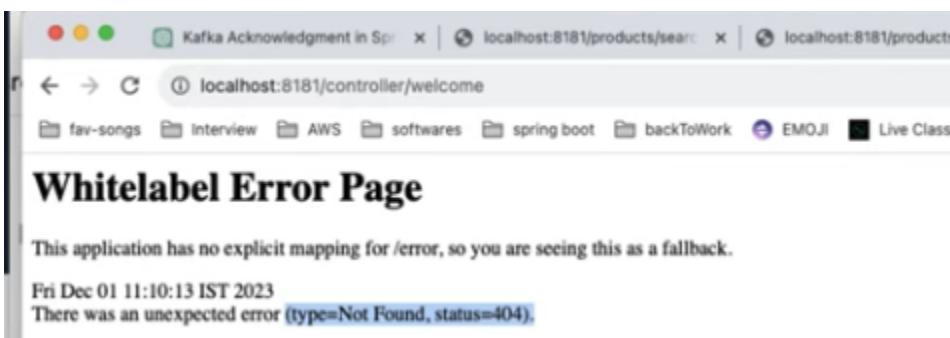
Fri Dec 01 10:57:16 IST 2023
There was an unexpected error (type=Not Found, status=404).

36. Why did you use `@RestController` why not `@Controller` ?

```
7  @RestController
8  public class RestControllerDemo {
9
10     @GetMapping("/restController/welcome")
11     public String welcome(){
12         return "hello";
13     }
14 }
```



```
6  @Controller
7  public class ControllerDemo {
8
9      @GetMapping("/controller/welcome")
10     public String welcome(){
11         return "hello";
12     }
13 }
```



In case of controller it spring expect a page jsp/html page with name hello.jsp
Controller works like a model and view. How this works is explained below.

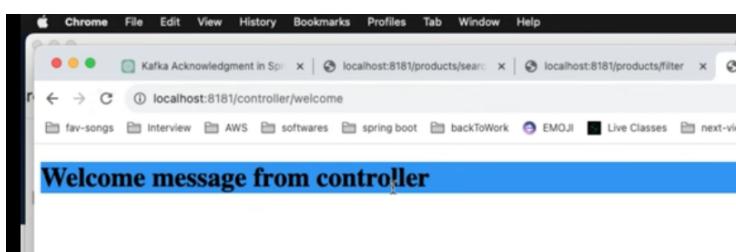
```
5 @Controller
6
7 public class ControllerDemo {
8
9     @GetMapping("/controller/welcome")
10    //modelAndView
```

Add below dependency

```
42 <artifactId>mysql-connector-j</artifactId>
43 <scope>runtime</scope>
44 </dependency>
45 <dependency>
46     <groupId>org.springframework.boot</groupId>
47     <artifactId>spring-boot-starter-thymeleaf</artifactId>
48 </dependency>
49 <dependency>
50     <groupId>org.springdoc</groupId>
```

Now add hello.html

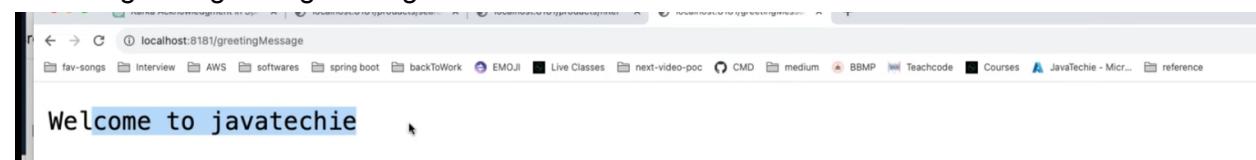
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>page loading</title>
</head>
<body>
    <h1 style="...>Welcome message from controller</h1>
</body>
</html>
```



How can we make the controller work like a rest controller? We have to explicitly add mediatype and response body as shown below.

```
1  @GetMapping(value = "/message", produces = MediaType.TEXT_PLAIN_VALUE)
2  @ResponseBody
3  public String message(){
4      return "Welcome to javatechie";
5  }
```

He changed to greetingMessage



RestController = Controller + ResponseBody

37. How can we deserialize a JSON request payload into an object within a Spring MVC controller ?

```
none form-data x-www-form-urlencoded
1 {
2     "bookId": 123,
3     "title": "Spring in Action",
4     "publicationYear": 2022,
5     "authors": [
6         {
7             "authorId": 1,
8             "name": "Craig Walls",
9             "birthYear": 1978
10        },
11        {
12            "authorId": 2,
13            "name": "John Doe",
```

A screenshot of an IDE showing the code for Book.java. The code defines a class Book with private fields bookId, title, publicationYear, and authors, which is a List of Author objects. The code includes annotations @Data, @AllArgsConstructor, and @NoArgsConstructor.



```
hello.html x pom.xml (interview-qa) x RestControllerD
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 @Data
8 @AllArgsConstructor
9 @NoArgsConstructor
10 public class Author {
11     private long authorId;
12     private String name;
13     private int birthYear;
14 }
```

Keys in the request and variable names must match. This is how we pass it to endpoint. Shown below.



```
13
14
15 @PostMapping("/books")
16     public String processBook(@RequestBody Book book){
17         return book.getTitle()+" New book has been published on year "+book.getPubli
18     }
```

38. Can we perform update operation in POST http method if yes then why do we need Put Mapping or put http method?



```
40
41     @PostMapping("/users")
42     public UserDetails addNewUser(@RequestBody UserDetails userDetails){
43         return repository.save(userDetails);
44     }
45
46     @PutMapping("/users/{id}")
47     public UserDetails addAndUpdateUser(@PathVariable int id,@RequestBody UserDetails userDetails){
48         UserDetails existingUserDetails = repository.findById(id).get();
49         existingUserDetails.setName(userDetails.getName());
50         existingUserDetails.setAge(userDetails.getAge());
51         return repository.save(existingUserDetails);
    }
```

Post is to create a record whereas put is to update an existing record.

Still we can do updates in the post mapping but it is a violation in restful design. In rest each http method has specific purpose and semantic meaning

Post method should not be idempotent - create new resource for each hit.

HTTP <http://localhost:8181/users>

POST <http://localhost:8181/users>

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

Body

```
1 {  
2   "name": "peter",  
3   "age": 36  
4 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

200 OK 22 ms 196 B Save as example

userdetails 1

SELECT * FROM UserDetails ud | Enter a SQL expression to filter result

	id	age	name
1	1	36	peter
2	2	36	peter
3	3	36	peter
4	4	36	peter
5	5	36	peter

and put method should be idempotent -

Hit below endpoint n point of time but it should not increase record count in the db.

POSTMAN Screenshot:

- Method: PUT
- URL: http://localhost:8181/users/5
- Body (JSON):


```

1 {
2   "name": "peter",
3   "age": 36
4 }
```
- Response Status: 200 OK

39. Can we pass Request Body in GET HTTP Method ?

We can pass technically but it is not recommended.

```

32
33     @GetMapping("/users/name")
34     public String checkBodyInGET(@RequestBody UserDetails userDetails)
35         return userDetails.getName();
36     }
37
  
```

POSTMAN Screenshot:

- Method: GET
- URL: http://localhost:8181/users/name
- Body (Text):


```

1 {
2   "name": "peter",
3   "age": 36
4 }
```
- Response Status: 200 OK
- Result: peter

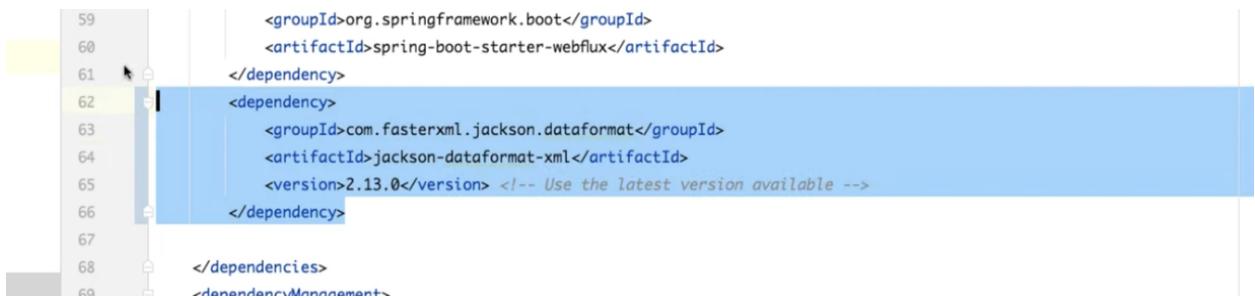
40. How can we perform content negotiation (XML/JSON) in Rest endpoint ?

```

32
33     @GetMapping(produces = {"application/json", "application/xml"})
34     public List<Product> products(@RequestParam(value = "productType", required = false) String productType)
35     {
36         return productType != null
37             ? service.getProductByType(productType)
38             : service.getProducts();
39     }

```

We need this one.



```

59     <groupId>org.springframework.boot</groupId>
60     <artifactId>spring-boot-starter-webflux</artifactId>
61   </dependency>
62   <dependency>
63     <groupId>com.fasterxml.jackson.dataformat</groupId>
64     <artifactId>jackson-dataformat-xml</artifactId>
65     <version>2.13.0</version> <!-- Use the latest version available -->
66   </dependency>
67
68 </dependencies>
69 <dependencyManagement>

```

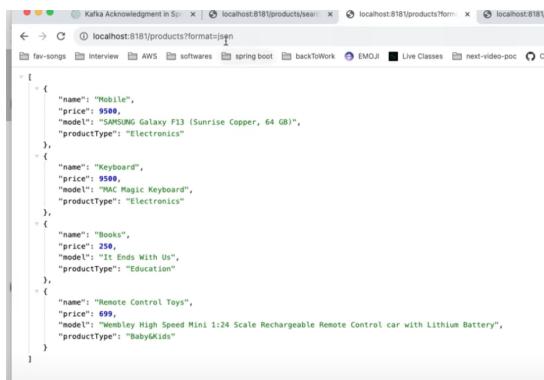
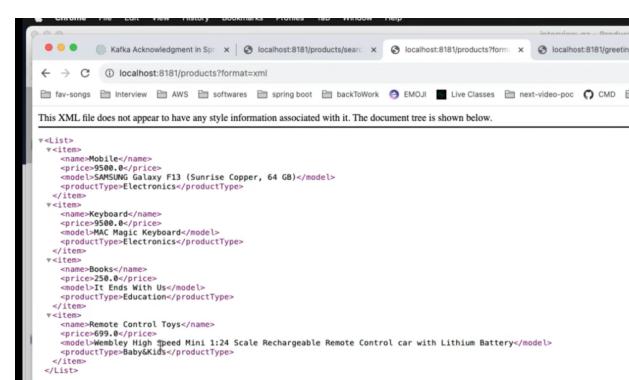


```

24     # Enable content negotiation
25     spring.mvc.contentnegotiation.favor-path-extension=true
26     spring.mvc.contentnegotiation.favor-parameter=true
27     spring.mvc.contentnegotiation.defaultContentType=application/xml
28

```

By default it will give xml if we not pass content type

The JSON response from `localhost:8181/products?format=json`:

```

[{"name": "Mobile", "price": 9999, "model": "SAMSUNG Galaxy F13 (Sunrise Copper, 64 GB)", "productType": "Electronics"}, {"name": "Keyboard", "price": 9999, "model": "MAC Magic Keyboard", "productType": "Electronics"}, {"name": "Books", "price": 250, "model": "It Ends With Us", "productType": "Education"}, {"name": "Remote Control Toys", "price": 699, "model": "Wembley High Speed Mini 1:24 Scale Rechargeable Remote Control car with Lithium Battery", "productType": "Baby&Kids"}]

```

The XML response from `localhost:8181/products?format=xml`:

```

<List>
<item>
<name>Mobile</name>
<price>9999</price>
<model>SAMSUNG Galaxy F13 (Sunrise Copper, 64 GB)</model>
<productType>Electronics</productType>
</item>
<item>
<name>Keyboard</name>
<price>9999</price>
<model>MAC Magic Keyboard</model>
<productType>Electronics</productType>
</item>
<item>
<name>Books</name>
<price>250</price>
<model>It Ends With Us</model>
<productType>Education</productType>
</item>
<item>
<name>Remote Control Toys</name>
<price>699</price>
<model>Wembley High Speed Mini 1:24 Scale Rechargeable Remote Control car with Lithium Battery</model>
<productType>Baby&Kids</productType>
</item>
</List>

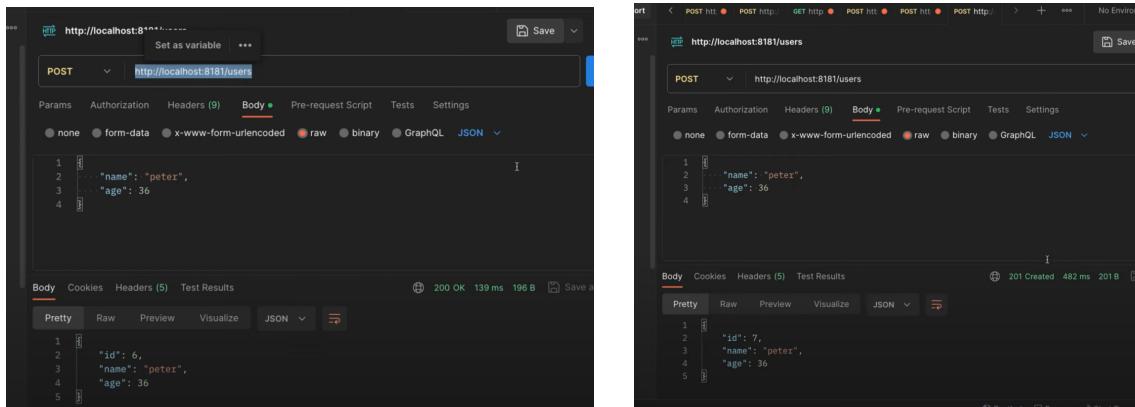
```

41. What all status code you have observed in your application ?

400 - bad request - some wrong input
 404 - resource (url) not found
 401 - authentication error
 403 - authentication successful but not authorized that resource (url)
 405 - method not allowed - get is accessed using post method like that
 415 - unsupported media error (content type is not correct)
 500 - Internal server error - server encountered unexpected condition that prevented fulfilling the request
 502 - BadGateway. Accessing other endpoints which is unhealthy/down

42. How can you customize the status code for your endpoint ?

Previously the below post method returning 200 after change 201



Before and After.

```

@PostMapping("/users") // not idempotent
@ResponseStatus(HttpStatus.CREATED)
public UserDetails addNewUser(@RequestBody UserI
    return repository.save(userDetails);
}
  
```

Similarly for put we can use no content

```

45
46     @PutMapping("/users/{id}") // idempotent
47     @ResponseStatus(HttpStatus.NO_CONTENT)
48     public UserDetails addAndUpdateUser(@PathVariable ir
49                                         UserDetails existingUserDetails = repository.fin

```

43. How can you enable cross origin ?

This will accept all the endpoints from the given origins only

```

16     @RestController
17     @CrossOrigin(origins = "http://localhost:4000")
18     public class DemoController {

```

The above one is annotation config the below one is java config

```

6 import org.springframework.web.servlet.config.annotation.EnableWebMvc;
7 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
8
9 @Configuration
10 @EnableWebMvc
11 public class CorsConfig implements WebMvcConfigurer {
12
13     @Override
14     public void addCorsMappings(CorsRegistry registry) {
15         registry.addMapping(pathPattern: "/api/**")
16             .allowedOrigins("http://localhost:3000") // Add your front-end application's origin
17             .allowedMethods("GET", "POST", "PUT", "DELETE")
18             .allowedHeaders("Origin", "Content-Type", "Accept", "Authorization")
19             .allowCredentials(true)
20             .maxAge(3600);
21     }
22 }

```

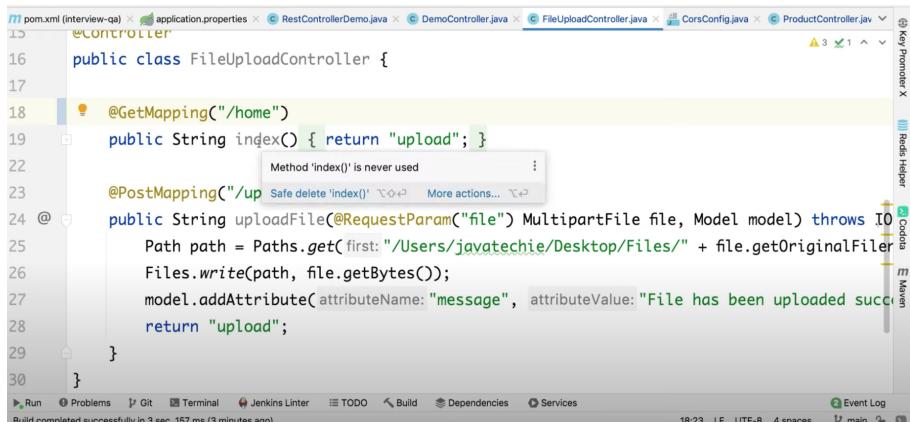
We can use wildcards as shown below

```

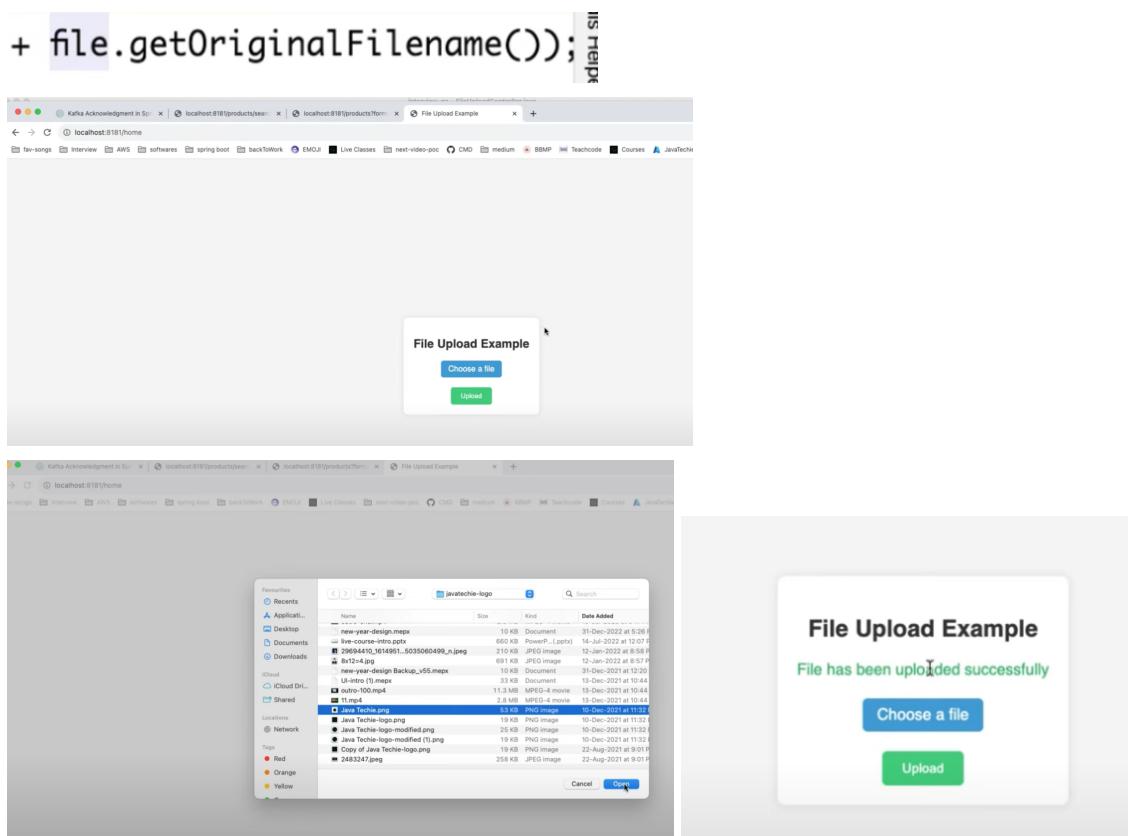
16     @RestController
17     @CrossOrigin(origins = "*")
18     public class DemoController {
19
20         @GetMapping("/users/{id}")
21         .allowedOrigins("*") // Add your front-end application's origin
22         .allowedMethods("GET", "POST", "PUT", "DELETE")

```

44. How can you upload a file in spring ?



```
pom.xml (interview-q-a) × application.properties ×RestController × DemoController.java × FileUploadController.java × CorsConfig.java × ProductController.java ×  
16 public class FileUploadController {  
17  
18     @GetMapping("/home")  
19     public String index() { return "upload"; }  
20  
21     @PostMapping("/up")  
22     public String uploadFile(@RequestParam("file") MultipartFile file, Model model) throws IOException {  
23         Path path = Paths.get(first: "/Users/javatechie/Desktop/Files/" + file.getOriginalFilename());  
24         Files.write(path, file.getBytes());  
25         model.addAttribute(attributeName: "message", attributeValue: "File has been uploaded successfully");  
26         return "upload";  
27     }  
28 }  
Run Problems JGit Terminal Jenkins Linter TODO Build Dependencies Services Event Log  
Build completed successfully in 3 sec, 157 ms (3 minutes ago)  
18:23 LF UTF-8 4 spaces main
```



+ file.getOriginalFilename());

localhost:8181/products/home

localhost:8181/productsForm

localhost:8181/products

localhost:8181/productsExample

File Upload Example

Choose a file

Upload

localhost:8181/home

javatechie-logo

Name Size Kind Date Added

- Java Techie.png 19 KB PNG image 10-Dec-2021 at 11:22
- Java Techie-logo-modified.png 25 KB PNG image 10-Dec-2021 at 11:22
- Java Techie-logo-modified1.png 19 KB PNG image 10-Dec-2021 at 11:22
- Copy of Java Techie-logo.png 19 KB PNG image 22-Aug-2021 at 9:01 PM
- 2483247.jpg 258 KB JPEG image 22-Aug-2021 at 9:01 PM

Cancel Open

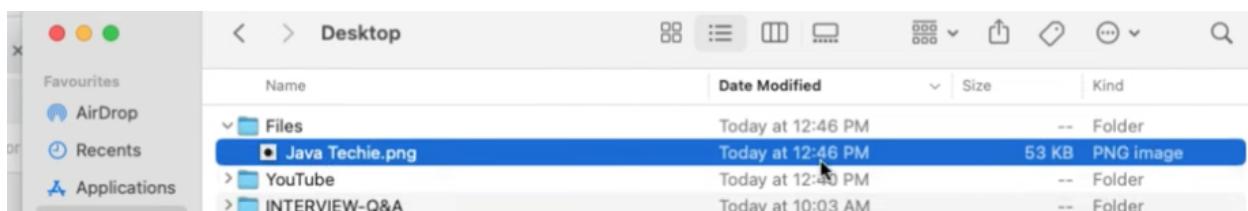
File Upload Example

File has been uploaded successfully

Choose a file

Upload

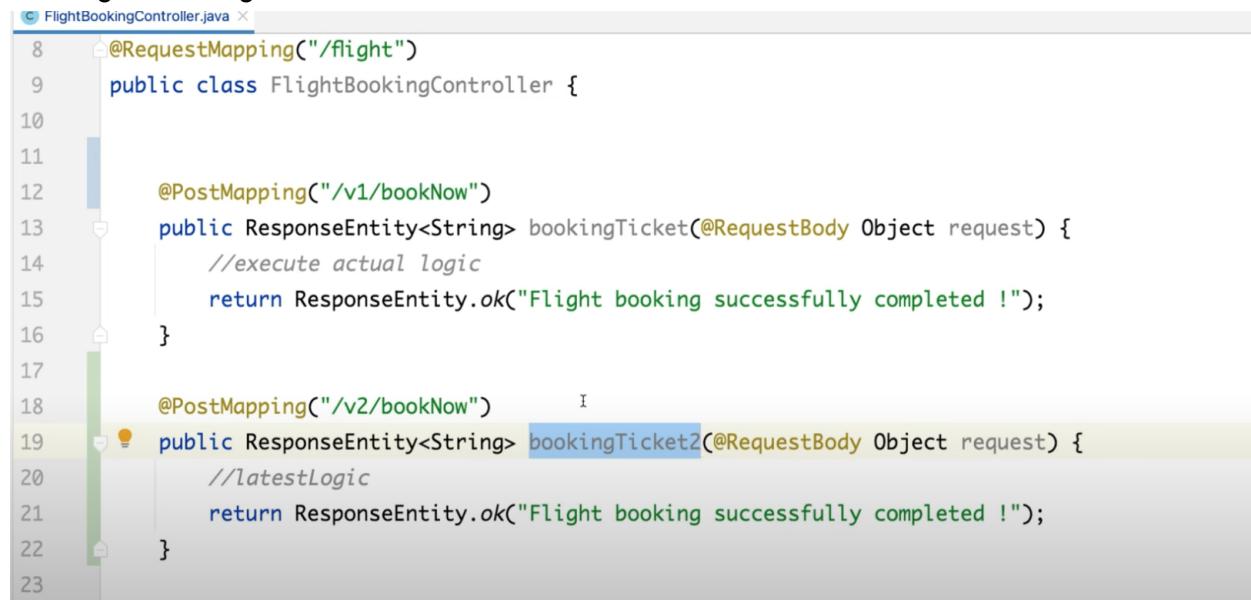
The location where the file is stored.



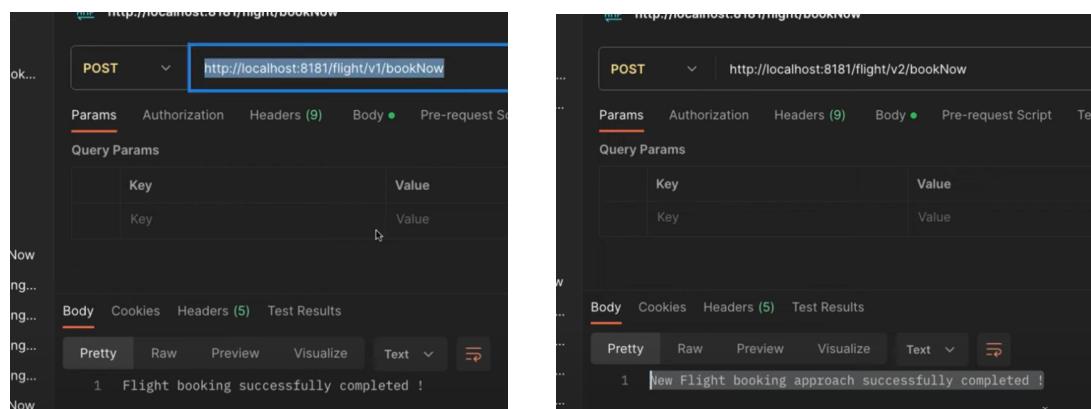
45. How do you maintain versioning for your REST API?

For backward compatibility and to add new features or changes we have to go for versioning our rest api.

1. Using versioning - like v1 and v2



```
8  @RequestMapping("/flight")
9  public class FlightBookingController {
10
11
12     @PostMapping("/v1/bookNow")
13     public ResponseEntity<String> bookingTicket(@RequestBody Object request) {
14         //execute actual logic
15         return ResponseEntity.ok("Flight booking successfully completed !");
16     }
17
18     @PostMapping("/v2/bookNow")
19     public ResponseEntity<String> bookingTicket2(@RequestBody Object request) {
20         //latestLogic
21         return ResponseEntity.ok("Flight booking successfully completed !");
22     }
23 }
```



The image shows two side-by-side Postman requests. Both requests are POST methods to `http://localhost:8181/flight/v1/bookNow` and `http://localhost:8181/flight/v2/bookNow`. The responses for both are identical, showing a body of `Flight booking successfully completed !`.

2. Using request params

```
@RequestParam Object request, @RequestParam(name = "version") int version) {
```

```
24
25 @PostMapping("/bookingNow")
26 public ResponseEntity<String> bookTicketVersionWithRequestParam(@RequestBody Object request,
27     //execute actual logic
28     if (version == 1) {
29         return ResponseEntity.ok("This is version 1 of the resource");
30     } else {
31         return ResponseEntity.ok("This is version 2 of the resource");
32     }
33 }
```

The screenshot shows two separate Postman requests for the endpoint `http://localhost:8181/flight/bookNow`.

Request 1 (Left): The URL is `http://localhost:8181/flight/bookingNow?version=1`. The "Params" tab is selected, showing a single parameter `version` with value `1`. The "Body" tab contains the response: `1 This is version 1 of the resource`.

Request 2 (Right): The URL is `http://localhost:8181/flight/bookingNow?version=2`. The "Params" tab is selected, showing a single parameter `version` with value `2`. The "Body" tab contains the response: `1 This is version 2 of the resource`.

Using request header

```
35
36 @PostMapping("/versionWithHeaderParam")
37 public ResponseEntity<String> bookTicketVersionWithHeaderParam(@RequestBody Object request, @RequestHeader(name = "Api-Version") int version) {
38     //execute actual logic
39     if (version == 1) {
40         return ResponseEntity.ok("This is version 1 of the resource");
41     } else {
42         return ResponseEntity.ok("This is version 2 of the resource");
43     }
44 }
```

46. How will you document your rest API ?

Previously it was swagger

```

</dependency>
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.0.4</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>

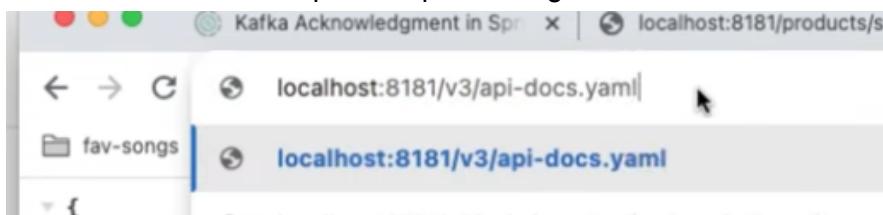
```

API Documents

API Description:

```
        "schema": {
            "$ref": "#/components/schemas/UserDetails"
        }
    },
    "/users": {
        "post": {
            "tags": [
                "demo_controller"
            ],
            "operationId": "addNewUser",
            "requestBody": {
                "content": {
                    "application/json": {
                        "schema": {
                            "$ref": "#/components/schemas/UserDetails"
                        }
                    }
                },
                "required": true
            },
            "responses": {
                "201": {
                    "description": "Created",
                    "content": {}
                }
            }
        }
    }
}
```

We can download this api description using the below url.

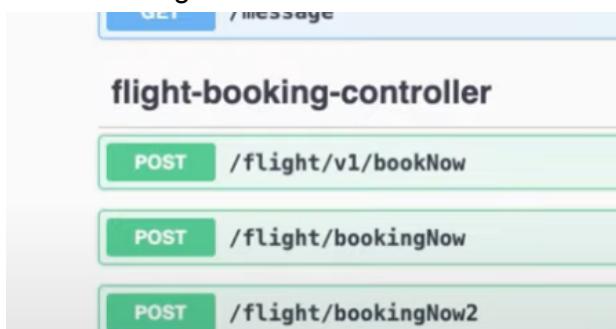


47. How can you hide certain REST endpoints to prevent them from being exposed externally?

Use `@Hidden` annotation as shown below. This will not be displayed in swagger.

```
12
13     @PostMapping("/v1/bookNow")
14     public ResponseEntity<String> bookingTicket(@RequestBody Object request) {
15         //execute actual logic
16         return ResponseEntity.ok("Flight booking successfully completed !");
17     }
18
19     @PostMapping("/v2/bookNow")
20     @Hidden
21     public ResponseEntity<String> bookingTicket2(@RequestBody Object request) {
22         //latest logic
Run: InterviewQaApplication
```

We are seeing V1 but not V2.



48. How will you consume restful API ?

1 RestTemplate

2 FeignClient

3 WebClient

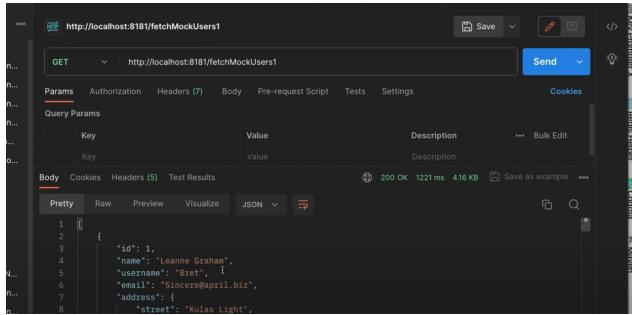
4 Advance Rest Client

Create a rest template bean and autowire it.

```
17
18     @Autowired
19     private RestTemplate template;
```



```
26
27     @GetMapping("/fetchMockUsers1")
28     public List<UserResponse> fetchMockUsersWithRestTemplate() {
29         return template.getForObject(url: "https://jsonplaceholder.typicode.com/users", List.class);
30     }
31 }
```



Feign Client -> proxy as interface



```
8
9     @FeignClient(url="https://jsonplaceholder.typicode.com", name = "USER-CLIENT")
10    public interface UserClient {
11
12        @GetMapping("/users")
13        public List<UserResponse> getUsers();
14    }
15 }
```

In controller

Autowire the above interface in the controller and use it like below.



```
32
33     @GetMapping("/fetchMockUsers2")
34     public List<UserResponse> fetchMockUsersWithFeignClient() {
35         return userClient.getUsers();
36     }
37 }
```

The screenshot shows a Postman collection interface. At the top, there's a header bar with tabs for 'HTTP' and 'http://localhost:8181/fetchMockUsers2'. Below this, a 'GET' method is selected, and the URL is set to 'http://localhost:8181/fetchMockUsers2'. The 'Params' tab is active, showing a 'Query Params' table with a single row: 'Key' (Value) and 'Value' (Description). Under the 'Body' tab, the response is displayed in 'Pretty' format, showing a JSON object with fields: id, name, username, email, and address (which contains street, suite, city, and zipcode). The status bar at the bottom indicates a 200 OK response with 587 ms duration and 4.16 kB size.

Webclient - unblocking and Asynchronous code.

The screenshot shows a Java code editor with syntax highlighting. The code is annotated with line numbers from 37 to 48. It defines a method `fetchMockUsersWithWebclient()` that uses the `webClient.get()` method to retrieve a list of `UserResponse` objects. The `uri` parameter is set to `"/users"`. The `bodyToFlux` method is used to convert the response body to a Flux of `UserResponse`. A comment indicates that the code is demonstrating a synchronous call by blocking with `.block()`. The code ends with a closing brace for the method definition.

```

37
38     @GetMapping("/fetchMockUsers3")
39     public List<UserResponse> fetchMockUsersWithWebclient() {
40         Flux<UserResponse> response = webClient.get() WebClient.RequestHeadersUriSpec<...>
41             .uri(uri: "/users") capture of ?
42             .retrieve() WebClient.ResponseSpec
43             .bodyToFlux(UserResponse.class);
44             // Block and get the result (synchronous call - for demonstration purposes only)
45             return response.collectList().block(); I
46     }
47
48

```