Links:

Spring Initializer

HTTP response status codes

Project Lombok

Maven Repository

https://s3-sa-east-1.amazonaws.com/thedevconf/presentations/TDC2019POA/designcodigo/DCQ-9841_2019-12-01T025820_tdc-nubank-ddd(2).pdf

Versões código

Etapa 1: criar projeto

criar projeto usando Spring Suite:

File > New > Spring Starter Project

Service url: https://start.spring.io

Name: payment-api-ss

Type: Maven

Packaging: Jar

Language: Java

Java Version: 21

Group: com.project.payment

Artifact: payment-api-ss

Package: com.project.payment

Next > Add dependências

Spring Web : Crie aplicativos da web, incluindo RESTful, usando Spring MVC. Usa Apache Tomcat como contêiner incorporado padrão.

Lombok: Biblioteca de anotação Java que ajuda a reduzir o código padrão.

Setar lombok no SS:

Help > Install New Software > Work with: https://projectlombok.org/p2 > Seleciona Lombok > Next > Finish > Restarta a SS > tudo ok

No IntelliJ

criar initializer do Spring no https://start.spring.io → gerar .zip → exportar o .zip criado → abrir pasta no IntelliJ

Rodar aplicação em porta diferente:

Alterar porta do serviço TomCat: src > main > resources > application.properties > server.port=SUA_PORTA

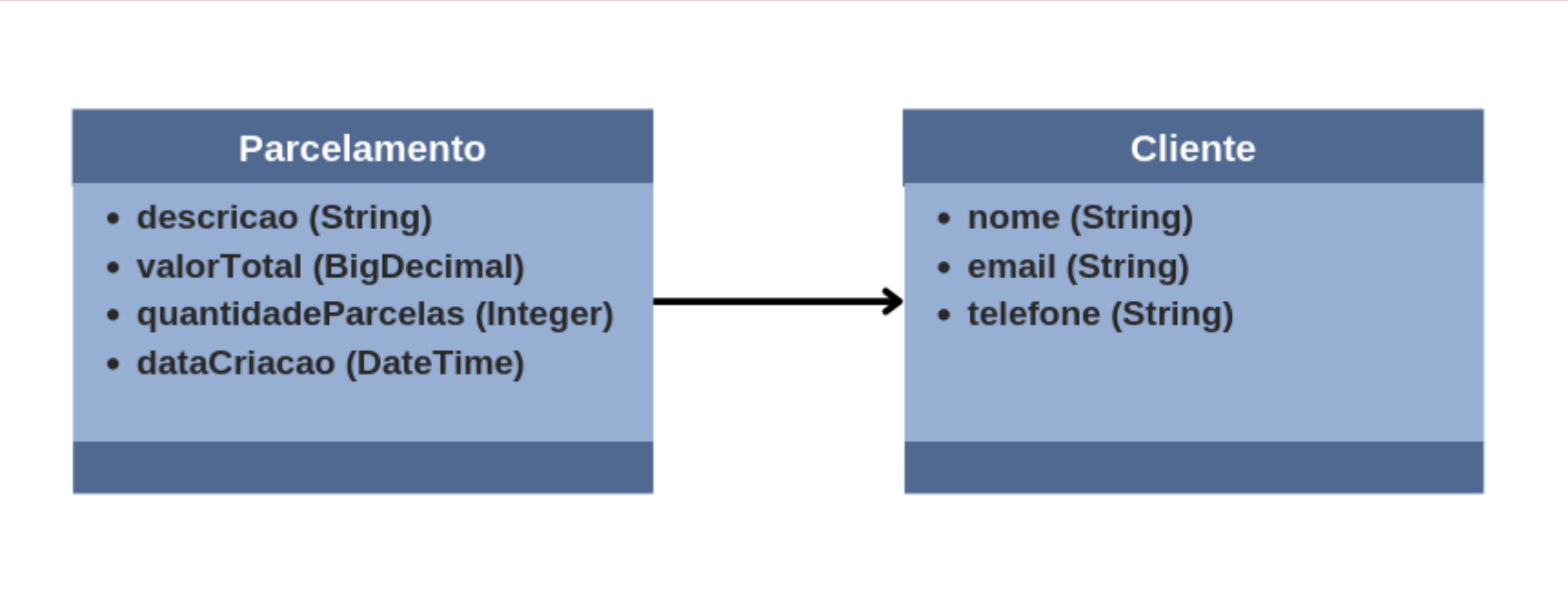
Gerar .jar no IntelliJ:

M > Lifecicle > seleciona clean e package > botão direito > Run Maven Build

Gerar .jar no SS:

botão direito sobre o projeto > Run as > Maven build > Goals: clean package > Run

UML


Parcelamento
descricao (String)
valorTotal (BigDecimal)
quantidadeParcelas (Integer)
dataCriacao (DateTime)
Cliente
nome (String)
email (String)
telefone (String)

Etapa 2: codar

CODE: Controlador de Cliente - criando um endpoint

1. criar um novo pacote com nomenclatura: com.project.payment.api.controller

2. criar nova classe dentro do controller: ClientController.java

3. criar um método que retorna um tipo:

```
public String list(){
    return "Endpoint Cliente";
}
```

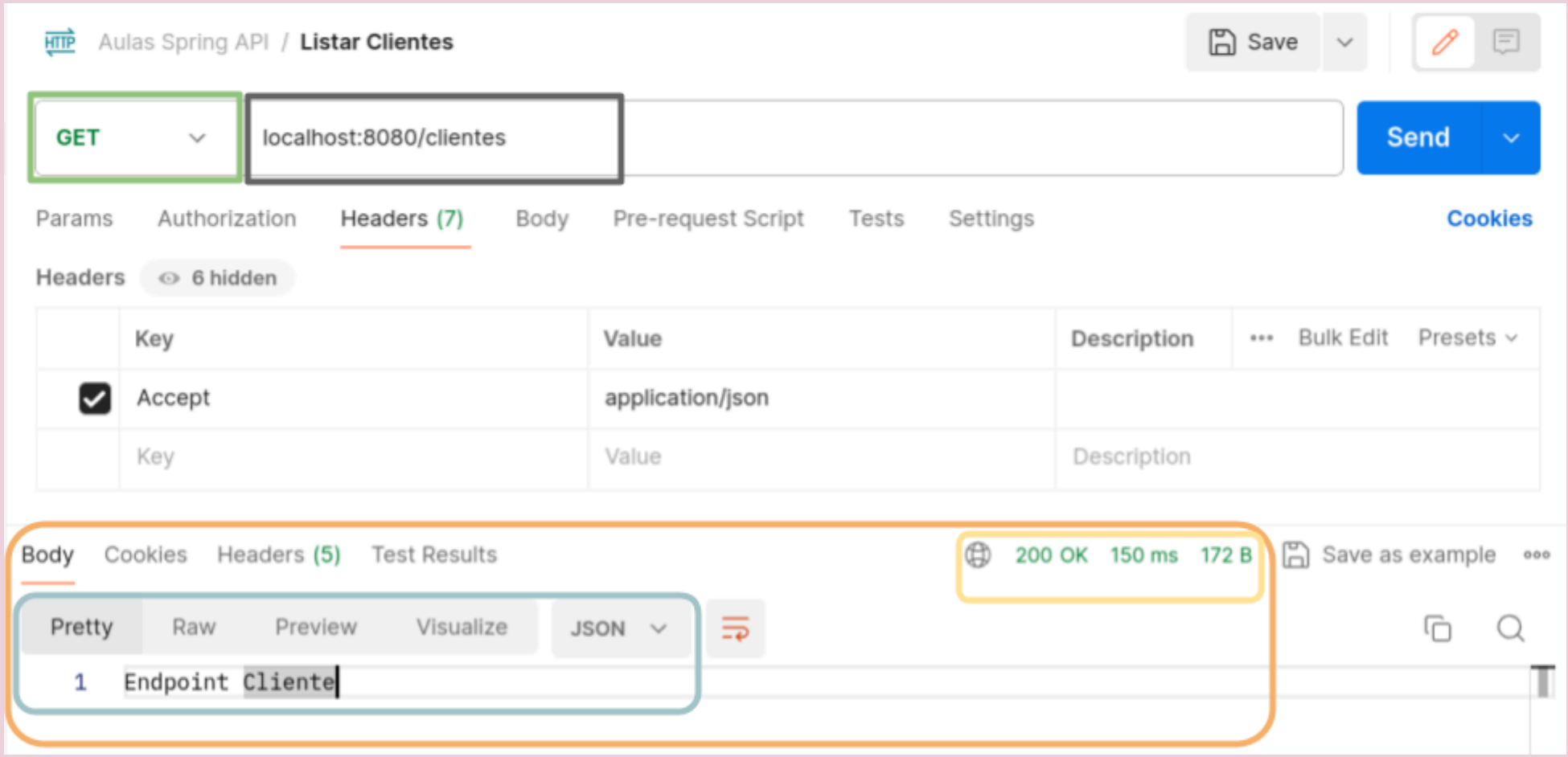
4. adicionar a anotação GetMapping (Spring Framework) com a uri de cliente:

```
@GetMapping("/clientes")
```

```
public String list(){
    return "Endpoint Cliente";
}
```

5. adicionar a anotação `@RestController` para que a classe seja lida como um controlador capaz de tratar requisições http

6. Testa endpoint no postman



CODE: Model de Cliente

1. criar um novo pacote com nomenclatura: `com.project.payment.domain.model;` (ou .entity) e criar uma classe java Client | ClientModel | ClientEntity

(no projeto foi criada a `class ClienteModel`)

2. adicionar os items:

```
public class ClienteModel {
    private Long id;
    private String name;
    private String email;
    private String phone;
}
```

3. adicionar as anotações @Getter e @Setter do Lombok

```
@Getter
@Setter
public class ClienteModel {...}
```

4. No controller de Cliente:

a. criar uma quantidade de clientes:

```
    @GetMapping("/clientes")
    public String listar() {
         var cliente1 = new ClienteModel();
        cliente1.setId(1L);
        cliente1.setName("Bruno S");
        cliente1.setEmail("bruno.s@email.com");
        cliente1.setPhone("77 9 0000-1111");

        var cliente2 = new ClienteModel();
        cliente2.setId(2L);
        cliente2.setName("Luana B");
        cliente2.setEmail("luana.b@email.com");
        cliente2.setPhone("77 9 0000-5555");

    }
```

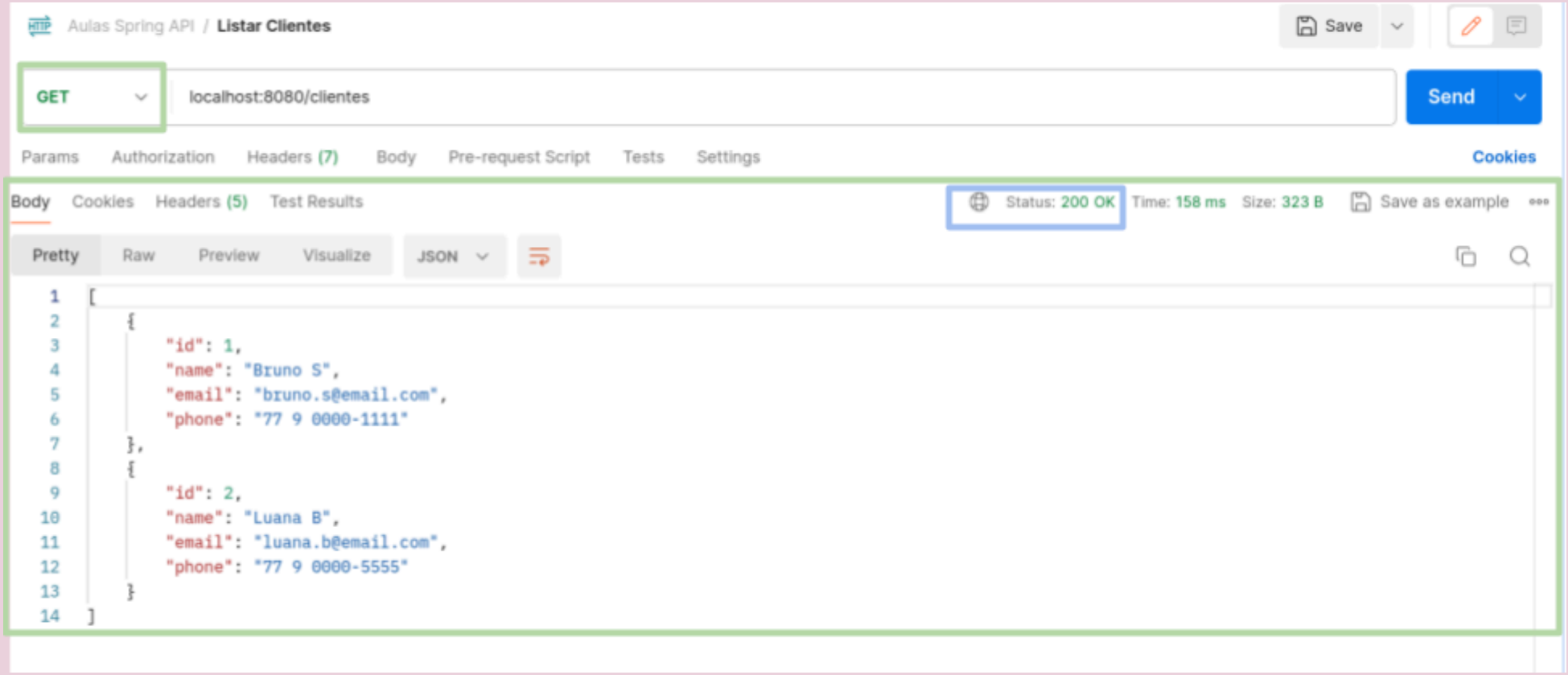
b. retorna esses clientes como uma lista:

```
return Arrays.asList(cliente1, cliente2);
```

c. altera o retorno da função de String para List<Client>

```
public List<ClienteModel> listar() {...}
```

5. testar a aplicação pelo postman



Content Negotiation:

Se quiser alterar a forma de retorno: p.e. no header do postman adicionar key: Accept e value: application/xml

no IntelliJ | no SS → no pom.xml:

```
        <dependency>
            <groupId>com.fasterxml.jackson.dataformat</groupId>
            <artifactId>jackson-dataformat-xml</artifactId>
        </dependency>
```

faz reload do projeto e ai agr, se no cabeçalho tiver requisição com tipo xml a api retorna com o tipo esperado.

Dependência DevTools

O problema no código de Controller de Cliente é que, se eu alterar um cliente que foi criado em tempo de compilação não consigo alterá-lo durante a execução sem ter que reiniciar o servidor. Para isso, adicionamos a dependência do Spring Boot DevTools, no IntelliJ adicione no pom.xml:

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

Se for no SS: botão direito na pasta raíz do projeto > Spring > Add Starters > Busca e seleciona Spring Boot DevTool > Next > seleciona pom.xml > Finish

Criando docker-compose.yml e .env

1. criar o docker-compose.yml:

```
version: '3.8'

services:

 db:

   image: postgres:16.2-alpine3.19

   restart: always

   environment:

     POSTGRES_USER: ${DB_USER}

     POSTGRES_PASSWORD: ${DB_PASSWORD}

   ports:

     - ${DB_PORT}:5432

 adminer:

   image: adminer

   restart: always

   ports:
```

```
      - ${ADMINER_PORT}:8080
```

2. criar o .env

```
DB_USER=dimas

DB_PASSWORD=2000

DB_PORT=2345

DB_HOST=localhost

DB_NAME=payment_api

DB_URL=jdbc:postgresql://localhost:2345/payment_api

ADMINER_PORT=9091
```

3. Suba os containers do Docker > Quando finalizado acesse: localhost:9091 > faça login > crie o banco payment_api

Adicionando as dependências Spring Data JPA, PostgreSQL JDBC e o FlyWay

1. Adicione No IntelliJ, se tiver dúvida sobre as dependências, busque no Maven Repository (link na 1ª pagina)
    a. Se for no SS: botão direito na pasta raíz do projeto > Spring > Add Starters > Busca e seleciona as dependências > Next > seleciona pom.xml > Finish
    b. no pom.xml alterar o dependency do postgres: add `<scope>runtime</scope>`

Como usar .env no projeto Spring:

1. Adiciona no pom.xml: (https://mvnrepository.com/artifact/io.github.cdimascio/dotenv-java/3.0.0)

```xml
<!-- https://mvnrepository.com/artifact/io.github.cdimascio/dotenv-java -->
<dependency>
    <groupId>io.github.cdimascio</groupId>
    <artifactId>dotenv-java</artifactId>
    <version>3.0.0</version>
</dependency>
```

2. Em PaymentApiApplication sete as propriedades para configurar a conexão com o banco de dados:

```java
    public static void main(String[] args) {
        Dotenv dotenv = Dotenv.load();
        String dbUrl = dotenv.get("DB_URL");
        String dbUser = dotenv.get("DB_USER");
        String dbPassword = dotenv.get("DB_PASSWORD");
        System.setProperty("spring.datasource.url", dbUrl);
        System.setProperty("spring.datasource.username", dbUser);
        System.setProperty("spring.datasource.password", dbPassword);
        SpringApplication.run(PaymentApiApplication.class, args);
```

```
    }
```

3. adicionar as variáveis do hibernate no projeto em src > main > resources > application.properties >

```
# jpa
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

4. inicia a aplicação e vê se tudo está ok.

Criando Migrations

1. crie uma pasta chamada db e outra dentro de db chamada migration dentro de src > main > resources.
    a. É nelas que estarão todos os arquivos referentes ao banco de dados, como os scripts de criação/alteração/deleção de tabelas, colunas, etc.
    b. Por padrão, o FW só lê os scripts com nome em determinado padrão, por padrão o FW percebe:
        i. arquivos iniciados com 'V', maiúsculo mesmo + identificação do arquivo,
            1. e aí a identificação do arquivo fica a seu critério, com índice (V0001) ou
            2. data e hora (V20221213213045 - meu formato aqui está YYYY/MM/DD-HH:MM:SS)
            3. (eu vou seguir com a primeira opção);
        ii. é inserido 2 underscore (__) após a versão do arquivo;
        iii. é inserido a descrição do arquivo junto com a extensão .sql (p.e. create-table-client.sql)
        iv. Então no final fica: **V0001__create-table-books.sql**

OBS.: Se o projeto estiver rodando, quando criar o arquivo com a extensão .sql o devtools vai fazer o seu papel de reiniciar o servidor e o FW vai rodar o script vazio e isso vai gerar uma mudança no DB e ficar registrado na tabela referente ao FW que foi criada, a com nome flyway_schema_history.

É nessa tabela que são guardadas as infos referentes ao banco, como por exemplo quais os schemas rodados.

Então, eu recomendo que, com o servidor inativo, crie os arquivos com a versão e extensão e, só depois de tudo certo, com o DDL inserido no arquivo, inicie o servidor.

Você pode também fazer isso com o servidor ativo, ai pode criar o arquivo com o nome (ex. create-table-books.sql) e só depois alterar o nome do arquivo para o padrão.

2. Crie um arquivo para tabela de cliente dentro da pasta migration:

   V0001__create-table-clients.sql

3. Adicione o DDL no arquivo criado:
   a. 1ª forma:

```sql
CREATE TABLE IF NOT EXISTS clients(
      id SERIAL PRIMARY KEY,
      name VARCHAR(200) NOT NULL,
      email VARCHAR(200) NOT NULL,
      telefone VARCHAR(20) NOT NULL,
);

ALTER TABLE clients ADD CONSTRAINT uk_client UNIQUE (email);
```

b. 2ª forma:

```
CREATE TABLE IF NOT EXISTS clients(
      id SERIAL PRIMARY KEY,
      name VARCHAR(200) NOT NULL,
      email VARCHAR(200) UNIQUE NOT NULL,
      telefone VARCHAR(20) NOT NULL,

      PRIMARY KEY (id)
);
```

4. SE quiser alterar o nome de uma coluna:
   a. cria-se uma nova migration
   b. adiciona o codigo:

```
ALTER TABLE client RENAME COLUMN telefone TO phone;
```

Alterando a Model de Cliente para ser lida como Entidade

Add as anotações:

```
@Data // lombok, para getter, setter constructors, toString, etc.
@EqualsAndHashCode(onlyExplicitlyIncluded = true) // lombok, adicionar apenas o id no hash
@Entity //javax.persistance indica que a classe é uma entidade do DB
@Table(name="client") //@Table(name=”client”) //javax.persistance referencia a table do DB
```

adiciona acima de id:

```
@EqualAndHashCode.Include
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

Alterando o Controller de Cliente:

```
@PersistenceContext

private EntityManager entityManager;

@GetMapping(“/clients”)

public String list(){

     return entityManager.createQuery(“from ClientModel”,ClientModel.class).getResultList();

}

// vai dar erro por causa dos nomes no banco x model


//necessário trocar name na class ClienteModel para nome
```

CODE: CRUD DE Cliente

1. Adicionando um repositório para cliente (usaremos para interfaces)
    a. crie um pacote repository dentro do pacote domain
    b. crie uma interface (class) ClienteRepository
    c. extenda nessa interface o JpaRepository
    d. adicione os tipos parametrizados dessa interface genérica <Cliente, Long> {entidade, identificador da entidade}
    e. adiciona a anotação @Repository
2. Altere o tipo EntityManager para o repositório criado:
    a. `private final ClienteRepository clienteRepository;`
    b. você pode criar um método que obriga a quando o controlador for instanciado, ele setar o repositório;
3. Adicione as anotações no topo da classe: `@AllArgsConstructor` e `@RequestMapping("/clientes")`
4. Altere a chamada de query para a desejada;
    a. return clienteRepository.findAll();
5. adicionando métodos GET, POST, PUT e DELETE

```
   // retorna todos os clientes encontrados, se não []
@GetMapping()
public List<ClienteModel> listAll() {
```

```
    return clienteRepository.findAll();
}

@GetMapping("/findByNome") // URI =
localhost:8080/clientes/findByNome?nome=Nome+Sobrenome
public List<ClienteModel> listByName(@RequestParam String nome) {
    return clienteRepository.findByNome(nome);
}

@GetMapping("/findWithNome") // URI =
localhost:8080/clientes/findWithNome?nome=PartNome
public List<ClienteModel> listWithName(@RequestParam String nome) {
    return clienteRepository.findByNomeContains(nome);
}

// retorna o cliente encontrado pelo id, se não uma mensagem de erro
@GetMapping("/{id}")
public Optional<ResponseEntity> findOne(@PathVariable Long id) {
    var client = clienteRepository.findById(id);
    if (client.isEmpty()) {
        return Optional.of(ResponseEntity
```

```
                .status(HttpStatus.NOT_FOUND)
                .body("Cliente nao encontrado"));
    }

    //op 1
    // return client.map(clienteModel -> ResponseEntity.ok(clienteModel));
    return client.map(ResponseEntity::ok);

    //op 2
    // return clienteRepository.findById(id)
    // .map(cliente -> ResponseEntity.ok(cliente))
    // .orElse(ResponseEntity.notFound().build());

    //op 3
    // return clienteRepository.findById(id)
    // .map(ResponseEntity::ok())
    // .orElse(ResponseEntity.notFound().build());
}

// cadastra um novo cliente
@PostMapping
```

```java
@ResponseStatus(HttpStatus.CREATED)
public ClienteModel create(@RequestBody ClienteModel cliente) {
    return clienteRepository.save(cliente);
}

// altera um cliente cadastrado se não retorna erro 404
@PutMapping("/{id}")
public ResponseEntity<ClienteModel> update(
        @PathVariable Long id,
        @RequestBody ClienteModel cliente) {
    if (!clienteRepository.existsById(id)) {
        return ResponseEntity.notFound().build();
    }

    cliente.setId(id);
    cliente = clienteRepository.save(cliente);

    return ResponseEntity.ok(cliente);
}

// remove um cliente
```

```java
@DeleteMapping("/{id}")
public ResponseEntity<ClienteModel> delete(@PathVariable Long id) {
    if (!clienteRepository.existsById(id)) {
        return ResponseEntity.notFound().build();
    }
    clienteRepository.deleteById(id);

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

Adicionando Validation

Adicione no pom.xml a dependência de Validation no JPA:

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

Adicione as anotações necessárias nos atributos de cliente, como:

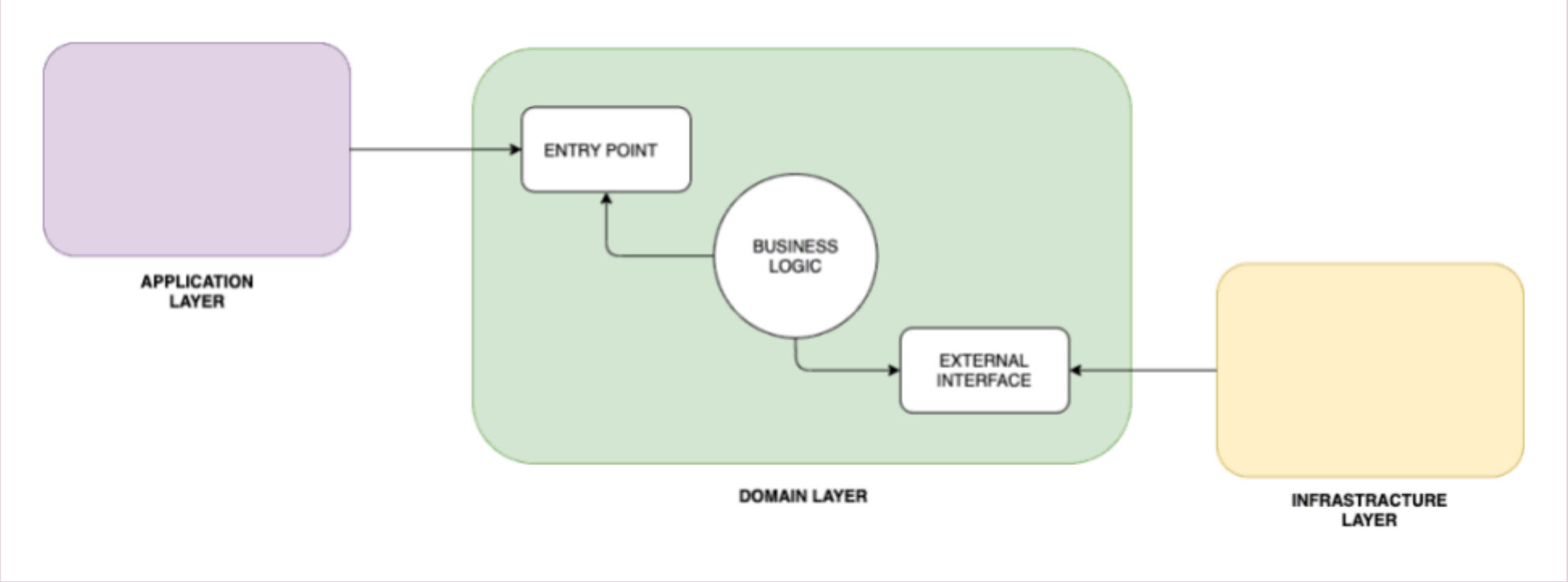
- @NotBlank que irá verificar se o atributo recebeu valores como: null, “” (string vazia) ou “ “ (string com espaço em branco).
- @Size passando por parâmetro os valores máximos da string
- @Email para validar o formato do email

Adicione antes do @RequestBody a anotação @Valid (pacote jakarta.validation.Valid) nos métodos de controle, para que de fato a anotação faça seu trabalho de validação.

Adicionando Domain Services

(https://s3-sa-east-1.amazonaws.com/thedevconf/presentations/TDC2019POA/designcodigo/DCQ-9841_2019-12-01T025820_tdc-nubank-ddd(2).pdf)

a ideia:



projeto crescendo e regras de negócio no controlador não se torna uma boa prática, porque fica tudo muito espalhado. A boa prática é termos classes bem definidas, com alta coesão. Uma possibilidade é colocar regras de negócio na própria entidade, tanto que é uma má prática usar classes (entidades de

modelo de domínio) que carregam apenas dados e não possuem comportamento. (ver sobre domínio rico x domínio anêmico)

Uma outra possibilidade é criar classes de serviço de domínio. O ideal é criar quando temos um processo de negócio que precisa trabalhar com diversas entidades ou ainda quando é necessário injetar um repositório, já que ele não pode ser injetado dentro de uma entidade. Não é necessário ter 1 para 1 (uma classe de serviço para gerenciamento de cada ação), pode ser uma classe para cada gerenciamento de entidade (dependendo do tamanho do projeto).

1. **crie um pacote chamado service, dentro de domain**

```
package com.project.payment.domain.service;
```

2. **crie uma classe do serviço de Cliente**
3. **use a anotação @Service**

```
import org.springframework.stereotype.Service;

@Service

public class GerenciamentoClienteService {

}
```

4. **crie uma variável do tipo ClienteRepository e use a anotação `@AllArgsConstructor`**
5. **crie o método de criação que retorna um cliente**
6. **salve e retorne o cliente recebido**
7. **adicione também a anotação @Transactional do**

   `org.springframework.transaction.annotation.Transactional`

```
@Transactional

public ClienteModel salvar(ClienteModel clienteModel) {

   return clienteRepository.save(clienteModel);

}
```

8. **faça a chamada do serviço no controlador para cadastro e alteração**
9. **crie um método para exclusão também e altere no controlador**

```
@Transactional

public void excluir(Long id) {

   clienteRepository.deleteById(id);

}
```

10. **altere o método de salvar para verificar se possui algum email já cadastrado e lançar uma excessão para o consumidor da API.**
    a. **crie um novo tipo de consulta dentro da interface do cliente**
    b. **adicione a regra para cadastro de cliente com uso de email não duplicado**
    c. **se, já está cadastrado, lança uma exceção, se não, salva o cliente.**

```
// class service
boolean emailExistente =
     clienteRepository.findByEmail(clienteModel.getEmail())
     .filter(cliente -> !cliente.equals(clienteModel)  // filtra o cliente quando faz put
     .isPresent();
if (emailExistente) {
   throw new ApiExceptionsHandler().handleException("email ja vinculado a outra conta.",
                                                    HttpStatus.NOT_ACCEPTABLE);
}

// class ApiExceptionHandler
public RuntimeException handleException(String message,  HttpStatus status) {
   return new RuntimeException(message);
```

```
}
```

Formatando retorno das exceções

1. criar um pacote chamado exception dentro do pacote domain
2. criar uma classe (NegotiationHandler) para o tratamento de exceções que extende a classe RuntimeException com construtor que recebe uma mensage e passa para super(message)

```
package com.project.payment.domain.exception;

public class NegotiationClass extends RuntimeException {

   public NegotiationClass(String message) {

       super(message);

   }

}
```

3. **chamar em salvar o throw new NegotiationHandler(“your_message”);**
4. **adicionar um método capturar no controlador de cliente**

```
@ExceptionHandler(NegotiationHandler.class)

public ResponseEntity<String> capture(NegotiationException e){

   return ResponseEntity.badRequest().body(e.getMessage());

}
```

podem ser alterados métodos de get, put, delete…

```
@PutMapping("/{id}")
public ResponseEntity<ClienteModel> update(
        @PathVariable Long id,
        @Valid @RequestBody ClienteModel cliente) {
    if (!clienteRepository.existsById(id)) {
        throw new NegotiationException("cliente com "+id+"nao encontrado");
    }

    cliente.setId(id);
    cliente = gerenciamentoClienteService.salvar(cliente);

    return ResponseEntity.ok(cliente);
}
```

```java
@GetMapping("/{id}")
   public Optional<ResponseEntity> findOne(@PathVariable Long id) {
        var client = clienteRepository.findById(id);
        if (client.isEmpty()) {
                throw new NegotiationException("cliente com id = "+id+" nao encontrado");
        }
        return client.map(ResponseEntity::ok);
   }
```