

# Desafio Orange Talents - ZUP

---

Projeto: Orange Bank

Desenvolvido por: Wagner Andrade

## O que você vai precisar

- [JDK 11](#)
- Spring 2.4.1
- Spring Data JPA
- Spring Validation
- Spring Devtools
- Maven
- [Insomnia](#)
- [Docker](#)
- [Lombok](#)
- [MapStruct](#)
- [MySQL 8+](#). Vamos usar o docker para executar o bando de dados como um [contêiner](#).
- [MySQL Driver](#)
- IDE favorita ([Intellij IDEA](#))

Este projeto está o Github, fique a vontade para consultar 😊

<https://github.com/WagnerpbAndrade/orange-bank>

## 1. Sobre as tecnologias

### 1.1 JDK 11

Para o desenvolvimento foi escolhido java 11 (ou poderia ser Kotlin), por ser uma versão mais "recente" e ser LTS (Long Term Support). Apesar de ter versões mais novas a cada 6 meses, utilizar a última versão LTS, te garante que as features implementadas neste versão permanecerão sem mudanças contantes.

### 1.2 Spring

Spring é o framework mais famoso do mundo Java. Devido a sua capacidade de produtividade, velocidade e simplicidade para desenvolver aplicações robustas. Com bibliotecas flexíveis e testadas, garante uma qualidade na entrega das soluções. Devido a isso, algumas bibliotecas foram escolhidas para o desenvolvimento do projeto.

#### 1.2.1 Spring Data JPA

Spring Data JPA, é uma abstração com o objeto de tornar mais fácil a implementação de repositórios baseados em JPA. Melhorando a implementação de camadas de acesso a dados.

Porém, é bom lembrar que o Spring Data, é apenas uma abstração, portando não é um provedor JPA, que fica a cargo do Hibernate por padrão ou outros (como Eclipse Link, por exemplo). Com isso, é importante saber escolher quando e como aplicar se baseando nas suas desvantagens (pois as vantagens já conhecemos, porém no longo prazo as desvantagens terão mais peso), como dificultar a criação de queries complexas, tornando verboso e até mesmo difícil de entender.

Para utilizar o Spring Data, é muito simples, basta criar uma interface que estenda a classe `JpaRepository<Entidade, Tipo_ID>`, como na imagem a seguir.

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
  
    Customer findByCpf(String cpf);  
  
    Customer findByEmail(String email);  
}
```

Com isso, ela já traz muitos métodos implementados, inclusive paginação e ordenação. E caso precise, você pode criar métodos específicos, como é o caso dos métodos: **findByCpf** e **findByEmail**.

### 1.2.2 Spring Validation

Spring Validation, provê anotações para campos das entidades, que facilitam a validação durante a manipulação desse dado. Na imagem a seguir, podemos observar alguns dessas validações e como é simples utilizá-las.

```
@NotEmpty(message = "Name cannot be empty")  
private String name;  
  
@Email(message = "Email cannot be invalid")  
@NotEmpty(message = "Email cannot be empty")  
private String email;  
  
@CPF(message = "CPF cannot be invalid")  
@NotNull(message = "CPF cannot be null")  
private String cpf;
```

Importante lembrar, que para as validações serem chamadas pelo Spring, é necessário anotar os parâmetros com **@Valid** nos endpoints onde serão chamadas, como podemos ver a seguir.

```
@PostMapping  
public ResponseEntity<CustomerDTO> insert(@Valid @RequestBody CustomerPostRequestDTO postRequestDTO) {  
    return new ResponseEntity(this.service.save(postRequestDTO), HttpStatus.CREATED);  
}
```

Analisando esse método, ele recebe um objeto no corpo da requisição (anotado com **@RequestBody**), e antes é anotado com **@Valid**, isso faz com que antes de chamar o service, o Spring valide os campos anotados antes de prosseguir, caso falhe em alguma validação, ele retorna um erro para o cliente.

Além disso, essa biblioteca permite que seja possível personalizar novas verificações, como foi o caso de validações personalizadas para evitar CPF e Email duplicados.

```
@Constraint(validatedBy = InsertValidatorImpl.class)
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface InsertValidator {

    String message() default "Validation error";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

Declaramos uma interface (InsertValidator) e dizemos no **@Constraint**, qual classe irá implementar essa validação.

```
14 @RequiredArgsConstructor
15 public class InsertValidatorImpl implements ConstraintValidator<InsertValidator, CustomerPostRequestDTO> {
16
17     private final CustomerRepository repository;
18
19     @Override
20     public boolean isValid(CustomerPostRequestDTO objDTO, ConstraintValidatorContext context) {
21         List<ValidationExceptionDetails> list = new ArrayList<>();
22
23         if (objDTO.getBirthdate() == null) {
24             list.add(new ValidationExceptionDetails( fields: "birthdate", fieldsMessage: "Birthdate cannot be null"));
25         }
26
27         Customer customer = this.repository.findByCpf(objDTO.getCpf());
28         if (customer != null) {
29             list.add(new ValidationExceptionDetails( fields: "cpf", fieldsMessage: "CPF already exists"));
30         }
31
32         customer = this.repository.findByEmail(objDTO.getEmail());
33         if (customer != null) {
34             list.add(new ValidationExceptionDetails( fields: "email", fieldsMessage: "E-mail already exists"));
35         }
36
37         for (ValidationExceptionDetails v : list) {
38             context.disableDefaultConstraintViolation();
39             context.buildConstraintViolationWithTemplate(v.getFieldsMessage()).addPropertyNode(v.getFields())
40                 .addConstraintViolation();
41         }
42         return list.isEmpty();
43     }
```

Aqui, estamos utilizando o **@RequiredArgsConstructor**, essa anotação é para injeção de dependência via construtor, o que é mais indicado que usar a anotação **@Autowired** (gerenciada pelo spring para fazer injeção de dependência). Mas para funcionar é necessário que a variável seja final.

Na linha 14, devemos implementar o a interface **ConstraintValidator**, tipando com a interface e a entidade a ser validada.

Essa interface, provê um método **isValid**, que é invocado sempre que usamos a anotação **@Valid** na nossa **CustomerController**, como já mencionamos.

Nas linhas 24 e 29, é feito uma verificação para saber se já existe algum objeto com o mesmo CPF ou Email no banco de dados. Caso exista, uma **ValidationExceptionDetails** é criada.

Na linha 33, devemos percorrer a lista de exceções e caso haja, desabilitar a validação padrão, para que o padrão da validação seja a nossa, quando for gerida pelo Spring.

### 1.2.3 Spring Devtools

Essa biblioteca, ajuda na inicialização mais rápida do sistema, com o hot reload durante o desenvolvimento, que melhora muito quando se faz modificações pontuais no código sem precisar reinicializar toda a aplicação, apenas a classe modificada.

### 1.3 Docker

Não é o objetivo mostrar como instalar e configurar o docker, mas a sua utilização é devido ao fato dele permitir que você use pedaços de software como processo, sem precisar instalar na sua máquina ou emular todo um sistema operacional. Para isso, existe um repositório chamado Docker Hub, onde possui milhares de imagens que podem ser utilizadas, no nosso caso, foi utilizado a imagem oficial do MySQL.

Para tornar mais fácil gerenciar essas imagens, foi criado um arquivo chamado docker-compose.yml, onde é possível configurar as imagens que a aplicação necessita.

A seguir mostro como ficou meu arquivo docker-compose.yml.

```
1  version: '3.1'
2
3  services:
4    db:
5      image: mysql
6      container_name: mysql
7      environment:
8        MYSQL_ROOT_PASSWORD: root
9      ports:
10     - "3306:3306"
11     volumes:
12       - spring_data:/var/lib/mysql
13       mem_limit: 1024m
14
15  volumes:
16    spring_data:
```

Neste arquivo, temos na linha:

1. versão do arquivo compose
2. declaração do serviço que vamos usar (services)
3. serviço utilizado (db)
4. qual a imagem do docker hub usaremos (mysql)
5. qual o nome será usado para identificar esse processo na máquina (mysql)
6. ambiente
7. variáveis de ambiente
8. mapeamento das portas, neste caso, estou dizendo que a porta da minha máquina 3306 irá se comunicar com a porta 3306 dentro do contêiner.
9. volumes, isso é para que os dados persistidos dentro do contêiner não se percam, pois por padrão sempre que o processo morre, ele limpa todos os dados.
10. limite de memória que o contêiner poderá utilizar, isso é importante para não ter desperdício de recurso. Mas em ambiente cloud, é possível gerenciar de forma automática usando Kubernetes, mas isso é assunto para outra hora.

Para iniciar o contêiner, basta abrir um terminal na sua IDE e digitar o comando: `docker-compose up`. Com isso, na primeira vez ela baixará a imagem para sua máquina e subirá seu bando de dados, tornando-o disponível para sua aplicação utilizar.

## 1.4 Lombok

Decidi usar o Lombok, pois ele diminui a verbosidade de se criar getters/setters, construtores, builder e etc.

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

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

    @NotEmpty(message = "Name cannot be empty")
    private String name;

    @Email(message = "Email cannot be invalid")
    @NotEmpty(message = "Email cannot be empty")
    private String email;

    @CPF(message = "CPF cannot be invalid")
    @NotNull(message = "CPF cannot be null")
    private String cpf;

    private LocalDate birthdate;
}
```

Vamos detalhar:

1. `@Data`: Possui as seguintes características: `@Getter/@Setter`, `@ToString`, `@EqualsAndHashCode`;
2. `@NoArgsConstructor`: Define um construtor padrão vazio;
3. `@AllArgsConstructor`: Cria todas as combinações possíveis.

Obs.: Para utilizar o Lombok, muitas vezes é necessário habilitar um plugin na sua IDE, atente-se a isso.

## 1.5 MapStruct

Para fazer o mapeamento das entidades para DTO e vice-versa, decidi utilizar o MapStruct, que gera esse código a partir da interface definida com seus métodos.

```
@Mapper(componentModel = "orange")
public interface CustomerMapper {

    CustomerMapper INSTANCE = Mappers.getMapper(CustomerMapper.class);

    Customer toCustomer(CustomerDTO customerDTO);

    Customer toCustomer(CustomerPostRequestDTO postRequestDTO);

    CustomerDTO toCustomerDTO(Customer customer);
}
```

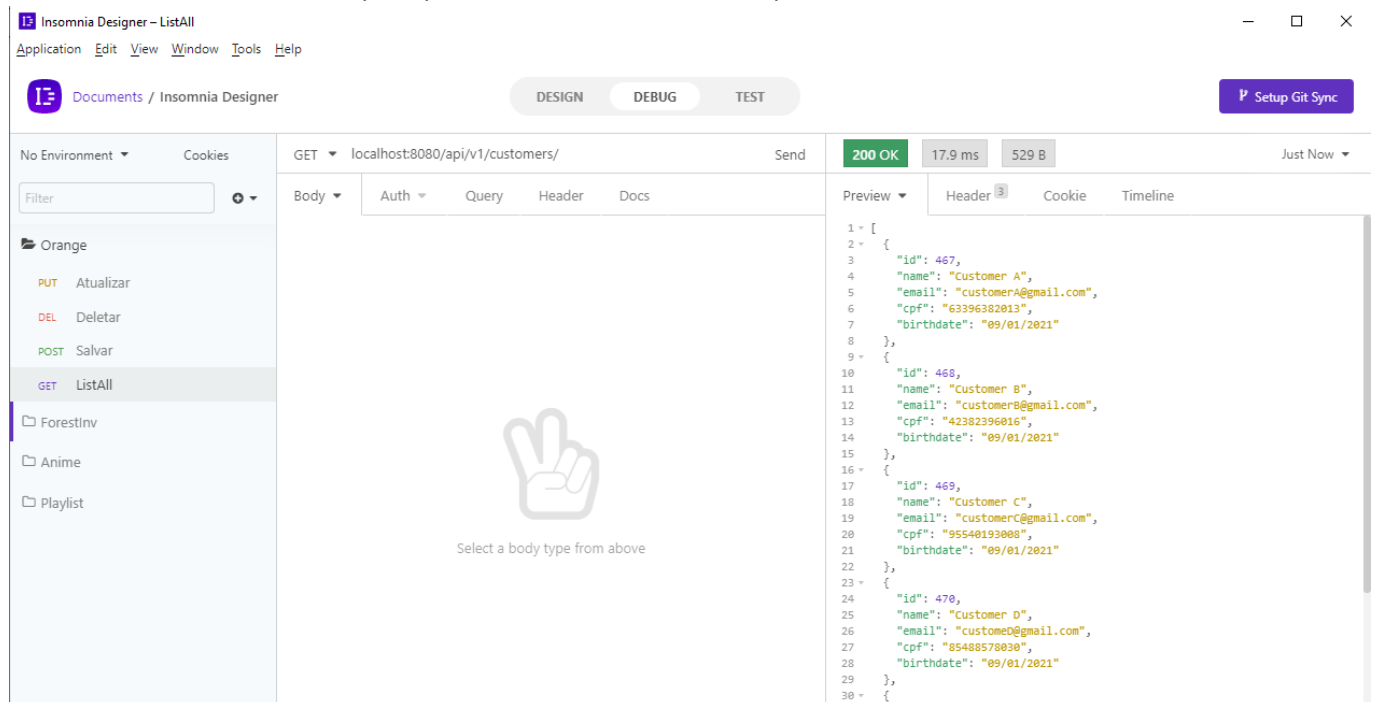
Obs.: Sempre que mudar alguma atributo da classe, delete a pasta target do se projeto, pois assim você terá certeza que ao gerar o novo código, estará com as modificações feitas.

## 1.6 MySQL

Para esse projeto foi escolhido utilizar o MySQL, um banco de dados relacional.

## 1.7 Insomnia

Essa ferramenta é muito útil para podermos testar nossos endpoints.



## 2 Arquitetura

Do mesmo modo que precisamos de uma planta baixa na hora de construir uma casa ou um prédio, é necessário também quando se vai projetar um sistema. Pois uma arquitetura bem definida é essencial para evolução, manutenção e para o desenvolvimento.

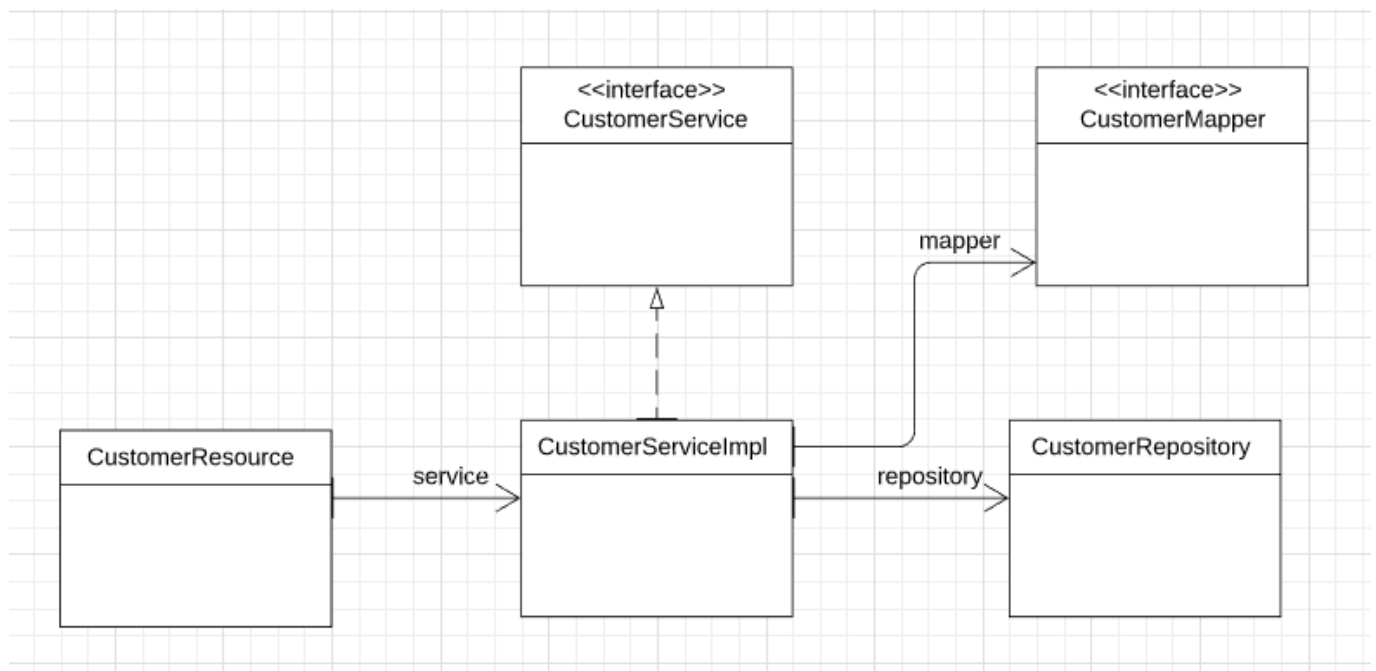
Porém, não é necessário criar uma super arquitetura, para projetos pequenos, sem escala. Pois devemos analisar cada caso, como é o caso do Orange Bank. Para esse projeto, a arquitetura segue um padrão MVC (sem utilizar a View, pois se trata de uma API), que é o suficiente para atender o objetivo.

## 2.1 Implementação

Para iniciar o desenvolvimento, primeiro identifiquei quais rotas eu precisava ter disponível na aplicação. Após isso, comecei a criação da model e das classes de transporte que serão usadas nas requisições. Com as rotas e model criadas, então o processo foi criar do mais interno para o mais externo, logo, iniciei pela CustomerRepository -> CustomerService -> CustomerMapper -> CustomerServiceImpl -> CustomerResource. Após isso, iniciei a criação das exceções personalizadas e validações e por fim e não mesmo importante, a criação dos testes.

Uma metodologia interessante de se aplicar, que não foi o caso aqui, é desenvolver utilizando TDD, onde você cria os testes antes da implementação.

## 2.2 Diagrama de Classe



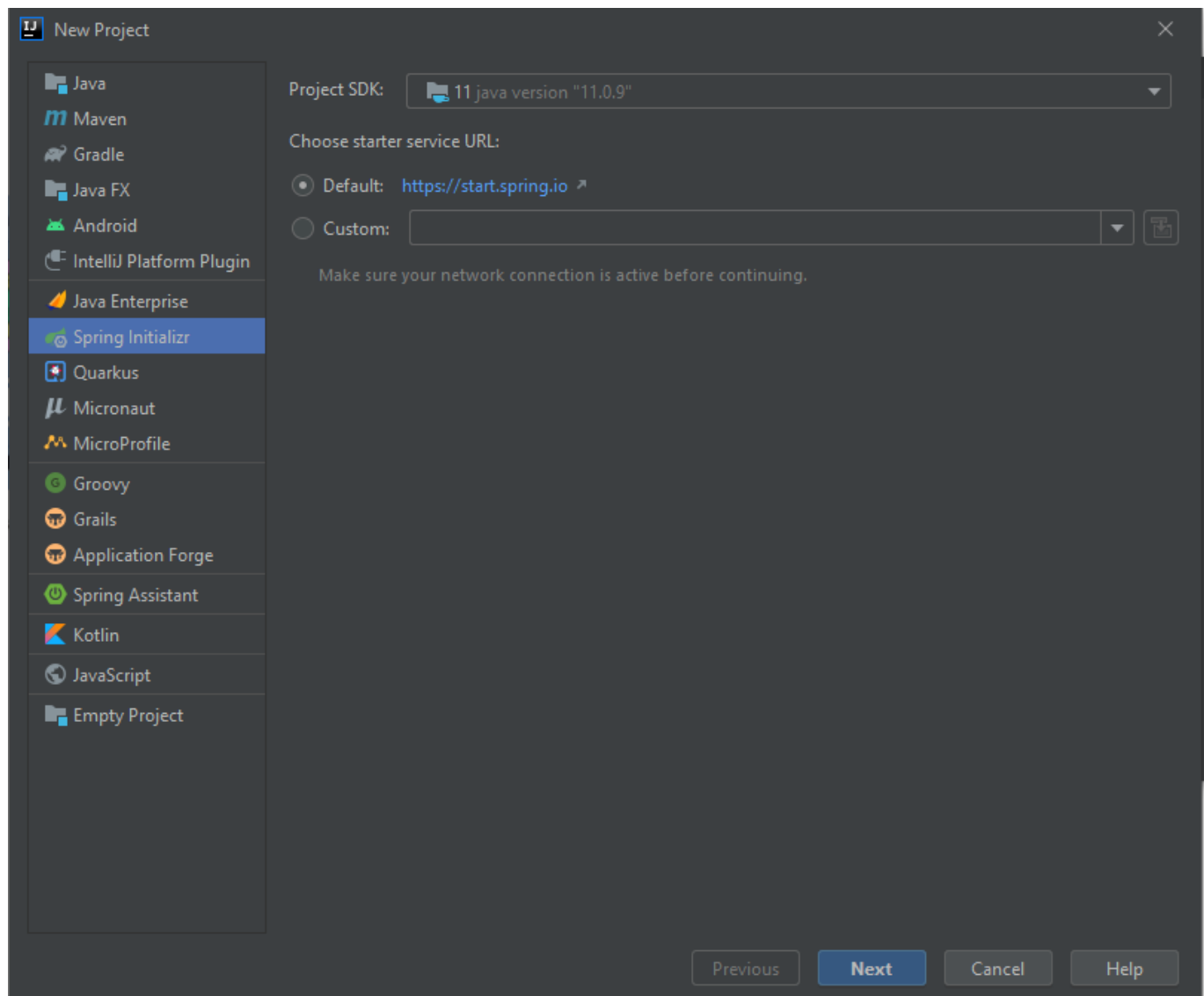
Para o desenvolvimento desse projeto, foi pensado nas seguintes classes:

1. CustomerRepository: Será responsável por fazer o CRUD no banco de dados. Lembrando que essa classe implementa a JpaRepository, podemos assim utilizar todos os métodos da classe.
2. CustomerService: Nossa interface de contrato para a Service, definindo quais métodos precisamos implementar.
3. CustomerServiceImpl: Implementação de fato dos métodos e das regras de negócios que precisamos atender para o correto funcionamento do sistema.
4. CustomerResource: Responsável pelas rotas (endpoints), é por essa classe que será feita a nossa entrada e saída de dados, de acordo com cada rota definida.

## 3 Criando o projeto do início


### 3.1 Spring Initializr

Para criar o projeto, utilizei o Spring Initializr do IntelliJ IDEA. Outra alternativa é criar pelo próprio site do [Spring](#). Após selecionar as dependências, será baixado um zip, e aí só importar na sua IDE.



### 3.2 Nome do projeto



 New Project ✕

### Spring Initializr Project Settings

Group:

Artifact:

Type: ☒ Maven ☐ Gradle

Language: ☒ Java ☐ Kotlin ☐ Groovy

Packaging: ☒ Jar ☐ War

Java Version:

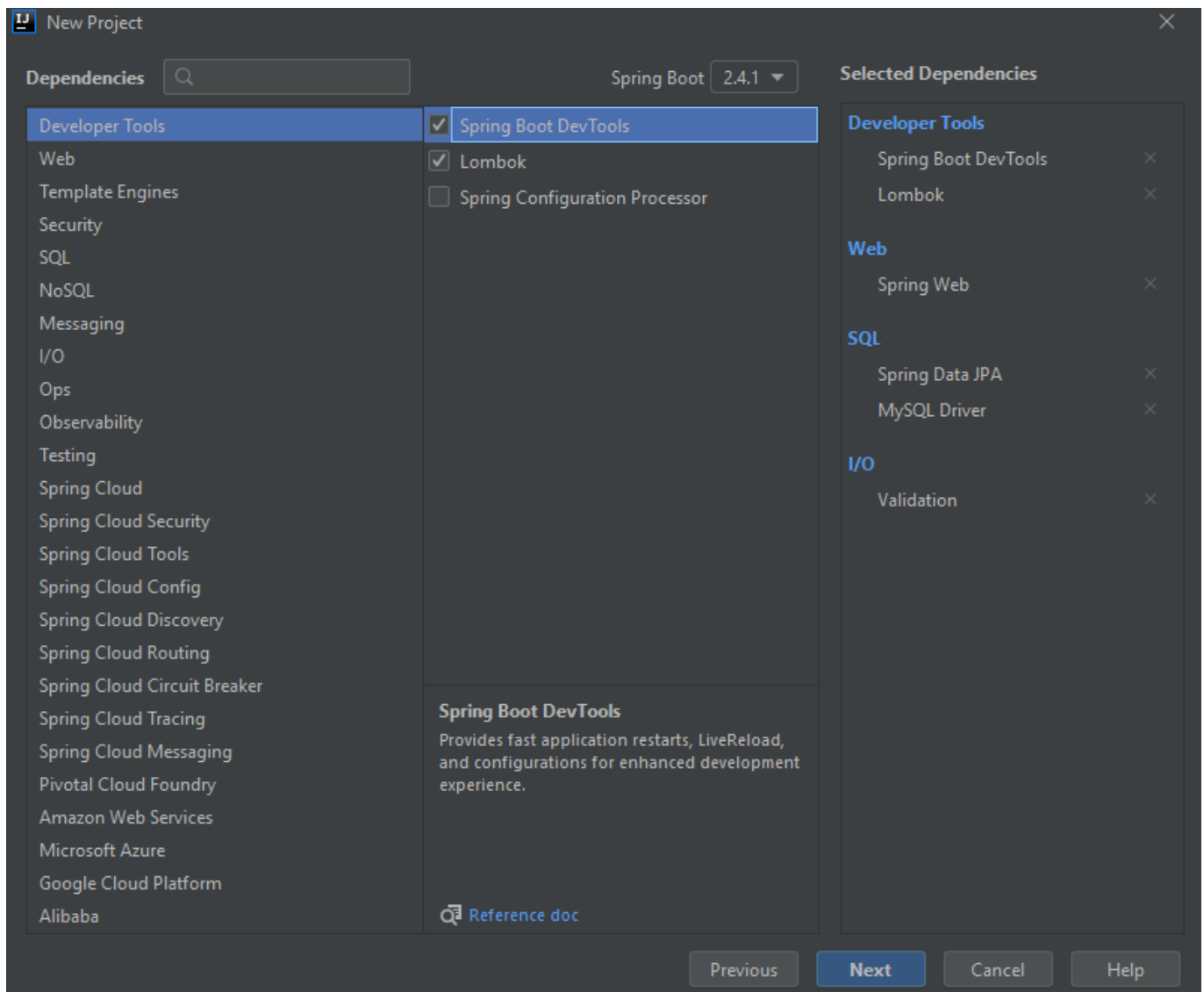
Version:

Name:

Description:

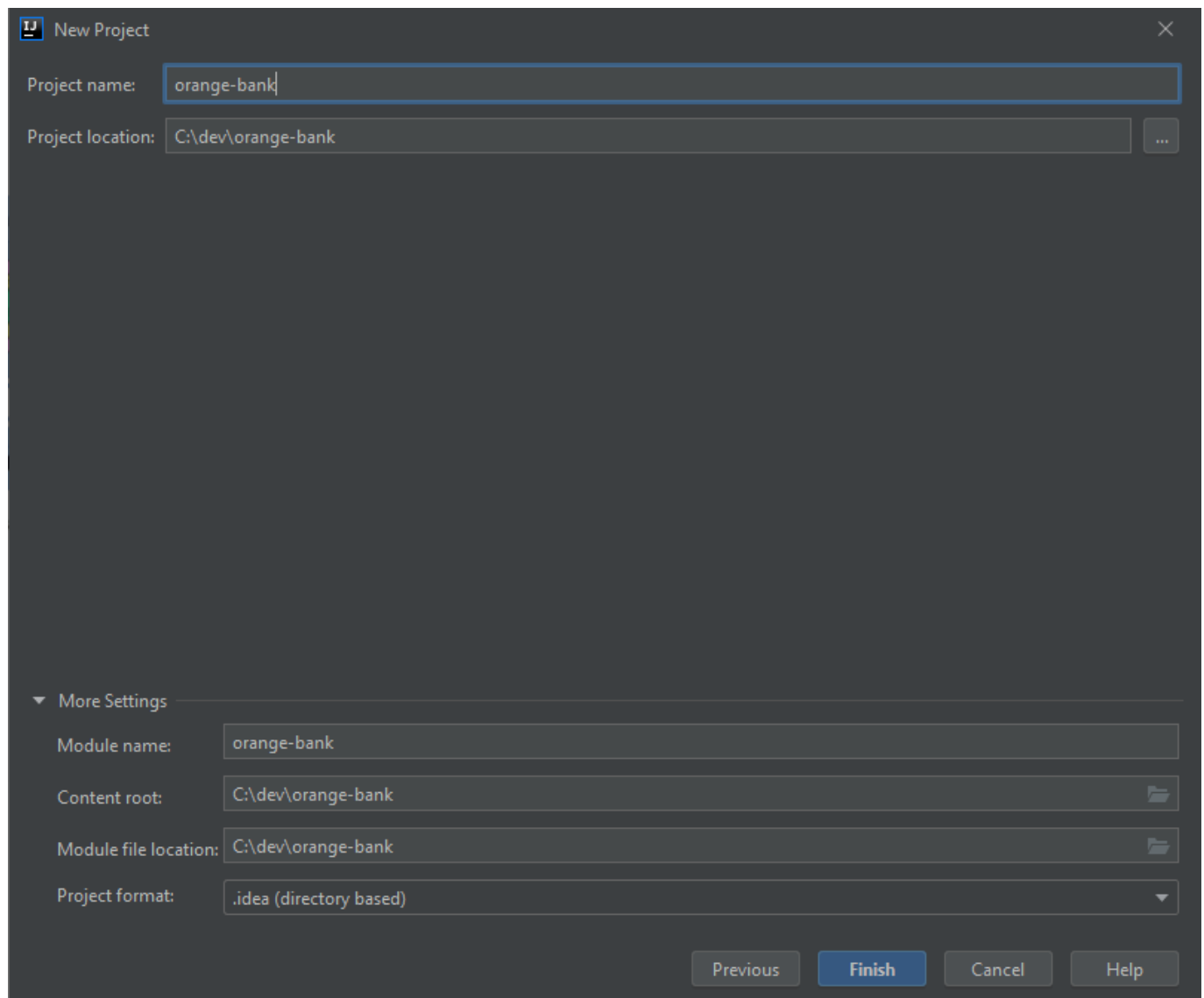
Package:

## 3.2 Dependências



O MapStruct, precisa se adicionado a parte, pode ser encontrado no site oficial ou nos repositórios do Maven. Encorajo olhar a documentação, tem muitas coisas legais.

### 3.3 Nome do projeto



New Project

Project name: orange-bank

Project location: C:\dev\orange-bank

More Settings

Module name: orange-bank

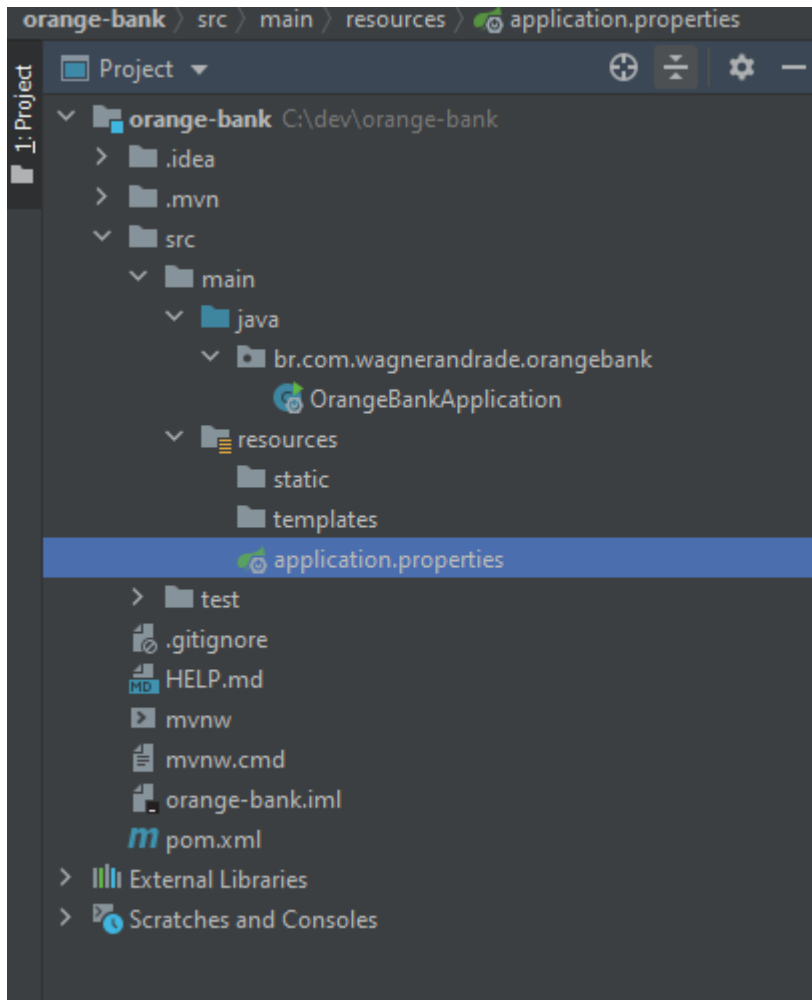
Content root: C:\dev\orange-bank

Module file location: C:\dev\orange-bank

Project format: .idea (directory based)

Previous Finish Cancel Help

E agora só finalizar e você terá a seguinte estrutura.



### 3.4 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.1</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>br.com.wagnerandrade</groupId>
  <artifactId>orange-bank</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>orange-bank</name>
  <description>Desafio Orange Talents</description>

  <properties>
    <java.version>11</java.version>
    <org.mapstruct.version>1.4.1.Final</org.mapstruct.version>
  </properties>
```

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

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>

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

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>

  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>

  <dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>${org.mapstruct.version}</version>
  </dependency>

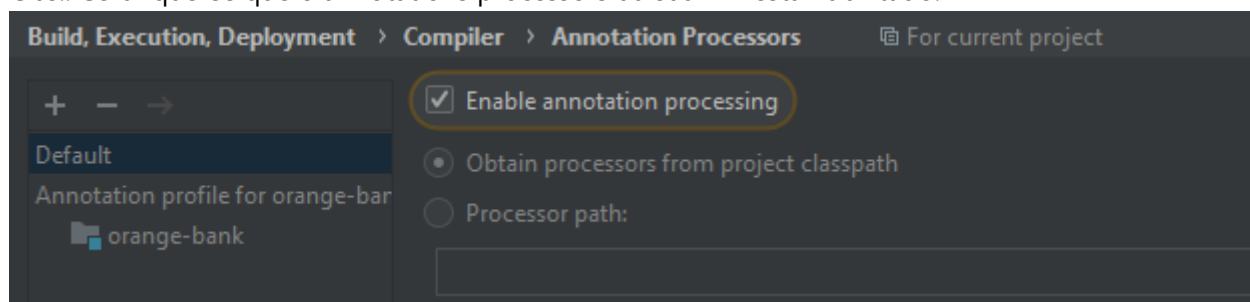
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
    <plugin>
```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>${maven-compiler-plugin.version}</version>
<configuration>
  <source>11</source>
  <target>11</target>
  <annotationProcessorPaths>
    <path>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>${lombok.version}</version>
    </path>
    <path>
      <groupId>org.mapstruct</groupId>
      <artifactId>mapstruct-processor</artifactId>
      <version>${org.mapstruct.version}</version>
    </path>
  </annotationProcessorPaths>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

Neste arquivo encontra-se todas as dependências que vamos utilizar. Mas um ponto de atenção está na hora de adicionar a `annotationProcessorPaths`, no build (linha 195 a 210). Pois elas são responsáveis por gerar o código baseado nas anotações utilizadas, mas ao combinar Lombok com MatStruct, descobri que a ordem importa, pois se o MapStruct vier antes, ele não consegue acesso aos Getters/Setters para fazer o mapeamento.

Obs.: Certifique-se que o annotations processors da sua IDE está habilitado.



### 3.5 application.yml

Por padrão, este arquivo vem com a extensão `.properties`, porém, arquivos `.yml`, tendem a serem mais fáceis de trabalhar e funcionam como uma hierarquia, possuindo muitos benefícios como organização, estrutura, leitura e etc.

Nesse arquivo está algumas configurações, mas a principal configuração é a de banco de dados com o

MySQL.

```
2  server:
3    error:
4      include-stacktrace: on_param
5
6  spring:
7    application:
8      name: orange-bank
9    datasource:
10     url: jdbc:mysql://localhost:3306/orange?createDatabaseIfNotExist=true
11     username: root
12     password: root
13  jpa:
14    hibernate:
15      ddl-auto: update
16      show-sql: true
17    properties:
18      hibernate:
19        format_sql: true
20
21  logging:
22    level:
23      org:
24        hibernate:
25          SQL: DEBUG
```

### 3.6 Criando a classe Customer

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Customer {

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

    @NotEmpty(message = "Name cannot be empty")
    private String name;

    @Email(message = "Email cannot be invalid")
    @NotEmpty(message = "Email cannot be empty")
    private String email;

    @CPF(message = "CPF cannot be invalid")
    @NotNull(message = "CPF cannot be null")
    private String cpf;
```

```
    private LocalDate birthdate;  
}
```

Alguns pontos a serem observados:

1. Anotação **@Entity**. Refere-se a tabela que será criada no banco, com o mesmo nome da classe.
2. Anotação **@Id**. Indentifica que esse atributo será a chave primária da entidade no banco.
3. Anotação **@GeneratedValue**. É como será gerado esse id no banco.
4. Demais anotações, são da biblioteca Lombok, bem como citados anteriormente.

## 3.7 Criando as classes de transporte (DTO)

### 3.7.1 CustomerDTO

Essa classe será o padrão de retorno quando houver alguma solicitação.

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class CustomerDTO {  
  
    private Long id;  
  
    private String name;  
  
    private String email;  
  
    private String cpf;  
  
    @JsonFormat(pattern = "dd/MM/yyyy")  
    private Date birthdate;  
}
```

### 3.7.2 CustomerPostRequestDTO

Essa classe representa o objeto esperado para inserção.

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@InsertValidator  
public class CustomerPostRequestDTO {  
  
    @NotEmpty(message = "Name cannot be empty")  
    private String name;  
}
```



```
@Email(message = "Email cannot be invalid")
@NotEmpty(message = "Email cannot be empty")
private String email;

@CPF(message = "CPF cannot be invalid")
private String cpf;

@JsonFormat(pattern = "dd/MM/yyyy")
@JsonDeserialize(using = LocalDateDeserializer.class)
@JsonSerialize(using = LocalDateSerializer.class)
private LocalDate birthdate;
}
```

Alguns pontos a serem observados:

1. Anotação **@InsertValidator**. Essa anotação é personalizada, vamos falar dela mais adiante.

### 3.7.3 CustomerPutRequestDto

Essa classe representa o objeto esperado para atualização.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@updateValidator
public class CustomerPutRequestDTO {

    private Long id;

    @NotEmpty(message = "Name cannot be empty")
    private String name;

    @Email(message = "Email cannot be invalid")
    @NotEmpty(message = "Email cannot be empty")
    private String email;

    @CPF(message = "CPF cannot be invalid")
    private String cpf;

    @JsonFormat(pattern = "dd/MM/yyyy")
    @JsonDeserialize(using = LocalDateDeserializer.class)
    @JsonSerialize(using = LocalDateSerializer.class)
    private LocalDate birthdate;
}
```

Alguns pontos a serem observados:

1. Anotação **@UpdateValidator**. Essa anotação é personalizada, vamos falar dela mais adiante.

### 3.8 Criando a classe CustomerMapper

```
@Mapper(componentModel = "orange")
public interface CustomerMapper {

    CustomerMapper INSTANCE = Mappers.getMapper(CustomerMapper.class);

    Customer toCustomer(CustomerDTO customerDTO);

    Customer toCustomer(CustomerPostRequestDTO postRequestDTO);

    CustomerDTO toCustomerDTO(Customer customer);
}
```

### 3.9 Criando a classe CustomerRepository

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    Customer findByCpf(String cpf);

    Customer findByEmail(String email);
}
```

No Spring Data, é possível criar consultas baseadas nas propriedades de um objeto. Como por padrão não existe os métodos buscar por cpf e email, então foi criado esses dois métodos. Lembrando que essa abordagem funciona bem aqui, por ser uma consulta simples.

Mas caso fosse preciso, poderíamos utilizar uma Query Annotation ou até mesmo query nativas.

### 3.10 Criando a classe CustomerService

```
public interface CustomerService {

    CustomerDTO findByIdOrThrowBadRequestException(Long id);

    List<CustomerDTO> findAll();

    CustomerDTO save(CustomerPostRequestDTO postRequestDTO);

    CustomerDTO update(CustomerPutRequestDTO putRequestDTO);

    void delete(Long id);
}
```

### 3.11 Criando a classe CustomerServiceImpl

```
@Service
@RequiredArgsConstructor
public class CustomerServiceImpl implements CustomerService {

    private final CustomerRepository repository;
    private final CustomerMapper mapper = CustomerMapper.INSTANCE;

    @Override
    @Transactional(readOnly = true, propagation = Propagation.SUPPORTS)
    public CustomerDTO findByIdOrThrowBadRequestException(Long id) {
        return this.mapper
            .toCustomerDTO(this.repository
                .findById(id)
                .orElseThrow(() -> new BadRequestException("Customer not
found"))));
    }

    @Override
    @Transactional(readOnly = true, propagation = Propagation.SUPPORTS)
    public List<CustomerDTO> findAll() {
        return this.repository.findAll().stream()
            .map(this.mapper::toCustomerDTO)
            .collect(Collectors.toList());
    }

    @Override
    @Transactional(rollbackFor = Exception.class)
    public CustomerDTO save(CustomerPostRequestDTO postRequestDTO) {
        Customer customer = this.mapper.toCustomer(postRequestDTO);
        return this.mapper.toCustomerDTO(this.repository.save(customer));
    }

    @Override
    @Transactional(rollbackFor = Exception.class)
    public CustomerDTO update(CustomerPutRequestDTO putRequestDTO) {
        Customer customer =
this.mapper.toCustomer(this.findByIdOrThrowBadRequestException(putRequestDTO.getId
()));

        updateData(customer, putRequestDTO);

        return this.mapper.toCustomerDTO(this.repository.save(customer));
    }

    @Override
    @Transactional(rollbackFor = Exception.class)
    public void delete(Long id) {
        Long customerId =
this.mapper.toCustomer(this.findByIdOrThrowBadRequestException(id)).getId();
        this.repository.deleteById(customerId);
    }

    private void updateData(Customer customer, CustomerPutRequestDTO
```

```
putRequestDTO) {  
    customer.setName(putRequestDTO.getName());  
    customer.setEmail(putRequestDTO.getEmail());  
    customer.setCpf(putRequestDTO.getCpf());  
    customer.setBirthdate(putRequestDTO.getBirthdate());  
}  
}
```

Alguns pontos a serem observados nessa classe:

1. Anotação **@Service**. Serve para indicar para o Spring que ela é um Bean, ou seja, um componente e portando ele vai gerenciar essa classe.
2. Anotação **@Transactional**. Uma transação garante que todo o processo deva ser executado com êxito (princípio da atomicidade). Com isso você tem a garantia que nenhuma informação será persistida se todo o processo não tiver 100% de êxito.
  1. Essa anotação possui alguns propriedades, uma delas é a **"readOnly"**, que por padrão é **false**. Porém, para métodos que não modificam a base de dados, como métodos GET, essa propriedade pode ser **true**.
  2. Outra propriedade é a **"propagation"**, que por padrão é **REQUIRED**. Ou seja, ela sempre precisa de uma transação, caso exista uma, ela usará esta, senão ela criará uma nova transação. Contudo, para métodos GET, não tem a necessidade, por isso, podemos utilizar a **SUPPORTS**, que caso haja uma transação, ela utilizará, mas senão houver, não criará uma nova.
  3. Por padrão ao se utilizar a anotação @Transactional, ela garante que caso ocorra alguma exceção do tipo **RuntimeException**, a operação será desfeita, ou seja um rollback dos dados. Porém, essa exceção é do tipo não checada, caso ocorra uma exceção do tipo **Exception**, mesmo ocorrendo um erro, os dados serão persistidos(podem testar ^^). Para que isso não aconteça, é importante usar uma outra propriedade do @Transactional, o **"rollbackFor = Exception.class"**, com isso garantimos que qualquer tipo de exceção lançada, os dados não serão persistidos no bando de dados.

Existem outras propriedades, mas por hora vamos ficar apenas com estas.

### 3.12 Criando a classe CustomerResource

```
@RestController  
@RequiredArgsConstructor  
@RequestMapping("/api/v1/customers")  
public class CustomerResource {  
  
    private final CustomerService service;  
  
    @GetMapping(value =("/{id}")  
    public ResponseEntity<CustomerDTO> findById(@PathVariable Long id) {  
        return  
        ResponseEntity.ok().body(this.service.findByIdOrThrowBadRequestException(id));  
    }  
  
    @GetMapping  
    public ResponseEntity<List<CustomerDTO>> findAll() {
```

```
        return ResponseEntity.ok().body(this.service.findAll());
    }

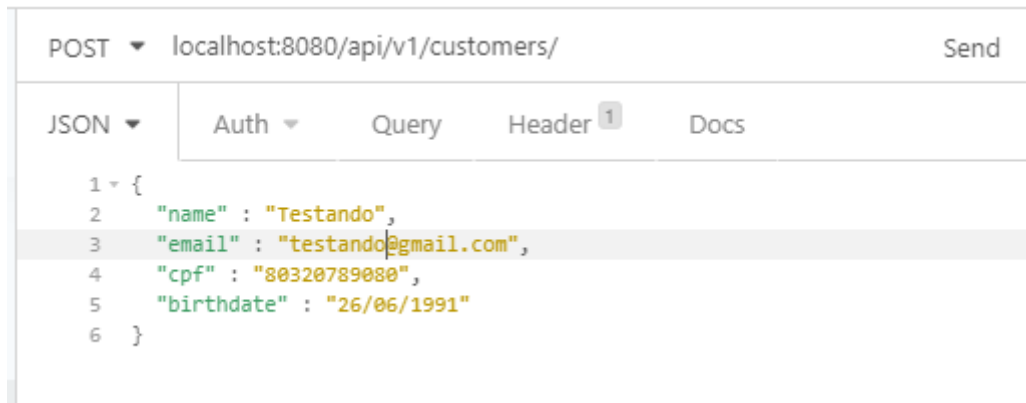
    @PostMapping
    public ResponseEntity<CustomerDTO> insert(@Valid @RequestBody
CustomerPostRequestDTO postRequestDTO) {
        return new ResponseEntity(this.service.save(postRequestDTO),
HttpStatus.CREATED);
    }

    @PutMapping
    public ResponseEntity<CustomerDTO> update(@Valid @RequestBody
CustomerPutRequestDTO putRequestDTO) {
        return ResponseEntity.ok().body(this.service.update(putRequestDTO));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        this.service.delete(id);
        return ResponseEntity.noContent().build();
    }
}
```

Alguns pontos a serem observados nessa classe:

1. Anotação **@RestController**. Para sinalizar para o Spring que a classe é uma controller existem duas anotações: **@Controller** e **@RestController**. A diferença, é que a **@RestController** por padrão já adiciona a anotação **@ResponseBody**, caso contrário seria necessário adicionar essa anotação no métodos criados.
2. Anotação **@RequestMapping**. Essa anotação especifica uma url base para todos os métodos que serão criados.
3. **ResponseEntity**. Essa classe possibilita configurar toda a resposta HTTP: código de status, cabeçalhos e corpo. Para que seja possível isso, é necessário que seja usando na extremidade, como é o caso das controllers.
4. Todos os métodos de uma API Rest, precisam ter uma caminho único.
5. Os códigos HTTP mais utilizados nesse projeto foram:
  1. 200 -> Quando uma ação foi bem sucedida. Podendo ter ou não um objeto de retorno
  2. 201 -> Quando um novo objeto é criado no banco de dados.
  3. 204 -> Quando uma requisição foi bem sucedida e não possui dados para retornar.
  4. 400 -> Quando existe algum erro por parte do cliente, uma requisição mal formatada ou inválida.
6. Anotação **@RequestBody**. Essa anotação quer dizer que o objeto vai vir no corpo da requisição em formato JSON, que é o padrão do REST.



Como podemos observar, estamos enviando uma requisição **POST**, ou seja, vamos criar um novo objeto, e no corpo da requisição estou passando os dados em formato **JSON**. Importante se atentar aos nomes dos campos, pois são os mesmo da classe `Customer`, dessa forma o Spring faz a deserialização do JSON para o objeto **`CustomerPostRequestDTO`** automaticamente.

7. Anotação **@PathVariable**. Essa anotação significa que estamos passando um identificador na URL. Se notarmos no método **`findById`** a seguir:

```
@GetMapping(value =("/{id}")
public ResponseEntity<CustomerDTO> findById(@PathVariable Long id) {
    return ResponseEntity.ok().body(this.service.findByIdOrThrowBadRequestException(id));
}
```

Estamos anotando a método com **@GetMapping**, ou seja, estamos buscando dados e passar nessa anotação como vamos receber esse identificador.

Supondo que queremos buscar o id (1), juntando a URL base com esse novo parâmetro, teremos: **`api/v1/customers/1`**, como estamos em localhost e usando a porta 8080, estão o caminho completo fica: **`localhost:8080/api/v1/customers/1`**. Caso estivesse hospedado em algum servidor, poseria ser algo como: **`servidor.xpto/api/v1/customers/1`**.

Com isso, ao criar um parâmetro no método e adicionar a anotação antes dele, queremos dizer para que seja preenchida como que vier na URL.

### 3.13 Criando as classes de validações personalizadas

#### 3.13.1 InsertValidation

```
@Constraint(validatedBy = InsertValidatorImpl.class)
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface InsertValidator {

    String message() default "Validation error";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

#### 3.13.2 InsertValidationImpl

```

@RequiredArgsConstructor
public class InsertValidatorImpl implements ConstraintValidator<InsertValidator,
CustomerPostRequestDTO> {

    private final CustomerRepository repository;

    @Override
    public boolean isValid(CustomerPostRequestDTO objDTO,
ConstraintValidatorContext context) {
        List<ValidationExceptionDetails> list = new ArrayList<>();

        if (objDTO.getBirthdate() == null) {
            list.add(new ValidationExceptionDetails("birthdate", "Birthdate cannot
be null"));
        }

        Customer customer = this.repository.findByCpf(objDTO.getCpf());
        if (customer != null) {
            list.add(new ValidationExceptionDetails("cpf", "CPF already exists"));
        }

        customer = this.repository.findByEmail(objDTO.getEmail());
        if (customer != null) {
            list.add(new ValidationExceptionDetails("email", "E-mail already
exists"));
        }

        for (ValidationExceptionDetails v : list) {
            context.disableDefaultConstraintViolation();

            context.buildConstraintViolationWithTemplate(v.getFieldsMessage()).addPropertyNode
(v.getFields())
                .addConstraintViolation();
        }
        return list.isEmpty();
    }
}

```

Essa classe é responsável por validar se existe birthdate nulo e verificar se já existe algum CPF ou Email cadastrado no banco de dados. Para inserção é simples, pois só preciso verificar se o objeto retornado é nulo ou não.

### 3.13.3 UpdateValidation

```

@Constraint(validatedBy = UpdateValidatorImpl.class)
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface UpdateValidator {

    String message() default "Validation error";
}

```

```

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}

```

### 3.13.4 UpdateValidationImpl

```

@RequiredArgsConstructor
public class UpdateValidatorImpl implements ConstraintValidator<UpdateValidator,
CustomerPutRequestDTO> {

    private final CustomerRepository repository;

    @Override
    public boolean isValid(CustomerPutRequestDTO objDTO,
ConstraintValidatorContext context) {
        List<ValidationExceptionDetails> list = new ArrayList<>();

        if (objDTO.getId() == null) {
            list.add(new ValidationExceptionDetails("id", "Id cannot be null"));
        }

        if (objDTO.getBirthdate() == null) {
            list.add(new ValidationExceptionDetails("birthdate", "Birthdate cannot
be null"));
        }

        Customer customer = this.repository.findByCpf(objDTO.getCpf());
        if (customer != null && !customer.getId().equals(objDTO.getId())) {
            list.add(new ValidationExceptionDetails("cpf", "CPF already exists"));
        }

        customer = this.repository.findByEmail(objDTO.getEmail());
        if (customer != null && !customer.getId().equals(objDTO.getId())) {
            list.add(new ValidationExceptionDetails("email", "E-mail already
exists"));
        }

        for (ValidationExceptionDetails v : list) {
            context.disableDefaultConstraintViolation();

            context.buildConstraintViolationWithTemplate(v.getFieldsMessage()).addPropertyNode
(v.getFields())
                .addConstraintViolation();
        }
        return list.isEmpty();
    }
}

```



Essas classes servem para validações que o Spring não consegue fornecer e vão de acordo com a regra de negócio de projeto. Nesse projeto, não se deve permitir cadastrar CPF e Email duplicado, com essas classes gerenciadas pelo Spring, garantimos isso.

Diferentemente da validação de inserção, durante a atualização eu preciso fazer uma checagem a mais. Pois, se o objeto retornado tiver um ID diferente do ID enviado no objDTO, então eu não posso permitir atualizar. Agora, caso o ID retornado seja igual o que está no objDTO, significa que são o "mesmo" objeto, portando pode ser atualizado.

A seguir, uma exemplo da mensagem de erro formatada para quando CPF e Email estão duplicados.

```
{
  "title": "Bad Request Exception, Invalid Fields",
  "status": 400,
  "details": "Check the field(s) error",
  "developerMessage":
    "org.springframework.web.bind.MethodArgumentNotValidException",
  "timestamp": "2021-01-10T21:57:13.8709599",
  "fields": "cpf,email",
  "fieldsMessage": "CPF already exists,E-mail already exists"
}
```

### 3.14 Criando classe de seed (DummyData)

```
@Component
@RequiredArgsConstructor
public class DummyData implements CommandLineRunner {

    private final CustomerRepository repository;

    @Override
    public void run(String... args) throws Exception {
        this.repository.deleteAll();
        List<Customer> customers = List.of(
            Customer.builder()
                .name("Customer A")
                .email("customerA@gmail.com")
                .cpf("63396382013")
                .birthdate(LocalDate.now())
                .build(),

            Customer.builder()
                .name("Customer B")
                .email("customerB@gmail.com")
                .cpf("42382396016")
                .birthdate(LocalDate.now())
                .build(),

            Customer.builder()
                .name("Customer C")
                .email("customerC@gmail.com")
                .build()
        );
        repository.saveAll(customers);
    }
}
```

```

        .cpf("95540193008")
        .birthdate(LocalDate.now())
        .build(),

        Customer.builder()
            .name("Customer D")
            .email("customeD@gmail.com")
            .cpf("85488578030")
            .birthdate(LocalDate.now())
            .build(),

        Customer.builder()
            .name("Customer E")
            .email("customeE@gmail.com")
            .cpf("63396382013")
            .birthdate(LocalDate.now())
            .build()
    );
    this.repository.saveAll(customers);
}
}

```

Essa classe implementa a classe `CommandLineRunner`, que possui o método **run**. Ela foi anotada como **@Component**, lembrando que tudo no Spring, é component, com isso, estamos dizendo que querendo que o Spring gerencie essa classe. A sua principal utilidade aqui é ser sempre executada quando o Spring é inicializado. Como sabemos desse comportamento, defini que toda vez que o Spring fosse inicializado, ela apagaria todos os dados do banco e faria a inserção novamente dos dados.

### 3.15 Criando nossas próprias exceções personalizadas

O intuito de criar exceções personalizadas, é que o Spring por padrão, fornece dados demais quando ocorre alguma exceção. Isso pode acarretar em falhas de segurança expondo dados que não deveriam.

#### 3.15.1 BadRequestException

```

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

    public BadRequestException(String message) {
        super(message);
    }
}

```

#### 3.15.2 ExceptionDetails

```

@Data
@NoArgsConstructor
@SuperBuilder

```

```
public class ExceptionDetails {
    protected String title;
    protected int status;
    protected String details;
    protected String developerMessage;
    protected LocalDateTime timestamp;
}
```

### 3.15.3 BadRequestExceptionDetails

```
@Getter
@SuperBuilder
public class BadRequestExceptionDetails extends ExceptionDetails{
}
```

### 3.15.4 ValidationExceptionDetails

```
@Data
@AllArgsConstructor
@SuperBuilder
public class ValidationExceptionDetails extends ExceptionDetails{
    private final String fields;
    private final String fieldsMessage;
}
```

## 3.16 Criando a classe RestExceptionHandler

```
@ControllerAdvice
public class RestExceptionHandler extends ResponseEntityExceptionHandler {
    @ExceptionHandler(BadRequestException.class)
    public ResponseEntity<BadRequestExceptionDetails>
    handlerBadRequestException(BadRequestException bre){
        return new ResponseEntity<>(
            BadRequestExceptionDetails.builder()
                .timestamp(LocalDateTime.now())
                .status(HttpStatus.BAD_REQUEST.value())
                .title("Bad Request Exception, Check the Documentation")
                .details(bre.getMessage())
                .developerMessage(bre.getClass().getName())
                .build(), HttpStatus.BAD_REQUEST
        );
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
        MethodArgumentNotValidException exception, HttpHeaders headers,
```

```

HttpStatus status, WebRequest request) {
    List<FieldError> fieldErrors =
exception.getBindingResult().getFieldErrors();
    String fields =
fieldErrors.stream().map(FieldError::getField).collect(Collectors.joining(","));
    String fieldsMessage =
fieldErrors.stream().map(FieldError::getDefaultMessage).collect(Collectors.joining(
","));

    return new ResponseEntity<>(
        ValidationExceptionDetails.builder()
            .timestamp(LocalDateTime.now())
            .status(HttpStatus.BAD_REQUEST.value())
            .title("Bad Request Exception, Invalid Fields")
            .details("Check the field(s) error")
            .developerMessage(exception.getClass().getName())
            .fields(fields)
            .fieldsMessage(fieldsMessage)
            .build(), HttpStatus.BAD_REQUEST
    );
}

@Override
protected ResponseEntity<Object> handleExceptionInternal(
    Exception ex, @Nullable Object body, HttpHeaders headers, HttpStatus
status, WebRequest request) {

    ExceptionDetails exceptionDetails = ExceptionDetails.builder()
        .timestamp(LocalDateTime.now())
        .status(status.value())
        .title(ex.getCause().getMessage())
        .details(ex.getMessage())
        .developerMessage(ex.getClass().getName())
        .build();

    return new ResponseEntity<>(exceptionDetails, headers, status);
}
}

```

Essa classe anotada com **@RestControllerAdvice**, permite que seja detectada automaticamente por meio de varredura do caminho da classe. Ela intercepta a lógica dos controladores e permite aplicar alguma lógica comum a eles.

Ao fazer ela estender da classe **ResponseEntityExceptionHandler**, nos permite sobrescrever as exceções que queremos padronizar no sistema, garantindo uma uniformidade.

## 4 Testes

Testes são essenciais para o programador. Eles são a prova de que a solução desenvolvida está funcionando e continuará funcionando. Caso haja alguma alteração no código, o testes acusarão falha e precisarão ser refeitos para estarem em conformidades novamente.

Podemos ter testes unitários e testes de integração. Testes unitário, é referente ao teste de toda uma classe,

aplicando testes de unidade, ou seja, testes em cada método dessa classe, procurando sempre varrer o maior número de cenários possíveis.

Testes de integração, já se diz respeito quando testamos nossa solução do início ao fim, garantindo que desde a entrada dos dados até a persistências dos mesmo, está tudo funcionando como o requisito solicita.

#### 4.1 Criando a CustomerRepositoryTest

```
@DataJpaTest
@DisplayName("Tests for Customer Repository")
@AutoConfigureTestDatabase(replace = NONE)
class CustomerRepositoryTest {

    @Autowired
    private CustomerRepository repository;

    @Test
    @DisplayName("save persists customer when successful")
    void save_PersistCustomer_WhenSuccessful() {
        Customer customerToBeSave = CustomerCreator.createCustomerToBeSave();

        Customer customerSaved = this.repository.save(customerToBeSave);

        Assertions.assertThat(customerSaved).isNotNull();

        Assertions.assertThat(customerSaved.getId()).isNotNull();

        Assertions.assertThat(customerSaved.getName()).isEqualTo(customerToBeSave.getName());

        Assertions.assertThat(customerSaved.getEmail()).isEqualTo(customerToBeSave.getEmail());

        Assertions.assertThat(customerSaved.getCpf()).isEqualTo(customerToBeSave.getCpf());
        ;

        Assertions.assertThat(customerSaved.getBirthdate()).isEqualTo(customerToBeSave.getBirthdate());
    }

    @Test
    @DisplayName("save updates customer when successful")
    void save_UpdatesCustomer_WhenSuccessful() {
        Customer customerToBeSaved = CustomerCreator.createCustomerToBeUpdate();

        Customer customerSaved = this.repository.save(customerToBeSaved);

        customerSaved.setName("Customer 1 updated");
    }
}
```

```
        Customer customerUpdated = this.repository.save(customerSaved);

        Assertions.assertThat(customerUpdated).isNotNull();

        Assertions.assertThat(customerUpdated.getId()).isNotNull();

    }

    Assertions.assertThat(customerUpdated.getName()).isEqualTo(customerToBeSaved.getName());

    Assertions.assertThat(customerUpdated.getEmail()).isEqualTo(customerToBeSaved.getEmail());

    Assertions.assertThat(customerUpdated.getCpf()).isEqualTo(customerToBeSaved.getCpf());

    Assertions.assertThat(customerUpdated.getBirthdate()).isEqualTo(customerToBeSaved.getBirthdate());
    }

    @Test
    @DisplayName("delete removes customer when successful")
    void delete_RemovesCustomer_WhenSuccessful() {
        Customer customerToBeSave = CustomerCreator.createCustomerToBeSave();

        Customer customerSaved = this.repository.save(customerToBeSave);

        this.repository.delete(customerSaved);

        Optional<Customer> customerOptional =
this.repository.findById(customerSaved.getId());

        Assertions.assertThat(customerOptional).isEmpty();
    }

    @Test
    @DisplayName("findByCpf returns customer when successful")
    void findByCpf_ReturnsCustomer_WhenSuccessful() {
        Customer customerToBeSave = CustomerCreator.createCustomerToBeSave();

        Customer customerSaved = this.repository.save(customerToBeSave);

        String cpf = customerSaved.getCpf();

        Customer customer = this.repository.findByCpf(cpf);

        Assertions.assertThat(customer)
            .isNotNull();
    }

    @Test
```

```
@DisplayName("findByEmail returns customer when successful")
void findByEmail_ReturnsCustomer_WhenSuccessful() {
    Customer customerToBeSave = CustomerCreator.createCustomerToBeSave();

    Customer customerSaved = this.repository.save(customerToBeSave);

    String email = customerSaved.getEmail();

    Customer customer = this.repository.findByEmail(email);

    Assertions.assertThat(customer)
        .isNotNull();
}

@Test
@DisplayName("findAll returns list of customer when successful")
void findAll_ReturnsListOfCustomer_WhenSuccessful() {
    Customer customerToBeSave = CustomerCreator.createCustomerToBeSave();

    Customer customerSaved = this.repository.save(customerToBeSave);

    List<Customer> customers = this.repository.findAll();

    Assertions.assertThat(customers)
        .isNotEmpty()
        .contains(customerSaved);
}

@Test
@DisplayName("save throw ConstraintViolationException when name is empty")
void save_ThrowConstraintViolationException_WhenNameIsEmpty() {
    Customer customerToBeSave = CustomerCreator.createCustomerNameEmpty();

    Assertions.assertThatExceptionOfType(ConstraintViolationException.class)
        .isThrownBy(() -> this.repository.save(customerToBeSave))
        .withMessageContaining("Name cannot be empty");
}

@Test
@DisplayName("save throw ConstraintViolationException when email is empty")
void save_ThrowConstraintViolationException_WhenEmailIsEmpty() {
    Customer customerToBeSave = CustomerCreator.createCustomerEmailEmpty();

    Assertions.assertThatExceptionOfType(ConstraintViolationException.class)
        .isThrownBy(() -> this.repository.save(customerToBeSave))
        .withMessageContaining("Email cannot be empty");
}

@Test
@DisplayName("save throw ConstraintViolationException when email is invalid")
void save_ThrowConstraintViolationException_WhenEmailIsInvalid() {
    Customer customerToBeSave = CustomerCreator.createCustomerEmailInvalid();

    Assertions.assertThatExceptionOfType(ConstraintViolationException.class)
```

```

        .isThrownBy(() -> this.repository.save(customerToBeSave))
        .withMessageContaining("Email cannot be invalid");
    }

    @Test
    @DisplayName("save throw ConstraintViolationException when cpf is null")
    void save_ThrowConstraintViolationException_WhenCPFIsNull() {
        Customer customerToBeSave = CustomerCreator.createCustomerCPFIsNull();

        Assertions.assertThatExceptionOfType(ConstraintViolationException.class)
            .isThrownBy(() -> this.repository.save(customerToBeSave))
            .withMessageContaining("CPF cannot be null");
    }

    @Test
    @DisplayName("save throw ConstraintViolationException when cpf is invalid")
    void save_ThrowConstraintViolationException_WhenCPFInvalid() {
        Customer customerToBeSave = CustomerCreator.createCustomerCPFInvalid();

        Assertions.assertThatExceptionOfType(ConstraintViolationException.class)
            .isThrownBy(() -> this.repository.save(customerToBeSave))
            .withMessageContaining("CPF cannot be invalid");
    }
}

```

Alguns pontos a serem observados:

1. Anotação **@DataJpaTest**. Essa anotação é utilizada para testar repositórios JPA. A anotação destina a configuração automática completa e aplica apenas a configuração relevante para testes JPA. Por padrão, os testes anotados com **@DataJpaTest** usando bando de dados embutido na memória.
2. Como exemplo de banco de dados embutido na memória, temos o H2, porém para esses testes usamos o banco de dados MySQL mesmo. Para que isso fosse possível, foi precisar utilizar a anotação **@AutoConfigureTestDatabase(replace - NONE)**, dessa forma, o Spring não tenta usar um banco em memória durante os testes.
3. Anotação **@DisplayName**. Essa anotação serve apenas para adicionar um descritivo sobre o que a classe ou o método faz ou deve fazer.
4. Para a verificação, usamos **Assertions** do assertj.

## 4.2 Criando a CustomerServiceTest

```

@ExtendWith(SpringExtension.class)
@DisplayName("Tests for Customer Service")
class CustomerServiceTest {

    @InjectMocks
    private CustomerServiceImpl service;

    @Mock
    private CustomerRepository repository;
}

```



```
@BeforeEach
void setUp() {
    BDDMockito.when(this.repository.findById(ArgumentMatchers.anyLong()))
        .thenReturn(Optional.of(CustomerCreator.createValidCustomer()));

    BDDMockito.when(this.repository.findAll())
        .thenReturn(List.of(CustomerCreator.createValidCustomer()));

    BDDMockito.when(this.repository.save(ArgumentMatchers.any(Customer.class)))
        .thenReturn(CustomerCreator.createValidCustomer());

    BDDMockito.doNothing().when(this.repository).deleteById(ArgumentMatchers.anyLong());
}

@Test
@DisplayName("findByIdOrThrowBadRequestException returns customer when successful")
void findByIdOrThrowBadRequestException_ReturnsCustomer_WhenSuccessful() {
    String expectedName = CustomerCreator.createValidCustomer().getName();

    Customer customer =
    CustomerMapper.INSTANCE.toCustomer(this.service.findByIdOrThrowBadRequestException(1L));

    Assertions.assertThat(customer)
        .isNotNull();

    Assertions.assertThat(customer.getName())
        .isNotNull()
        .isNotEmpty()
        .isEqualTo(expectedName);
}

@Test
@DisplayName("findAll returns a list of customer when successful")
void findAll_ReturnsListOfCustomer_WhenSuccessful() {
    String expectedName = CustomerCreator.createValidCustomer().getName();

    List<Customer> customers = this.service.findAll().stream()
        .map(CustomerMapper.INSTANCE::toCustomer)
        .collect(Collectors.toList());

    Assertions.assertThat(customers)
        .isNotNull()
        .isNotEmpty()
        .hasSize(1);

    Assertions.assertThat(customers.get(0).getName())
        .isNotNull()
        .isNotEmpty()
        .isEqualTo(expectedName);
}
```

```
}

@Test
@DisplayName("save returns customer when successful")
void save_ReturnsCustomer_WhenSuccessful() {
    String expectedName = CustomerCreator.createValidCustomer().getName();

    Customer customer =
    CustomerMapper.INSTANCE.toCustomer(this.service.save(CustomerPostRequestCreator.createCustomerPostRequestDTO()));

    Assertions.assertThat(customer)
        .isNotNull();

    Assertions.assertThat(customer.getName())
        .isNotNull()
        .isNotEmpty()
        .isEqualTo(expectedName);
}

@Test
@DisplayName("update updates customer when successful")
void update_UpdatesCustomer_WhenSuccessful() {
    String expectedName = CustomerCreator.createValidCustomer().getName();

    BDDMockito.when(this.repository.save(ArgumentMatchers.any(Customer.class)))
        .thenReturn(CustomerCreator.createValidCustomer());

    CustomerPutRequestDTO putRequestDTO =
    CustomerPutRequestCreator.createCustomerPutRequestDTO();

    Customer customer =
    CustomerMapper.INSTANCE.toCustomer(this.service.update(putRequestDTO));

    Assertions.assertThat(customer)
        .isNotNull();

    Assertions.assertThat(customer.getName())
        .isNotNull()
        .isNotEmpty()
        .isEqualTo(expectedName);
}

@Test
@DisplayName("delete removes customer when successful")
void delete_RemovesCustomer_WhenSuccessful() {
    Assertions.assertThatCode(() -> this.service.delete(1L))
        .doesNotThrowAnyException();
}
}
```

Alguns pontos a serem observados:

1. Anotação **@InjectMocks**. Essa anotação quer dizer que queremos mockar uma classe que usaremos no teste. Mais especificamente, a classe que queremos testar, que no nosso caso é a **CustomerService**.
2. Anotação **@Mock**. Essa anotação é referente as classes que a CustomerService utiliza, nesse caso é a **CustomerRepository**. Mas poderiam ter mais classes.
3. Anotação **@BeforeEach**. Essa anotação sobre o método, quer dizer que ele sempre acontecerá antes de qualquer outro método. No nosso caso, ela simula o que esperamos de retorno ao fazer alguma ação com a **CustomerRepository**.

## 4.3 Criando classes de apoio para os testes

### 4.3.1 CustomerCreator

```
public class CustomerCreator {
    private final static LocalDate BIRTHDATE = LocalDate.now();

    public static Customer createCustomerToBeSave() {
        return Customer.builder()
            .name("Customer 1")
            .email("customer1@gmail.com")
            .cpf("80320789080")
            .birthdate(BIRTHDATE)
            .build();
    }

    public static Customer createCustomerToBeUpdate() {
        return Customer.builder()
            .id(1L)
            .name("Customer 1 updated")
            .email("customer1@gmail.com")
            .cpf("88197762007")
            .birthdate(BIRTHDATE)
            .build();
    }

    public static Customer createValidCustomer() {
        return Customer.builder()
            .id(1L)
            .name("Customer 1")
            .email("customer1@gmail.com")
            .cpf("88197762007")
            .birthdate(BIRTHDATE)
            .build();
    }

    public static Customer createCustomerNameEmpty() {
        return Customer.builder()
            .id(1L)
            .name("")
            .email("customer1@gmail.com")
    }
```

```
        .cpf("88197762007")
        .birthdate(BIRTHDATE)
        .build();
    }

    public static Customer createCustomerEmailEmpty() {
        return Customer.builder()
            .id(1L)
            .name("Customer 1")
            .email("")
            .cpf("88197762007")
            .birthdate(BIRTHDATE)
            .build();
    }

    public static Customer createCustomerCPFIsNull() {
        return Customer.builder()
            .id(1L)
            .name("Customer 1")
            .email("customer1@gmail.com")
            .cpf(null)
            .birthdate(BIRTHDATE)
            .build();
    }

    public static Customer createCustomerCPFInvalid() {
        return Customer.builder()
            .id(1L)
            .name("Customer 1")
            .email("customer1@gmail.com")
            .cpf("234")
            .birthdate(BIRTHDATE)
            .build();
    }

    public static Customer createCustomerEmailInvalid() {
        return Customer.builder()
            .id(1L)
            .name("Customer 1")
            .email("customer1gmail.com")
            .cpf("88197762007")
            .birthdate(BIRTHDATE)
            .build();
    }
}
```

#### 4.3.2 CustomerPostRequestCreator

```
public class CustomerPostRequestCreator {

    public static CustomerPostRequestDTO createCustomerPostRequestDTO() {
```

```
        return CustomerPostRequestDTO.builder()
            .name(CustomerCreator.createCustomerNameEmpty().getName())
            .email(CustomerCreator.createValidCustomer().getEmail())
            .cpf(CustomerCreator.createValidCustomer().getCpf())
            .birthdate(CustomerCreator.createValidCustomer().getBirthdate())
            .build();
    }
}
```

#### 4.3.3 CustomerPutRequestCreator

```
public class CustomerPutRequestCreator {

    public static CustomerPutRequestDTO createCustomerPutRequestDTO() {
        return CustomerPutRequestDTO.builder()
            .id(CustomerCreator.createCustomerToBeUpdate().getId())
            .name(CustomerCreator.createCustomerToBeUpdate().getName())
            .email(CustomerCreator.createCustomerToBeUpdate().getEmail())
            .cpf(CustomerCreator.createCustomerToBeUpdate().getCpf())

            .birthdate(CustomerCreator.createCustomerToBeUpdate().getBirthdate())
            .build();
    }
}
```

## 5 Conclusão

Por fim, chegamos ao fim do desenvolvimento desse projeto. Todo o código pode ser encontrado aqui:

<https://github.com/WagnerpbAndrade/orange-bank>.

## 5 Referências

<https://spring.io/>

<https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#validation>

<https://www.baeldung.com/hibernate-identifiers>

<https://www.devmedia.com.br/conheca-o-spring-transactional-annotations/32472>

<https://www.zup.com.br/blog/desenvolvimento-de-apis-design-first-e-code-first>

<https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-three-custom-queries-with-query-methods/>

<https://mapstruct.org/documentation/installation/>

<https://docs.docker.com/compose/>