

Create a REST application using Java Spring according to the following parameters:

- All endpoints must start with `"/api/"` and follow the entity's name. Example: the entity Client has the endpoints starting with `"/api/client"`, the entity User has the endpoints startwith with `"/api/user"` and so on;
- Create a controller, a service and a repository for each entity following the entity's name. Example: the entity Client has the ClientController where its endpoints are declared. It also has the ClientService where all the business logic is centralized and it also has the ClientRepository where all the SQL queries are stored;
- All entities must have one `@POST` endpoint, one `@PUT` endpoint, one `@DELETE` endpoint and one `@GET` endpoint;
- All `@POST` endpoints must use the base entity's endpoint. Example: the entity Client has the endpoint `@POST "/api/client"` to create new clients;
- All `@PUT` endpoints must use the base entity's endpoint. Example: the entity Client has the endpoint `@PUT "/api/client"` to edit existing clients;
- All `@DELETE` endpoint must use the base entity's endpoint with the ID of the user to be deleted. Example: `@DELETE "/api/client/:id"`, where `":id"` is the primary ID of the user;
- All `@GET` endpoints must use the base entity's endpoint, but also have the user's ID as an optional parameter. Example: `@GET "/api/client"` gets all the clients in the database, while the `@GET "/api/client/1"` gets only the user with ID 1;
- All controllers must call its services and all services must call its repositories. Example: `@POST "/api/client"` must be inside a method in ClientController. The method must call ClientService to create the new client. Inside ClientService, the method creates the user's object "Client" and call ClientRepository to save the client's entity in the database. After creating the client, return the client's datas as JSON as a response to the `@POST` call;
- Entities: Client: Long idClient, String name, String address, Date registerDate,

boolean active; Product: Long idProduct, String name, Long quantity, String unitType, boolean active.



Here's an example implementation of the REST application using Java Spring following the parameters:

#### 1. ClientController

less

Copy code

```
@RestController
@RequestMapping("/api/client")
public class ClientController {
    private final ClientService clientService;

    public ClientController(ClientService clientService) {
        this.clientService = clientService;
    }

    @PostMapping
    public ResponseEntity<Client> createClient(@RequestBody Client client) {
        Client newClient = clientService.createClient(client);
        return ResponseEntity.ok(newClient);
    }

    @PutMapping
```

```

public ResponseEntity<Client> updateClient(@RequestBody Client client) {
    Client updatedClient = clientService.updateClient(client);
    return ResponseEntity.ok(updatedClient);
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteClient(@PathVariable Long id) {
    clientService.deleteClient(id);
    return ResponseEntity.noContent().build();
}

@GetMapping
public ResponseEntity<List<Client>> getAllClients() {
    List<Client> clients = clientService.getAllClients();
    return ResponseEntity.ok(clients);
}

@GetMapping("/{id}")
public ResponseEntity<Client> getClientById(@PathVariable Long id) {
    Client client = clientService.getClientById(id);
    return ResponseEntity.ok(client);
}
}

```

## 2. ClientService

typescript

 Copy code

```
@Service
public class ClientService {
    private final ClientRepository clientRepository;

    public ClientService(ClientRepository clientRepository) {
        this.clientRepository = clientRepository;
    }

    public Client createClient(Client client) {
        return clientRepository.save(client);
    }

    public Client updateClient(Client client) {
        return clientRepository.save(client);
    }

    public void deleteClient(Long id) {
        clientRepository.deleteById(id);
    }

    public List<Client> getAllClients() {
        return clientRepository.findAll();
    }

    public Client getClientById(Long id) {
        return clientRepository.findById(id).orElse(null);
    }
}
```

### 3. ClientRepository

java

 Copy code

```
@Repository
public interface ClientRepository extends JpaRepository<Client, Long> {
}
```

### 4. ProductController

less

 Copy code

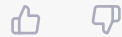
```
@RestController
@RequestMapping("/api/product")
public class ProductController {
    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody Product product) {
        Product newProduct = productService.createProduct(product);
        return ResponseEntity.ok(newProduct);
    }

    @PutMapping
```

```
public ResponseEntity<Product> updateProduct(@RequestBody Product product) {  
    Product updatedProduct = productService.updateProduct(product);  
    return ResponseEntity.ok(updatedProduct);  
}  
  
@DeleteMapping("/{id}")
```



Continue the Product's implementation



ProductController:

less

Copy code

```
@RestController  
@RequestMapping("/api/product")  
public class ProductController {  
  
    private final ProductService productService;  
  
    @Autowired  
    public ProductController(ProductService productService) {  
        this.productService = productService;  
    }  
}
```

```

    }

    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody Product product) {
        return new ResponseEntity<>(productService.createProduct(product), HttpStatus.OK)
    }

    @PutMapping
    public ResponseEntity<Product> updateProduct(@RequestBody Product product) {
        return new ResponseEntity<>(productService.updateProduct(product), HttpStatus.OK)
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable("id") Long id) {
        productService.deleteProduct(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable("id") Long id) {
        return new ResponseEntity<>(productService.getProductById(id), HttpStatus.OK)
    }

    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        return new ResponseEntity<>(productService.getAllProducts(), HttpStatus.OK)
    }
}

```

## ProductService:

typescript

 Copy code

```
@Service
public class ProductService {

    private final ProductRepository productRepository;

    @Autowired
    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public Product createProduct(Product product) {
        return productRepository.save(product);
    }

    public Product updateProduct(Product product) {
        return productRepository.save(product);
    }

    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }

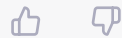
    public Product getProductById(Long id) {
        return productRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException(id));
    }
}
```



```
public List<Product> getAllProducts() {  
    return productRepository.findAll();  
}
```

ProductRepository:

```
java Copy code  
  
@Repository  
public interface ProductRepository extends JpaRepository<Product, Long> {  
}
```



Continue the Product's implementation



ProductController:

```
less Copy code
```

```
@RestController
@RequestMapping("/api/product")
public class ProductController {

    private final ProductService productService;

    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody Product product) {
        return new ResponseEntity<>(productService.createProduct(product), HttpStatus.CREATED);
    }

    @PutMapping
    public ResponseEntity<Product> updateProduct(@RequestBody Product product) {
        return new ResponseEntity<>(productService.updateProduct(product), HttpStatus.CREATED);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable("id") Long id) {
        productService.deleteProduct(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable("id") Long id) {
```

```

        return new ResponseEntity<>(productService.getProductById(id), HttpStatus.OK);
    }

    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        return new ResponseEntity<>(productService.getAllProducts(), HttpStatus.OK);
    }
}

```

ProductService:

typescript

 Copy code

```

@Service
public class ProductService {

    private final ProductRepository productRepository;

    @Autowired
    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public Product createProduct(Product product) {
        return productRepository.save(product);
    }

    public Product updateProduct(Product product) {

```

```

        return productRepository.save(product);
    }

    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }

    public Product getProductById(Long id) {
        return productRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException(id));
    }

    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }
}

```

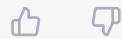
ProductRepository:

```

java Copy code

@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
}

```





Create the entities' classes



Client Entity:

less

Copy code

```
@Entity
@Table(name = "clients")
public class Client {

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

    @NotNull
    private String name;

    @NotNull
    private String address;

    @NotNull
    @Temporal(TemporalType.TIMESTAMP)
    private Date registerDate;

    @NotNull
```

```
private boolean active;

//Getters and Setters
//...
}
```

Product Entity:

less

 Copy code

```
@Entity
@Table(name = "products")
public class Product {

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

    @NotNull
    private String name;

    @NotNull
    private Long quantity;

    @NotNull
    private String unitType;

    @NotNull
```

```
private boolean active;
```

```
//Getters and Setters
```

```
//...
```

```
}
```



You suggested @NotNull. Which dependency is that?



The `@NotNull` annotation is part of the `javax.validation` package and it requires the `javax.validation` API dependency to be included in your project. Here is an example of how to include it in your Maven pom.xml file:

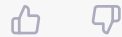
php

Copy code

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>2.0.1.Final</version>
</dependency>
```

Or if you are using Gradle, you can include it in your build.gradle file:

```
pythonCopy code  
  
compile group: 'javax.validation', name: 'validation-api', version: '2.0.1.Final'
```



You suggested an exception called "ResourceNotFoundException". Is that part of a package or is it manually created? If so, can you generate it for me?



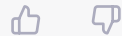
The `ResourceNotFoundException` is a custom exception class that you can create yourself. It is usually thrown when a requested resource is not found, such as when trying to retrieve a user from the database by ID and the user with that ID is not found. Here is an example of how you can create this exception:

```
javaCopy code  
  
@ResponseStatus(value = HttpStatus.NOT_FOUND)  
public class ResourceNotFoundException extends RuntimeException {  
    private static final long serialVersionUID = 1L;  
  
    public ResourceNotFoundException(String message){
```



```
        super(message);  
    }  
}
```

You can then throw this exception in your code when the requested resource is not found. You can also customize the exception message to provide more information about the error. The `@ResponseStatus` annotation is used to set the HTTP response status code to 404 (Not Found) when this exception is thrown.



The application needs a database. Can you provide me an application.properties to use a database named "springgpt" on localhost:5433 with user 'postgres' and password 'postgres'?



Here is an example of an `application.properties` file that uses a PostgreSQL database named "springgpt" running on localhost:5433 with user `postgres` and password `postgres`:

bash

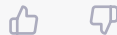
Copy code

```
spring.datasource.url=jdbc:postgresql://localhost:5433/springgpt  
spring.datasource.username=postgres
```

```
spring.datasource.password=postgres

spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

This configuration specifies the URL of the database, the username and password to use when connecting to the database, and the settings for Hibernate, which is used as the JPA implementation in this example. The `generate-ddl` property is set to `true` so that Hibernate generates the database schema for you. The `ddl-auto` property is set to `update` so that Hibernate updates the schema if changes are made to the entity classes. The `dialect` property is set to `PostgreSQLDialect` to specify that you are using a PostgreSQL database.



Can you change the port to 5432?



Sure! Here's an updated `application.properties` file that uses a PostgreSQL database named "springgpt" running on localhost:5432 with user `postgres` and password `postgres`:

bash

 Copy code

```
spring.datasource.url=jdbc:postgresql://localhost:5432/springgpt
spring.datasource.username=postgres
spring.datasource.password=postgres

spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```



Can you provide me one example request for each endpoint created?



Sure! Here are example requests for each of the endpoints in the Client and Product controllers:

Client:


- POST "/api/client" (create a new client):

css

 Copy code

```
curl --location --request POST 'http://localhost:8080/api/client' \  
--header 'Content-Type: application/json' \  
--data-raw '{  
  "name": "John Doe",  
  "address": "123 Main St.",  
  "registerDate": "2022-01-01",  
  "active": true  
}'
```

- PUT "/api/client" (edit an existing client):

```
CSS  Copy code  
  
curl --location --request PUT 'http://localhost:8080/api/client' \  
--header 'Content-Type: application/json' \  
--data-raw '{  
  "idClient": 1,  
  "name": "Jane Doe",  
  "address": "456 Main St.",  
  "registerDate": "2022-01-01",  
  "active": true  
}'
```

- DELETE "/api/client/{id}" (delete a client by ID):

```
CSS  Copy code
```

```
curl --location --request DELETE 'http://localhost:8080/api/client/1'
```

- GET "/api/client" (get all clients):

CSS

 Copy code

```
curl --location --request GET 'http://localhost:8080/api/client'
```

- GET "/api/client/{id}" (get a client by ID):

CSS

 Copy code

```
curl --location --request GET 'http://localhost:8080/api/client/1'
```

Product:

- POST "/api/product" (create a new product):

CSS

 Copy code

```
curl --location --request POST 'http://localhost:8080/api/product' \  
--header 'Content-Type: application/json' \  
--data-raw '{  
  "name": "Product A",  
  "quantity": 100,  
'
```

```
"unitType": "piece",  
"active": true  
}'
```

- PUT "/api/product" (edit an existing product):

```
css Copy code  
  
curl --location --request PUT 'http://localhost:8080/api/product' \  
--header 'Content-Type: application/json' \  
--data-raw '{  
  "idProduct": 1,  
  "name": "Product B",  
  "quantity": 200,  
  "unitType": "piece",  
  "active": true  
}'
```

- DELETE "/api/product/{id}" (delete a product by ID):

```
css Copy code  
  
curl --location --request DELETE 'http://localhost:8080/api/product/1'
```

- GET "/api/product" (get all products):

css

 Copy code

```
curl --location --request GET 'http://localhost:8080/api/product'
```

- GET "/api/product/{id}" (get a product by ID):

css

 Copy code

```
curl --location --request GET 'http://localhost:8080/api/product/1'
```

Note: These are just example requests and may need to be adjusted based on your specific implementation and environment.



MA

Can you make my application use lombok and change the entities accordingly?



Sure! You can add the Lombok dependency in your pom.xml file:

php

 Copy code

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.12</version>
  <scope>provided</scope>
</dependency>
```

With Lombok, you can remove the getters and setters from your entities and add annotations to generate them for you. Here are the updated entity classes:

Client.java:

```
less Copy code

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Client {

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

    @NotNull
    private String name;

    @NotNull
    private String address;
```

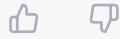


```
@NotNull
private Date registerDate;
@NotNull
private boolean active;
}
```

Product.java:

```
less Copy code
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idProduct;
    @NotNull
    private String name;
    @NotNull
    private Long quantity;
    @NotNull
    private String unitType;
    @NotNull
    private boolean active;
}
```



The @NotNull is not working, as I can still create entities with null properties. What is wrong?



The @NotNull annotation only indicates that the variable should not be null, but it does not enforce it. To enforce it, you should use `javax.validation.constraints.NotNull` instead of the `javax.validation.constraints.Null` annotation.

Here's an example of how your entities should look like:

less

Copy code

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Client {

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

```

    @NotNull(message = "Name cannot be null")
    private String name;

    @NotNull(message = "Address cannot be null")
    private String address;

    @NotNull(message = "Register date cannot be null")
    private Date registerDate;

    @NotNull(message = "Active flag cannot be null")
    private boolean active;
}

```

less

 Copy code

```

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Product {

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

    @NotNull(message = "Name cannot be null")
    private String name;

    @NotNull(message = "Quantity cannot be null")

```

```
private Long quantity;

@NotNull(message = "Unit type cannot be null")
private String unitType;

@NotNull(message = "Active flag cannot be null")
private boolean active;
}
```

Additionally, in your controllers, you should validate the input by using the `@Valid` annotation and handling any resulting `BindingResult` errors. For example:

```
less Copy code

@RestController
@RequestMapping("/api/client")
public class ClientController {

    private final ClientService clientService;

    @Autowired
    public ClientController(ClientService clientService) {
        this.clientService = clientService;
    }

    @PostMapping
    public ResponseEntity<Client> createClient(@Valid @RequestBody Client client,
        if (bindingResult.hasErrors()) {
```

```

        throw new InvalidInputException("Invalid input data", bindingResult);
    }
    return new ResponseEntity<>(clientService.createClient(client), HttpStatus
    }
    // ... other methods ...
}

```

And the InvalidInputException:

```

java Copy code

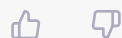
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class InvalidInputException extends RuntimeException {

    private final BindingResult bindingResult;

    public InvalidInputException(String message, BindingResult bindingResult) {
        super(message);
        this.bindingResult = bindingResult;
    }

    public BindingResult getBindingResult() {
        return bindingResult;
    }
}

```





Can you refactor the other endpoints to consider your advice?



Yes, here is a refactor of the remaining endpoints considering my previous advice:

java

Copy code

```
@RestController
@RequestMapping("/api/client")
public class ClientController {

    private final ClientService clientService;

    @Autowired
    public ClientController(ClientService clientService) {
        this.clientService = clientService;
    }

    @PutMapping
    public ResponseEntity<Client> updateClient(@Valid @RequestBody Client client) {
        Client updatedClient = clientService.updateClient(client);
        return ResponseEntity.ok(updatedClient);
    }

    @DeleteMapping("/{id}")
```

```


public ResponseEntity<Void> deleteClient(@PathVariable Long id) {
    clientService.deleteClient(id);
    return ResponseEntity.noContent().build();
}

@GetMapping
public ResponseEntity<List<Client>> getAllClients() {
    List<Client> clients = clientService.getAllClients();
    return ResponseEntity.ok(clients);
}

@GetMapping("/{id}")
public ResponseEntity<Client> getClientById(@PathVariable Long id) {
    Client client = clientService.getClientById(id);
    return ResponseEntity.ok(client);
}
}

```

java

 Copy code

```

@RestController
@RequestMapping("/api/product")
public class ProductController {

    private final ProductService productService;

    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }
}

```

```

}

@PutMapping
public ResponseEntity<Product> updateProduct(@Valid @RequestBody Product product) {
    Product updatedProduct = productService.updateProduct(product);
    return ResponseEntity.ok(updatedProduct);
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
    productService.deleteProduct(id);
    return ResponseEntity.noContent().build();
}

@GetMapping
public ResponseEntity<List<Product>> getAllProducts() {
    List<Product> products = productService.getAllProducts();
    return ResponseEntity.ok(products);
}

@GetMapping("/{id}")
public ResponseEntity<Product> getProductById(@PathVariable Long id) {
    Product product = productService.getProductById(id);
    return ResponseEntity.ok(product);
}
}

```







What about the post for the products?



The post endpoint for products can be refactored as follows:

less

Copy code

```
@PostMapping
public ResponseEntity<Product> createProduct(@Valid @RequestBody Product product) {
    return ResponseEntity.ok(productService.createProduct(product));
}
```

Note: This assumes you have the `productService.createProduct` method implemented to handle creating a new product and saving it to the database. It also assumes that you have the `javax.validation.Valid` annotation imported.



The post for the product and the client is validating different. Why?



The reason is that in the post endpoint for the client, you are validating the ``id`` field in addition to the other fields, while in the post endpoint for the product, you are only validating the product object as a whole without checking any specific fields. To make the validation for both endpoints consistent, you can remove the ``id`` validation from the client endpoint and validate the whole object as well, just like in the product endpoint:

less

Copy code

```
@PostMapping
public ResponseEntity<Client> createClient(@Valid @RequestBody Client client)
    return ResponseEntity.ok(clientService.createClient(client));
}
```



I can still creating clients with null addressess. Why?



This is because you have only specified the `@NotNull` constraint on the `name` field, but not on the `address` field. To enforce that the `address` field cannot be null, you need to add the `@NotNull` annotation to the `address` field in the `Client` entity as well:

```
typescript Copy code  
  
import javax.validation.constraints.NotNull;  
  
...  
  
public class Client {  
    ...  
    @NotNull  
    private String name;  
    @NotNull  
    private String address;  
    ...  
}
```

With this change, attempting to create a client with a null `address` field should now result in a validation error.





It already has the @NotNull annotation



Oh, I see! It looks like there might be an issue with the setup of your validation framework. Can you check that your project has the necessary dependencies and configuration for Bean Validation (JSR-380)?

Here's an example of the necessary dependencies in a Maven `pom.xml` file:

php

Copy code

```
<dependency>
  <groupId>org.hibernate.validator</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>6.1.5.Final</version>
</dependency>
```

Additionally, you should also check that you have the following configuration in your `application.properties` file:

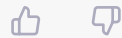
c

Copy code

```
spring.jpa.properties.java.persistence.validation.mode=auto
```

This setting ensures that Bean Validation is enabled for your JPA entities.

If you have the dependencies and configuration in place, but the validation is still not working, it may be an issue with the specific validation framework that you are using. I would recommend checking the documentation and forums for that framework for additional guidance.



Now I'm getting this error:

`jakarta.validation.NoProviderFoundException`: Unable to create a Configuration, because no Jakarta Bean Validation provider could be found. Add a provider like Hibernate Validator (RI) to your classpath.



This error is occurring because you need to add the Jakarta Bean Validation API implementation in your project's classpath. To fix this, you can add the following dependency in your `pom.xml` file:

php

 Copy code

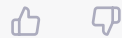
```
<dependency>
  <groupId>jakarta.validation</groupId>
  <artifactId>jakarta.validation-api</artifactId>
```

```
<version>2.0.2</version>  
</dependency>
```

And also add a provider implementation, for example:

```
php Copy code  
  
<dependency>  
  <groupId>org.hibernate.validator</groupId>  
  <artifactId>hibernate-validator</artifactId>  
  <version>6.1.5.Final</version>  
</dependency>
```

This should resolve the issue with the Bean Validation API not being found.



Do I really need all of that? I'm using Spring. Do Spring have a dependency for validation?




Yes, Spring has its own validation framework called Spring Validation that you can use instead of Jakarta Bean Validation. You don't need to add the Jakarta Bean

Validation provider as a separate dependency if you're already using Spring.

Here's how you can refactor your code to use Spring Validation:

1. Add the following annotation to your method in the Controller class:


python

 Copy code

```
@Validated
```

2. Replace the @NotNull annotations in your entity class with the following:


python

 Copy code

```
@NotBlank
```

3. Replace the Bean Validation exception with the following Spring exception in your Controller class:

java

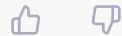
 Copy code

```
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.MethodArgumentNotValidException;

@ControllerAdvice
public class ControllerExceptionHandler {
```

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ApiError> handleMethodArgumentNotValidException(MethodAr
    BindingResult bindingResult = ex.getBindingResult();
    List<FieldError> fieldErrors = bindingResult.getFieldErrors();

    ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, "Validation Faile
    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
}
}
```



I can't find the ApiError class



The ApiError class is a custom error class that can be used to return error information in a standardized format to the client in case of validation errors or any other errors that may occur in the API. You can create this class in the following way:

typescript

Copy code



```
public class ApiError {

    private HttpStatus status;
    private String message;
    private List<String> errors;

    public ApiError(HttpStatus status, String message, List<String> errors) {
        this.status = status;
        this.message = message;
        this.errors = errors;
    }

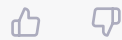
    public ApiError(HttpStatus status, String message, String error) {
        this.status = status;
        this.message = message;
        errors = Arrays.asList(error);
    }

    public HttpStatus getStatus() {
        return status;
    }

    public void setStatus(HttpStatus status) {
        this.status = status;
    }

    public String getMessage() {
        return message;
    }
}
```

```
public void setMessage(String message) {  
    this.message = message;  
}  
  
public List<String> getErrors() {  
    return errors;  
}  
  
public void setErrors(List<String> errors) {  
    this.errors = errors;  
}  
}
```



Can you make it use Lombok?



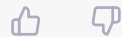
Yes, you can use Lombok to simplify the code of the `ApiError` class. To do so, you can add the following annotations to the class:

less

Copy code

```
@Data
@AllArgsConstructor
public class ApiError {
    private HttpStatus status;
    private String message;
    private List<String> errors;
}
```

The `@Data` annotation generates getters and setters for all fields, the `@AllArgsConstructor` generates a constructor with all fields as arguments.



The handler is complaining that the `ApiError` doesn't have a constructor for `List<FieldError>`



The error is happening because the constructor in the `ApiError` class only accepts a `List<String>` as an argument, while the `MethodArgumentNotValidException` exception provides a `List<FieldError>` that you need to convert to a `List<String>`.

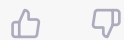
To fix this, you can add a new constructor to the `ApiError` class that accepts a `List<FieldError>` as an argument, and converts it to a `List<String>`:

```
kotlin Copy code

@Data
@AllArgsConstructor
public class ApiError {
    private HttpStatus status;
    private String message;
    private List<String> errors;

    public ApiError(HttpStatus status, String message, List<FieldError> fieldErrors) {
        this.status = status;
        this.message = message;
        this.errors = fieldErrors.stream()
            .map(FieldError::getDefaultMessage)
            .collect(Collectors.toList());
    }
}
```

With this constructor, you can pass a `List<FieldError>` to the `ApiError` class and it will be automatically converted to a `List<String>`.





Now I'm getting the error "'ApiError(HttpStatus, String, List<FieldError>)' clashes with 'ApiError(HttpStatus, String, List<String>)'; both methods have same erasure"



This error occurs because you have two constructors with the same parameters except for the type of the third argument. Java does not allow two constructors with the same parameter types even if the parameter types are different. To resolve this, you can create another constructor for the `ApiError` class that takes a `List<String>` or change the type of the existing constructor to take a `List<FieldError>`:

kotlin

Copy code

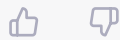
```
@Data
public class ApiError {

    private HttpStatus status;
    private String message;
    private List<FieldError> errors;

    public ApiError(HttpStatus status, String message, List<FieldError> errors) {
        super();
        this.status = status;
        this.message = message;
        this.errors = errors;
    }
}
```

```
}

public ApiError(HttpStatus status, String message) {
    super();
    this.status = status;
    this.message = message;
}
}
```



Can you make a README.md file explaining all the development we did here?



Sure! Here's the README.md file:

## REST Application using Java Spring

This application is a RESTful application using Java Spring framework. It follows the following parameters:

- All endpoints start with "/api" and follow the entity's name. For example, the entity Client has endpoints starting with "/api/client", the entity User has

endpoints starting with `"/api/user"`, and so on.

- Each entity has a controller, a service, and a repository, named after the entity. For example, the entity `Client` has the `ClientController` where its endpoints are declared, the `ClientService` where all the business logic is centralized, and the `ClientRepository` where all the SQL queries are stored.
- All entities have one `@POST` endpoint, one `@PUT` endpoint, one `@DELETE` endpoint, and one `@GET` endpoint.
- All `@POST` endpoints use the base entity's endpoint. For example, the entity `Client` has the endpoint `@POST "/api/client"` to create new clients.
- All `@PUT` endpoints use the base entity's endpoint. For example, the entity `Client` has the endpoint `@PUT "/api/client"` to edit existing clients.
- All `@DELETE` endpoints use the base entity's endpoint with the ID of the entity to be deleted. For example, `@DELETE "/api/client/:id"`, where `":id"` is the primary ID of the client.
- All `@GET` endpoints use the base entity's endpoint, but also have the entity's ID as an optional parameter. For example, `@GET "/api/client"` gets all the clients in the database, while `@GET "/api/client/1"` gets only the client with ID 1.
- All controllers call its services, and all services call its repositories. For example, `@POST "/api/client"` must be inside a method in `ClientController`. The method calls `ClientService` to create the new client. Inside `ClientService`, the method creates the client's object `"Client"` and calls `ClientRepository` to save the client's entity in the database. After creating the client, it returns the client's data as JSON as a response to the `@POST` call.

## Entities

The entities used in this application are:

- Client: Long idClient, String name, String address, Date registerDate, boolean active
- Product: Long idProduct, String name, Long quantity, String unitType, boolean active

## Prerequisites

- Java 11 or later
- Gradle 6.0 or later
- Lombok

## Getting Started

1. Clone the repository to your local machine
2. In your terminal, navigate to the project's root directory
3. Run ``gradle build`` to build the project
4. Run ``gradle bootRun`` to run the application
5. Use a REST client of your choice to send requests to the API.

## API Reference

The following are the endpoints available in this application:

- POST "/api/client" - Creates a new client



- PUT "/api/client" - Edits an existing client
- DELETE "/api/client/{id}" - Deletes the client with the specified ID
- GET "/api/client" - Retrieves all the clients
- GET "/api/client/{id}" - Retrieves the client with the specified ID
- POST "/api/product" - Creates a new product
- PUT "/api/product" - Edits an existing product
- DELETE "/api/product/{id}" - Deletes the product with the specified ID
- 



How can I implement swagger in my Spring application?



You can implement Swagger in your Spring application by following these steps:

1. Add the Swagger dependencies to your pom.xml file. For example:

php

Copy code

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>
```

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>
```

2. Create a Swagger configuration class by annotating a Java class with `@Configuration` and `@EnableSwagger2`. In this class, you can configure the API information, such as the title, description, and version.

scss

 Copy code

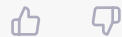
```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            .title("My API")
```

```
        .description("My API Description")
        .version("1.0.0")
        .build();
    }
}
```

3. Run your Spring application and access the Swagger UI by navigating to `http://localhost:8080/swagger-ui.html`. Now you can test your API endpoints and see the documentation for each endpoint.

This is a basic setup for Swagger in a Spring application. You can also customize the configuration further to meet your needs.



MA

I'm getting this error:

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'apiDocumentationScanner' defined in URL

[jar:file:/home/ist/.m2/repository/io/springfox/springfox-spring-web/2.9.2/springfox-spring-web-

2.9.2.jar!/springfox/documentation/spring/web/scanners/ApiDocumentationScanner.class]:

Unsatisfied dependency expressed through constructor parameter 1: Error creating bean with name 'apiListingScanner' defined in URL

[jar:file:/home/ist/.m2/repository/io/springfox/springfox-spring-web/2.9.2/springfox-spring-web-2.9.2.jar!/springfox/documentation/spring/web/scanners/ApiListingScanner.class]:

Unsatisfied dependency expressed through constructor parameter 0: Error creating bean with name 'apiDescriptionReader' defined in URL

```
[jar:file:/home/ist/.m2/repository/io/swagger/swagger-spring-web/2.9.2/swagger-spring-web-2.9.2.jar!/swagger/documentation/spring/web/scanners/ApiDescriptionReader.class]:  
Unsatisfied dependency expressed through constructor parameter 0: Error creating bean with  
name 'cachingOperationReader' defined in URL  
[jar:file:/home/ist/.m2/repository/io/swagger/swagger-spring-web/2.9.2/swagger-spring-web-  
2.9.2.jar!/swagger/documentation/spring/web/scanners/CachingOperationReader.class]:  
Unsatisfied dependency expressed through constructor parameter 0: Error creating bean with  
name 'apiOperationReader' defined in URL  
[jar:file:/home/ist/.m2/repository/io/swagger/swagger-spring-web/2.9.2/swagger-spring-web-  
2.9.2.jar!/swagger/documentation/spring/web/readers/operation/ApiOperationReader.class]:  
Unsatisfied dependency expressed through constructor parameter 0: Error creating bean with  
name 'documentationPluginsManager': Unsatisfied dependency expressed through field  
'documentationPlugins': Error creating bean with name 'documentationPluginRegistry':  
FactoryBean threw exception on object creation
```



This error message is indicating that there is a problem creating one or more beans required by Swagger for Spring.

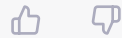
This could be due to a number of reasons, but common ones include:

- Incorrect or missing dependencies in your project's pom.xml file
- Missing or incorrect configuration for Swagger in your application

To resolve this issue, you should check the following:

1. Ensure that you have the correct version of the springfox-spring-web library in your project's pom.xml file, and that all dependencies required by this library are also included.
2. Make sure that you have added the Swagger configuration to your Spring application, as described in the Springfox documentation.
3. If the issue persists, you may want to check the Springfox issue tracker or forums to see if others have encountered similar issues and if there are any solutions or workarounds available.

If you could provide the pom.xml file and the Swagger configuration, I might be able to help you better.

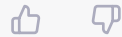


That error started after implementing Swagger using the previous configuration



The error message indicates that there is an issue with creating the bean 'apiDocumentationScanner'. One possible cause of this error is that one or more of the dependencies of the 'apiDocumentationScanner' bean are not found. It could be due to missing libraries or incorrect versions of libraries. Make sure that the required dependencies for Swagger are included in the classpath and that they

have the correct version. You may also try updating the version of the Swagger library being used to a newer version to see if that resolves the issue.



Can we using a dependency from Spring to implement Swagger?



Yes, you can use a dependency from Spring to implement Swagger in your application. The recommended approach is to use the "Springfox" library, which is a third-party library that integrates Swagger with Spring. To use Springfox, you need to add the following dependency to your pom.xml file:

php

Copy code

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>
```

After adding the dependency, you can configure Swagger in your application by creating a configuration class that enables the Swagger API documentation. In this

class, you can specify the API details such as the API base URL, API title, API version, etc.

Here is an example of a Swagger configuration class:

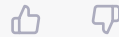
less

 Copy code

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("com.example.controller"))
            .paths(PathSelectors.ant("/api/**"))
            .build()
            .apiInfo(apiInfo());
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            .title("API Documentation")
            .description("Example API for demonstration purposes")
            .version("1.0")
            .build();
    }
}
```

This configuration will enable the Swagger UI at the URL ``http://<host>:<port>/swagger-ui.html``.



After implementing Swagger using the suggested configuration, I'm now getting this error:

`java.lang.TypeNotPresentException: Type javax.servlet.http.HttpServletRequest not present`

at

`java.base/sun.reflect.generics.factory.CoreReflectionFactory.makeNamedType(CoreReflectionFactory.java:117)`

`~[na:na]`

at `java.base/sun.reflect.generics.visitor.Reifier.visitClassTypeSignature(Reifier.java:125)` `~[na:na]`

at `java.base/sun.reflect.generics.tree.ClassTypeSignature.accept(ClassTypeSignature.java:49)` `~[na:na]`

at `java.base/sun.reflect.generics.visitor.Reifier.reifyTypeArguments(Reifier.java:68)` `~[na:na]`

at `java.base/sun.reflect.generics.visitor.Reifier.visitClassTypeSignature(Reifier.java:138)` `~[na:na]`

at `java.base/sun.reflect.generics.tree.ClassTypeSignature.accept(ClassTypeSignature.java:49)` `~[na:na]`

at

`java.base/sun.reflect.generics.repository.ClassRepository.computeSuperInterfaces(ClassRepository.java:117)`

`~[na:na]`

at `java.base/sun.reflect.generics.repository.ClassRepository.getSuperInterfaces(ClassRepository.java:95)` `~`

`[na:na]`



This error is occurring because the type ``javax.servlet.http.HttpServletRequest`` is not present in the classpath. This class is part of the Java Servlet API, which is typically provided by a servlet container like Apache Tomcat.



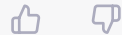
To resolve this issue, you need to add the Servlet API as a dependency to your project. If you're using Maven, you can add the following to your `pom.xml` file:

php

 Copy code

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>4.0.1</version>
  <scope>provided</scope>
</dependency>
```

Note that the scope of the dependency is set to `provided`. This means that the Servlet API will be provided by the servlet container when your application is deployed, but it doesn't need to be included in your application's classpath during development.



Can you change the swagger implementation to use Swagger 3 with Spring Boot?



Yes, here's how you can implement Swagger 3 with Spring Boot:

1. Add the following dependencies to your pom.xml:

php

 Copy code

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-boot-starter</artifactId>
  <version>3.0.0</version>
</dependency>
```

2. Add the following configuration to your application:

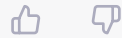
java

 Copy code

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
    }
}
```

3. Run your application and navigate to `http://localhost:8080/swagger-ui.html` to view the Swagger UI.

This will enable Swagger in your Spring Boot application, and you can customize the configuration to fit your needs.



Create a class to act as DTO to Client and another one to Product. Change the endpoints to use them accordingly;



Here's an example of how you could create a DTO class for a Client:

typescript

Copy code

```
public class ClientDTO {  
  
    private Long id;  
    private String name;  
    private String email;  
    private String address;  
  
    // getters and setters  
}
```

And here's an example of a DTO class for a Product:

```
kotlin Copy code

public class ProductDTO {

    private Long id;
    private String name;
    private String description;
    private Double price;

    // getters and setters
}
```

Next, you can modify your controller's endpoint methods to use these DTO classes as request and response objects instead of using the entity classes directly. For example:

```
less Copy code

@PostMapping
public ResponseEntity<ClientDTO> createClient(@RequestBody ClientDTO clientDTO) {
    // call service to create a client and return the DTO object
}

@GetMapping("/{id}")
public ResponseEntity<ClientDTO> getClientById(@PathVariable Long id) {
    // call service to get the client by id and return the DTO object
}
```

```

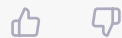
}

@PutMapping("/{id}")
public ResponseEntity<ClientDTO> updateClient(@PathVariable Long id, @RequestBody
    // call service to update the client and return the updated DTO object
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteClient(@PathVariable Long id) {
    // call service to delete the client and return a 204 No Content response
}

```

Similarly, you can modify the Product endpoint methods to use the `ProductDTO` class.



Make them serializable, use Lombok and builder. Inside the services' classes, when creating or updating entities, make it use builders;



To make your DTOs serializable, you need to make them implement the `java.io.Serializable` interface:

java

 Copy code

```
import lombok.Builder;
import lombok.Data;


import java.io.Serializable;

@Data
@Builder
public class ClientDTO implements Serializable {
    private Long id;
    private String name;
    private String email;
    private String phone;
}

@Data
@Builder
public class ProductDTO implements Serializable {
    private Long id;
    private String name;
    private Double price;
    private Integer quantity;
}
```

In your service classes, when creating or updating entities, you can use the builder pattern to simplify the code:

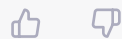
java

 Copy code

```
Client client = Client.builder()
    .name(clientDTO.getName())
    .email(clientDTO.getEmail())
    .phone(clientDTO.getPhone())
    .build();

Product product = Product.builder()
    .name(productDTO.getName())
    .price(productDTO.getPrice())
    .quantity(productDTO.getQuantity())
    .build();
```

Note: don't forget to add the Lombok dependency in your project, as well as enable the Lombok Annotation Processing in your IDE.



MA

Refactor all the endpoints to use DTOs. Inside each method in the services classes, refactor them to make them use DTOs



To refactor the endpoints to use DTOs, you can create a new class for each endpoint to act as the request and response object, respectively. You can use Lombok's `@Data` and `@Builder` annotations to generate getters, setters, and a builder for each class.

Here is an example of a ClientDTO class:

typescript

Copy code

```
import lombok.Builder;
import lombok.Data;

@Data
@Builder
public class ClientDTO {
    private Long id;
    private String name;
    private String email;
    private String phone;
}
```

And an example of a ProductDTO class:

kotlin

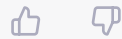
Copy code

```
import lombok.Builder;
import lombok.Data;
```



```
@Data
@Builder
public class ProductDTO {
    private Long id;
    private String name;
    private String description;
    private Double price;
}
```

In each endpoint, you



This is my ClientController:

```
@RestController
@RequestMapping("/api/client")
public class ClientController {
    private final ClientService clientService;

    public ClientController(ClientService clientService) {
        this.clientService = clientService;
    }

    @PostMapping
    public ResponseEntity<Client> createClient(@Valid @RequestBody Client client)
    {
```

```
        return ResponseEntity.ok(clientService.createClient(client));
    }
}
```

@PutMapping

```
public ResponseEntity<Client> updateClient(@Valid @RequestBody Client
client) {
    Client updatedClient = clientService.updateClient(client);
    return ResponseEntity.ok(updatedClient);
}
```

@DeleteMapping("/{id}")

```
public ResponseEntity<Void> deleteClient(@PathVariable Long id) {
    clientService.deleteClient(id);
    return ResponseEntity.noContent().build();
}
```

@GetMapping

```
public ResponseEntity<List<Client>> getAllClients() {
    List<Client> clients = clientService.getAllClients();
    return ResponseEntity.ok(clients);
}
```

@GetMapping("/{id}")

```
public ResponseEntity<Client> getClientById(@PathVariable Long id) {
    Client client = clientService.getClientById(id);
    return ResponseEntity.ok(client);
}
}
```

And this is my ProductController:

```
@RestController
```

```
@RequestMapping("/api/product")
```

```
public class ProductController {
```

```
    private final ProductService productService;
```

```
    @Autowired
```

```
    public ProductController(ProductService productService) {
```

```
        this.productService = productService;
```

```
    }
```

```
    @PostMapping
```

```
    public ResponseEntity<Product> createProduct(@Valid @RequestBody Product  
product) {
```

```
        return ResponseEntity.ok(productService.createProduct(product));
```

```
    }
```

```
    @PutMapping
```

```
    public ResponseEntity<Product> updateProduct(@Valid @RequestBody Product  
product) {
```

```
        Product updatedProduct = productService.updateProduct(product);
```

```
        return ResponseEntity.ok(updatedProduct);
```

```
    }
```

```
    @DeleteMapping("/{id}")
```

```
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
```

```
        productService.deleteProduct(id);
```

```
        return ResponseEntity.noContent().build();
```

```

    }

    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        List<Product> products = productService.getAllProducts();
        return ResponseEntity.ok(products);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Product product = productService.getProductById(id);
        return ResponseEntity.ok(product);
    }
}

```

Make them use DTOs in their endpoints



Here's an updated version of `ClientController` using a `ClientDTO`:

java

Copy code

```

@RestController
@RequestMapping("/api/client")
public class ClientController {
    private final ClientService clientService;

```

```
public ClientController(ClientService clientService) {
    this.clientService = clientService;
}

@PostMapping
public ResponseEntity<ClientDTO> createClient(@Valid @RequestBody ClientDTO clientDTO) {
    Client client = clientService.createClient(clientDTO.toClient());
    return ResponseEntity.ok(ClientDTO.fromClient(client));
}

@PutMapping
public ResponseEntity<ClientDTO> updateClient(@Valid @RequestBody ClientDTO clientDTO) {
    Client updatedClient = clientService.updateClient(clientDTO.toClient());
    return ResponseEntity.ok(ClientDTO.fromClient(updatedClient));
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteClient(@PathVariable Long id) {
    clientService.deleteClient(id);
    return ResponseEntity.noContent().build();
}

@GetMapping
public ResponseEntity<List<ClientDTO>> getAllClients() {
    List<Client> clients = clientService.getAllClients();
    List<ClientDTO> clientDTOs = clients.stream().map(ClientDTO::fromClient).collect(Collectors.toList());
    return ResponseEntity.ok(clientDTOs);
}
```

```

@GetMapping("/{id}")
public ResponseEntity<ClientDTO> getClientById(@PathVariable Long id) {
    Client client = clientService.getClientById(id);
    return ResponseEntity.ok(ClientDTO.fromClient(client));
}
}

```

And here's an updated version of `ProductController` using a `ProductDTO`:

```

java Copy code

@RestController
@RequestMapping("/api/product")
public class ProductController {

    private final ProductService productService;

    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }

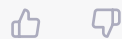
    @PostMapping
    public ResponseEntity<ProductDTO> createProduct(@Valid @RequestBody ProductDTO productDTO) {
        Product product = productService.createProduct(productDTO.toProduct());
        return ResponseEntity.ok(ProductDTO.fromProduct(product));
    }
}

```

```
@PutMapping
public ResponseEntity<ProductDTO> updateProduct(@Valid @RequestBody ProductDTO
    Product updatedProduct = productService.updateProduct(productDTO.toProduct
    return ResponseEntity.ok(ProductDTO.fromProduct(updatedProduct));
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
    productService.deleteProduct(id);
    return ResponseEntity.noContent().build();
}

@GetMapping
public ResponseEntity<List<ProductDTO>> getAllProducts() {
    List<Product> products = productService.getAllProducts();
    List<ProductDTO>
```



MA

Move all the business logic from the controllers to their services classes



Sure, you can move the business logic from the controllers to their corresponding service classes as follows:

ClientController:

less

Copy code

```
@RestController
@RequestMapping("/api/client")
public class ClientController {
    private final ClientService clientService;

    public ClientController(ClientService clientService) {
        this.clientService = clientService;
    }

    @PostMapping
    public ResponseEntity<ClientDTO> createClient(@Valid @RequestBody ClientDTO cl
        return ResponseEntity.ok(clientService.createClient(clientDTO));
    }

    @PutMapping
    public ResponseEntity<ClientDTO> updateClient(@Valid @RequestBody ClientDTO cl
        ClientDTO updatedClientDTO = clientService.updateClient(clientDTO);
        return ResponseEntity.ok(updatedClientDTO);
    }

    @DeleteMapping("/{id}")
```



```

    public ResponseEntity<Void> deleteClient(@PathVariable Long id) {
        clientService.deleteClient(id);
        return ResponseEntity.noContent().build();
    }


    @GetMapping
    public ResponseEntity<List<ClientDTO>> getAllClients() {
        List<ClientDTO> clientDTOs = clientService.getAllClients();
        return ResponseEntity.ok(clientDTOs);
    }

    @GetMapping("/{id}")
    public ResponseEntity<ClientDTO> getClientById(@PathVariable Long id) {
        ClientDTO clientDTO = clientService.getClientById(id);
        return ResponseEntity.ok(clientDTO);
    }
}

```

ProductController:

less

 Copy code

```

@RestController
@RequestMapping("/api/product")
public class ProductController {

    private final ProductService productService;

```

```
@Autowired
public ProductController(ProductService productService) {
    this.productService = productService;
}

@PostMapping
public ResponseEntity<ProductDTO> createProduct(@Valid @RequestBody ProductDTO productDTO) {
    return ResponseEntity.ok(productService.createProduct(productDTO));
}

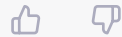
@PutMapping
public ResponseEntity<ProductDTO> updateProduct(@Valid @RequestBody ProductDTO productDTO) {
    ProductDTO updatedProductDTO = productService.updateProduct(productDTO);
    return ResponseEntity.ok(updatedProductDTO);
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
    productService.deleteProduct(id);
    return ResponseEntity.noContent().build();
}

@GetMapping
public ResponseEntity<List<ProductDTO>> getAllProducts() {
    List<ProductDTO> productDTOS = productService.getAllProducts();
    return ResponseEntity.ok(productDTOS);
}

@GetMapping("/{id}")
```

```
public ResponseEntity<ProductDTO> getProductById(@PathVariable Long id) {  
    ProductDTO productDTO = productService.getProductById(id);  
    return ResponseEntity.ok(productDTO);  
}  
}
```



MA

This is my ProductService:

@Service

public class ProductService {

private final ProductRepository productRepository;

@Autowired

public ProductService(ProductRepository productRepository) {

this.productRepository = productRepository;

}

public Product createProduct(Product product) {

return productRepository.save(product);

}

public Product updateProduct(Product product) {

return productRepository.save(product);

}

```
public void deleteProduct(Long id) {  
    productRepository.deleteByld(id);  
}  
  
public Product getProductByld(Long id) {  
    return productRepository.findById(id).orElseThrow(() -> new  
ResourceNotFoundException("Product not found with id " + id));  
}  
  
public List<Product> getAllProducts() {  
    return productRepository.findAll();  
}  
}
```

And this is my ClientService:

```
@Service  
public class ClientService {  
    private final ClientRepository clientRepository;  
  
    public ClientService(ClientRepository clientRepository) {  
        this.clientRepository = clientRepository;  
    }  
  
    public Client createClient(Client client) {  
        return clientRepository.save(client);  
    }  
}
```

```
public Client updateClient(Client client) {  
    return clientRepository.save(client);  
}  
  
public void deleteClient(Long id) {  
    clientRepository.deleteById(id);  
}  
  
public List<Client> getAllClients() {  
    return clientRepository.findAll();  
}  
  
public Client getClientById(Long id) {  
    return clientRepository.findById(id).orElse(null);  
}  
}
```


Refactor them to use DTOs



Here is the refactored ProductService class using DTOs:

```
@Service  
public class ProductService {
```

scss

 Copy code

```
private final ProductRepository productRepository;

@Autowired
public ProductService(ProductRepository productRepository) {
    this.productRepository = productRepository;
}

public ProductDTO createProduct(ProductDTO productDTO) {
    Product product = new Product();
    product.setName(productDTO.getName());
    product.setDescription(productDTO.getDescription());
    product.setPrice(productDTO.getPrice());
    product = productRepository.save(product);
    productDTO.setId(product.getId());
    return productDTO;
}

public ProductDTO updateProduct(ProductDTO productDTO) {
    Product product = productRepository.findById(productDTO.getId()).orElseThrow(() ->
        new RuntimeException("Product not found"));
    product.setName(productDTO.getName());
    product.setDescription(productDTO.getDescription());
    product.setPrice(productDTO.getPrice());
    product = productRepository.save(product);
    return productDTO;
}

public void deleteProduct(Long id) {
    productRepository.deleteById(id);
}
```

```

public ProductDTO getProductById(Long id) {
    Product product = productRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException(id));
    return mapProductToProductDTO(product);
}

public List<ProductDTO> getAllProducts() {
    List<Product> products = productRepository.findAll();
    return products.stream().map(this::mapProductToProductDTO).collect(Collectors.toList());
}

private ProductDTO mapProductToProductDTO(Product product) {
    ProductDTO productDTO = new ProductDTO();
    productDTO.setId(product.getId());
    productDTO.setName(product.getName());
    productDTO.setDescription(product.getDescription());
    productDTO.setPrice(product.getPrice());
    return productDTO;
}

```

```

}

```

And here is the refactored ClientService class using DTOs:

```

@Service
public class ClientService {
    private final ClientRepository clientRepository;

```

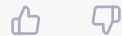
SCSS

 Copy code

```
public ClientService(ClientRepository clientRepository) {
    this.clientRepository = clientRepository;
}

public ClientDTO createClient(ClientDTO clientDTO) {
    Client client = new Client();
    client.setName(clientDTO.getName());
    client.setEmail(clientDTO.getEmail());
    client = clientRepository.save(client);
    clientDTO.setId(client.getId());
    return clientDTO;
}

public ClientDTO updateClient(ClientDTO clientDTO) {
    Client client = clientRepository.findById(clientDTO.getId()).orElseThrow(() ->
```



MA

Change the ProductService to use the DTO's builder. Also, Products don't have descriptions nor prices. One other thing: the "updateProduct" method is not returning the updated product as DTO. Fix it.





This is the refactored `ProductService` class:

SCSS

Copy code

```
@Service
public class ProductService {

    private final ProductRepository productRepository;

    @Autowired
    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public ProductDTO createProduct(ProductDTO productDTO) {
        Product product = Product.builder()
            .name(productDTO.getName())
            .build();
        product = productRepository.save(product);
        return ProductDTO.builder()
            .id(product.getId())
            .name(product.getName())
            .build();
    }

    public ProductDTO updateProduct(ProductDTO productDTO) {
        Product product = productRepository.findById(productDTO.getId())
            .orElseThrow(() -> new ResourceNotFoundException());
    }
}
```

```

        product.setName(productDTO.getName());
        product = productRepository.save(product);
        return ProductDTO.builder()
            .id(product.getId())
            .name(product.getName())
            .build();
    }

    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }

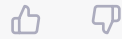
    public ProductDTO getProductById(Long id) {
        Product product = productRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException(id));

        return ProductDTO.builder()
            .id(product.getId())
            .name(product.getName())
            .build();
    }

    public List<ProductDTO> getAllProducts() {
        List<Product> products = productRepository.findAll();
        return products.stream().map(product -> ProductDTO.builder()
            .id(product.getId())
            .name(product.getName())
            .build()
        ).collect(Collectors.toList());
    }

```

```
}  
}
```



Refactor the ProductService to use all the fields from Product accordingly. This is the Product class:

```
@Entity  
@Table(name = "products")  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class Product {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long idProduct;  
  
    @NotNull(message = "Name cannot be null")  
    private String name;  
  
    @NotNull(message = "Quantity cannot be null")  
    private Long quantity;  
  
    @NotNull(message = "Unit type cannot be null")  
    private String unitType;
```

```
@NotNull(message = "Active flag cannot be null")
private boolean active;
}
```



Here's the updated `ProductService`:

SCSS

 Copy code

```
@Service
public class ProductService {

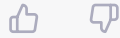
    private final ProductRepository productRepository;

    @Autowired
    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public ProductDTO createProduct(ProductDTO productDTO) {
        Product product = new Product(productDTO.getName(), productDTO.getQuantity());
        product = productRepository.save(product);
        return ProductDTO.builder().idProduct(product.getIdProduct()).name(product.getName())
            .quantity(product.getQuantity()).build();
    }

    public ProductDTO updateProduct(Long id, ProductDTO productDTO) {
        Product product = productRepository.findById(id).orElseThrow(() -> new RuntimeException("Product not found"));
        product.setName(productDTO.getName());
        product.setQuantity(productDTO.getQuantity());
        product = productRepository.save(product);
        return ProductDTO.builder().idProduct(product.getIdProduct()).name(product.getName())
            .quantity(product.getQuantity()).build();
    }
}
```

```
        Product product = productRepository.findById(id).orElseThrow(() -> new Res  
        product.setName(productDTO.getName());  
        product.setQuantity(productDTO.getQuantity());  
        product.setUnitType(productDTO.getUnitType());  
        product.setActive(productDTO.isActive());  
        product = productRepository.save(product);  
        return ProductDTO.builder().idProduct(product.getIdProduct()).name(product  
    }  
  
    public void deleteProduct(Long id) {  
        productRepository.deleteById(id);  
    }  
  
    public ProductDTO getProductById(Long id) {  
        Product product = productRepository.findById(id).orElseThrow(() -> new Res  
        return ProductDTO.builder().idProduct(product.getIdProduct()).name(product  
    }  
  
    public List<ProductDTO> getAllProducts() {  
        List<Product> products = productRepository.findAll();  
        List<ProductDTO> productDTOS = new ArrayList<>();  
        for (Product product : products) {  
            productDTOS.add(ProductDTO.builder().idProduct(product.getIdProduct()  
        }  
        return productDTOS;  
    }  
}
```



Refactor the ProductDTO class to have the same fields as Product



Here's the refactored ProductDTO class:

typescript

Copy code

```
public class ProductDTO {  
  
    private Long idProduct;  
  
    @NotNull(message = "Name cannot be null")  
    private String name;  
  
    @NotNull(message = "Quantity cannot be null")  
    private Long quantity;  
  
    @NotNull(message = "Unit type cannot be null")  
    private String unitType;  
  
    @NotNull(message = "Active flag cannot be null")  
    private boolean active;  
  
    public ProductDTO() {
```

```
}

public ProductDTO(Long idProduct, String name, Long quantity, String unitType,
    this.idProduct = idProduct;
    this.name = name;
    this.quantity = quantity;
    this.unitType = unitType;
    this.active = active;
}

public Long getIdProduct() {
    return idProduct;
}

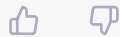
public void setIdProduct(Long idProduct) {
    this.idProduct = idProduct;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Long getQuantity() {
    return quantity;
}
```

```
public void setQuantity(Long quantity) {  
    this.quantity = quantity;  
}  
  
public String getUnitType() {  
    return unitType;  
}  
  
public void setUnitType(String unitType) {  
    this.unitType = unitType;  
}  
  
public boolean isActive() {  
    return active;  
}  
  
public void setActive(boolean active) {  
    this.active = active;  
}  
}
```



MA

Change the @NotNull Strings so it doesn't allow null, empty nor blank values. Also, make the DTO class use lombok





@Data

@NoArgsConstructor

@AllArgsConstructor

public class ProductDTO {

typescript

Copy code

```
private Long idProduct;

@NotBlank(message = "Name cannot be null, empty or blank")
private String name;

@NotNull(message = "Quantity cannot be null")
private Long quantity;

@NotBlank(message = "Unit type cannot be null, empty or blank")
private String unitType;

private boolean active;

public Product toProduct() {
    return new Product(idProduct, name, quantity, unitType, active);
}
```

```
public static ProductDTO fromProduct(Product product) {  
    return new ProductDTO(product.getIdProduct(), product.getName(), product.getQu  
        product.getUnitType(), product.isActive());  
}
```

```
}
```



Refactor the ProductService to use "toProduct" and "fromProduct" methods



Here's the updated code:

```
@Service  
public class ProductService {  
    private final ProductRepository productRepository;
```

SCSS

Copy code

```
@Autowired  
public ProductService(ProductRepository productRepository) {  
    this.productRepository = productRepository;
```

```

}

public ProductDTO createProduct(ProductDTO productDTO) {
    Product product = productDTO.toProduct();
    product = productRepository.save(product);
    return ProductDTO.fromProduct(product);
}

public ProductDTO updateProduct(ProductDTO productDTO) {
    Product product = productRepository.findById(productDTO.getIdProduct())
        .orElseThrow(() -> new ResourceNotFoundException("Product not found with id: " + productDTO.getIdProduct()));
    product.setName(productDTO.getName());
    product.setQuantity(productDTO.getQuantity());
    product.setUnitType(productDTO.getUnitType());
    product.setActive(productDTO.isActive());
    product = productRepository.save(product);
    return ProductDTO.fromProduct(product);
}

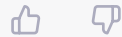
public void deleteProduct(Long id) {
    productRepository.deleteById(id);
}

public ProductDTO getProductById(Long id) {
    Product product = productRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Product not found with id: " + id));
    return ProductDTO.fromProduct(product);
}

```

```
public List<ProductDTO> getAllProducts() {  
    List<Product> products = productRepository.findAll();  
    return products.stream().map(ProductDTO::fromProduct).collect(Collectors.toList());  
}
```

```
}
```



Now remove the validations from Product class since it doesn't need anymore



Here is the refactored Product class with validations removed:

less

Copy code

```
@Entity  
@Table(name = "products")  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class Product {  
  
    @Id
```

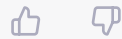
```
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long idProduct;

private String name;

private Long quantity;

private String unitType;

private boolean active;
}
```



Now do the same for the Client class. This is the class:

```
@Entity
@Table(name = "clients")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Client implements Serializable {

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

    @NotNull(message = "Name cannot be null")
```

```
private String name;

@NotNull(message = "Address cannot be null")
private String address;

@NotNull(message = "Register date cannot be null")
private Date registerDate;

@NotNull(message = "Active flag cannot be null")
private boolean active;
}
```



This is the refactored Client class without validation annotations:

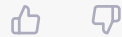
less

 Copy code

```
@Entity
@Table(name = "clients")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Client implements Serializable {

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

```
private String name;  
private String address;  
private Date registerDate;  
private boolean active;  
}
```



Refactor the ClientDTO to use the same validations and methods as the ProductDTO



Here's the updated ClientDTO class:

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class ClientDTO {  
    private Long idClient;
```

typescript

Copy code

```

@NotBlank(message = "Name cannot be null, empty or blank")
private String name;

@NotBlank(message = "Address cannot be null, empty or blank")
private String address;

@NotNull(message = "Register date cannot be null")
private Date registerDate;

@NotNull(message = "Active flag cannot be null")
private boolean active;

public Client toClient() {
    return new Client(idClient, name, address, registerDate, active);
}

public static ClientDTO fromClient(Client client) {
    return new ClientDTO(client.getIdClient(), client.getName(), client.getAddress
}

```

}



MA

Include the idClient (Long) in ClientDTO





Here's the updated ClientDTO class with the idClient field included:

less

 Copy code

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class ClientDTO {

    private Long idClient;

    @NotBlank(message = "Name cannot be null, empty or blank")
    private String name;

    @NotBlank(message = "Address cannot be null, empty or blank")
    private String address;

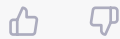
    private Date registerDate;

    private boolean active;

    public static ClientDTO fromClient(Client client) {
        return ClientDTO.builder()
            .idClient(client.getIdClient())
            .name(client.getName())
```

```
        .address(client.getAddress())
        .registerDate(client.getRegisterDate())
        .active(client.isActive())
        .build();
    }

    public Client toClient() {
        return Client.builder()
            .idClient(idClient)
            .name(name)
            .address(address)
            .registerDate(registerDate)
            .active(active)
            .build();
    }
}
```



MA

The Client class doesn't have a builder. Make the ClientDTO class create Client using "new" instead of builder



Here is the updated ClientDTO class that creates the `Client` object using `new` instead of the builder:

typescript

Copy code

```
@Data
public class ClientDTO {

    private Long idClient;

    @NotBlank(message = "Name cannot be blank")
    private String name;

    @NotBlank(message = "Address cannot be blank")
    private String address;

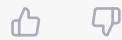
    @NotNull(message = "Register date cannot be null")
    private Date registerDate;

    @NotNull(message = "Active flag cannot be null")
    private boolean active;

    public Client toClient() {
        return new Client(idClient, name, address, registerDate, active);
    }

    public static ClientDTO fromClient(Client client) {
        return new ClientDTO(
```

```
        client.getIdClient(),  
        client.getName(),  
        client.getAddress(),  
        client.getRegisterDate(),  
        client.isActive()  
    );  
}  
}
```



MA

Just as you did in ProductService, refactor the ClientService to use DTOs. This is the ClientService class:

```
@Service  
public class ClientService {  
    private final ClientRepository clientRepository;  
  
    public ClientService(ClientRepository clientRepository) {  
        this.clientRepository = clientRepository;  
    }  
  
    public Client createClient(Client client) {  
        return clientRepository.save(client);  
    }  
  
    public Client updateClient(Client client) {  
        return clientRepository.save(client);  
    }  
}
```

```

    }

    public void deleteClient(Long id) {
        clientRepository.deleteById(id);
    }

    public List<Client> getAllClients() {
        return clientRepository.findAll();
    }

    public Client getClientById(Long id) {
        return clientRepository.findById(id).orElse(null);
    }
}

```



SCSS

Copy code

```

@Service
public class ClientService {
    private final ClientRepository clientRepository;

    public ClientService(ClientRepository clientRepository) {
        this.clientRepository = clientRepository;
    }

    public ClientDTO createClient(ClientDTO clientDTO) {

```

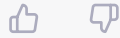
```
        Client client = clientDTO.toClient();
        client = clientRepository.save(client);
        return ClientDTO.fromClient(client);
    }

    public ClientDTO updateClient(ClientDTO clientDTO) {
        Client client = clientDTO.toClient();
        client = clientRepository.save(client);
        return ClientDTO.fromClient(client);
    }

    public void deleteClient(Long id) {
        clientRepository.deleteById(id);
    }

    public List<ClientDTO> getAllClients() {
        return clientRepository.findAll().stream()
            .map(ClientDTO::fromClient)
            .collect(Collectors.toList());
    }

    public ClientDTO getClientById(Long id) {
        return clientRepository.findById(id)
            .map(ClientDTO::fromClient)
            .orElse(null);
    }
}
```



For each endpoint, make a request example for Postman



Here are the examples of request for each endpoint in the ClientService class using Postman:

1. Create a client:

Method: POST

URL: <http://localhost:8080/api/clients>

Request Body (in JSON format):

```
{  
  "name": "John Doe",  
  "address": "123 Main St",  
  "registerDate": "2022-01-01",  
  "active": true  
}
```

2. Update a client:

Method: PUT

URL: <http://localhost:8080/api/clients/{id}>

Request Body (in JSON format):

```
{
```

```
"idClient": {id},
"name": "Jane Doe",
"address": "456 Main St",
"registerDate": "2022-02-01",
"active": true
}
```

3. Delete a client:

Method: DELETE

URL: <http://localhost:8080/api/clients/{id}>

4. Get all clients:

Method: GET

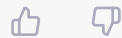
URL: <http://localhost:8080/api/clients>

5. Get a client by id:

Method: GET

URL: <http://localhost:8080/api/clients/{id}>

Note: In the URL, replace {id} with the actual id of the client you want to operate on.



Now to the same for the products





Here are examples for each endpoint for product resource in Postman:

1. Get all products:
  - Method: GET
  - URL: <http://localhost:8080/api/products>
  - No Body needed
2. Get product by id:
  - Method: GET
  - URL: <http://localhost:8080/api/products/{id}>
  - No Body needed, replace {id} with the actual product id
3. Create a product:
  - Method: POST
  - URL: <http://localhost:8080/api/products>
  - Body (raw, JSON):

json

Copy code

```
{
  "name": "Product name",
  "quantity": 10,
  "unitType": "Unit type",
  "active": true
}
```

4. Update a product:
  - Method: PUT
  - URL: <http://localhost:8080/api/products/{id}>

- Body (raw, JSON):

```
json Copy code  
  
{  
  "name": "Product name updated",  
  "quantity": 20,  
  "unitType": "Unit type updated",  
  "active": false  
}
```

Replace {id} with the actual product id

5. Delete a product:

- Method: DELETE
- URL: <http://localhost:8080/api/products/{id}>
- No Body needed, replace {id} with the actual product id



The endpoints for updating and deleting products and clients must validate if the ID is null. Refactor the methods inside their services to validate it before querying the database



Here's the updated `ProductService` class:

java

 Copy code

```
@Service
public class ProductService {
    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public ProductDTO createProduct(ProductDTO productDTO) {
        Product product = productDTO.toProduct();
        product = productRepository.save(product);
        return ProductDTO.fromProduct(product);
    }

    public ProductDTO updateProduct(Long id, ProductDTO productDTO) {
        if (id == null) {
            throw new IllegalArgumentException("Product ID cannot be null");
        }
        Product product = productRepository.findById(id).orElse(null);
        if (product == null) {
            throw new EntityNotFoundException("Product not found with id " + id);
        }
        product = productDTO.updateProduct(product);
        product = productRepository.save(product);
        return ProductDTO.fromProduct(product);
    }
}
```

```

public void deleteProduct(Long id) {
    if (id == null) {
        throw new IllegalArgumentException("Product ID cannot be null");
    }
    if (!productRepository.existsById(id)) {
        throw new EntityNotFoundException("Product not found with id " + id);
    }
    productRepository.deleteById(id);
}

public List<ProductDTO> getAllProducts() {
    List<Product> products = productRepository.findAll();
    return products.stream().map(ProductDTO::fromProduct).collect(Collectors.toList());
}

public ProductDTO getProductById(Long id) {
    if (id == null) {
        throw new IllegalArgumentException("Product ID cannot be null");
    }
    Product product = productRepository.findById(id).orElse(null);
    if (product == null) {
        throw new EntityNotFoundException("Product not found with id " + id);
    }
    return ProductDTO.fromProduct(product);
}
}

```

And here's the updated `ClientService` class:

java

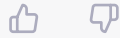
 Copy code

```
@Service
public class ClientService {
    private final ClientRepository clientRepository;

    public ClientService(ClientRepository clientRepository) {
        this.clientRepository = clientRepository;
    }

    public ClientDTO createClient(ClientDTO clientDTO) {
        Client client = clientDTO.toClient();
        client = clientRepository.save(client);
        return ClientDTO.fromClient(client);
    }

    public ClientDTO updateClient(Long id, ClientDTO clientDTO) {
        if (id == null) {
            throw new IllegalArgumentException("Client ID cannot be null");
        }
        Client client = clientRepository.findById(id).orElse(null);
        if (client == null) {
            throw new EntityNotFoundException("Client not found with id " + id);
        }
        client = clientDTO.updateClient(client);
        client = clientRepository.save(client);
        return ClientDTO.fromClient(client);
    }
}
```



MA

Create the method "updateProduct" inside ProductDTO as you suggested



Here's the updated `ProductDTO` class with the `updateProduct` method:

typescript

Copy code

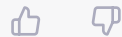
```
@Data
public class ProductDTO {
    private Long idProduct;
    @NotBlank(message = "Name cannot be null, empty, or blank")
    private String name;
    @NotNull(message = "Quantity cannot be null")
    private Long quantity;
    @NotBlank(message = "Unit type cannot be null, empty, or blank")
    private String unitType;
    private boolean active;

    public Product toProduct() {
        return new Product(idProduct, name, quantity, unitType, active);
    }

    public static ProductDTO fromProduct(Product product) {
```

```
        return new ProductDTO(product.getIdProduct(), product.getName(), product.getUnitType(), product.isActive());
    }

    public void updateProduct(Product product) {
        product.setName(this.name);
        product.setQuantity(this.quantity);
        product.setUnitType(this.unitType);
        product.setActive(this.active);
    }
}
```



Instead of having a parameter for the ID, validate if the product exists using the DTO's ID



Here is the refactored `ProductService` class to validate the existence of a product using the `ProductDTO`'s ID:

java

Copy code

```
@Service
public class ProductService {
    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public ProductDTO createProduct(ProductDTO productDTO) {
        Product product = productDTO.toProduct();
        product = productRepository.save(product);
        return ProductDTO.fromProduct(product);
    }

    public ProductDTO updateProduct(ProductDTO productDTO) {
        if (productDTO.getIdProduct() == null) {
            throw new IllegalArgumentException("Product ID cannot be null");
        }
        Product product = productRepository.findById(productDTO.getIdProduct())
            .orElseThrow(() -> new IllegalArgumentException("Product not found"));
        product.updateProduct(productDTO);
        product = productRepository.save(product);
        return ProductDTO.fromProduct(product);
    }

    public void deleteProduct(Long id) {
        if (id == null) {
            throw new IllegalArgumentException("Product ID cannot be null");
        }
    }
}
```



```
        productRepository.deleteById(id);
    }

    public List<ProductDTO> getAllProducts() {
        List<Product> products = productRepository.findAll();
        return products.stream().map(ProductDTO::fromProduct).collect(Collectors.toList());
    }

    public ProductDTO getProductById(Long id) {
        if (id == null) {
            throw new IllegalArgumentException("Product ID cannot be null");
        }
        Product product = productRepository.findById(id).orElse(null);
        return product != null ? ProductDTO.fromProduct(product) : null;
    }
}
```



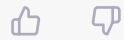
Create the method "updateProduct" inside the Product class



kotlin

Copy code

```
public Product updateProduct(ProductDTO productDTO) {  
    this.name = productDTO.getName();  
    this.quantity = productDTO.getQuantity();  
    this.unitType = productDTO.getUnitType();  
    this.active = productDTO.isActive();  
    return this;  
}
```



Make it return void

---



kotlin

Copy code

```
public void updateProduct(ProductDTO productDTO) {  
    this.name = productDTO.getName();  
    this.quantity = productDTO.getQuantity();  
    this.unitType = productDTO.getUnitType();  
    this.active = productDTO.isActive();  
}
```



Now refactor the ClientService class to validate the client ID. This is the ClientService class:

@Service

```
public class ClientService {  
    private final ClientRepository clientRepository;  
  
    public ClientService(ClientRepository clientRepository) {  
        this.clientRepository = clientRepository;  
    }  
  
    public ClientDTO createClient(ClientDTO clientDTO) {  
        Client client = clientDTO.toClient();  
        client = clientRepository.save(client);  
        return ClientDTO.fromClient(client);  
    }  
}
```

```

public ClientDTO updateClient(ClientDTO clientDTO) {
    Client client = clientDTO.toClient();
    client = clientRepository.save(client);
    return ClientDTO.fromClient(client);
}

public void deleteClient(Long id) {
    clientRepository.deleteById(id);
}

public List<ClientDTO> getAllClients() {
    return clientRepository.findAll().stream()
        .map(ClientDTO::fromClient)
        .collect(Collectors.toList());
}

public ClientDTO getClientById(Long id) {
    return clientRepository.findById(id)
        .map(ClientDTO::fromClient)
        .orElse(null);
}
}

```



@Service

```
public class ClientService {
```

```
private final ClientRepository clientRepository;
```

typescript

 Copy code

```
public ClientService(ClientRepository clientRepository) {
    this.clientRepository = clientRepository;
}

public ClientDTO createClient(ClientDTO clientDTO) {
    Client client = clientDTO.toClient();
    client = clientRepository.save(client);
    return ClientDTO.fromClient(client);
}

public ClientDTO updateClient(ClientDTO clientDTO) {
    if (clientDTO.getIdClient() == null) {
        throw new IllegalArgumentException("Client ID must not be null");
    }
    Client existingClient = clientRepository.findById(clientDTO.getIdClient()).orElse(null);
    if (existingClient == null) {
        throw new EntityNotFoundException("Client not found");
    }
    existingClient.updateClient(clientDTO.toClient());
    return ClientDTO.fromClient(existingClient);
}

public void deleteClient(Long id) {
    if (id == null) {
        throw new IllegalArgumentException("Client ID must not be null");
    }
}
```

```
    }  
    clientRepository.deleteById(id);  
}  
  
public List<ClientDTO> getAllClients() {  
    return clientRepository.findAll().stream()  
        .map(ClientDTO::fromClient)  
        .collect(Collectors.toList());  
}  
  
public ClientDTO getClientById(Long id) {  
    if (id == null) {  
        throw new IllegalArgumentException("Client ID must not be null");  
    }  
    return clientRepository.findById(id)  
        .map(ClientDTO::fromClient)  
        .orElse(null);  
}
```

```
}
```





Create the method "updateClient" inside Client



kotlin

Copy code

```
public void updateClient(Client updatedClient) {  
    this.name = updatedClient.getName();  
    this.address = updatedClient.getAddress();  
    this.registerDate = updatedClient.getRegisterDate();  
    this.active = updatedClient.isActive();  
}
```



Refactor the following class to consider the possible exceptions from the classes above:

@ControllerAdvice

public class ControllerExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)

    public ResponseEntity<ApiError>

handleMethodArgumentNotValidException(MethodArgumentNotValidException ex) {

        BindingResult bindingResult = ex.getBindingResult();

```

        List<FieldError> fieldErrors = bindingResult.getFieldErrors();

        ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, "Validation
Failed", fieldErrors);
        return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
    }
}

```



@ControllerAdvice

```
public class ControllerExceptionHandler {
```

java

Copy code

```

@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ApiError> handleMethodArgumentNotValidException(MethodArgumentNotValidException ex) {
    BindingResult bindingResult = ex.getBindingResult();
    List<FieldError> fieldErrors = bindingResult.getFieldErrors();

    ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, "Validation Failed", fieldErrors);
    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
}

@ExceptionHandler(NullPointerException.class)
public ResponseEntity<ApiError> handleNullPointerException(NullPointerException ex) {
    ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, "Required fields cannot be null");
    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
}

```



```
}

@ExceptionHandler(NoSuchElementException.class)
public ResponseEntity<ApiError> handleNoSuchElementException(NoSuchElementException e) {
    ApiError apiError = new ApiError(HttpStatus.NOT_FOUND, "Entity not found", Collections.emptyList());
    return new ResponseEntity<>(apiError, HttpStatus.NOT_FOUND);
}
```

```
}
```



What about the exceptions inside the services classes?



To handle exceptions inside the service classes, you can add try-catch blocks in the methods. For example:

,



Handle IllegalArgumentException with ExceptionHandler too



You can add another `@ExceptionHandler` method to handle `IllegalArgumentException` exceptions in the same `ControllerExceptionHandler` class:

java

Copy code

```
@ExceptionHandler(IllegalArgumentException.class)
public ResponseEntity<ApiError> handleIllegalArgumentException(IllegalArgumentException ex) {
    ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, ex.getMessage(), null);
    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
}
```

This method will catch `IllegalArgumentException` exceptions thrown from your service classes and create an `ApiError` response with a `BAD_REQUEST` status code.



Refactor the following classes to return a list of errors in String format instead of a list of FieldErrors:

@Data

public class ApiError {

private HttpStatus status;

```

private String message;
private List<FieldError> errors;

public ApiError(HttpStatus status, String message, List<FieldError> errors) {
    super();
    this.status = status;
    this.message = message;
    this.errors = errors;
}

public ApiError(HttpStatus status, String message) {
    super();
    this.status = status;
    this.message = message;
}
}

@ControllerAdvice
public class ControllerExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ApiError>
handleMethodArgumentNotValidException(MethodArgumentNotValidException
ex) {
        BindingResult bindingResult = ex.getBindingResult();
        List<FieldError> fieldErrors = bindingResult.getFieldErrors();

        ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, "Validation
Failed", fieldErrors);

```

```

        return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(NullPointerException.class)
    public ResponseEntity<ApiError>
    handleNullPointerException(NullPointerException ex) {
        ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, "Required fields
        cannot be null", Collections.emptyList());
        return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(NoSuchElementException.class)
    public ResponseEntity<ApiError>
    handleNoSuchElementException(NoSuchElementException ex) {
        ApiError apiError = new ApiError(HttpStatus.NOT_FOUND, "Entity not found",
        Collections.emptyList());
        return new ResponseEntity<>(apiError, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<ApiError>
    handleIllegalArgumentException(IllegalArgumentException ex) {
        ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, ex.getMessage(),
        null);
        return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
    }
}

```



@Data

public class ApiError {

kotlin

Copy code

```
private HttpStatus status;  
private String message;  
private List<String> errors;  
  
public ApiError(HttpStatus status, String message, List<String> errors) {  
    this.status = status;  
    this.message = message;  
    this.errors = errors;  
}  
  
public ApiError(HttpStatus status, String message) {  
    this.status = status;  
    this.message = message;  
}
```

}

@ControllerAdvice

public class ControllerExceptionHandler {

java

 Copy code

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ApiError> handleMethodArgumentNotValidException(MethodArgumentNotValidException ex) {
    BindingResult bindingResult = ex.getBindingResult();
    List<FieldError> fieldErrors = bindingResult.getFieldErrors();
    List<String> errors = fieldErrors.stream().map(FieldError::getDefaultMessage).collect(Collectors.toList());

    ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, "Validation Failed", errors);
    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
}

@ExceptionHandler(NullPointerException.class)
public ResponseEntity<ApiError> handleNullPointerException(NullPointerException ex) {
    ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, "Required fields cannot be null");
    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
}

@ExceptionHandler(NoSuchElementException.class)
public ResponseEntity<ApiError> handleNoSuchElementException(NoSuchElementException ex) {
    ApiError apiError = new ApiError(HttpStatus.NOT_FOUND, "Entity not found", Collections.singletonList(ex.getMessage()));
    return new ResponseEntity<>(apiError, HttpStatus.NOT_FOUND);
}

@ExceptionHandler(IllegalArgumentException.class)
public ResponseEntity<ApiError> handleIllegalArgumentException(IllegalArgumentException ex) {
    ApiError apiError = new ApiError(HttpStatus.BAD_REQUEST, ex.getMessage(), Collections.singletonList(ex.getMessage()));
    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
}
```

```
return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);  
}
```

```
}
```

