

MERGULHO SPRING REST  
ALGAWORKS

PEDRO GURGEL DINIZ NETO

**ANOTAÇÕES DO MSR #27**  
Módulos 1 a 3

Natal, RN  
05/2021

## Sumário

1.1.	Fundamentos de REST .....	3
1.2.	Ecosistema Spring .....	5
1.3.	Criando o projeto Spring Boot .....	6
1.4.	Implementando Collection Resource .....	7
1.5.	Métodos e códigos de status HTTP .....	11
1.6.	Content Negotiation .....	11
2.1.	Configurando o Flyway .....	14
2.2.	Usando o Jakarta Persistence (JPA) .....	18
2.3.	Usando o Spring Data JPA .....	21
2.4.	Implementando o CRUD de cliente .....	25
2.5.	Validando com Bean Validation .....	34
2.6.	Implementando Exception Handler .....	38
2.7.	Implementando Domain Services .....	46
3.1.	Implementando solicitação de entrega .....	53
3.2.	Validação em cascata e Validation Groups .....	63
3.3.	Boas práticas para trabalhar com data hora .....	75
3.4.	Isolando Domain Model do Representation Model .....	75
3.5.	Simplificando a transformação de objetos com ModelMapper .....	78
3.6.	Implementando sub recursos .....	89
3.7.	Implementando ação não CRUD .....	99

# MÓDULO 1 – 17/05/2021

## 1.1. Fundamentos de REST

**O que é uma API?** É um conjunto de funções que faz a intermediação de acesso às funcionalidades de algum sistema.

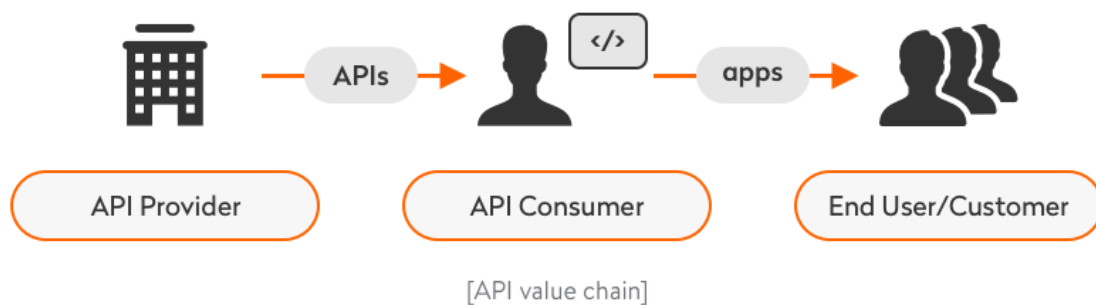


Figura 1 - Intermediação realizada por API.

Todo webservice é uma API, mas nem toda API é um webservice. A conectividade atual da internet do mundo se deve aos webservices.

**O que é REST?** Representational State Transfer, é um conjunto de especificações e boas práticas que define a forma de comunicação entre componentes de softwares na web.

### Quais as vantagens do REST?

- Separação entre cliente e servidor (consumidor e provedor). Dessa forma, um sistema pode evoluir sem o outro, tendo maior flexibilidade e portabilidade.
- Escalabilidade (qualquer servidor provedor pode atender qualquer requisição).
- Independência de linguagens de programação (APIs escritas em diferentes linguagens podem interagir entre si).
- Muitas oportunidades de mercado (integrações entre maiores números de empresas demandam melhores práticas de desenvolvimento e mais profissionais capacitados).

### E quais os constraints do REST?

- Cliente-servidor: a API sempre intermedia um cliente e um servidor. Não há dependência entre eles.
- Stateless: as requisições HTTP devem conter todas as informações necessárias para o seu processamento.
- Cache: a aplicação pode fazer caching do resultado de requisições na aplicação consumidora. Isso diminui a carga no servidor provedor e agiliza a resposta de requisições repetidas ao cliente.
- Interface uniforme: tanto para requisições quanto para métodos, devemos utilizar padrões.
- Sistema em camadas: possibilidade de inclusão de camadas entre o cliente e o servidor, por exemplo, para autenticação, segurança de dados ou balanceamento de carga.
- Código sob demanda: pode retornar código para ser executado na aplicação do cliente.

## Protocolo HTTP

O protocolo HTTP (Hypertext Transfer Protocol) é o protocolo utilizado para comunicação na web. Ele consiste em um padrão de requisições e respostas entre clientes e servidores.

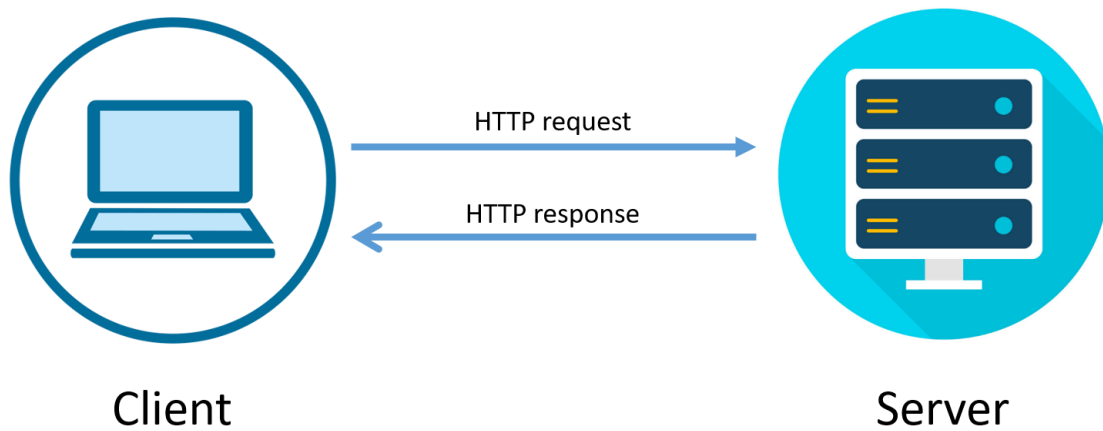


Figura 2 - Comunicação no protocolo HTTP.

**Composição de uma requisição HTTP:** método, URI, Versão HTTP, Cabeçalhos, Corpo/Payload.

- Método: verbos HTTP (GET, PUT, POST, PATCH, ..., DELETE). É a ação que desejamos realizar.
- URI: caminho que identifica o recurso que estamos buscando em nosso servidor.
- Versão HTTP: auto-explicativo.
- Cabeçalhos: chaves e valores contendo informações sobre a requisição que podem ser usados pelo servidor para interpretar a requisição e executar a operação.
  - Content-Type: tipo de conteúdo enviado ao servidor no corpo da requisição.
  - Accept: o que é aceito como resposta pelo cliente.
- Corpo/Payload: opcional, depende do método HTTP. É aqui onde enviamos os dados ao servidor.

**Composição de uma resposta HTTP:** versão HTTP, status, cabeçalhos e corpo.

- Status: código numérico com o resultado da requisição, retornado pelo servidor. Cada código representa uma resposta diferente.
- Cabeçalhos: informações sobre o corpo da resposta.
  - Location: URI acessada.
- Corpo: conteúdo da resposta, opcional, dependendo da requisição.

## Recursos REST

“É qualquer coisa exposta na web, como um documento, uma página, um vídeo, uma imagem ou até mesmo um processo de negócio. É algo importante o suficiente para ser referenciado como uma coisa no software.”

Recurso único (singleton resource): um produto, um usuário.

Coleção de recursos (collection resource): lista de usuários, catálogo de produtos.

Recursos são identificados por URIs (Uniform Resource Identifiers), conjuntos de caracteres que endereçam recursos de forma não ambígua.

URLs, diferentes de URIs, são localizadores completos (links de sites), que especificam também como chegar em um recurso (normalmente pelo protocolo HTTP).

Exemplos de URIs:

- /listarProdutos × - Não devemos usar verbos em nossas URIs.
- /produtos ✓
- /produto/{codigo} × - Nossas URIs devem pluralizar nosso recurso.
- /produtos/{código} ✓

## 1.2. Ecossistema Spring

### O que é Spring?

- Ecossistema de projetos para simplificar a vida do programador Java.
- + desenvolvimento de regras de negócio.
- - tempo perdido em código boilerplate.

### Por que Spring?

- Canivete suíço do Java.
- Simplicidade (pouco complexo).
- Maturidade (alto uso e confiança por empresas).
- Modularidade.
- Evolução constante.
- Open Source.
- Comunidade grande e forte.
- Popularidade.
- Empregabilidade.

**Spring Framework:** é a base do ecossistema Spring.

**Spring Data:** acesso a dados.

- Spring Data JPA: especificação de persistência de dados Java.
- Diminui boilerplate code.

**Spring Boot:** criação de projetos facilmente configuráveis que seguem convenções pré-definidas de acordo com as necessidades do projeto.

- Mais agilidade.
- Não substitui o Spring MVC.
- Não compete com outros componentes do Spring Framework, apenas os complementa.

### 1.3. Criando o projeto Spring Boot

Criaremos um aplicativo básico de entregas. Abaixo, vemos o diagrama de classes.

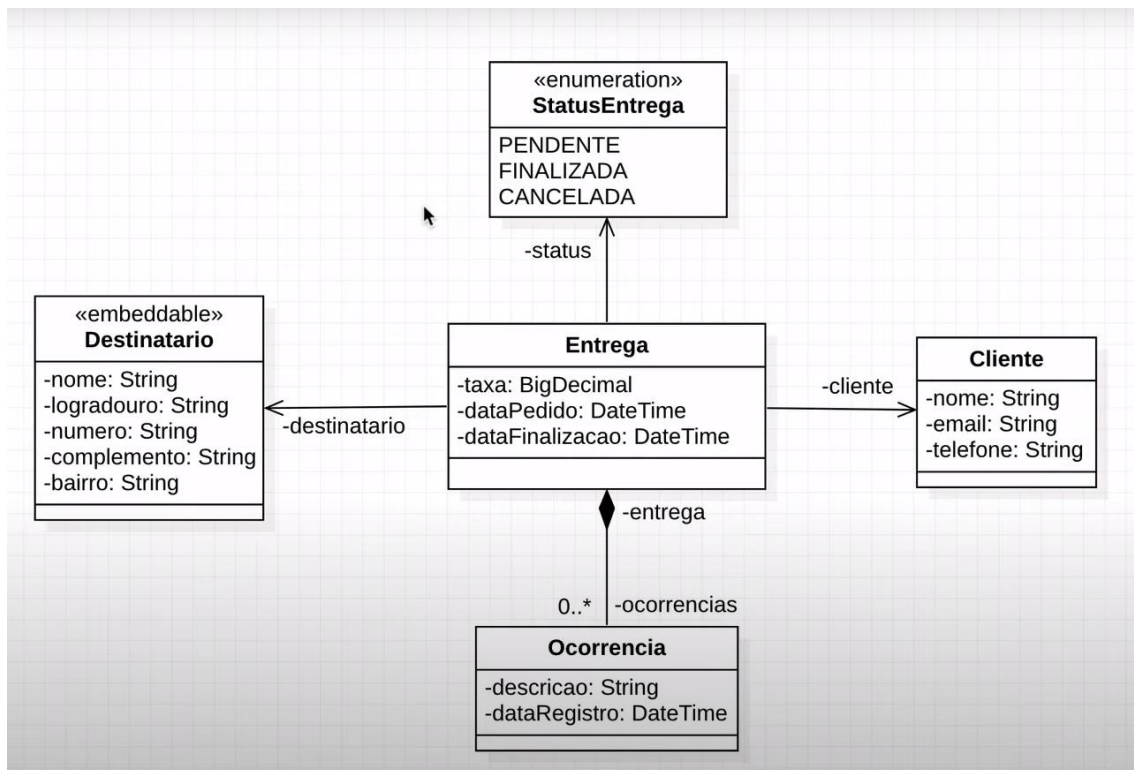


Figura 3 - Diagrama de Classes do projeto AlgaLog.

Para isso, abrimos o Spring Tools Suite (STS) e criamos um Spring Start Project, fazendo as seguintes alterações:

- NAME -> algalog-api
- GROUP -> com.algaworks.algalog
- DESCRIPTION -> API da AlgaLog
- PACKAGE -> com.algaworks.algalog

Na próxima tela, escolhemos os starters do projeto. Starters automaticamente adicionam conjuntos de dependências, agilizando a configuração, evitando adições manuais de dependências e conflitos entre elas. Aqui, adicionamos:

- STARTERS -> Spring Web

O STS nos permite criar os projetos diretamente por ele, mas caso desejemos usar outra IDE, o projeto pode ser criado no Spring Initializr e importado para a IDE de escolha.

No caso de erros na criação do projeto:

- RIGHT CLICK -> MAVEN -> UPDATE PROJECT -> FORCE UPDATE -> OK

Uma vez criado o projeto, encontramos nosso arquivo pom.xml (Project Object Model), que é o coração dos projetos Maven.

Dentro da tag <parent> do pom.xml, herdamos as configurações do Spring Boot Start Parent.

Também no pom.xml, encontramos informações sobre o pacote, group e description criados previamente, assim como a versão do Java e sobre as dependências do projeto.

Em <dependencies>, adicionamos as bibliotecas que usaremos no projeto. Uma dependência padrão para testes, a spring-boot-starter-test, é adicionada em todos os projetos, mas não é compilada no projeto final, servindo apenas para testes internos da aplicação.

Dependências que estão dentro de outras dependências são chamadas de dependências transitivas.

Podemos ver uma lista de todas as dependências transitivas ao clicar em Maven Dependencies. Por mais que até agora só tenhamos duas dependências no nosso pom.xml, todas as dependências transitivas dessas duas estão instaladas. Essas dependências transitivas ficam no repositório remoto do Maven, na pasta .m2.

Para rodar nossa aplicação, precisamos construí-la primeiro. Em consequência disso, geramos um “fat jar”, ou jar gordo, que contém todas as dependências do projeto em um arquivo compilado. Para construir o projeto no STS:

- RIGHT CLICK(ROOT FOLDER) -> RUN AS -> MAVEN BUILD... -> GOALS(clean package) -> RUN.

Após esse processo, o Maven preenche uma pasta chamada target com vários arquivos e, entre eles, o nosso fat jar.

Já podemos rodar nosso fat jar ao entrarmos na pasta target pelo terminal digitando:

- Java -jar algalog-api-0.0.1-SNAPSHOT.jar

Ainda não conseguimos fazer nada com nossa aplicação, pois não implementamos nada nela. A aplicação já funciona pois roda em um “servlet container”, o TomCat, que funciona como um servidor web embarcado na aplicação que compilamos.

Não precisamos, entretanto, ficar iniciando nossa API pelo prompt de comando. É mais cômodo fazer isso pela IDE.

## 1.4. Implementando Collection Resource

No Postman, uma Collection é um grupo de requisições. Vamos criar uma collection lá:

- COLLECTIONS -> CREATE COLLECTION(AlgaLog) -> ADD A REQUEST(Clientes – Listar)

Preenchemos a nossa request da seguinte forma:

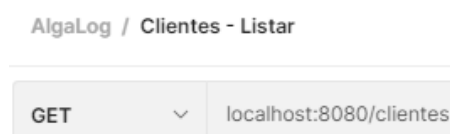


Figura 4 - GET request na URI clientes.

Ao executá-la, recebemos o erro mostrado na figura 5. Por quê? Bom, não programamos nada até agora. Nossa API ainda não possui nenhum tratamento para requisições de clientes, logo, recebemos essa resposta. Vamos dar uma olhada na estrutura do nosso projeto antes de mais nada.

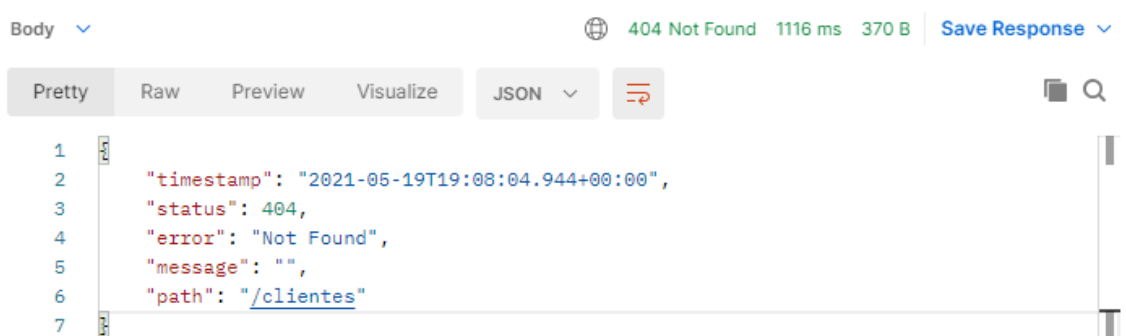


Figura 5 - Resposta da nossa GET request. Ainda não criamos o nosso endpoint.

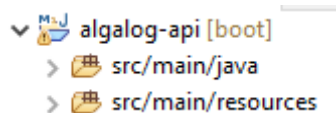


Figura 6 - Estrutura de pastas do projeto.

Em `src/main/java` ficam todos os nossos arquivos de código Java.

Em `src/main/resources` ficam os arquivos estáticos, como imagens ou páginas HTML, por exemplo.

Vamos começar criando uma **classe Controller**, que é **responsável por receber requisições HTTP e retornar uma resposta**.

Em `src/main/java/com.algaworks.algalog`, criamos nosso `ClienteController`:

- PACKAGE -> `com.algaworks.algalog.api.controller`
- NAME -> `ClienteController`

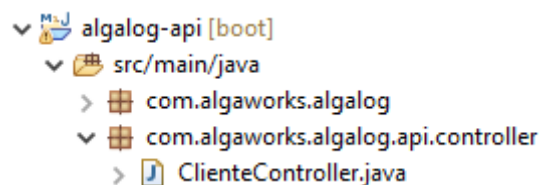


Figura 7 - Criação do `ClienteController`.

Dentro do nosso `ClienteController`, incluímos o seguinte código:

```
@RestController
public class ClienteController {

    @GetMapping("/clientes")
    public String listar() {
        return "Teste";
    }
}
```

- `@RestController` anota o `ClienteController` como capaz de lidar com requisições HTTP.
- `@GetMapping` cria o endpoint `/clientes` para requisições HTTP GET.

Ao recarregar nossa API e executar nossa GET request novamente pelo Postman, recebemos:



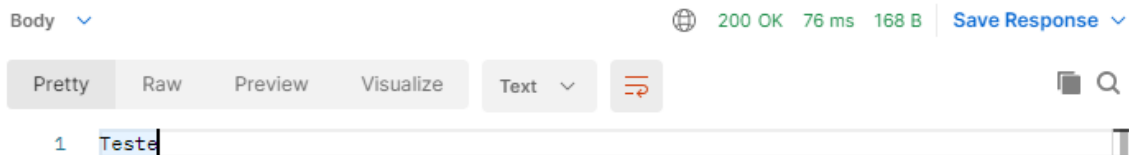


Figura 8 - Nova resposta à nossa primeira GET request.

Entretanto, não é do nosso interesse receber “Teste”. Queremos receber uma lista de clientes. Logo, precisamos implementar a classe `Cliente`. Para isso, em `src/main/java/com.algaworks.algalog`, criamos:

- PACKAGE -> `com.algaworks.algalog.domain.model`
- NAME -> `Cliente`

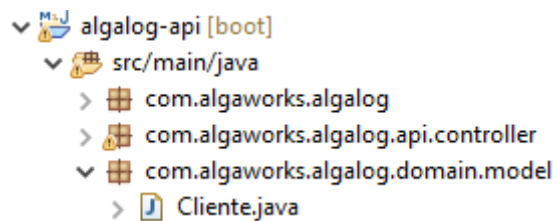


Figura 9 - Criação da classe `Cliente`.

Dentro da classe, inserimos:

```
@Getter
@Setter
public class Cliente {

    private Long id;
    private String nome;
    private String email;
    private String telefone;

}
```

`@Getter` e `@Setter` são anotações do Lombok, uma biblioteca que diminui o volume de código boilerplate em nosso projeto. Nesse caso, essas anotações equivalem aos métodos `Get` e `Set` de cada propriedade da classe.

Porém, ainda não instalamos o Lombok! E agora? Bom...

RIGHT CLICK(ROOT PROJECT) -> SPRING -> ADD STARTERS -> LOMBOK -> NEXT -> pom.xml -> FINISH.

Isso não só instala o Lombok no projeto, como atualiza o nosso `pom.xml` automaticamente para incluí-lo.

Agora que a classe `Cliente` está pronta, voltemos ao método `listar()` no `ClienteController`. Lá, vamos criar dois objetos da classe `Cliente` para testarmos o retorno da API. Nosso `ClienteController` vai ficar assim:

```
@RestController
public class ClienteController {

    @GetMapping("/clientes")
    public List<Cliente> listar() {
        Cliente cliente1 = new Cliente();
    }
}
```

```

        cliente1.setId(1L);
        cliente1.setNome("João");
        cliente1.setTelefone("34 99999-1111");
        cliente1.setEmail("joaodascouves@algaworks.com");

        Cliente cliente2 = new Cliente();
        cliente2.setId(2L);
        cliente2.setNome("Maria");
        cliente2.setTelefone("11 97777-2222");
        cliente2.setEmail("mariadasilva@algaworks.com");

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

```

Primeiro, alteramos o tipo do retorno do método de String para uma List<Cliente>, ou lista de clientes. Em seguida, instanciamos dois objetos da classe Cliente e os preenchemos com as informações mostradas acima. Por último, dizemos que o retorno do método é uma lista (Arrays.asList) com os dois objetos.

Repetindo nossa GET request anterior:

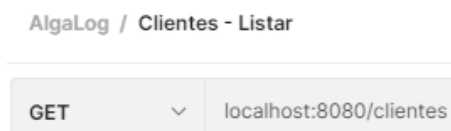


Figura 10 - Re-execução da GET request anterior.

Obtemos a resposta abaixo:

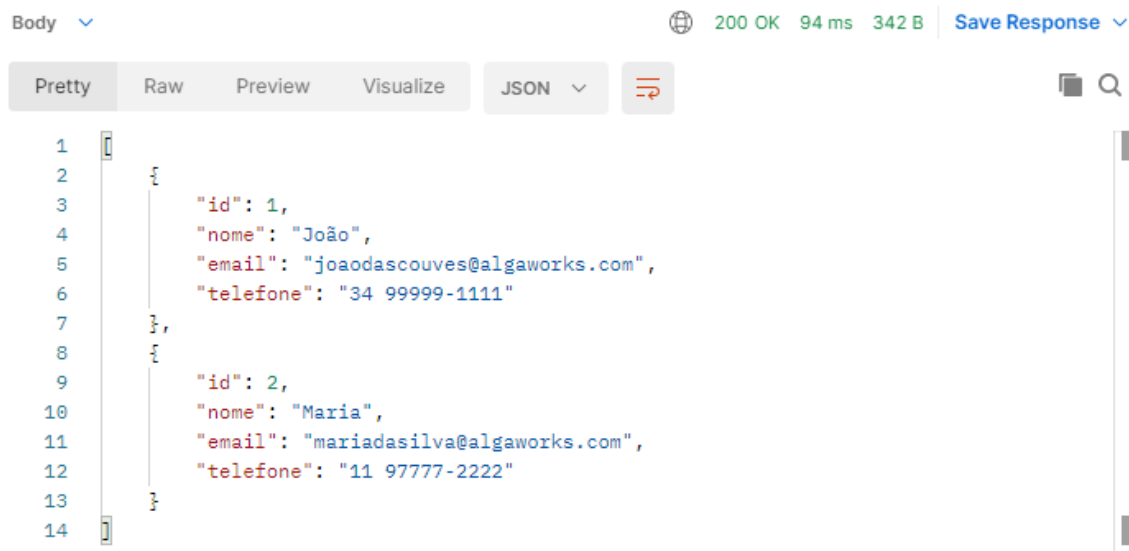


Figura 11 - Resposta à nossa GET request, agora retornando uma lista de clientes.

Recebemos os dois clientes no formato de dados JSON. O JSON é o formato tradicional de dados para REST APIs. Os colchetes [] indicam o início e o fim do nosso JSON, envolvendo uma Collection Resource. As chaves {} indicam objetos individuais, ou singleton resources. Cada linha da representação acima representa uma propriedade de um objeto.

## 1.5. Métodos e códigos de status HTTP

Métodos HTTP possuem a semântica do que queremos fazer com o nosso recurso.

Através do método nós dizemos ao servidor qual tipo de ação queremos executar em um determinado recurso identificado pela URI.

Até então vimos as GET requests, que obtêm uma representação de um recurso. GET requests costumam ser idempotentes, retornam o mesmo resultado em todas as suas chamadas de uma mesma URI, caso não haja nenhuma regra de negócio que as altere.

As respostas HTTP sempre possuem um código numérico HTTP. Esses códigos são explanados em diversas fontes. Uma delas, onde podemos pesquisar, é o [httpstatuses.com](http://httpstatuses.com).

**Os códigos no padrão 2xx indicam que uma requisição foi bem-sucedida.**

- 200 OK
- 201 CREATED
- 204 NO CONTENT

**Os códigos no padrão 3xx indicam redirecionamento.**

- 301 MOVED PERMANENTLY

**Códigos 4xx indicam erro do cliente.**

- 400 BAD REQUEST
- 404 NOT FOUND

**Códigos 5xx indicam erros no servidor.**

- 500 INTERNAL SERVER ERROR

## 1.6. Content Negotiation

Como dito anteriormente, JSON é o formato de representação padrão para recursos em REST APIs, mas podemos visualizar um mesmo recurso em diferentes formatos, e isso pode ser especificado pelo cliente da aplicação.

O cliente da API especifica o formato do retorno no campo “Accept” da requisição.

No Postman, podemos alterar os Headers da requisição para obter respostas em diferentes formatos.

- KEY -> Accept
- VALUE -> application/json (parâmetro conhecido como media type)

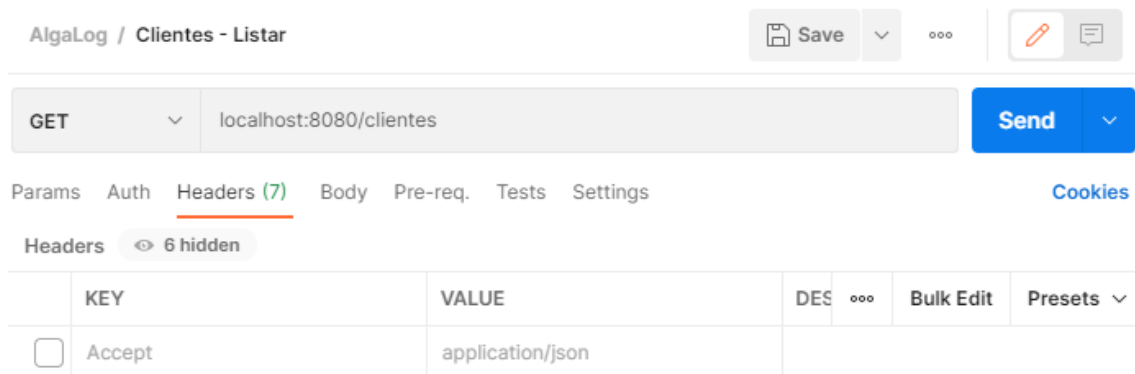


Figura 12 - Headers de uma GET request no Postman.

Executar essa requisição nos retorna a mesma resposta da requisição anterior, pois esses KEY e VALUE já são valores padrão para as respostas, que naturalmente retornam JSON. Mas e se mudássemos o VALUE para `application/xml`?

Recebemos a seguinte resposta, mostrada na imagem. Repare no código HTTP 406, NOT ACCEPTABLE. Isso significa que não é permitido ao cliente realizar requisições que pedem um formato de retorno xml. E agora?

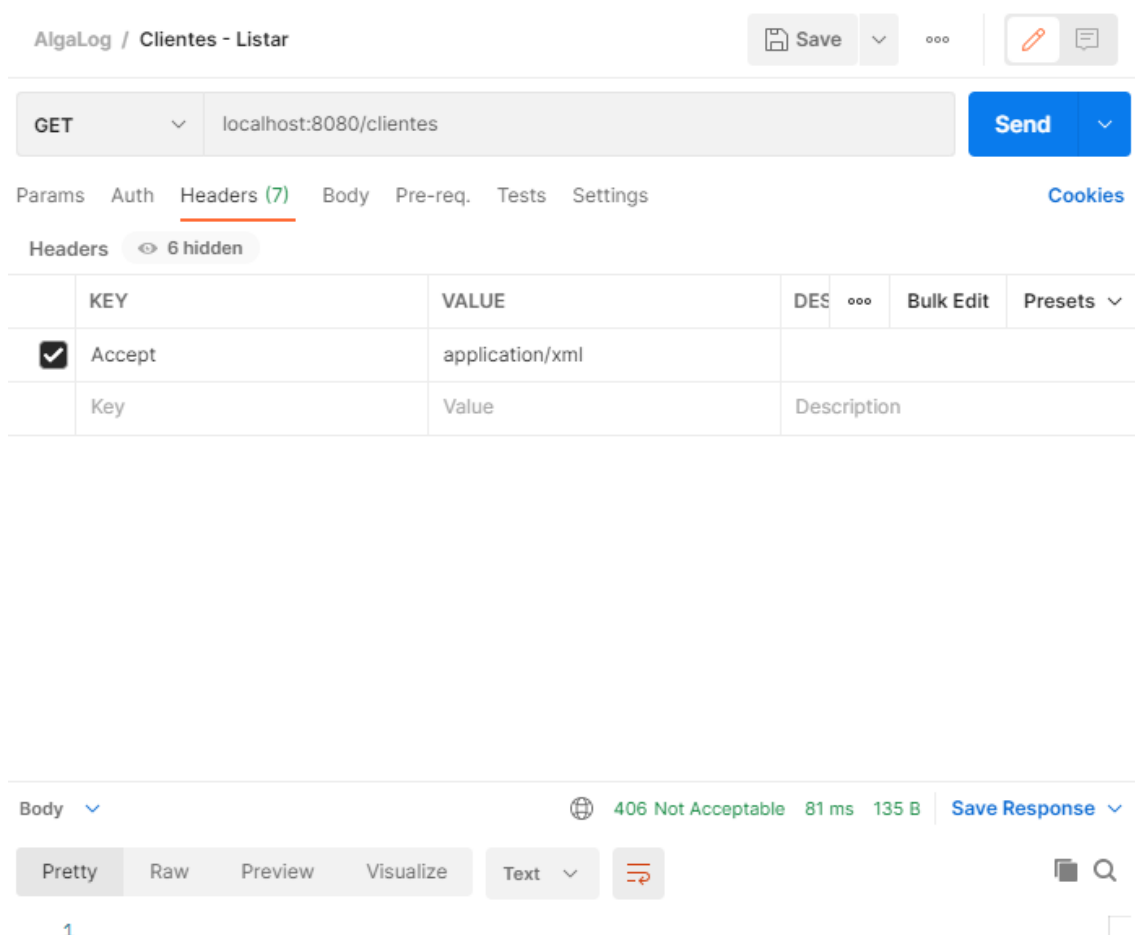


Figura 13 - GET request malsucedida. Requisições que esperam uma resposta em .xml não são aceitas.

Bom, podemos incluir uma dependência em nosso `pom.xml` para realizar a serialização de nossos dados para o formato xml.

```

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

```

Figura 14 - Inclusão de novo artefato da dependência Jackson para tratar requisições em xml.

O Jackson é a dependência para transformar objetos Java em outros tipos de objeto, e ela é usada por padrão para fazer as transformações para JSON, mas não para xml. Por isso, adicionamos essa dependência: para suportarmos respostas em xml.

Repetindo a requisição anterior com o Header Accept application/xml, obtemos uma nova resposta.

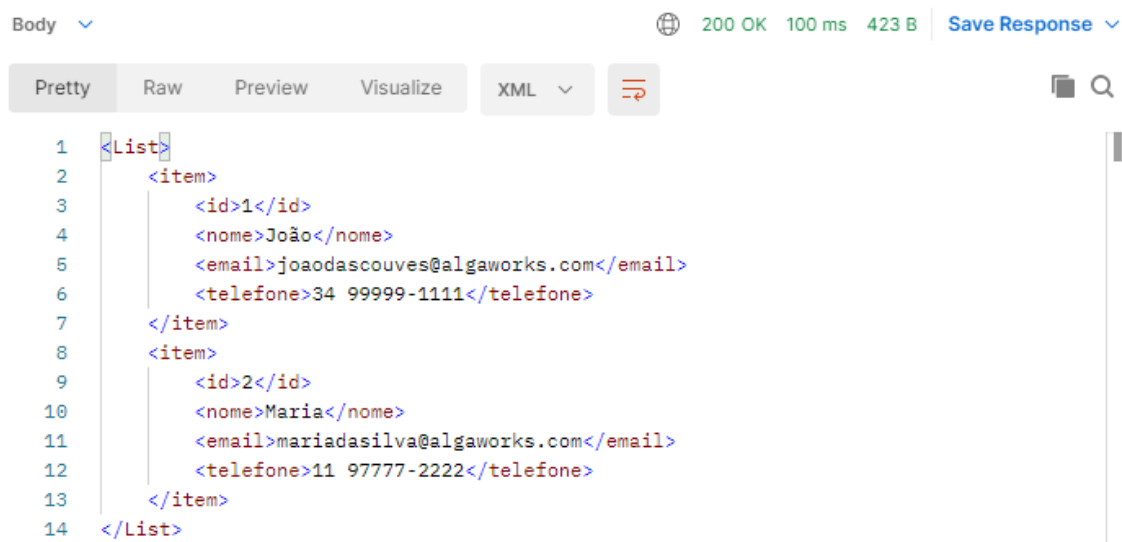


Figura 15 - Resposta xml à GET request.

E se repetirmos nossa requisição com o VALUE = application/json? Também funciona! Ao incluir um novo tipo de representação em nossa API, não perdemos a capacidade de lidar com os formatos de representação anteriores. Agora nossa aplicação é capaz de retornar JSON e xml.

O nome disso é **Content Negotiation**: permitir que o consumidor da API escolha em qual formato ele quer manusear e representar os dados.

A maioria das APIs trabalham com JSON. Por isso, seguiremos o curso focando somente em JSON, e não em xml. Portanto, removeremos a dependência que nos permite suportar xml.

No geral, **é sempre bom consultar os consumidores da nossa API para sabermos com qual tipo de dado eles preferem trabalhar.**

# MÓDULO 2 – 19/05/2021

Até então, quando alteramos algo no código do nosso sistema, temos que reiniciar o TomCat manualmente. É possível, entretanto, automatizar esse processo, aumentando nossa produtividade no processo de desenvolvimento.

Para isso, adicionamos as DevTools em:

- RIGHT CLICK(ROOT FOLDER) -> SPRING -> ADD DEVTOOLS

Feito isso, na hora que salvarmos qualquer alteração no nosso código, ele recarrega o TomCat.

## 2.1. Configurando o Flyway

A partir de agora, vamos iniciar a persistência de dados em nossa aplicação. Vamos instalar o Spring Data JPA em nossa aplicação, que não será utilizada de cara, mas que pré-configura o nosso projeto para utilizarmos bancos de dados, e o My SQL Driver, propriamente para o banco de dados.

- RIGHT CLICK(ROOT PROJECT) -> SPRING -> ADD STARTERS -> SPRING DATA JPA, MY SQL DRIVER -> NEXT -> pom.xml -> FINISH.

Ao adicionar os dois starters mencionados acima e tentar executar a aplicação, nos deparamos com a seguinte mensagem de erro:

```
Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2021-05-19 17:41:04.661 ERROR 19056 --- [ restartedMain] o.s.b.d.LoggingFailureAnalysisReporter :

*****
APPLICATION FAILED TO START
*****

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Action:

Consider the following:
  If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
  If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).
```

Figura 16 - Erro ao tentar inicializar a aplicação sem definir as propriedades do banco de dados.

**Para configurarmos o banco de dados, vamos em application.properties, que fica em src/main/resources. Esse arquivo é utilizado para configurar nossa aplicação.**

A conexão feita com o banco é feita pela seguinte linha:

```
spring.datasource.url=jdbc:mysql://localhost:3306/algalog?createDatabaseIfNotExist=true&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=12345678
```

Em que:

- localhost é o IP da máquina,
- 3306 é a porta padrão do MySQL, podendo ser ocultada,
- algalog é o nome da nossa database,
- ?createDatabaseIfNotExist significa que o Spring irá analisar se o banco já existe e, se não existir, ele o cria para nós.

- &serverTimezone=UTC é uma boa prática de salvar as horas no banco de dados em UTC, ou GMT 00.
- username é o nome do nosso usuário do banco de dados
- password é a senha do nosso usuário do banco de dados

Em ambientes de teste, podemos usar “root” e uma senha simples ou até não usar senha, mas o ideal é que autentiquemos um usuário na hora de nos conectarmos ao banco de dados.

Podemos criar nossa schema do banco de dados diretamente no MySQL Workbench, mas já que configuramos a própria aplicação para fazê-lo, vamos executar a aplicação e ver o resultado.

Ao irmos em schema no nosso MySQL Workbench, já conseguimos ver que o banco algalog foi criado, e que nele ainda não consta nenhuma tabela.

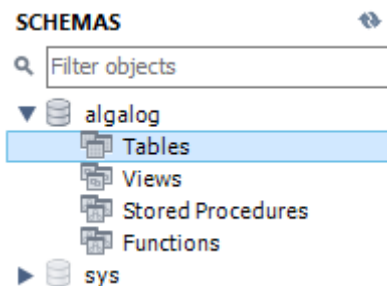


Figura 17 - Schema "algalog" criada pela nossa aplicação Spring, visualizada dentro do MySQL Workbench.

Tendo a schema criada, podemos começar a estruturar nossa aplicação, criando suas tabelas. Mas como organizar a criação de tabelas e seu manuseio? **Alterações em bancos de dados costumam causar problemas quando não são organizadas por migrations.**

**Migrations são uma forma de versionamento de bancos de dados**, presentes em frameworks de várias linguagens, como o Laravel (PHP) e o Django (Python). **No Spring, podemos utilizar o Flyway para versionar e organizar as migrations em nosso banco de dados.**

Começamos, portanto, adicionando o starter do Flyway:

- RIGHT CLICK(ROOT PROJECT) -> SPRING -> ADD STARTERS -> FLYWAY MIGRATION -> NEXT -> pom.xml -> FINISH.

Ao executar o projeto após a inclusão do Flyway, nos deparemos com o seguinte erro:

```
Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2021-05-19 18:48:39.538 ERROR 984 --- [ restartedMain] o.s.b.d.LoggingFailureAnalysisReporter :

*****
APPLICATION FAILED TO START
*****

Description:

Flyway failed to initialize: none of the following migration scripts locations could be found:

- classpath:db/migration

Action:

Review the locations above or check your Flyway configuration
```

Figura 18 - Erro causado pelo Flyway na inicialização do projeto.

O erro indica que não existe uma pasta chamada db/migration. Portanto, vamos criá-la em src/main/resources.

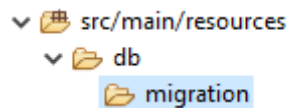


Figura 19 - Criação da pasta db e, dentro dela, da pasta migration.

Isso ainda não é suficiente para que a aplicação funcione. Precisa criar os scripts de migration. Cada alteração na estrutura do nosso banco de dados será um arquivo dentro da pasta db/migration.

A criação de arquivos de migration seguem a seguinte convenção:

- V maiúsculo
- String crescente (normalmente um número ou uma data)
- Dois underscores \_\_
- Informação sobre a utilidade daquela migration
- .sql

Seguindo essa convenção, criamos o nosso arquivo V001\_\_cria-tabela-cliente.sql. Agora, já conseguimos rodar nossa aplicação, entretanto, esse arquivo está vazio. Para abri-lo dentro da IDE, podemos:

- RIGHT CLICK(V001\_\_cria-tabela-cliente.sql) -> OPEN WITH -> TEXT EDITOR

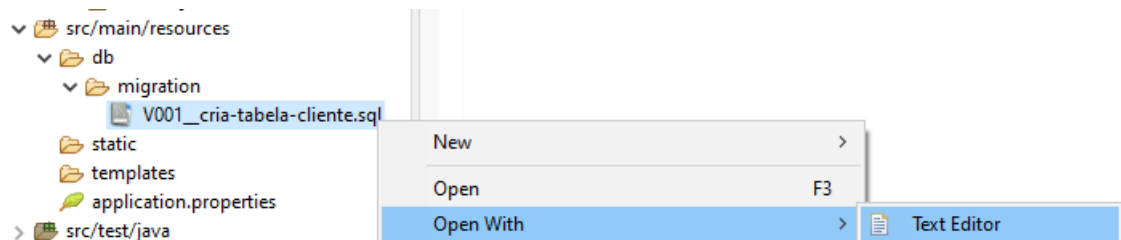


Figura 20 - Abrindo nossa migration dentro do STS. Podemos também arrastá-lo para as abas da IDE.

Isso abrirá o arquivo em uma janela dentro da IDE.

É complicado criar o nosso código SQL diretamente dentro da migration, pois não temos como testá-lo lá se não inicializando a aplicação inteira. Por isso, é melhor abrir um query builder dentro do MySQL Workbench para testar o código SQL.

Para facilitar os testes de SQL no Workbench, podemos:

- RIGHT CLICK(algalog) -> SET AS DEFAULT SCHEMA

Agora, testamos o código de criação da tabela cliente no Workbench.

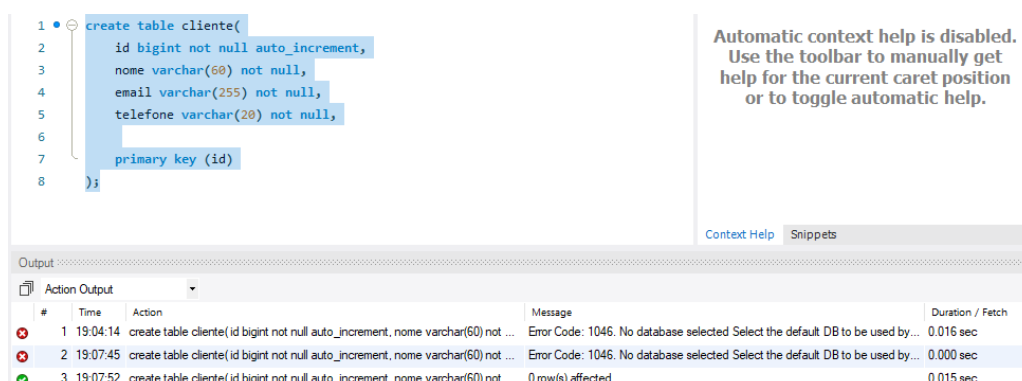


Figura 21 - Código de criação da tabela cliente sendo testado no Workbench.



Entretanto, não queremos criar o código pelo MySQL Workbench, apenas testá-lo. Por isso, vamos deletar a tabela cliente criada (Drop Table) e copiar o código SQL para o nosso arquivo V001\_\_cria-tabela-cliente.sql, que deve conter a migration.

Ao tentar executar a aplicação novamente, nos deparamos com um novo erro:

```
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'flywayInitializer' defined in class path resource [org/springframework/boot/autoconfigure/flyway/FlywayAutoConfiguration$FlywayConfiguration.class]: Invocation of init method failed; nested exception is org.flywaydb.core.api.exception.FlywayValidateException: Validate failed: Migrations have failed validation
```

**E por que esse erro aconteceu?** Cada arquivo migration é único. Na primeira vez que executamos a aplicação após criarmos o V001\_\_cria-tabela-cliente.sql, esse arquivo se encontrava vazio. Portanto, na tabela flyway\_schema\_history, onde se mantém o controle das migrations, já há um registro de que para esse arquivo, no qual ele aparecia vazio. Tentar modificá-lo gera esse erro. Na prática, devemos criar uma nova migration para inserirmos as mudanças.

Para desfazer essa primeira migration, podemos simplesmente deletá-la da tabela flyway\_schema\_history e, agora, com o nosso código SQL dentro dele, executar a aplicação.

Finalmente, ao tentar rodar nossa aplicação mais uma vez, nossa aplicação roda, e a tabela “cliente” é criada em nosso banco de dados.

Vamos praticar criar uma segunda migration, alterando o nome da coluna telefone na tabela cliente para fone.

Queremos evitar o mesmo problema que ocorreu vez passada: que ao criar uma nova migration, o sistema recarregue automaticamente e rode a migration vazia. Podemos contornar esse problema de duas formas:

- Podemos parar a aplicação, criar nossa migration corretamente e, assim que terminarmos, executá-la.
- Ou podemos criar a migration com um nome fora do padrão, preenchê-la e, depois, mudar o nome para um nome condizente com o padrão.

Escolhi parar o servidor para criar nossa migration, V002\_\_renomeia-coluna-telefone.sql, e, dentro dela, inseri o código:

```
alter table cliente rename column telefone to fone;
```

Ao executar o sistema, vemos que a coluna foi alterada na tabela do banco de dados.

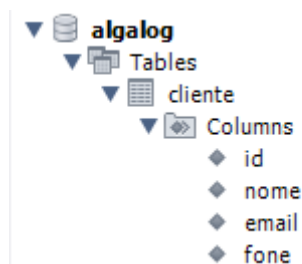


Figura 22 - Tabela cliente após alteração da coluna "telefone" para "fone".

## 2.2. Usando o Jakarta Persistence (JPA)

Para começar a implementação da persistência de dados (código Java que dá acesso ao nosso banco de dados), usaremos, em conjunto com o Spring, a especificação de persistência de dados do Jakarta EE, que é uma especificação geral do Java. Essa especificação de persistência se chama Jakarta Persistence API, ou JPA.

O JPA nos ajuda a mapear as tabelas de um banco de dados relacional em classes Java, e ao fazer isso, podemos fazer operações no banco de dados usando a API do JPA, que deixa nosso trabalho muito mais simples no banco de dados.

Se verificarmos nossa hierarquia de dependências no pom.xml, podemos ver a especificação jakarta.persistence-api lá.

**Toda especificação precisa de uma implementação. Quem implementa a Jakarta Persistence API? O Hibernate Core.**

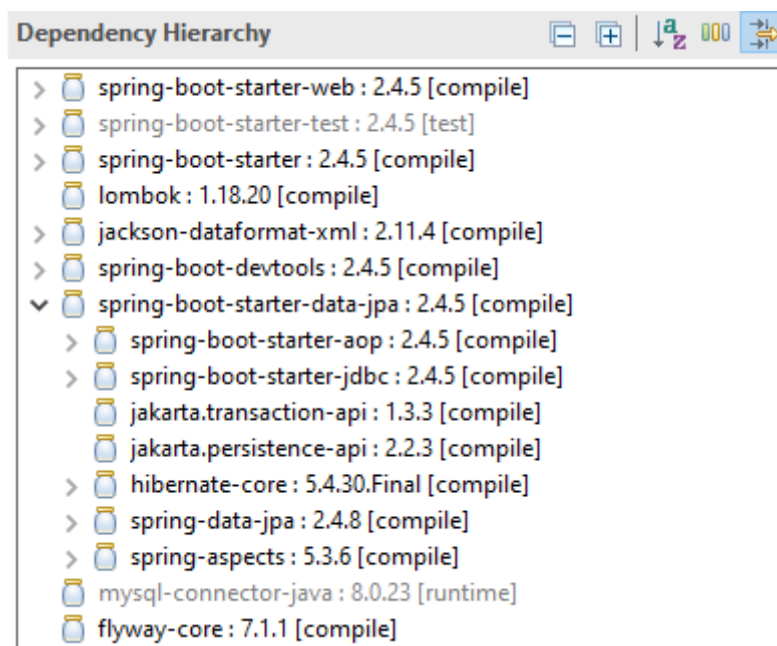


Figura 23 - Hierarquia de dependências.

**Spring Data JPA não é uma implementação da especificação Jakarta Persistence. Spring Data JPA é apenas uma biblioteca que nos ajuda a trabalhar com Jakarta Persistence.**

Agora, precisamos fazer nosso mapeamento objeto-relacional, indicando que uma propriedade de um objeto Java seja equivalente uma coluna em uma tabela do banco de dados.

Até então, só temos uma classe e uma tabela: cliente. Para realizar essa ligação, anotamos a classe Cliente com @Entity.

```
@Getter
@Setter
@Entity
public class Cliente {

    private Long id;
    private String nome;
    private String email;
    private String telefone;
```

```
}
```

Por padrão, o `@Entity` associa um objeto Java a uma tabela do mesmo nome. Se um objeto deve ser relacionado a uma tabela que não tem o mesmo nome que a classe Java, podemos anotar com `@Table`. Nesse caso, nossa classe ficaria da seguinte forma:

```
@Getter
@Setter
@Table(name="cliente")
public class Cliente {

    private Long id;
    private String nome;
    private String email;
    private String telefone;

}
```

Na verdade, temos mais alterações e mais anotações feitas, então vamos entendê-las:

```
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Getter
@Setter
@Entity
public class Cliente {

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

    private String nome;
    private String email;

    @Column(name="fone")
    private String telefone;

}
```

Começando pelo `@Column`. Da mesma forma que temos o `@Table` para especificar relações com tabelas de nomes diferente do nosso objeto, temos o `@Column` para especificar relações de propriedades com nomes diferentes das suas respectivas colunas na tabela do banco de dados. Nesse caso, estamos dizendo que a propriedade “telefone” está vinculada à coluna “fone” da tabela “cliente”.

`@Id` anota a propriedade `id` do nosso objeto como a chave primária da tabela, e o `@GeneratedValue(strategy = GenerationType.IDENTITY)` explicita o autoincremento de ids na criação de objetos em nossa aplicação.

`@EqualsAndHashCode(onlyExplicitlyIncluded = true)` criam esses métodos para as propriedades da nossa classe, e o argumento da anotação significa que só queremos criar para propriedades específicas, explicitamente incluídas. Mais especificamente o `id`. Por isso, anotamos o `id` com `@EqualsAndHashCode.Include`.

Agora que preparamos a classe `Cliente` e fizemos todas as devidas conexões com o banco de dados, vamos lá no `ClienteController` apagar aqueles dois objetos da classe `Cliente` que criamos estaticamente, pois agora queremos interagir com o banco de dados.

Para isso, vamos instanciar um tipo chamado EntityManager, do javax.persistence. Esse EntityManager é uma classe do javax.persistence utilizada para fazer as operações com as entidades que são refletidas no banco de dados.

Depois de mais algumas alterações, a classe ClienteController ficou com essa cara:

```
@RestController
public class ClienteController {

    @PersistenceContext
    private EntityManager manager;

    @GetMapping("/clientes")
    public List<Cliente> listar() {
        return manager.createQuery("from Cliente",
        Cliente.class).getResultList();
    }

}
```

Além de adicionarmos o EntityManager, precisamos anotá-lo com a anotação @PersistenceContext. Esse novo tipo, instanciado na variável “manager”, é capaz de realizar as operações de persistência em nossas entidades Java que se comunicam com o banco de dados, evitando a necessidade de instanciar cada nova entidade abordada.

Para buscar a lista de clientes no banco de dados, agora, a função listar() retorna:

- O método .getResultList(), que obtém a lista de objetos referentes à query construída em
- .createQuery(), que recebe dois parâmetros: uma String de JPQL (Java Persistence Query Language, bem similar a SQL) e o nome da classe da entidade, proveniente de um objeto
- EntityManager, chamado manager.

Se, nesse momento, fizermos uma GET request pelo Postman em localhost:8080/clientes, receberemos uma lista vazia, pois até então não temos nenhum registro em nossa tabela cliente.

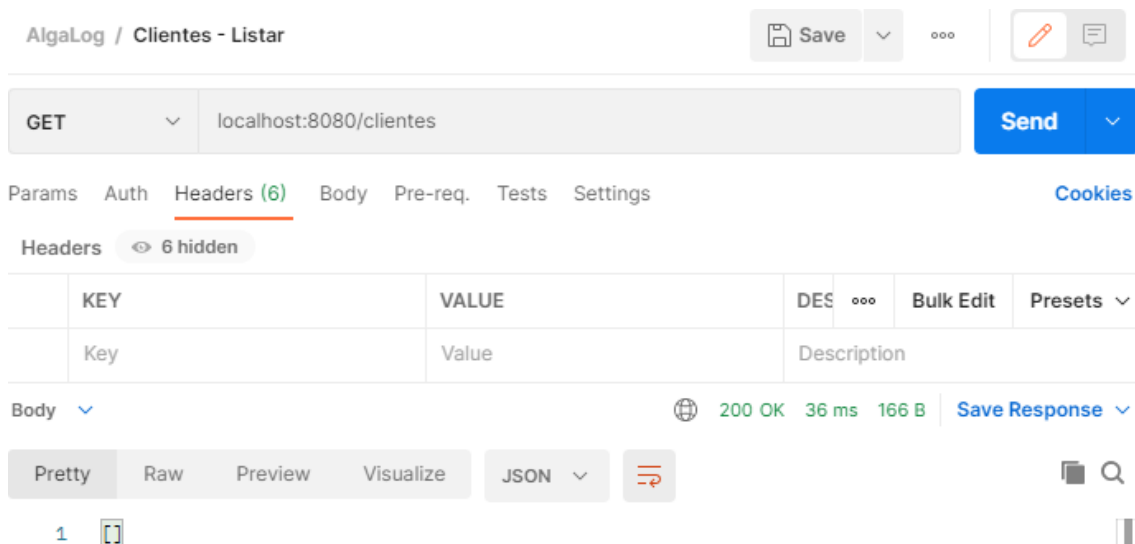


Figura 24 - GET request após configurarmos nosso controller para buscar registros na tabela cliente.

Vamos popular nossa tabela com alguns registros.

	id	nome	email	fone
	NULL	João da Silva	joaodasilva@algaworks.com	34 99990-3333
	NULL	Maria Abadia	mariaabadia@algaworks.com	11 8888-3333
	NULL	NULL	NULL	NULL

Figura 25 - Inclusão de registros na tabela cliente diretamente pelo MySQL Workbench. Para confirmar, basta clicar em Apply.

A mesma requisição, agora, nos retorna o seguinte:

The screenshot shows a REST client interface with a 'Body' tab selected. The response is a JSON array containing two objects, each representing a client record. The status is 200 OK, and the response size is 354 B. The JSON content is as follows:

```

1  [
2    {
3      "id": 1,
4      "nome": "João da Silva",
5      "email": "joaodasilva@algaworks.com",
6      "telefone": "34 99990-3333"
7    },
8    {
9      "id": 2,
10     "nome": "Maria Abadia",
11     "email": "mariaabadia@algaworks.com",
12     "telefone": "11 8888-3333"
13   }
14 ]

```

Figura 26 - Resposta da GET request em localhost:8080/clientes após a adição de registros no banco de dados.

Para termos mais visibilidade de como nossa aplicação está processando as transações com o banco de dados, podemos alterar o nosso arquivo application.properties para mostrar o SQL gerado pela aplicação no console da IDE.

Basta incluir a seguinte linha:

- `spring.jpa.show-sql=true`

## 2.3. Usando o Spring Data JPA

Spring Data JPA é uma biblioteca que usaremos em nossa aplicação para criar repositórios com o Jakarta Persistence.

**E o que é um repositório?** É uma classe que implementa métodos que fazem as operações de persistência de dados. Atualmente, **nossa aplicação está operando com o banco de dados diretamente no Controller, o que não é uma boa prática.**

Em projetos profissionais, devemos separar as responsabilidades das classes, e denotarmos que o acesso aos dados deve ser feito por uma classe especializada, os repositórios.

Partindo do src/main/java, vamos criar nosso primeiro repositório, o ClienteRepository, que ao invés de uma classe, é criada como uma interface:

- RIGHT CLICK(src/main/java) -> NEW -> INTERFACE
- PACKAGE -> com.algaworks.algalog.domain.repository
- NAME -> ClienteRepository

Dentro do nosso clienteRepository, o configuramos da seguinte forma:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.algaworks.algalog.domain.model.Cliente;

@Repository
public interface ClienteRepository extends JpaRepository<Cliente, Long>{

}
```

Em que a interface herdada JpaRepository é a interface padrão do Spring Data JPA, os parâmetros são Cliente (classe da entidade que será gerenciada pelo repositório) e Long (tipo do id da entidade).

A anotação @Repository define que a interface ClienteRepository é um componente do Spring com uma interface bem definida, um repositório, que gerencia as entidades.

**E o que é um componente Spring? É um tipo onde as instâncias desse tipo são gerenciadas pelo próprio container do Spring, e nisso podemos injetar uma instância desse repositório ClienteRepository de uma forma muito fácil em objetos de outras classes usando injeção de dependências.**

Lembra quando criamos o EntityManager ali em cima? Pois é, utilizá-lo no nosso Controller não é uma boa prática. Portanto, vamos injetar uma instância de ClienteRepository no nosso Controller.

```
@Autowired
private ClienteRepository clienteRepository;
```

**O @Autowired nos poupa de instanciarmos uma ClienteRepository vazia, e sim injetarmos uma instância que está sendo gerenciada pelo Spring. Não é possível instanciar interfaces, apenas instanciar classes, e o ClienteRepository é uma interface. Entretanto, o Spring Data JPA oferece uma implementação da interface do repositório automaticamente, e em tempo de execução pelo seu IoC Container.**

Feito isso, podemos apagar o EntityManager, e nosso Controller fica assim:

```
@RestController
public class ClienteController {

    @Autowired
    private ClienteRepository clienteRepository;

    @GetMapping("/clientes")
    public List<Cliente> listar() {
        return clienteRepository.findAll();
    }

}
```

Agora, o consumidor da nossa API receba uma lista de clientes em sua requisição, utilizamos apenas o método .findAll() do nosso ClienteRepository, que retorna uma lista com todos os clientes. Simples, né?

Também podemos injetar o repositório sem utilizar o `@Autowired`, gerando um construtor para o nosso Controller recebendo o Repository como parâmetro.

- RIGHT CLICK(CONTROLLER) -> SOURCE -> GENERATE CONSTRUCTOR USING FIELDS.

Isso gera o seguinte código em nosso Controller:

```
public ClienteController(ClienteRepository clienteRepository) {  
    super();  
    this.clienteRepository = clienteRepository;  
}
```

Para deixar o código mais limpo, podemos usar o Lombok para gerar o mesmo construtor de forma oculta. Veja só:

```
@AllArgsConstructor  
@RestController  
public class ClienteController {  
  
    private ClienteRepository clienteRepository;  
  
    @GetMapping("/clientes")  
    public List<Cliente> listar() {  
        return clienteRepository.findAll();  
    }  
  
}
```

Certo, mas... **Se eu posso escrever `@Autowired` em cada repositório injetado, por que usar o `@AllArgsConstructor`?** O `@AllArgsConstructor` cria construtores para todos os repositórios localizados dentro de um Controller. Ou seja, se eu tiver 3 repositórios ali, seriam 3 `@Autowired` que eu substituo apenas por uma anotação no topo, o `@AllArgsConstructor`.

Repetindo a requisição em `localhost:8080/clientes`, recebemos a mesma resposta de antes:

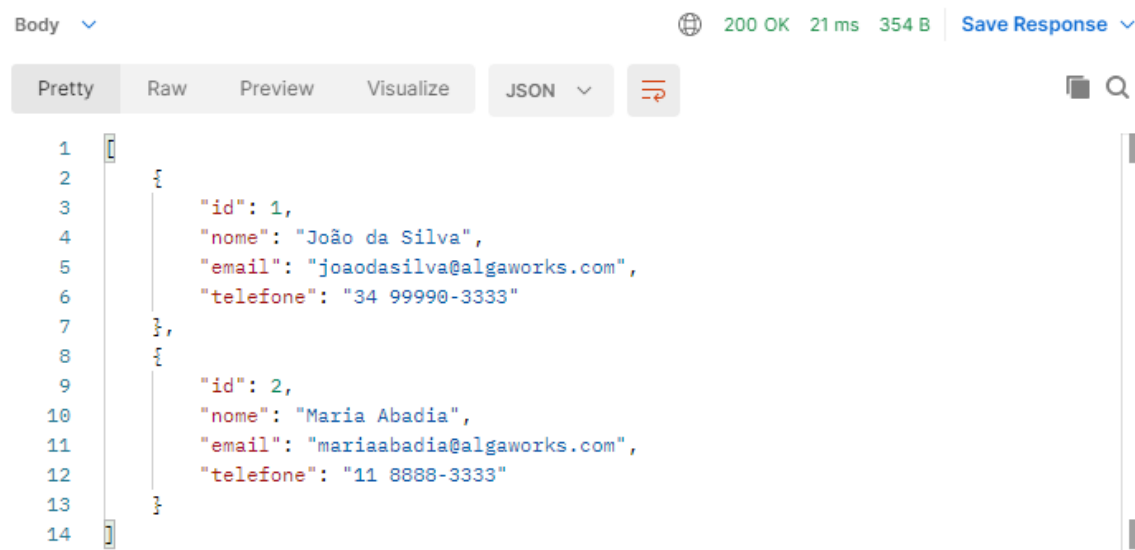


Figura 27 - GET request em `localhost:8080/clientes` após deixarmos de usar o `EntityManager` e começarmos a usar o `Repository`.

Por padrão, o Repository só dispõe de dois métodos para encontrar registros em nosso banco de dados: `.findAll()` e `.findById()`, além de variações desses dois métodos. **E se quisermos métodos para achar por nome, por telefone ou por e-mail?** Implementamos esses métodos em nosso Repository. Vamos iniciar criando um método para pesquisar por nome.

```
@Repository
public interface ClienteRepository extends JpaRepository<Cliente, Long>{

    List<Cliente> findByNome(String nome);
}
```

Ao invés da criação como um método comum, criamos o `findByNome` como uma propriedade do nosso Repository, que, ao seguir o `JpaRepository`, converte para um método real. Destrinchando-a, entendemos que:

- `List<Cliente>` é o tipo de retorno que queremos do método.
- `findBy` é um prefixo padrão do Spring JPA para criação do método.
- `Nome` é o nome da propriedade que queremos integrar ao nosso método para buscar na entidade. Seguindo o camel case, sua primeira letra precisa ser maiúscula mesmo.
- `(String nome)` é a propriedade que queremos buscar em nossa entidade.

Vamos alterar o `ClienteController` para testar esse novo método e enviar uma nova requisição:

```
@GetMapping("/clientes")
public List<Cliente> listar() {
    return clienteRepository.findByNome("João da Silva");
}
```



Figura 28 - Resposta para GET request feita com nosso Controller utilizando o método `.findByNome("João da Silva")`.

Da forma como implementamos o `findByNome`, ele faz uma pesquisa exata por um nome de cliente em nosso banco de dados. Para pesquisar não exatas, podemos criar um método com o sufixo `Containing`, que gera uma pesquisa não exata, utilizando o `LIKE` do SQL.

```
List<Cliente> findByNomeContaining(String nome);
```

Fazendo uma nova requisição, mudando nossa função `listar()` para:

```
@GetMapping("/clientes")
public List<Cliente> listar() {
    return clienteRepository.findByNomeContaining("Silva");
}
```

Recebemos o mesmo retorno da imagem acima.



## 2.4. Implementando o CRUD de cliente

Agora que aprendemos a utilizar o Jakarta Persistence e o Spring Data JPA, vamos implementar o nosso CRUD. Vamos começar criando um endpoint para realizar a busca de um cliente por id.

```
@GetMapping("clientes/{clienteId}")
public Cliente buscar(@PathVariable Long clienteId) {
    Optional<Cliente> cliente =
        clienteRepository.findById(clienteId);

    return cliente.orElse(null);
}
```

Explicando cada item:

- `@GetMapping("clientes/{clienteId}")` – criamos um novo endpoint, que agora recebe uma path variable, ou variável de caminho, que é o `{clienteId}`, envolvida por chaves. O valor imputado aqui será recebido dentro de nossa função `buscar()`
- `public Cliente` – retorno da função `buscar()`
- `buscar(@PathVariable Long clienteId)` – a anotação `@PathVariable` serve justamente para injetar a variável de caminho, `{clienteId}`, dentro da nossa função `buscar()`.
- `Optional<Cliente> cliente` – ao realizarmos uma busca em nosso banco de dados, podemos encontrar ou não um registro. Sendo assim, utilizamos um container chamado `Optional`, que poderá ou não conter um objeto da classe `Cliente` dentro da variável `cliente`.
- `clienteRepository.findById(clienteId)` – método de busca por Id do repository.
- `return cliente.orElse(null)` – retorno da função. Nos utilizando nos métodos do container `Optional<Cliente>`, verificamos se um `Cliente` foi encontrado e, se não, retornamos `null`.

Para facilitar, duplicamos nossa antiga request no Postman, a renomeamos para `Clientes – Obter` e editamos a URI para agora receber o `clienteId`.

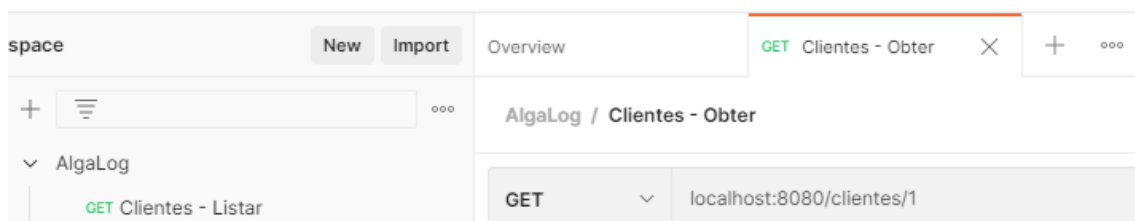


Figura 29 - Nova request, Clientes - Obter, no Postman.

Ao criarmos requests com os `clienteId` 1 e 2, recebemos os registros que temos em nosso banco de dados, mas ao passarmos `clienteId` 10, recebemos uma resposta vazia.



Figura 30 - Resposta para a request em `/clientes/1`.

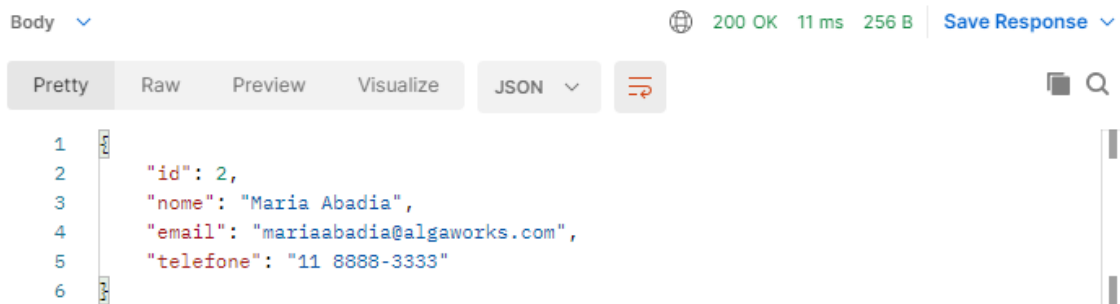


Figura 31 - Resposta para a request em /clientes/2.

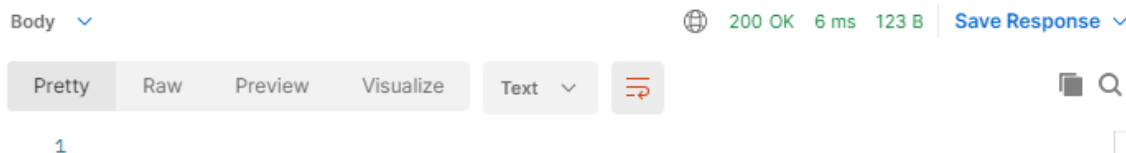


Figura 32 - Resposta para a request em /clientes/10.

Repare, entretanto, que o código HTTP da resposta, até onde não obtivemos resposta, foi 200 OK. Não é o melhor código para se retornar no caso de um retorno nulo, portanto, por boa prática, devemos retornar 404 NOT FOUND.

Considerando que, atualmente, estamos retornando um objeto do tipo Cliente no método buscar(), não temos muita flexibilidade de alterar os códigos HTTP retornados. Por isso, vamos encapsular nosso retorno em um container chamado `ResponseEntity`, conforme mostrado abaixo:

```
@GetMapping("/clientes/{clienteId}")
public ResponseEntity<Cliente> buscar(@PathVariable Long clienteId) {
    Optional<Cliente> cliente =
        clienteRepository.findById(clienteId);

    if (cliente.isPresent()) {
        return ResponseEntity.ok(cliente.get());
    }

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

Agora, ao invés de retornarmos um objeto `Cliente`, retornamos uma `ResponseEntity` que contém um `Cliente`. **Legal, e qual a vantagem?** O container `ResponseEntity` possui métodos para alterar o código HTTP da nossa resposta.

Então utilizando o método `.isPresent()` do container `Optional<Cliente>`, verificamos se nossa requisição retornou um cliente ou não. Se houver um cliente dentro do `Optional`, retornamos:

- `ResponseEntity` – tipo do retorno
- `.ok()` – marca o código HTTP para 200
- `cliente.get()` – o objeto cliente dentro do container `Optional<Cliente>`.

Se não encontrarmos um cliente dentro do `Optional`, retornamos:

- `ResponseEntity` – tipo do retorno
- `.notFound()` – marca o código HTTP para 404
- `.build()` – constrói uma resposta HTTP sem corpo.

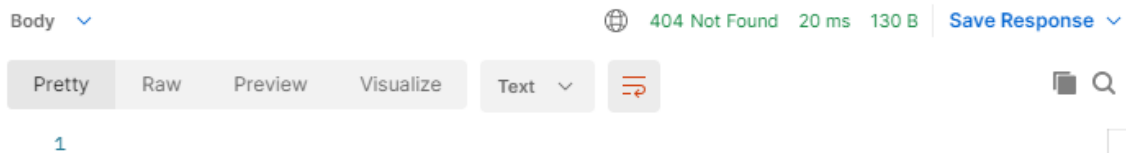


Figura 33 - GET request em /clientes/10. Perceba o status 404.

Podemos enxugar esse código utilizando o método `.map` com uma lambda expression:

```
@GetMapping("clientes/{clienteId}")
public ResponseEntity<Cliente> buscar(@PathVariable Long clienteId) {
    return clienteRepository.findById(clienteId)
        .map(cliente -> ResponseEntity.ok(cliente))
        .orElse(ResponseEntity.notFound().build());
}
```

Dessa forma, não precisamos escrever o nosso `if` nem o `Optional`. Dá para enxugar mais ainda a chamada do `ResponseEntity.ok()`, pois fica implícito que estamos invocando o método a partir de um objeto cliente, que era a classe do seu argumento:

```
@GetMapping("clientes/{clienteId}")
public ResponseEntity<Cliente> buscar(@PathVariable Long clienteId) {
    return clienteRepository.findById(clienteId)
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
}
```

As requisições feitas anteriormente continuam funcionando da mesma forma após as últimas alterações.

Concluídos os nossos métodos GET, vamos criar outros métodos para manuseio de requisições HTTP no `ClienteController`, que atualmente está assim:

```
@AllArgsConstructor
@RestController
public class ClienteController {

    private ClienteRepository clienteRepository;

    @GetMapping("/clientes")
    public List<Cliente> listar() {
        return clienteRepository.findAll();
    }

    @GetMapping("clientes/{clienteId}")
    public ResponseEntity<Cliente> buscar(@PathVariable Long clienteId) {
        return clienteRepository.findById(clienteId)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }
}
```

Nos nossos dois métodos, colocamos `“clientes/”` no `@GetMapping`. Vamos precisar fazer isso para todos os outros métodos? Sim, mas há como definir a URI clientes automaticamente para todos os métodos do Controller, através do `@RequestMapping`.

```
@RequestMapping("/clientes")
@AllArgsConstructor
```

```

@RestController
public class ClienteController {

    private ClienteRepository clienteRepository;

    @GetMapping
    public List<Cliente> listar() {
        return clienteRepository.findAll();
    }

    @GetMapping("/{clienteId}")
    public ResponseEntity<Cliente> buscar(@PathVariable Long clienteId) {
        return clienteRepository.findById(clienteId)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }
}

```

Agora que simplificamos a criação de novos métodos no nosso Controller, estamos prontos para criar nosso método POST. Vamos começar no Postman:

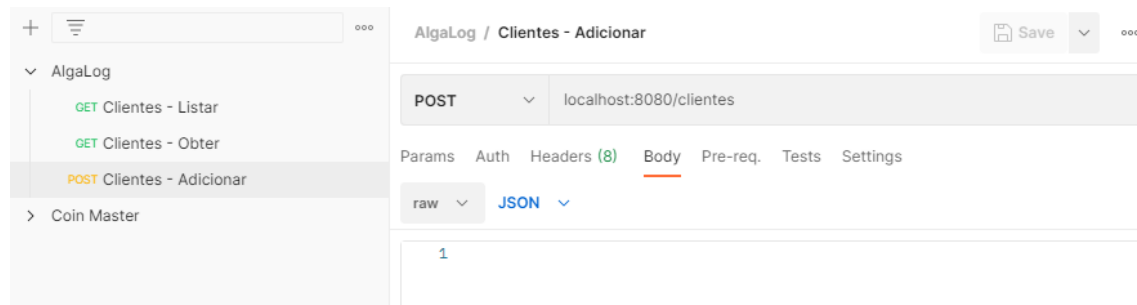


Figura 34 - Criação de request Clientes - Adicionar em nossa Collection no Postman.

Se preencheremos o corpo do método POST com dados e tentarmos submeter, receberemos um erro:

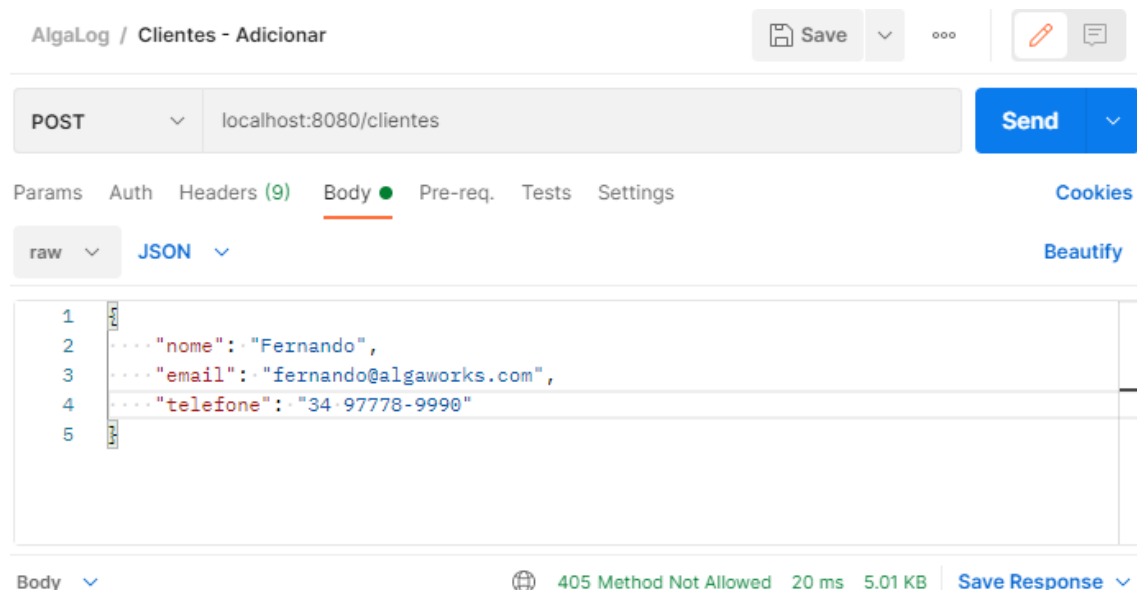


Figura 35 - POST request em /clientes. Ainda não implementamos um método no nosso ClienteController para lidar com POST requests.

Repare no código HTTP retornado: 405 METHOD NOT ALLOWED. Isso quer dizer que não é permitido realizar POST requests nesse endpoint. Pelo menos não ainda. Vamos criar o método para POST requests no ClienteController.

```
@PostMapping
public Cliente adicionar(@RequestBody Cliente cliente) {
    return clienteRepository.save(cliente);
}
```

- @PostMapping – anota o método para receber POST requests.
- @RequestBody Cliente cliente – desserializa o JSON enviado pela requisição e o converte automaticamente em um objeto do tipo cliente.
- return clienteRepository.save(cliente) – salva o nosso objeto cliente, preenchido com os dados enviados no corpo da POST request, em nosso banco de dados, e o retorna na API para mostrar o que foi salvo no banco de dados.

Ao repetir a requisição anterior, obtemos esse resultado:

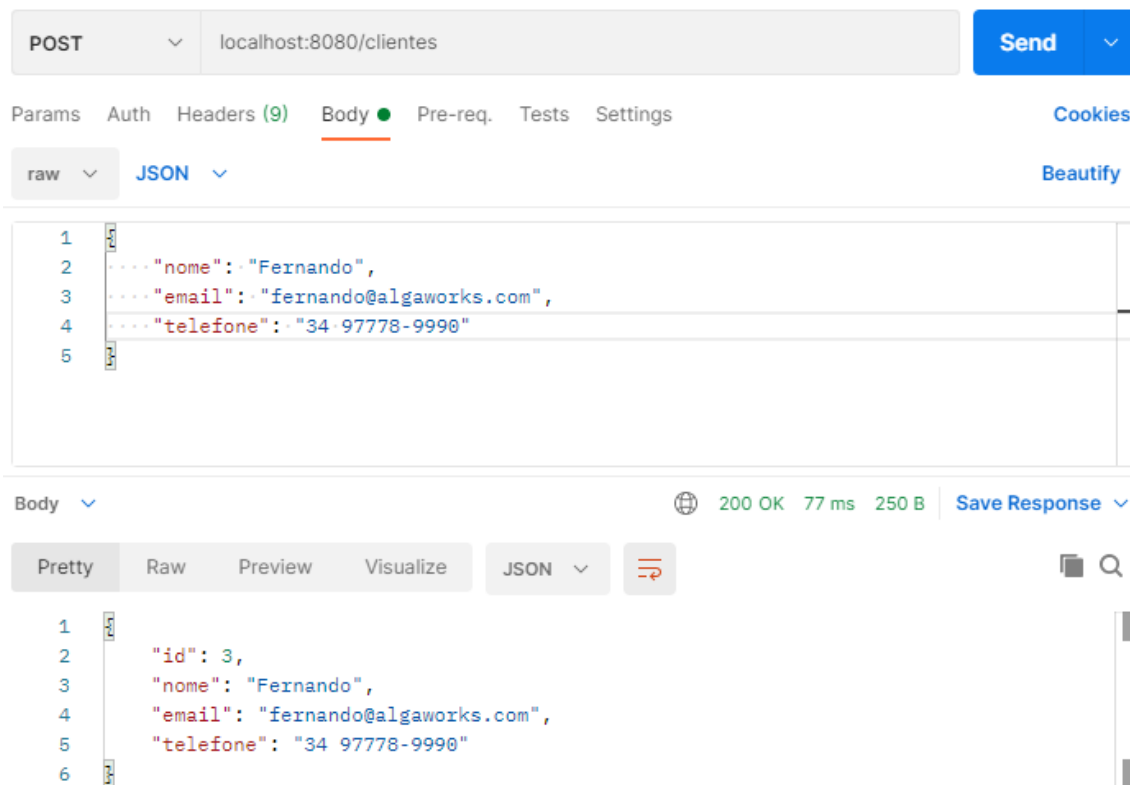


Figura 36 - POST request em /clientes. A requisição foi bem-sucedida, gerando um id para o cliente e salvando-o no banco de dados.

Apesar da nossa requisição ter retornado 200 OK para demonstrar que houve sucesso, considerando que criamos um registro em nosso banco de dados, é mais indicado usar o código 201 CREATED.

Para isso, podemos encapsular o Cliente em uma ResponseEntity ou, mais fácil ainda, anotamos a resposta que queremos de um método do Controller logo acima dele com o @ResponseStatus:

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Cliente adicionar(@RequestBody Cliente cliente) {
    return clienteRepository.save(cliente);
}
```

}

Enviando uma nova POST request...

POST localhost:8080/clientes Send

Params Auth Headers (9) Body Pre-req. Tests Settings Cookies

raw JSON Beautify

```
1 {
2   "nome": "Fernando Silva",
3   "email": "fernandosilva@algaworks.com",
4   "telefone": "34 97778-9990"
5 }
```

Body 201 Created 14 ms 266 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 4,
3   "nome": "Fernando Silva",
4   "email": "fernandosilva@algaworks.com",
5   "telefone": "34 97778-9990"
6 }
```

Figura 37 - Nova POST request em /clientes. Observe o código HTTP retornado.

Deu certo retornar nosso código 201. Vamos testar agora a listagem:

GET localhost:8080/clientes

Params Auth Headers (6) Body Pre-req. Tests Settings

Headers 6 hidden

KEY	VALUE	DE
-----	-------	----

Body 200 OK 14 ms

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "nome": "João da Silva",
4   "email": "joaodasilva@algaworks.com",
5   "telefone": "34 99990-3333"
6 },
7 {
8   "id": 2,
9   "nome": "Maria Abadia",
10  "email": "mariaabadia@algaworks.com",
11  "telefone": "11 8888-3333"
12 },
13 {
14  "id": 3,
15  "nome": "Fernando",
16  "email": "fernando@algaworks.com",
17  "telefone": "34 97778-9990"
18 },
19 {
20  "id": 4,
21  "nome": "Fernando Silva",
22  "email": "fernandosilva@algaworks.com",
23  "telefone": "34 97778-9990"
24 }
25 }
```

Figura 38 - Listagem de clientes após inserções.

Tudo certo! Agora, vamos implementar uma forma de atualizar os registros no banco de dados.

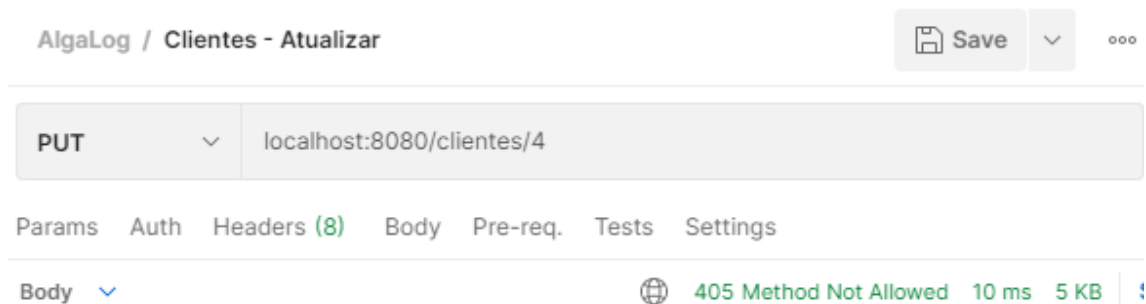


Figura 39 - Tentativa de PUT request. Como ainda não implementamos no Controller, recebemos erro 405.

Uma vez criado o método atualizar(), ele fica com essa cara:

```
@PutMapping("/{clienteId}")
public ResponseEntity<Cliente> atualizar(@PathVariable Long clienteId,
    @RequestBody Cliente cliente) {
    if (!clienteRepository.existsById(clienteId)) {
        return ResponseEntity.notFound().build();
    }

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

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

Perceba que ele parece uma mistura dos métodos buscar() e adicionar(). Vamos destrinchá-lo:

- `@PutMapping("/{clienteId}")` – anotação para método HTTP Put, recebendo uma Path Variable, `clienteId`.
- `ResponseEntity<Cliente>` – ao invés de utilizarmos a anotação `@ResponseStatus`, precisamos encapsular essa resposta em uma `ResponseEntity`, pois precisamos retornar diferentes códigos HTTP caso encontremos um cliente ou não.
- `atualizar(@PathVariable Long clienteId, @RequestBody Cliente cliente)` – recebe por parâmetros da função tanto o `clienteId` como variável de caminho como o corpo da requisição para atualizar o registro.
- `if (!clienteRepository.existsById(clienteId))` – verifica se existe um cliente referente àquele `clienteId` e, se não existir, retorna um código HTTP 404.
- `cliente.setId(clienteId)` – associa o `clienteId` recebido pela `@PathVariable` às propriedades do cliente recebido no `@RequestBody`. Dessa forma, impedimos que um novo cliente seja criado com o método `clienteRepository.save(cliente)`.

Se der tudo certo, atualizamos o cliente, e o retornamos junto com o código HTTP 200 OK. Vamos testar:

PUT localhost:8080/clientes/4

Params Auth Headers (9) **Body** Pre-req. Tests Settings

raw JSON

```
1 {
2   "nome": "Fernando Silva2222",
3   "email": "fernandosilva@algaworks.com",
4   "telefone": "34 97778-9990"
5 }
```

Body 200 OK 28 ms 265 B

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 4,
3   "nome": "Fernando Silva2222",
4   "email": "fernandosilva@algaworks.com",
5   "telefone": "34 97778-9990"
6 }
```

Figura 40 - PUT request alterando o nome do nosso cliente 4 de Fernando Silva para Fernando Silva2222.

GET localhost:8080/clientes/4

Params Auth Headers (7) Body Pre-req. Tests Settings

Body 200 OK 9 ms 265 B

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 4,
3   "nome": "Fernando Silva2222",
4   "email": "fernandosilva@algaworks.com",
5   "telefone": "34 97778-9990"
6 }
```

Figura 41 - GET request no cliente modificado.

Assim, conseguimos ver que deu tudo certo com nossa alteração. Se tentarmos fazer uma PUT request em um cliente que não existe, recebemos um código HTTP 404.





Figura 42 - Tentativa de PUT request em cliente inexistente.

Para finalizar o nosso CRUD, vamos implementar o método de exclusão.

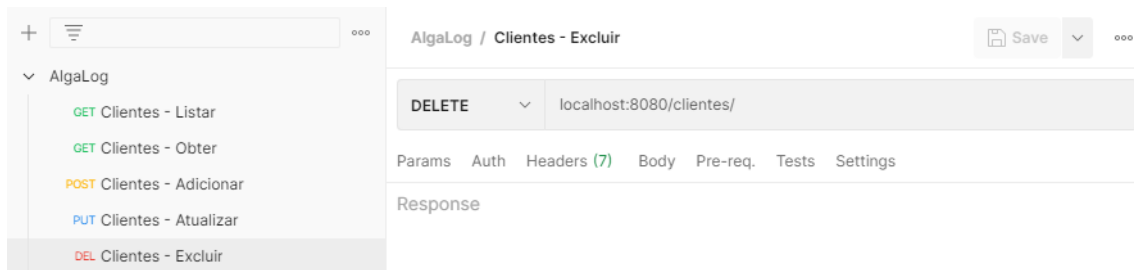


Figura 43 - DELETE request em nossa coleção do Postman.

Se tentarmos executar essa request agora, recebemos 405 METHOD NOT ALLOWED. Vamos implementá-lo:

```
@DeleteMapping("/{clienteId}")
public ResponseEntity<Void> remover(@PathVariable Long clienteId) {
    if (!clienteRepository.existsById(clienteId)) {
        return ResponseEntity.notFound().build();
    }

    clienteRepository.deleteById(clienteId);

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

Destrinchando o código acima:

- `ResponseEntity<Void>` – o método HTTP DELETE não retorna nenhum corpo, portanto, o corpo da nossa `ResponseEntity` é `Void`.
- `@PathVariable Long clienteId` – quando deletamos um registro, fazemos isso por id. Recebemos o `clienteId` pela nossa variável de caminho.
- `if (!clienteRepository.existsById(clienteId))` – verifica se existe um cliente referente àquele `clienteId` e, se não existir, retorna um código HTTP 404.
- `clienteRepository.deleteById(clienteId)` – se o cliente existir, o deleta a partir do `clienteId` recebido pela variável de caminho.

- `return ResponseEntity.noContent().build()` – retorna o código HTTP 204 NO CONTENT, que não possui corpo, logo, usa o `.build()`.

Fazendo o teste no nosso novo método no cliente 1, obtemos a seguinte resposta:

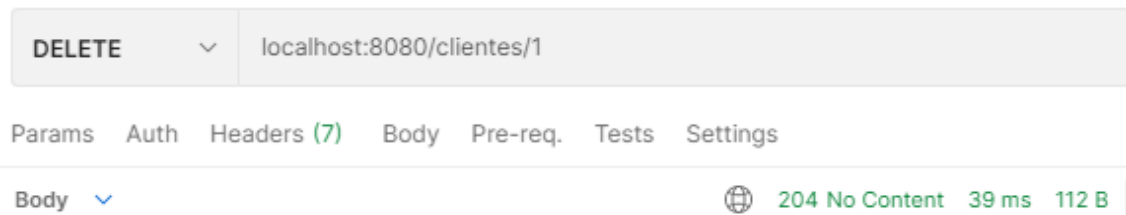


Figura 44 - DELETE request do cliente com `clienteId = 1`.

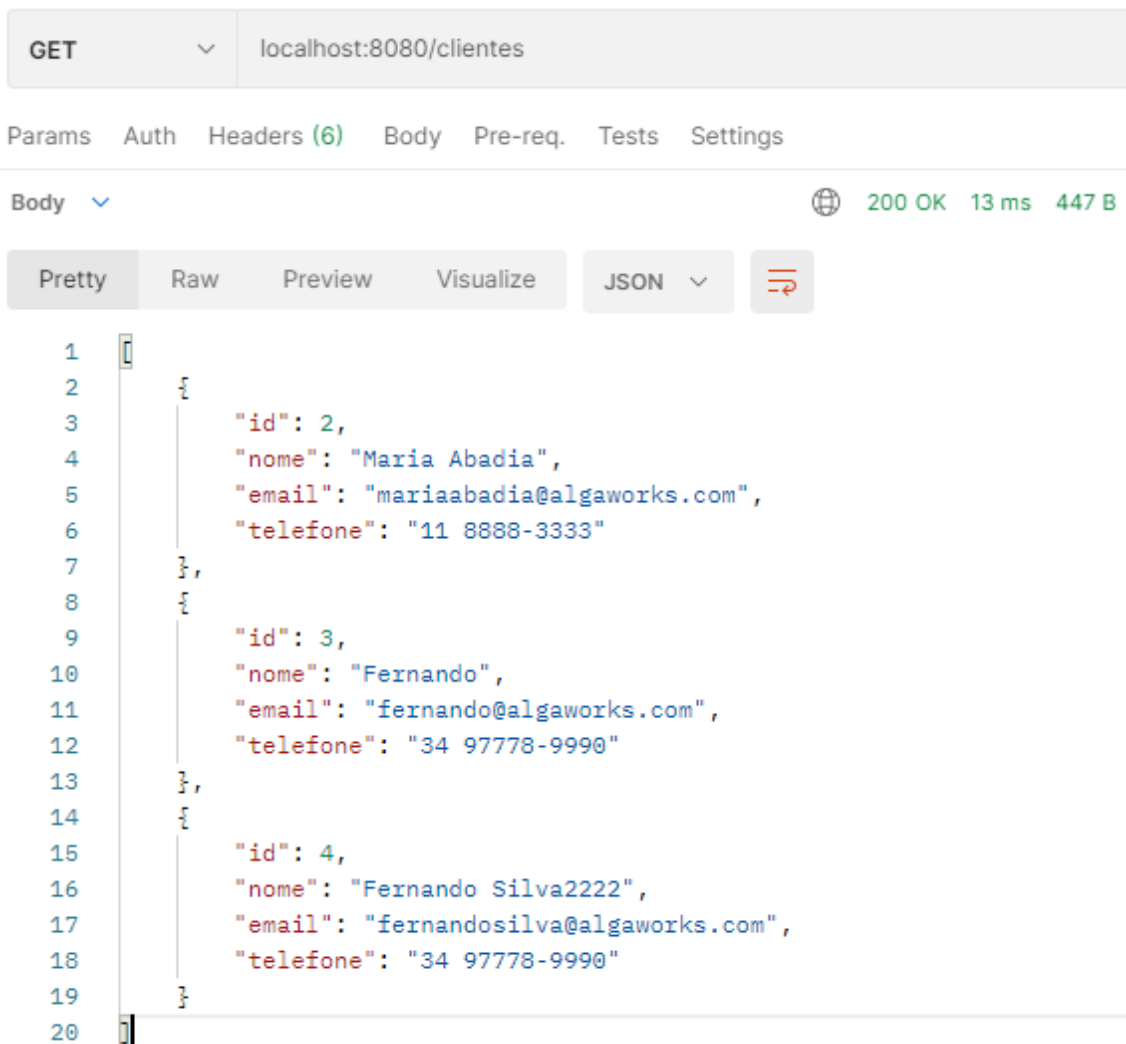


Figura 45 - GET request de clientes. Perceba que o cliente 1 foi apagado com sucesso.

## 2.5. Validando com Bean Validation

O que acontece se a gente tentar cadastrar um cliente sem nome? Vamos ver:

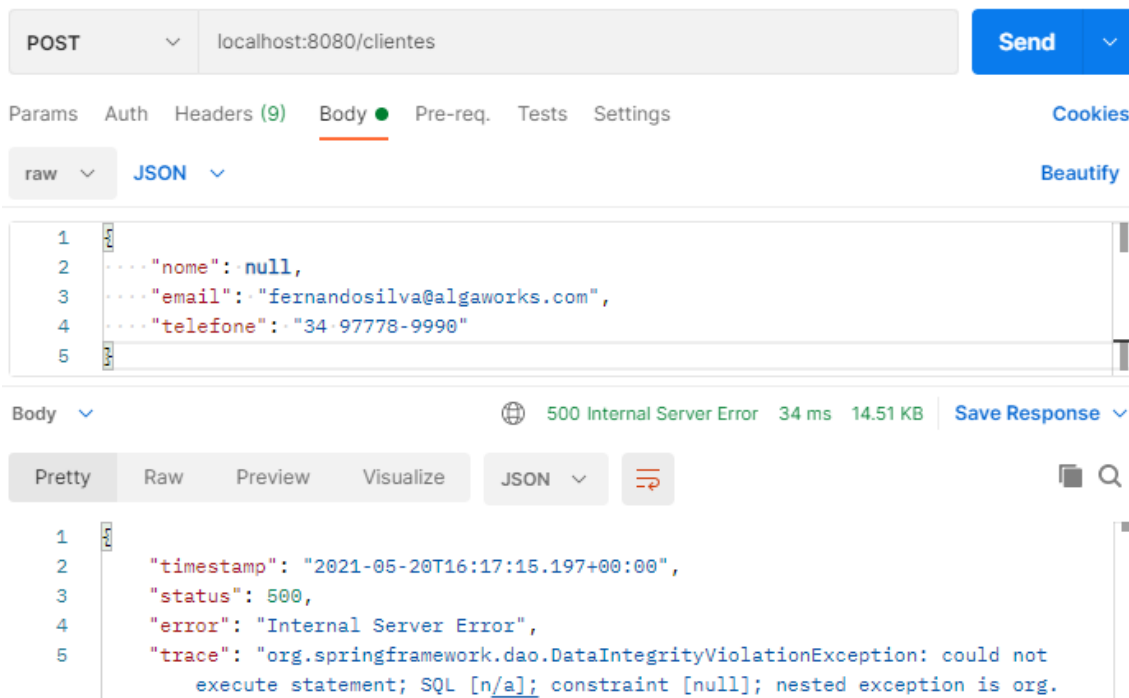


Figura 46 - Tentativa de POST request para adicionar um cliente com nome nulo.

Quando criamos a tabela cliente, definimos que a coluna “nome” não pode ser nula. Nesse caso, o consumidor da API não deveria ver esse tipo de erro. **Precisamos criar uma validação intermediária para que essa requisição não chegue ao banco de dados e viole seus constraints.** Para isso, vamos usar uma especificação do Jakarta EE chamado Jakarta Bean Validation.

O Bean Validation é uma especificação para validar objetos Java no Spring. Começamos adicionando um novo starter.

RIGHT CLICK(ROOT PROJECT) -> SPRING -> ADD STARTERS -> VALIDATION -> NEXT -> pom.xml -> FINISH.

Ao conferir nossa Hierarquia de Dependências do pom.xml, podemos ver que a especificação de validação é implementada pelo Hibernate Validator.

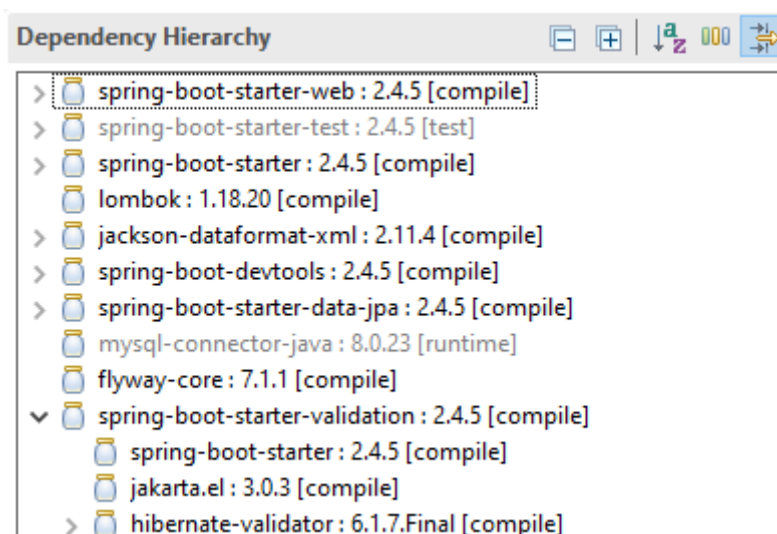


Figura 47 - Implementação do Bean Validation pelo Hibernate.

Para validarmos os nossos objetos Java, precisamos ir à classe Cliente e adicionar anotações nas propriedades da classe. Neste caso, anotamos as propriedades nome, email e telefone.

```
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Getter
@Setter
@Entity
public class Cliente {

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

    @NotBlank
    @Size(max = 60)
    private String nome;

    @NotBlank
    @Email
    @Size(max = 255)
    private String email;

    @NotBlank
    @Size(max = 20)
    @Column(name="fone")
    private String telefone;
}
```

- @NotBlank – garante que a propriedade do objeto não será nula nem vazia.
- @Email – garante que a propriedade seja condizente com a sintaxe de um e-mail.
- @Size – define o tamanho máximo da propriedade. Deve ser condizente com o tamanho da respectiva coluna no banco de dados.

Repetindo nossa requisição, obtemos outro erro, um pouco diferente do anterior:

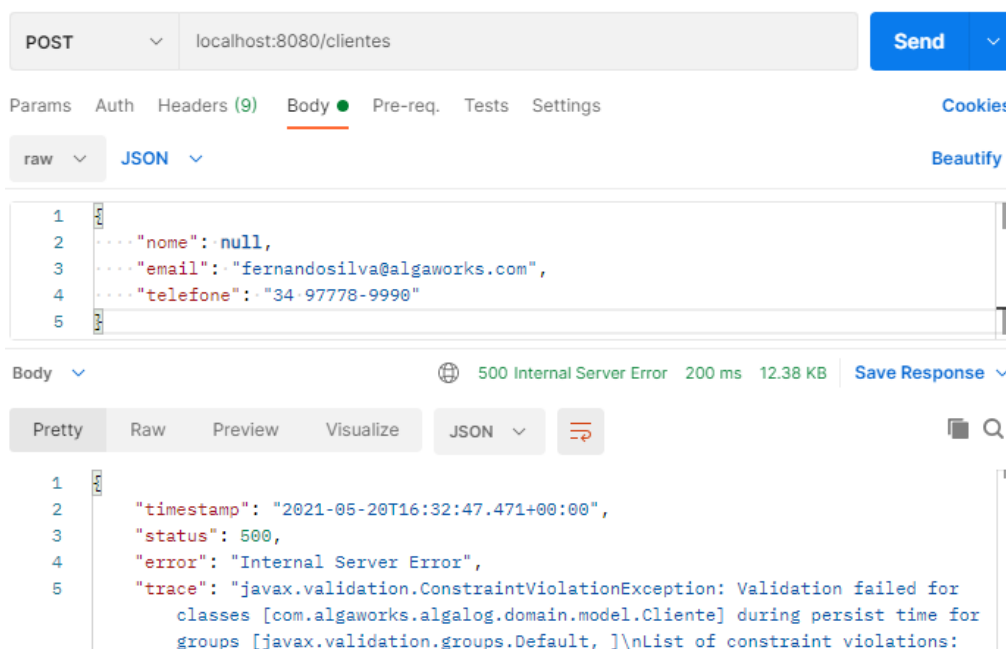


Figura 48 - Nova POST request com nome = null. Repare que agora o erro é de validação, não diretamente no SQL.

A validação está funcionando, mas ainda não impede que o usuário receba uma extensa mensagem de erro com código HTTP 500, indicando erro interno do servidor, pois ela está ocorrendo na hora de persistir os dados, ou seja, na última camada antes da transação no banco de dados.

Queremos que a validação seja feita na entrada dos dados pelo corpo da requisição. Fazer isso é bem simples. Adicionamos uma anotação em nosso Controller, antes de `@RequestBody`:

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Cliente adicionar(@Valid @RequestBody Cliente cliente) {
    return clienteRepository.save(cliente);
}

@PutMapping("/{clienteId}")
public ResponseEntity<Cliente> atualizar(@PathVariable Long clienteId,
    @Valid @RequestBody Cliente cliente) {
    if (!clienteRepository.existsById(clienteId)) {
        return ResponseEntity.notFound().build();
    }

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

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

A anotação `@Valid` executa a validação na hora da HTTP request que submete um `@RequestBody` (POST e PUT). Repetindo a última requisição, obtemos um novo código de erro:

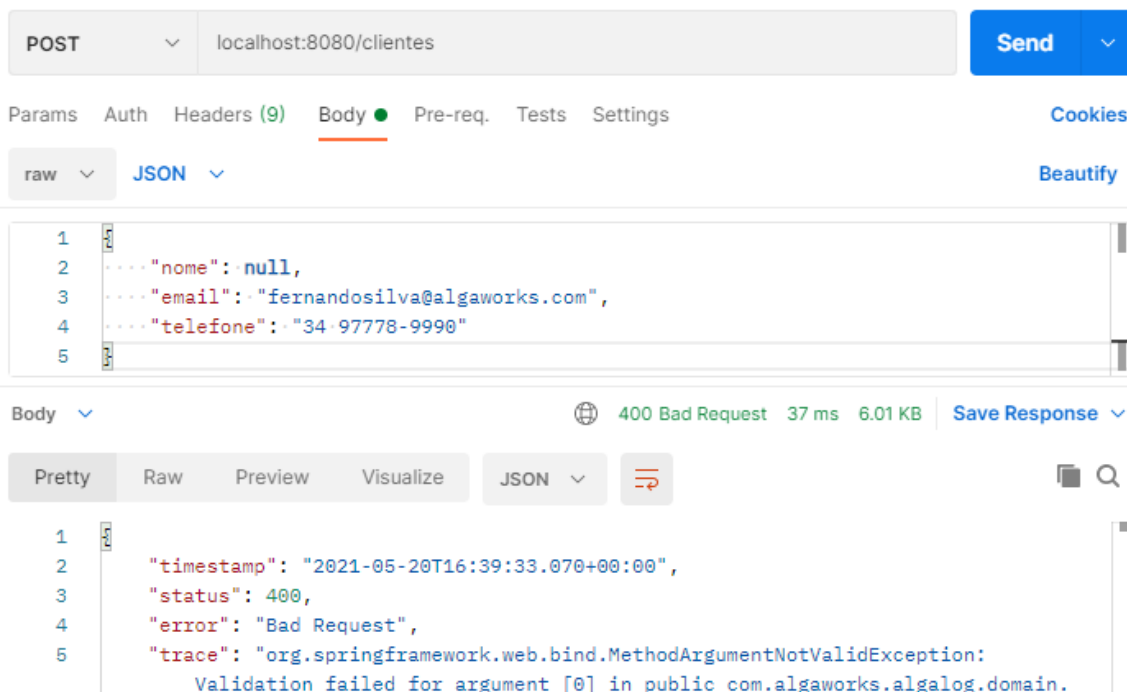


Figura 49 - POST request com `nome = null`. O erro 400 BAD REQUEST indica que o erro ocorreu do lado do consumidor da API, que inseriu dados inválidos.

Nesse curso, ainda melhoraremos o erro retornado.

Podemos também realizar outros testes, como passar nomes vazios (""), e-mail sem @, telefone vazio, e a resposta da API será a mesma: 400 BAD REQUEST. Testando, agora, a adição de um cliente que não viola nenhum princípio do Jakarta Bean Validation, vemos que nossos métodos continuam funcionando normalmente:

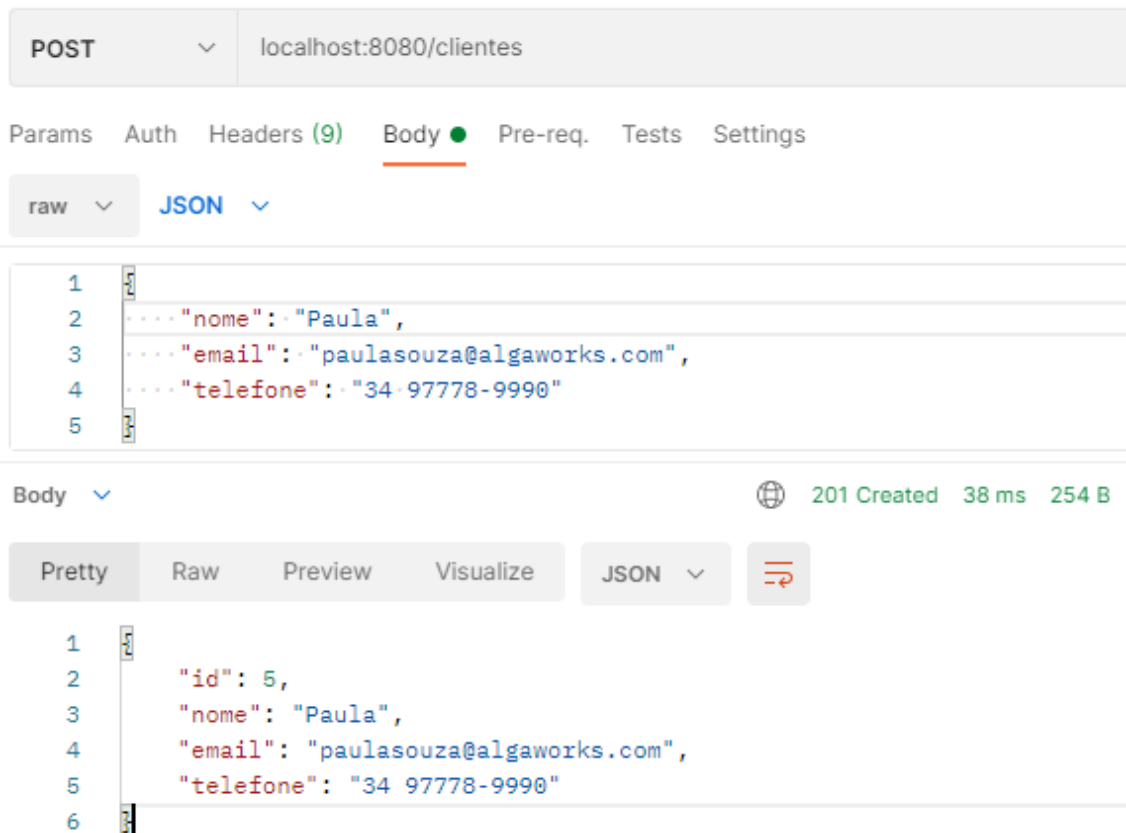


Figura 50 - Adição de um novo cliente por um POST request com propriedades válidas.

## 2.6. Implementando Exception Handler

Atualmente, as respostas de erro da API estão enormes, apresentando informações desnecessárias ou até confidenciais sobre a estrutura do projeto para o consumidor. A ideia é que tratemos essa mensagem para que elas fiquem menores e mais legíveis.

Em `src/main/java/com.algaworks.algalog`, criamos nossa classe `ApiExceptionHandler`:

- PACKAGE -> `com.algaworks.algalog.api.exceptionhandler`
- NAME -> `ApiExceptionHandler`

```
@ControllerAdvice
public class ApiExceptionHandler extends ResponseEntityExceptionHandler {
}
```

- `@ControllerAdvice` – indica que a classe é um componente Spring e que trata exceções de forma global, para todos os Controllers da aplicação.
- `extends ResponseEntityExceptionHandler` – classe base de conveniência para tratar Exceptions, permitindo à nossa classe herdar todos os seus métodos e nos ligarmos à classe `ResponseEntity`.

Ao tentar uma nova POST request com um corpo inválido:

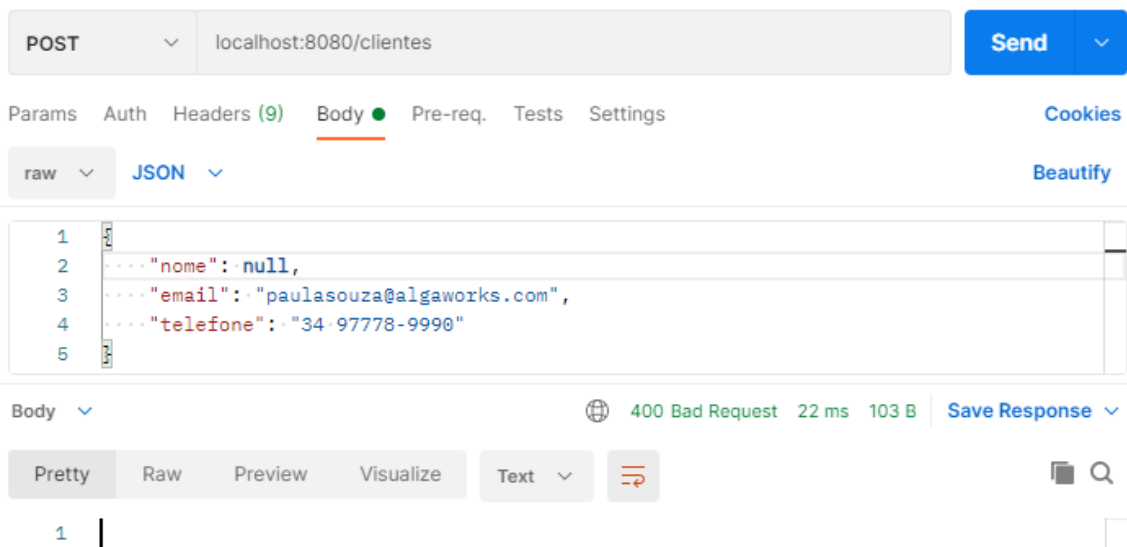


Figura 51 - POST request inválida. Não há mais mensagem de erro no corpo da resposta.

É bom não retornar um erro imenso ao consumidor, mas melhor ainda é retornar um erro informativo ao consumidor.

Ao consultar as logs no nosso console:

```
_[2m2021-05-20 14:09:45.129_[0;39m _[33m WARN_[0;39m _[35m12884_[0;39m _[2m--
-[0;39m _[2m[nio-8080-exec-2]_[0;39m
_[36m.m.m.a.ExceptionHandlerExceptionHandler_[0;39m _[2m:_[0;39m Resolved
[org.springframework.web.bind.MethodArgumentNotValidException: Validation
failed for argument [0] in public com.algaworks.algalog.domain.model.Cliente
com.algaworks.algalog.api.controller.ClienteController.adicionar(com.algawork
s.algalog.domain.model.Cliente): [Field error in object 'cliente' on field
'nome': rejected value [null]; codes
[NotBlank.cliente.nome,NotBlank.nome,NotBlank.java.lang.String,NotBlank];
arguments
[org.springframework.context.support.DefaultMessageSourceResolvable: codes
[cliente.nome,nome]; arguments []; default message [nome]]; default message
[não deve estar em branco]] ]
```

Podemos ver que o nosso erro está sendo tratado pelo `MethodArgumentNotValidException`. O que precisamos fazer para modificá-lo e permitir que ele exiba as informações sobre a nossa exceção é um `Override`. O importaremos dentro da nossa classe `ApiExceptionHandler` para isso.

```
@ControllerAdvice
public class ApiExceptionHandler extends ResponseEntityExceptionHandler {
    @Override
    protected ResponseEntity<Object>
    handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
                                HttpHeaders headers, HttpStatus status, WebRequest
    request) {
        // TODO Auto-generated method stub
        return super.handleMethodArgumentNotValid(ex, headers, status,
        request);
    }
}
```

Em seguida, alteramos o return, deixando o método assim:

```

@ControllerAdvice
public class ApiExceptionHandler extends ResponseEntityExceptionHandler {

    @Override
    protected ResponseEntity<Object>
    handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
                                HttpHeaders headers, HttpStatus status, WebRequest
    request) {
        // TODO Auto-generated method stub

        return handleExceptionInternal(ex, "Valor inválido", headers,
        status, request);
    }
}

```

Os argumentos desse retorno são:

- Exception ex
- Object body
- HTTP headers
- HTTP status
- Request

Se repetirmos nossa última requisição, já recebemos uma mensagem de erro:

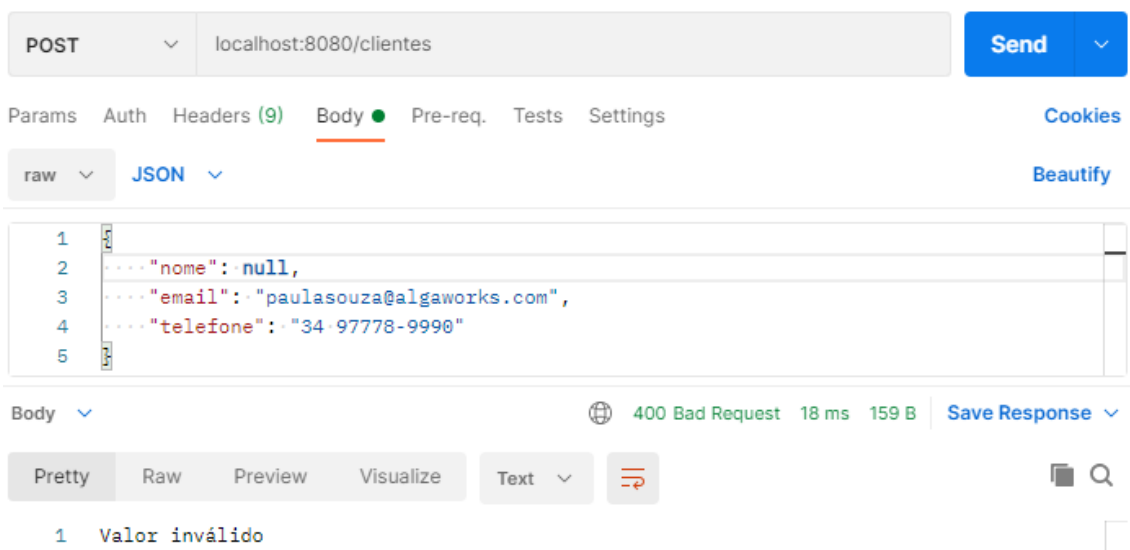


Figura 52 - POST request após primeiro tratamento via ApiExceptionHandler.

Sendo assim, podemos adicionar qualquer coisa ao corpo da nossa resposta. Visto que o parâmetro body pode receber um objeto, criaremos uma classe Problema que será instanciada e retornada nesses casos, em vez de “Valor inválido”.

Para isso, em src/main/java/ com.algaworks.algalog.api.exceptionhandler, criamos:

- NAME -> Problema

```

@Getter
@Setter
public class Problema {

    private Integer status;
    private LocalDateTime dataHora;
}

```



```

        private String titulo;
    }

```

Por enquanto, retornaremos o status HTTP, a DataHora da requisição, e o título do erro. Agora, instanciaremos um objeto da classe Problema na nossa ApiExceptionHandler.

```

@ControllerAdvice
public class ApiExceptionHandler extends ResponseEntityExceptionHandler {

    @Override
    protected ResponseEntity<Object>
    handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
                                HttpHeaders headers, HttpStatus status, WebRequest
    request) {
        // TODO Auto-generated method stub

        Problema problema = new Problema();
        problema.setStatus(status.value());
        problema.setDataHora(LocalDateTime.now());
        problema.setTitulo("Um ou mais campos estão inválidos. Faça o
        preenchimento correto e tente novamente.");

        return handleExceptionInternal(ex, problema, headers, status,
        request);
    }
}

```

Ao repetir a última requisição, já recebemos um retorno instrutivo.

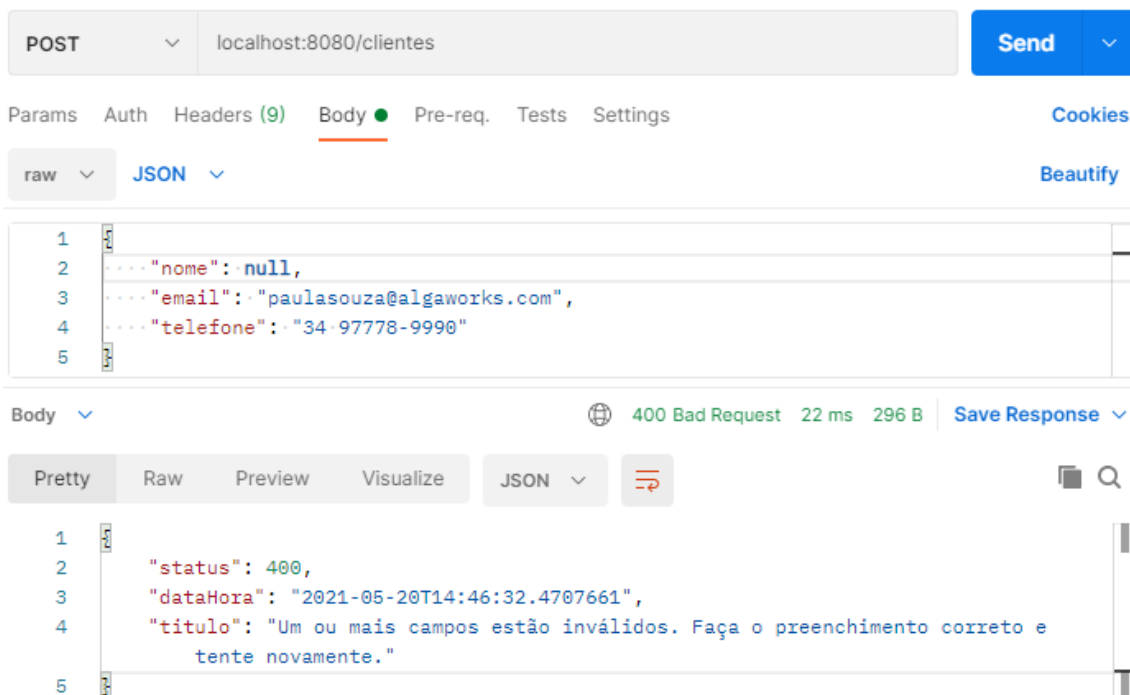


Figura 53 - Tratamento de erro pelo ApiExceptionHandler com um objeto da classe Problema.

A mensagem do erro ainda está muito genérica. Por isso, vamos adicionar uma lista de campos à nossa classe Problema. Que campos? Uma subclasse interna de Problema, olha só:

```

@Getter
@Setter

```

```

public class Problema {

    private Integer status;
    private LocalDateTime dataHora;
    private String titulo;
    private List<Campo> campos;

    @AllArgsConstructor
    @Getter
    public static class Campo {
        private String nome;
        private String mensagem;
    }

}

```

Ao invés de criarmos os Setters da subclasse Campo, decidimos criar o objeto completo puramente através do construtor, por isso o `@AllArgsConstructor`.

Agora, para conseguirmos adicionar a lista de erros à resposta de exceção, vamos incluir os campos no `ApiExceptionHandler`.

```

@ControllerAdvice
public class ApiExceptionHandler extends ResponseEntityExceptionHandler {

    @Override
    protected ResponseEntity<Object>
    handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
                                HttpHeaders headers, HttpStatus status, WebRequest
    request) {
        // TODO Auto-generated method stub

        List<Problema.Campo> campos = new ArrayList<>();

        for (ObjectError error : ex.getBindingResult().getAllErrors()) {
            String nome = ((FieldError) error).getField();
            String mensagem = error.getDefaultMessage();

            campos.add(new Problema.Campo(nome, mensagem));
        }

        Problema = new Problema();
        problema.setStatus(status.value());
        problema.setDataHora(LocalDateTime.now());
        problema.setTitulo("Um ou mais campos estão inválidos. Faça o
preenchimento correto e tente novamente.");
        problema.setCampos(campos);

        return handleExceptionInternal(ex, problema, headers, status,
request);
    }
}

```

Vamos entender as alterações que fizemos:

- Criamos uma `List<Problema.Campo>`, ou seja, de objetos do tipo Campo, subclasse de Problema, que contêm um “nome” e uma “mensagem”.
- Varremos individualmente cada `ObjectError` “error” levantado pela variável de exceção “ex”.

- A partir daqui cada `ObjectError` tem seu nome dado como uma instância da classe `FieldError`. Para acessarmos o método `.getField()`, que nos retorna o nome do erro, a partir da classe `FieldError`, precisamos fazer uma castagem do `ObjectError` para `FieldError`, por meio do código `((FieldError) error)`.
- A mensagem do erro é obtida diretamente por `error.getDefaultMessage()`.
- Adicionamos à propriedade “campos” do nosso “problema” cada objeto `Problema.Campo` preenchido pelo laço de repetição acima.

Vamos refazer nossa requisição, agora sem nome e com e-mail inválido, para vermos o que é retornado:

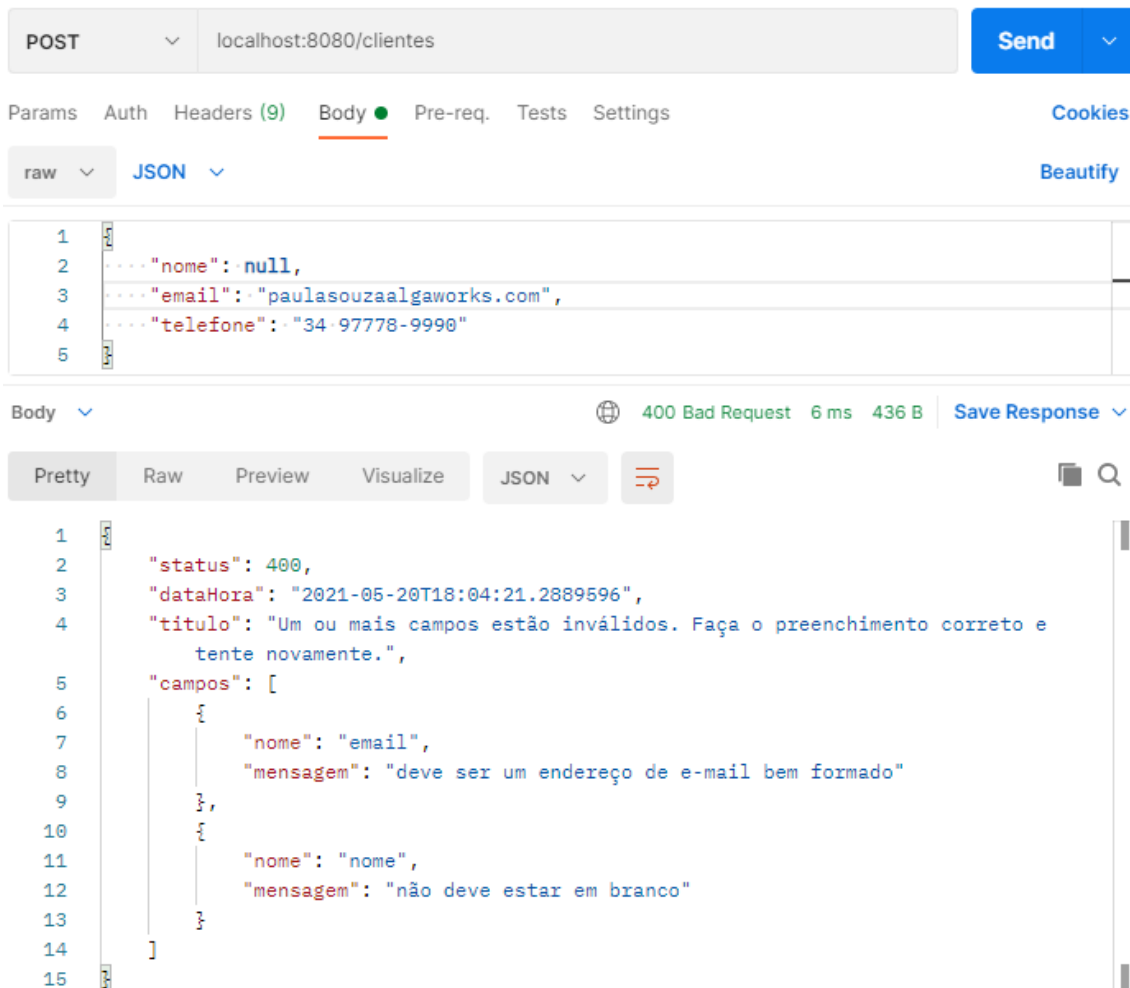


Figura 54 - Resposta de POST request inválida. Os campos com informação inválida e o motivo estão explícitos.

Muito melhor para o consumidor da API, agora, enxergar o que há de errado.

Para melhorar ainda mais a experiência do consumidor da API, podemos customizar as mensagens de erro que mostramos. Para isso, em `src/main/resources`, criamos um arquivo chamado `messages.properties` e o preenchemos com mensagens para cada tipo de validação que criamos e que desejamos alterar a mensagem de erro.

`NotBlank`=é obrigatório.

`Size`=deve ter no mínimo {2} e no máximo {1} caracteres.

`Email`=deve ser um e-mail válido.

Ao repetir nossa última requisição, recebemos a mesma resposta, com as mesmas mensagens de erro, pois ainda não integramos essas novas mensagens à nossa API. Para isso, voltamos lá no nosso `ApiExceptionHandler`:

```
@AllArgsConstructor
@ControllerAdvice
public class ApiExceptionHandler extends ResponseEntityExceptionHandler {

    private MessageSource messageSource;

    @Override
    protected ResponseEntity<Object>
    handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
                                HttpHeaders headers, HttpStatus status, WebRequest
    request) {
        // TODO Auto-generated method stub

        List<Problema.Campo> campos = new ArrayList<>();

        for (ObjectError error : ex.getBindingResult().getAllErrors()) {
            String nome = ((FieldError) error).getField();
            String mensagem = messageSource.getMessage(error,
                LocaleContextHolder.getLocale());

            campos.add(new Problema.Campo(nome, mensagem));
        }

        Problema problema = new Problema();
        problema.setStatus(status.value());
        problema.setDataHora(LocalDateTime.now());
        problema.setTitulo("Um ou mais campos estão inválidos. Faça o
        preenchimento correto e tente novamente.");
        problema.setCampos(campos);

        return handleExceptionInternal(ex, problema, headers, status,
        request);
    }
}
```

Instanciamos um objeto da classe `MessageSource`, que resgata nossas mensagens do `messages.properties`, e alteramos a propriedade “mensagem” para, agora, receber as mensagens de erro do objeto da classe `MessageSource`.

- Ainda recebemos a mensagem do `ObjectError error`.
- O idioma da mensagem depende do segundo parâmetro, obtido por `LocaleContextHolder.getLocale()`.

Refazendo a última requisição, podemos observar que:

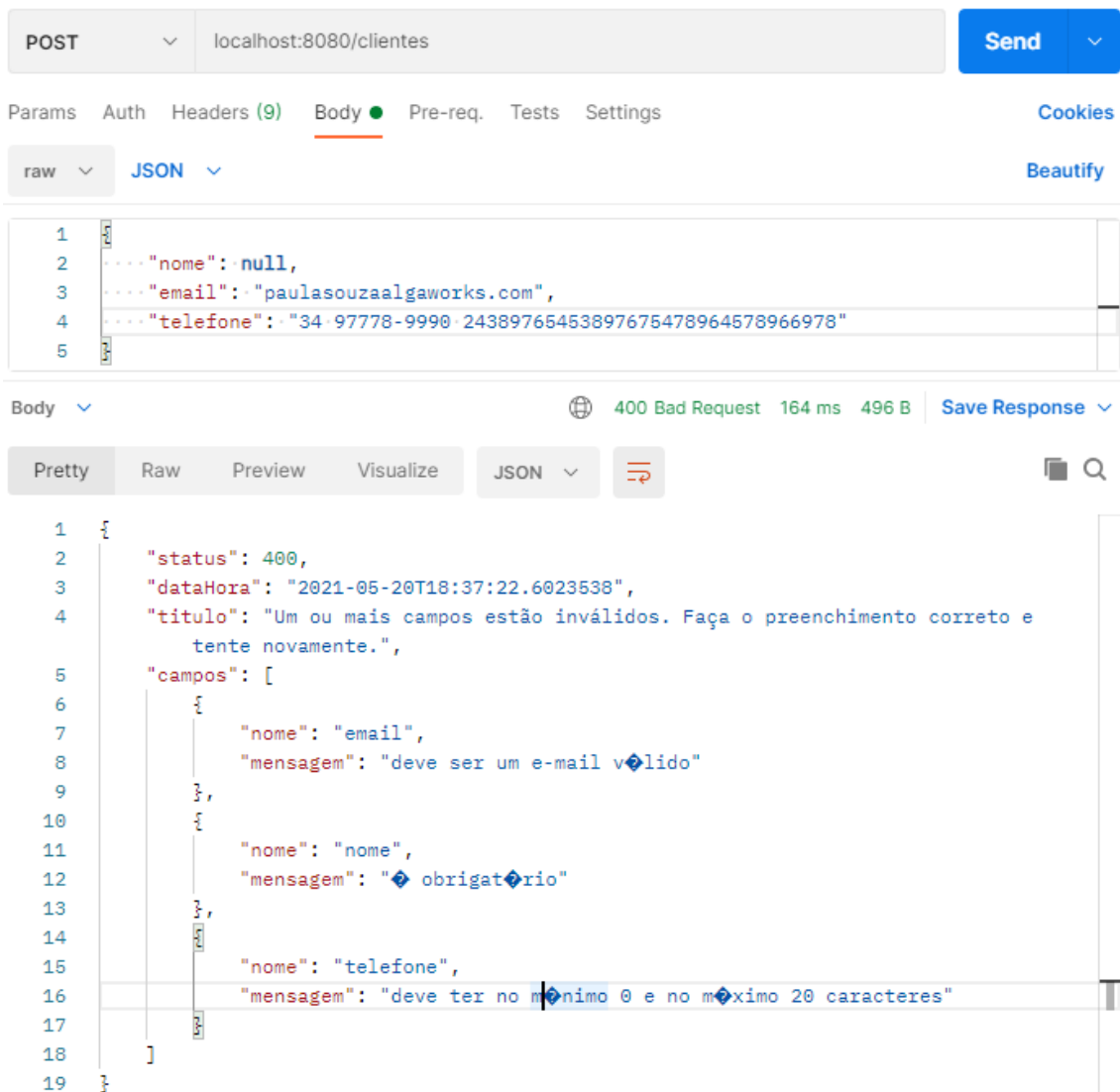


Figura 55 - Tratamento de exceção com mensagens customizadas.

Funcionou! Agora temos mensagens de erro customizadas.

Para evitar os erros de codificação de caracteres exibidos, devemos verificar se a codificação padrão do STS é UTF-8.

STS -> WINDOW -> PREFERENCES -> GENERAL -> CONTENT TYPES -> JAVA PROPERTIES FILE -> DEFAULT ENCONDING(UTF-8) -> UPDATE -> APPLY AND CLOSE.

Assim que confirmamos, nosso arquivo `messages.properties` “quebra” onde há acentuação.

NotBlank=é obrigatório

Size=deve ter no mínimo {2} e no máximo {1} caracteres

Email=deve ser um e-mail válido

Por isso, devemos configurar a codificação padrão para UTF-8 logo no começo do projeto, para evitar retrabalho e correção de texto posteriormente.

NotBlank=é obrigatório.

Size=deve ter no mínimo {2} e no máximo {1} caracteres.

Email=deve ser um e-mail válido.

Uma vez corrigido o texto, fazemos uma nova requisição, dessa vez com as 3 propriedades inválidas.

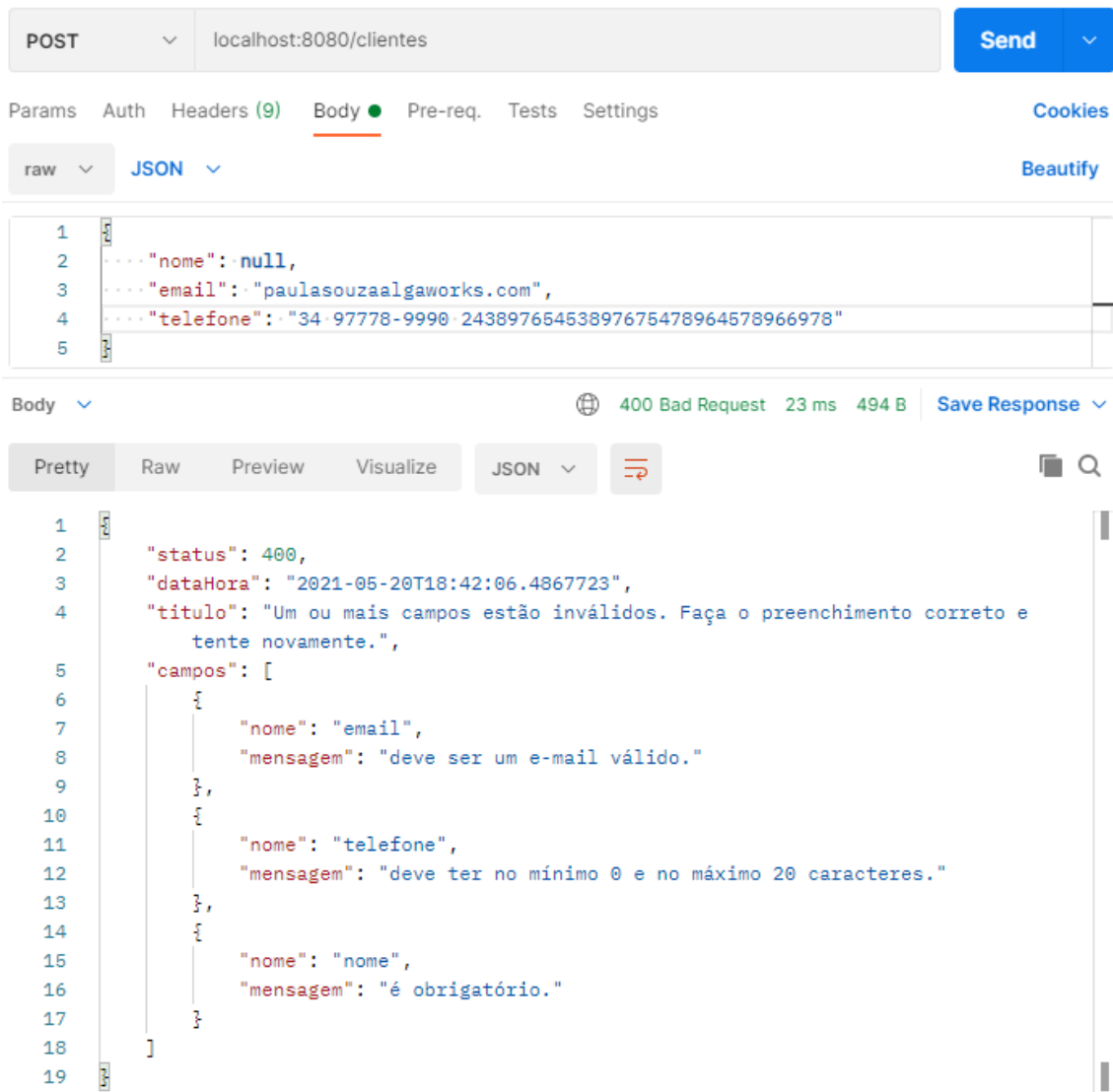


Figura 56 - Exceções tratadas e codificadas corretamente.

Obtemos em nossa resposta mensagens customizadas e devidamente codificadas em UTF-8.

## 2.7. Implementando Domain Services

À medida que o projeto cresce, há uma tendência de colocarmos regras de negócio dentro do Controller, o que não é uma boa prática de negócio.

Podemos e **devemos colocar algumas regras básicas de negócios dentro das nossas classes que representam entidades**, mas a partir do momento em que as classes precisam se relacionar entre si, ou quando precisamos injetar objetos, criamos uma classe para representar um ou mais casos de uso, a classe **Service**. Para isso, em `src/main/java/com.algaworks.algalog`, criamos:

- PACKAGE -> `com.algaworks.algalog.domain.service`
- NAME -> `CatalogoClienteService`

Nela, implementamos:

```

@AllArgsConstructor
@Service
public class CatalogoClienteService {

    private ClienteRepository clienteRepository;

    @Transactional
    public Cliente salvar(Cliente cliente) {
        return clienteRepository.save(cliente);
    }

    @Transactional
    public void excluir(Long clienteId) {
        clienteRepository.deleteById(clienteId);
    }
}

```

- @Service – marca a classe como um serviço.
- @Transactional – marca uma função como transacional, indicando que ela é executada durante uma transação com o banco de dados e que, diante de um erro, restaura o estado anterior da aplicação, desfazendo tudo que foi feito.

Implementamos os métodos salvar() e excluir() diretamente no nosso CatalogoClienteService com o intuito de diminuir a interação do ClienteController com o ClienteRepository. A ideia é que tudo seja intermediado pelo Service.

Em nosso ClienteController, portanto, alteramos os métodos adicionar(), atualizar() e deletar().

```

@RequestMapping("/clientes")
@AllArgsConstructor
@RestController
public class ClienteController {

    private ClienteRepository clienteRepository;
    private CatalogoClienteService catalogoClienteService;

    @GetMapping
    public List<Cliente> listar() {
        return clienteRepository.findAll();
    }

    @GetMapping("/{clienteId}")
    public ResponseEntity<Cliente> buscar(@PathVariable Long clienteId) {
        return clienteRepository.findById(clienteId)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Cliente adicionar(@Valid @RequestBody Cliente cliente) {
        return catalogoClienteService.salvar(cliente);
    }

    @PutMapping("/{clienteId}")
    public ResponseEntity<Cliente> atualizar(@PathVariable Long clienteId,
        @Valid @RequestBody Cliente cliente) {
        if (!clienteRepository.existsById(clienteId)) {

```

```

        return ResponseEntity.notFound().build();
    }

    cliente.setId(clienteId);
    cliente = catalogoClienteService.salvar(cliente);

    return ResponseEntity.ok(cliente);
}

@DeleteMapping("/{clienteId}")
public ResponseEntity<Void> remover(@PathVariable Long clienteId) {
    if (!clienteRepository.existsById(clienteId)) {
        return ResponseEntity.notFound().build();
    }
    catalogoClienteService.excluir(clienteId);

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

```

Não é considerado uma má prática deixar o Controller com acesso ao repositório, mas, por questões de padronização, podemos fazer dessa forma, para que o Controller nunca tenha acesso direto ao Repository.

Vamos, agora, adicionar uma regra de negócio para impedir o cadastro ou atualização de clientes com e-mails duplicados.

Começamos criando um método em nosso ClienteRepository, findByEmail:

```

@Repository
public interface ClienteRepository extends JpaRepository<Cliente, Long>{

    List<Cliente> findByNome(String nome);
    List<Cliente> findByNomeContaining(String nome);
    Optional<Cliente> findByEmail(String email);
}

```

O findByEmail retorna um Optional, que pode conter ou não um cliente.

Agora que movemos a função salvar() para o nosso CatalogoClienteService, vamos alterá-la para verificar se um e-mail já está em uso ou não.

```

@Transactional
public Cliente salvar(Cliente cliente) {
    boolean emailEmUso =
        clienteRepository.findByEmail(cliente.getEmail())
            .stream()
            .anyMatch(clienteExistente ->
                !clienteExistente.equals(cliente));

    if (emailEmUso) {
        throw new NegocioException("Já existe um cliente
        cadastrado com este e-mail");
    }

    return clienteRepository.save(cliente);
}

```



```
}
```

Destrinchando:

- Boolean emailEmUso – variável Boolean para representar a expressão seguinte, que é baseada em um Optional, podendo ser verdadeira caso o e-mail já esteja em uso ou falsa se não estiver.
- clienteRepository.findByEmail(cliente.getEmail()) – método do clienteRepository para encontrar um usuário por e-mail.
- .stream() – biblioteca Java para manuseio de coleções. Possui métodos de programação funcional, permitindo o uso de lambda expressions.
- .anyMatch(clienteExistente -> !clienteExistente.equals(cliente)) – verifica se algo é encontrado ao comparar o clienteExistente (cliente encontrado na busca) com o cliente que possui o e-mail pesquisado. Se o cliente que encontramos no nosso repositório após a pesquisa por e-mail for diferente do cliente que estamos salvando agora, vai dar match. Se der match, emailEmUso = true, logo, impedimos a transação.

Podemos encontrar o mesmo cliente na hora de utilizar o método salvar(), pois esse método serve tanto para criar registros quanto para atualizá-los.

Uma vez que detectamos que emailEmUso = true, vamos criar uma exceção para lidar com esse caso. Para isso, em src/main/java/com.algaworks.algalog, criamos:

- PACKAGE -> com.algaworks.algalog.domain.exception
- NAME -> NegocioException

Dentro de NegocioException:

```
public class NegocioException extends RuntimeException{  
  
    private static final long serialVersionUID = 1L;  
  
    public NegocioException(String message) {  
        super(message);  
    }  
}
```

Para criar o serialVersionUID, declaramos nossa exceção e clicamos no símbolo amarelo ao lado:

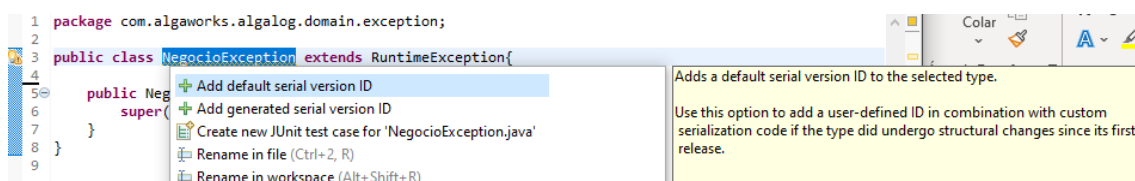


Figura 57 - Criando o serialVersionUID da nossa exceção.

O objetivo dessa Exception é repassar uma mensagem de erro para a classe pai, RuntimeException, através do seu construtor, que a recebe em:

```
public NegocioException(String message) {  
    super(message);  
}
```

Vamos testar cadastrar um novo cliente e, logo em seguida, tentar duplicar o cadastro para ver o que acontece.

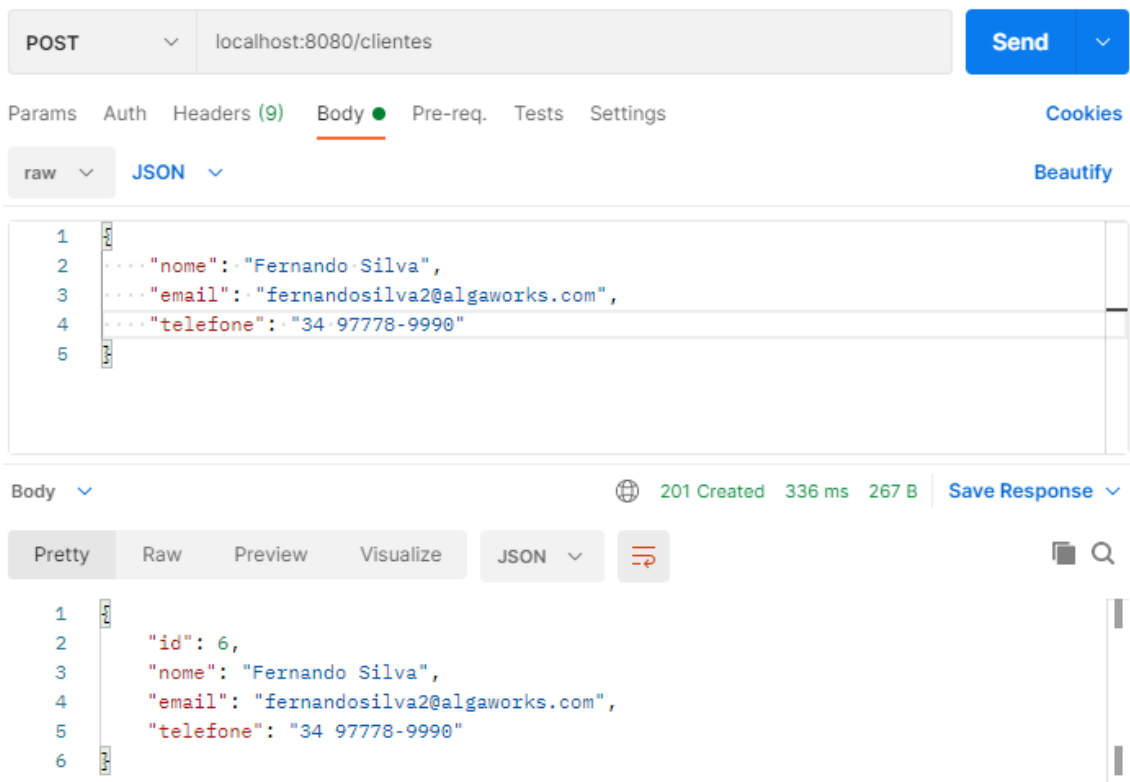


Figura 58 - Cadastro de novo cliente.

Ao repetir a POST request da figura passada, recebemos uma mensagem de erro.

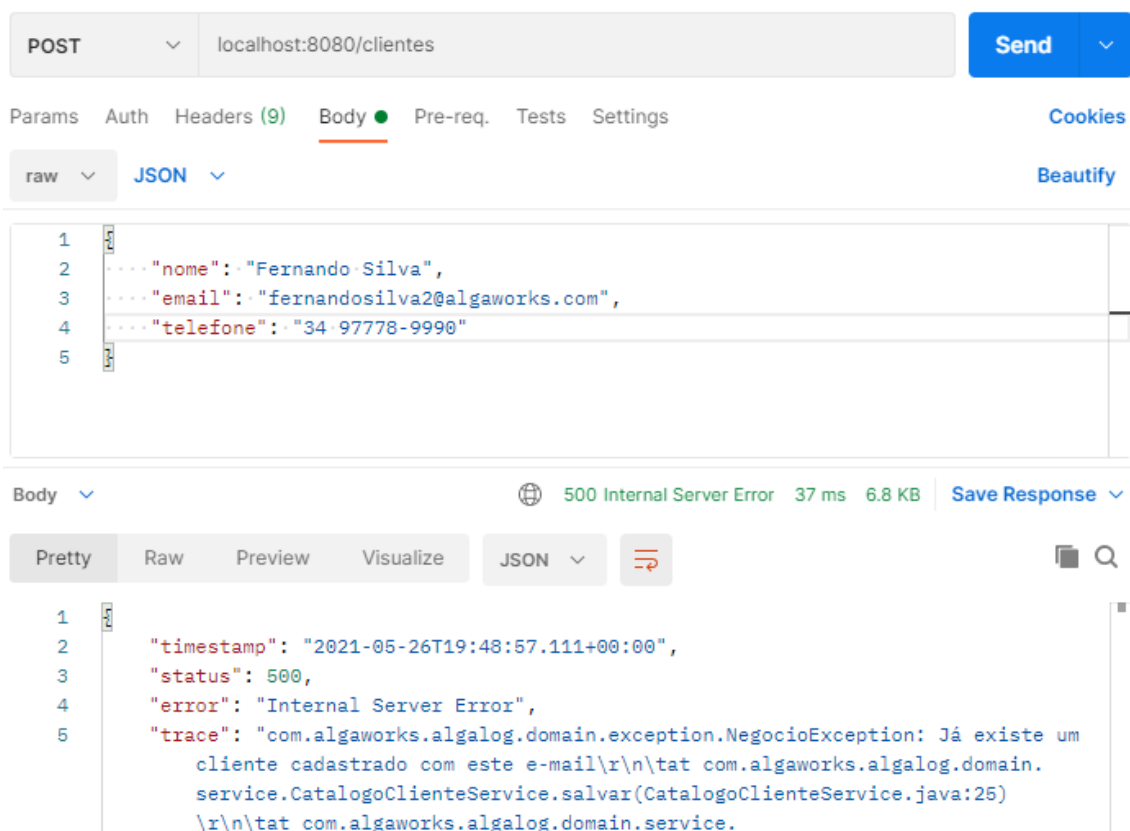


Figura 59 - Erro ao acionar exceção de cadastro de e-mail duplicado.

O próximo passo é tratar esse erro em nosso ApiExceptionHandler.

```

@ExceptionHandler(NegocioException.class)
public ResponseEntity<Object> handleNegocio(NegocioException ex,
WebRequest request) {

    HttpStatus status = HttpStatus.BAD_REQUEST;

    Problema problema = new Problema();
    problema.setStatus(status.value());
    problema.setDataHora(LocalDateTime.now());
    problema.setTitulo(ex.getMessage());

    return handleExceptionInternal(ex, problema, new HttpHeaders(),
status, request);
}

```

De maneira similar ao método `handleMethodArgumentNotValid`, o método `handleNegocio` tratará uma exceção `NegocioException`, que estende `RunTimeException`, causado por uma `WebRequest request`.

Destrinchando:

- O método `handleNegocio` tratará uma exceção `NegocioException` causado por uma `WebRequest request`.
- A anotação `@ExceptionHandler(NegocioException.class)` indica a relação dessa exceção com a classe `NegocioException`.
- Geramos um problema com status HTTP definido acima, data e hora, e o título do erro é a própria mensagem passada por parâmetro, "Já existe um cliente cadastrado com este e-mail."
- Retornamos uma `ResponseEntity` que encapsula a exceção na resposta, justamente com a exceção, o problema (com as informações do nosso tratamento de exceção), o cabeçalho HTTP, o status HTTP e a própria requisição.

Feito isso, a nova resposta da nossa requisição é a seguinte:

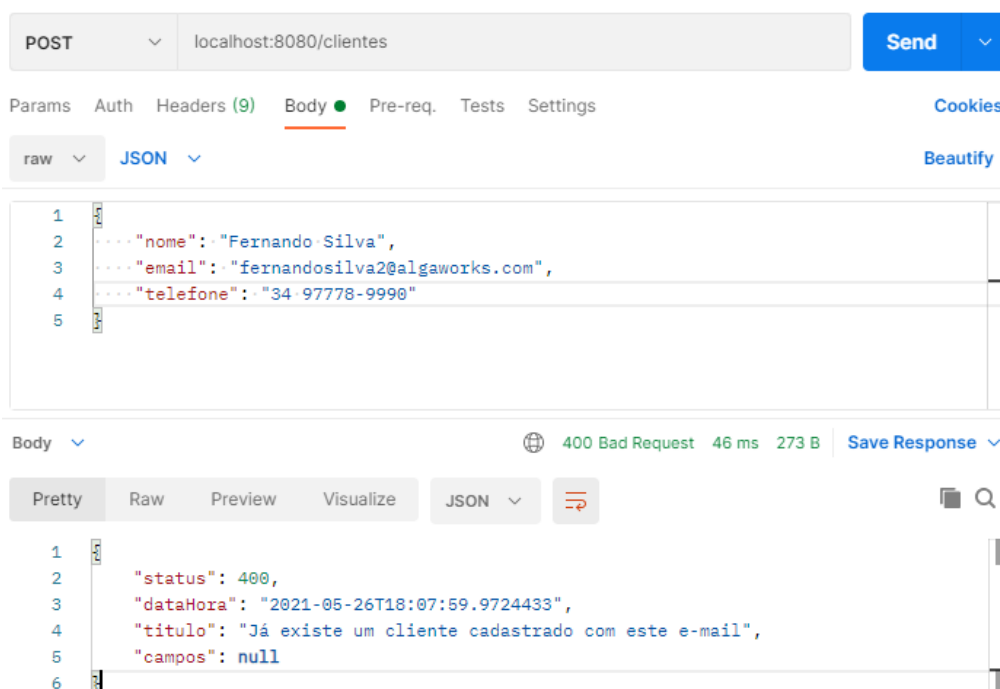


Figura 60 - Resposta tratada à POST request com e-mail já cadastrado.

Para limpar mais ainda, podemos tirar essa propriedade “campo” da nossa resposta, já que ela não acrescenta em nada ao erro. Nesse caso, editamos a nossa classe Problema.

```
@JsonInclude(Include.NON_NULL)
@Getter
@Setter
public class Problema {

    private Integer status;
    private LocalDateTime dataHora;
    private String titulo;
    private List<Campo> campos;

    @AllArgsConstructor
    @Getter
    public static class Campo {
        private String nome;
        private String mensagem;
    }
}
```

Adicionamos a nova anotação `@JsonInclude(Include.NON_NULL)` para que somente os campos não-nulos sejam incluídos na exibição do objeto. Repetindo nossa última requisição:

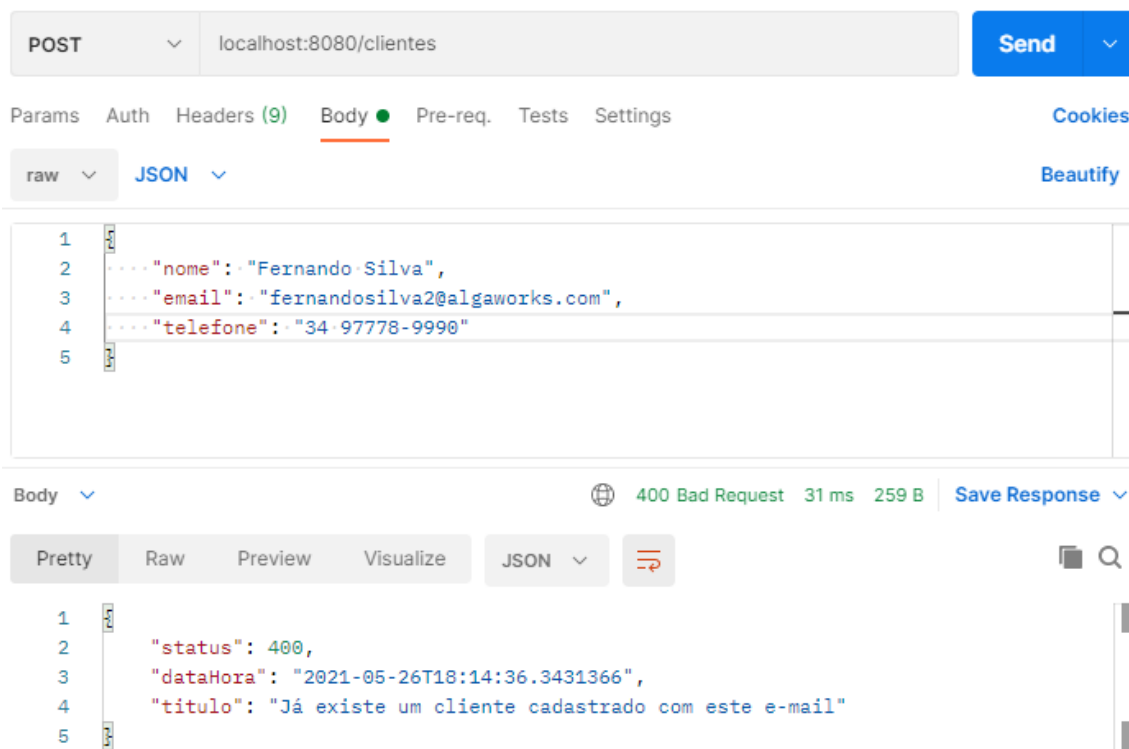


Figura 61 - Exibição de mensagem de erro sem campos nulos.

Se tentarmos atualizar um cadastro, veremos que nossa aplicação está permitindo a alteração.

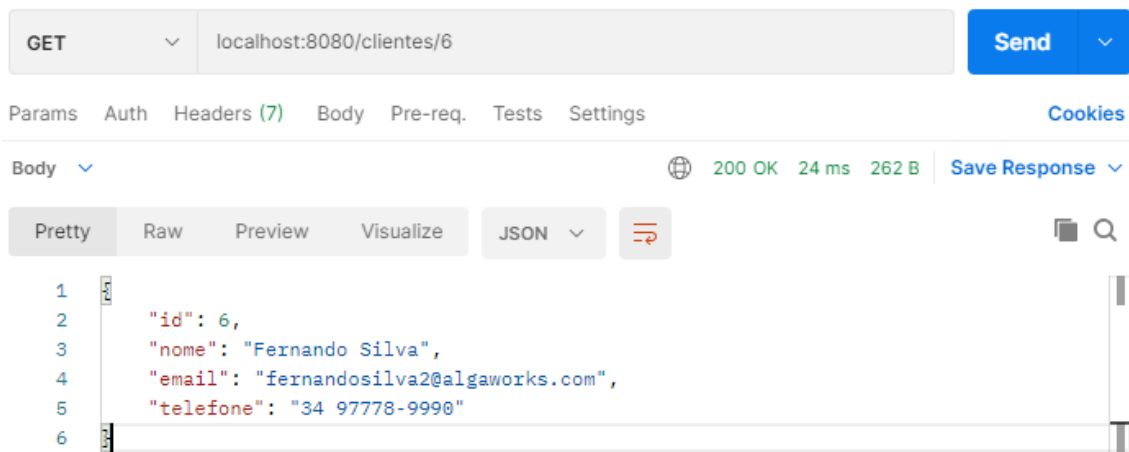


Figura 62 - GET request antes de atualização.

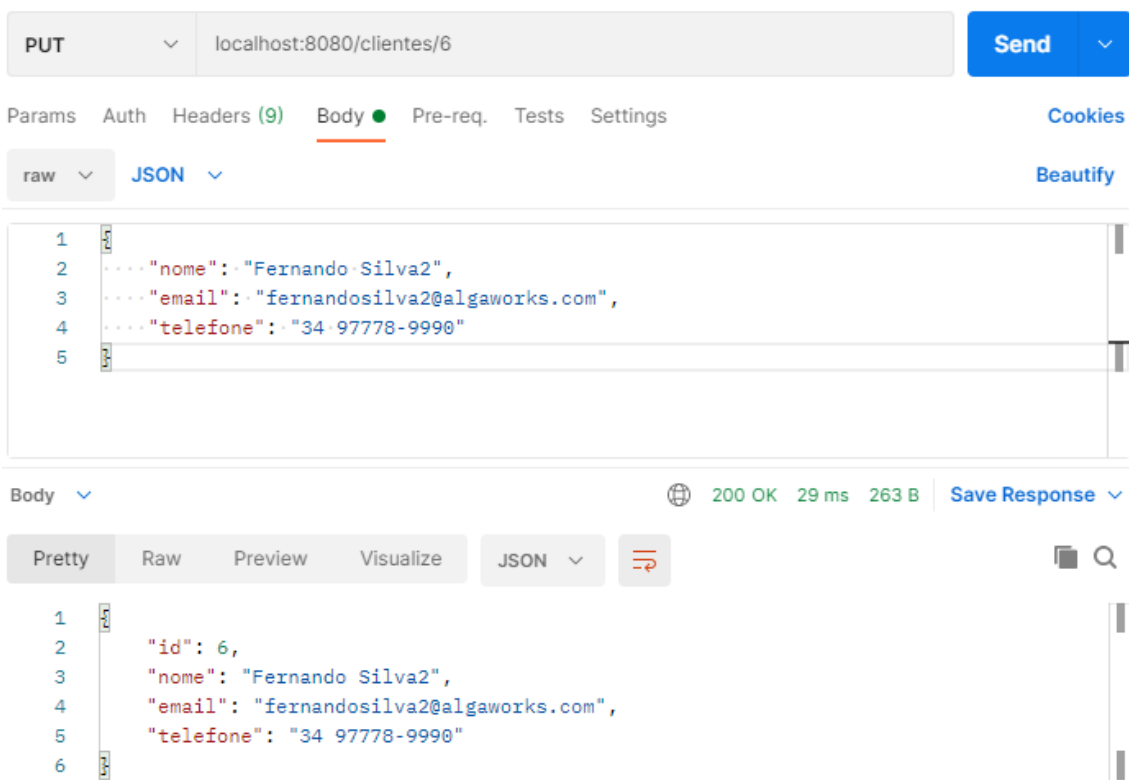


Figura 63 - PUT request: atualização bem-sucedida.

Se tentarmos alterar o e-mail para vazio ou inválido, receberemos as exceções implementadas anteriormente (e-mail inválido, e-mail vazio) como resposta.

## MÓDULO 3 – 20/05/2021

### 3.1. Implementando solicitação de entrega

Nosso endpoint Cliente já está bem desenvolvido e consolidado. O próximo passo da nossa aplicação é criar o endpoint Entrega. Em `src/main/java/com.algaworks.algalog`, criamos:

- PACKAGE -> `com.algaworks.algalog.domain.model`
- NAME -> Entrega

Nossa classe Entrega ficou dessa forma:

```
@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Entrega {

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

    @ManyToOne // relacionamento de chave estrangeira. Por padrão,
    utiliza o <nomeDaEntidade>_id. Nesse caso, cliente_id. Se quiséssemos fazer o
    relacionamento a outra coluna, usaríamos também @JoinColumn(name =
    "<nomeColuna>")
    private Cliente cliente;

    @Embedded // usamos quando queremos abstrair os dados de uma classe
    para outra, porém, embutida na mesma tabela. Tudo fica na tabela Entrega.
    Fazemos isso para diminuir a classe Entrega e organizá-la.
    private Destinatario destinatario;
    private BigDecimal taxa;

    @Enumerated(EnumType.STRING) // indica que estamos usando um
    Enumerator e que seu conteúdo é uma String, garantindo que os status
    PENDENTE, FINALIZADA, CANCELADA sejam salvos diretamente no nosso BD.
    private StatusEntrega status;
    private LocalDateTime dataPedido;
    private LocalDateTime dataFinalizacao;

}
```

Dessa vez, incluímos algumas anotações novas, @ManyToOne, @Embedded e @Enumerated, além de duas subclasses, Destinatario e StatusEntrega.

- @ManyToOne – denota um relacionamento de chave estrangeira, indicando que pode haver várias entregas para um mesmo cliente.
- @Embedded – indica que uma classe está embutida em outra, dividindo a mesma tabela no banco de dados. A classe Destinatario está embutida em Entrega.
- @Enumerated(EnumType.STRING) – indica que o campo abaixo é derivado de uma classe Enumerator.

Nossa classe Destinatario ficou dessa forma:

```
@Getter
@Setter
@Embeddable // embutível, será embutida na classe Entrega
public class Destinatario {

    @Column(name = "destinatario_nome")
    private String nome;

    @Column(name = "destinatario_logradouro")
    private String logradouro;

    @Column(name = "destinatario_numero")
```

```

private String numero;

@Column(name = "destinatario_complemento")
private String complemento;

@Column(name = "destinatario_bairro")
private String bairro;
}

```

A anotação `@Embeddable` complementa a `@Embedded` da classe `Entrega`: a `Entrega` possui um `Destinatario` embutido, o `Destinatario` é embutível.

Considerando que `Destinatario` não tem sua própria tabela, mapeamos cada uma de suas propriedades com uma anotação `@Column`, para exprimir o relacionamento com a tabela onde serão armazenados os dados.

Por fim, vejamos a classe `StatusEntrega`:

```

public enum StatusEntrega {

    PENDENTE, FINALIZADA, CANCELADA

}

```

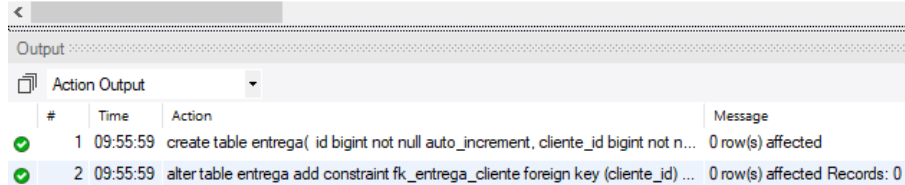
Classes `Enumerator` contêm listas de constantes, e apresentam sintaxe bem simples.

Declaradas as classes, vamos criar nossa `Migration`. Primeiro, testamos o código SQL diretamente no `MySQL Workbench`:

```

1 • create table entrega(
2
3     id bigint not null auto_increment,
4     cliente_id bigint not null,
5     taxa decimal(10,2) not null,
6     status varchar(20) not null,
7     data_pedido datetime not null,
8     data_finalizacao datetime,
9
10    destinatario_nome varchar(60) not null,
11    destinatario_logradouro varchar(255) not null,
12    destinatario_numero varchar(30) not null,
13    destinatario_complemento varchar(60) not null,
14    destinatario_bairro varchar(30) not null,
15
16    primary key (id)
17
18 );
19
20 • alter table entrega add constraint fk_entrega_cliente
21 foreign key (cliente_id) references cliente (id)

```



#	Time	Action	Message
1	09:55:59	create table entrega( id bigint not null auto_increment, cliente_id bigint not n...	0 row(s) affected
2	09:55:59	alter table entrega add constraint fk_entrega_cliente foreign key (cliente_id) ...	0 row(s) affected Records: 0

Figura 64 - Criação de tabela `Entrega`.

Assim que verificamos que a criação da tabela diretamente pelo MySQL Workbench foi bem-sucedida, devemos dropá-la para que o FlyWay possa executar esse script de criação por nós.

Criamos a migration `V003__cria-tabela-entrega.sql` em nossa pasta `src/main/resources/db/migration`, soltamos o código DDL lá dentro e rodamos nosso servidor.

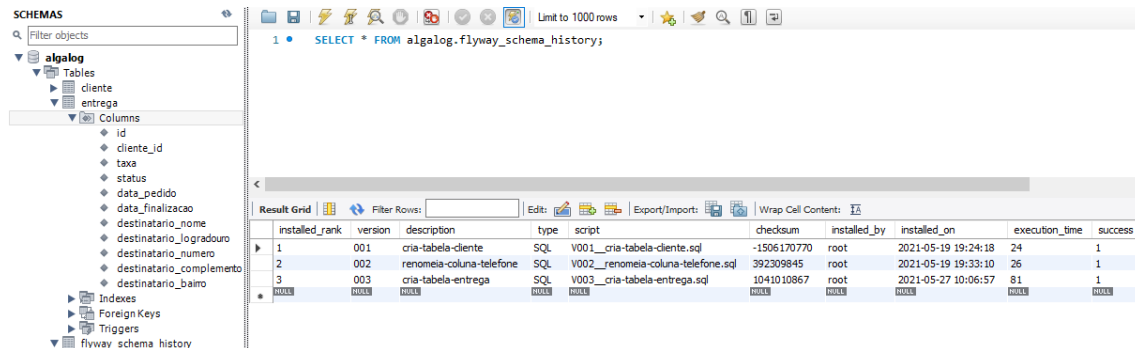


Figura 65 – Migration bem-sucedida.

Nossa terceira Migration foi bem-sucedida. Repare que a tabela `flyway_schema_history` contém um registro da migration e que a tabela `entrega` aparece na nossa lista de tabelas.

Feito isso, vamos criar o nosso `EntregaRepository`:

```
@Repository
public interface EntregaRepository extends JpaRepository<Entrega, Long>{

}
```

E, em seguida, nosso Service, o `SolicitacaoEntregaService`.

```
@AllArgsConstructor
@Service
public class SolicitacaoEntregaService {

    private EntregaRepository entregaRepository;

    @Transactional // será totalmente revertida no BD caso falhe
    public Entrega solicitar(Entrega entrega) {
        entrega.setStatus(StatusEntrega.PENDENTE);
        entrega.setDataPedido(LocalDate.now());

        return entregaRepository.save(entrega);
    }
}
```

Agora que já temos nossas Classes, nossa tabela no banco de dados, nosso Repository e nosso Service, vamos criar o `EntregaController`, que, por enquanto, só terá o método `solicitar()`.

```
@AllArgsConstructor
@RestController
@RequestMapping("/entregas")
public class EntregaController {

    private SolicitacaoEntregaService solicitacaoEntregaService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Entrega solicitar(@RequestBody Entrega entrega) {
```



```

        return solicitacaoEntregaService.solicitar(entrega);
    }
}

```

Vamos reorganizar a nossa Collection no Postman em pastas e, então, começar a testar o nosso novo endpoint.

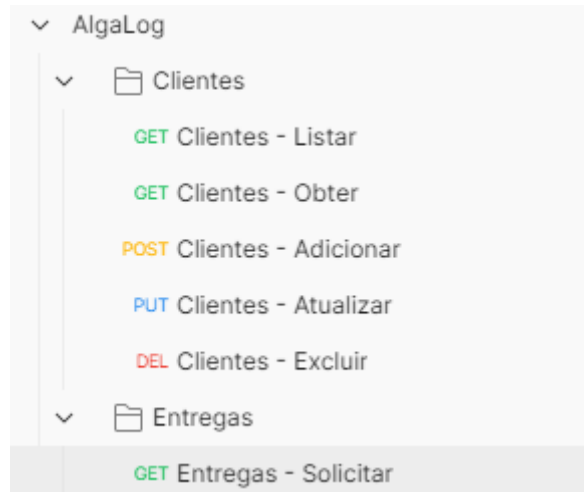


Figura 66 - Organização de requisições dentro de pastas no Postman.

Agora, podemos testar nosso método `solicitar()`. Dessa vez, a estrutura do corpo da requisição será mais complexa do que o corpo das requisições de Cliente, pois possuímos outros objetos dentro de Entrega, que serão aninhados no nosso JSON.

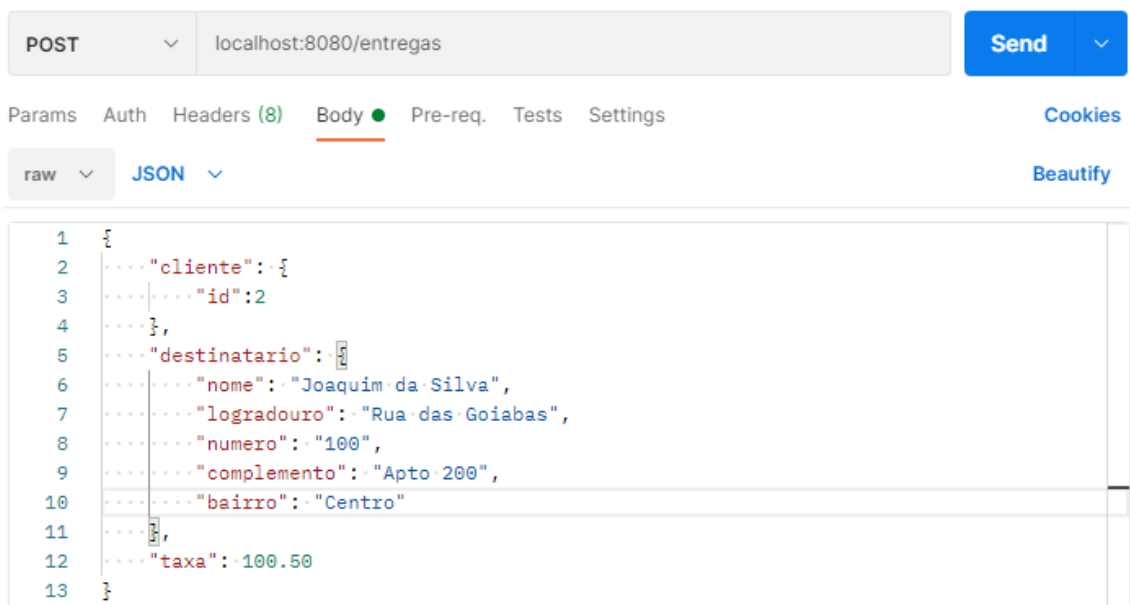


Figura 67 - POST request para criação de solicitação de entrega.

Nosso objeto Entrega possui como propriedade um Cliente, entretanto, nosso objetivo aqui não é criar clientes. Precisamos apenas referenciar o Id de algum cliente já existente. Nesse caso, escolhemos o cliente 2, pois apagamos o cliente 1 anteriormente. O destinatário, porém, precisa ter todas as suas propriedades especificadas, pois estamos criando-o.

A resposta para a POST request da imagem anterior pode ser encontrada na imagem a seguir:

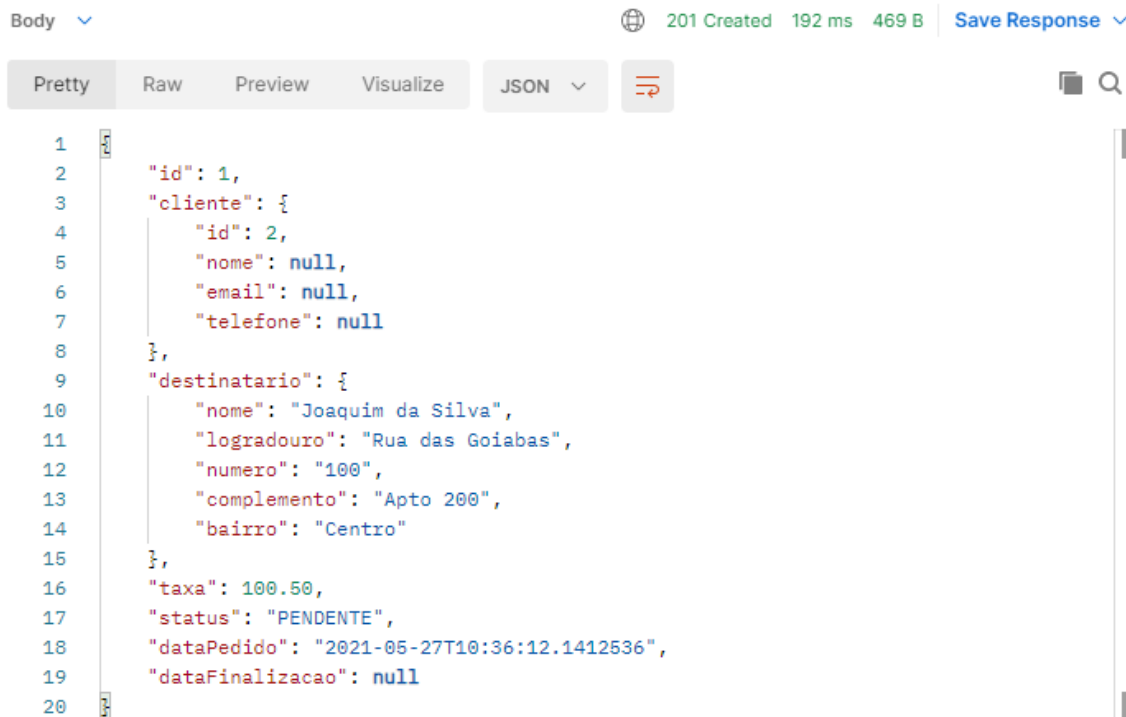


Figura 68 - Resposta à POST request da figura 67. Repare que conseguimos visualizar todas as propriedades de uma entrega agora: as não definidas como null e as definidas pelo nosso Service com seus devidos valores.

Os dados do cliente não foram recarregados no momento do cadastro, pois passamos apenas o id para referenciar um cliente existente.

Se tentarmos referenciar um cliente que não existe, obtemos um erro do servidor:

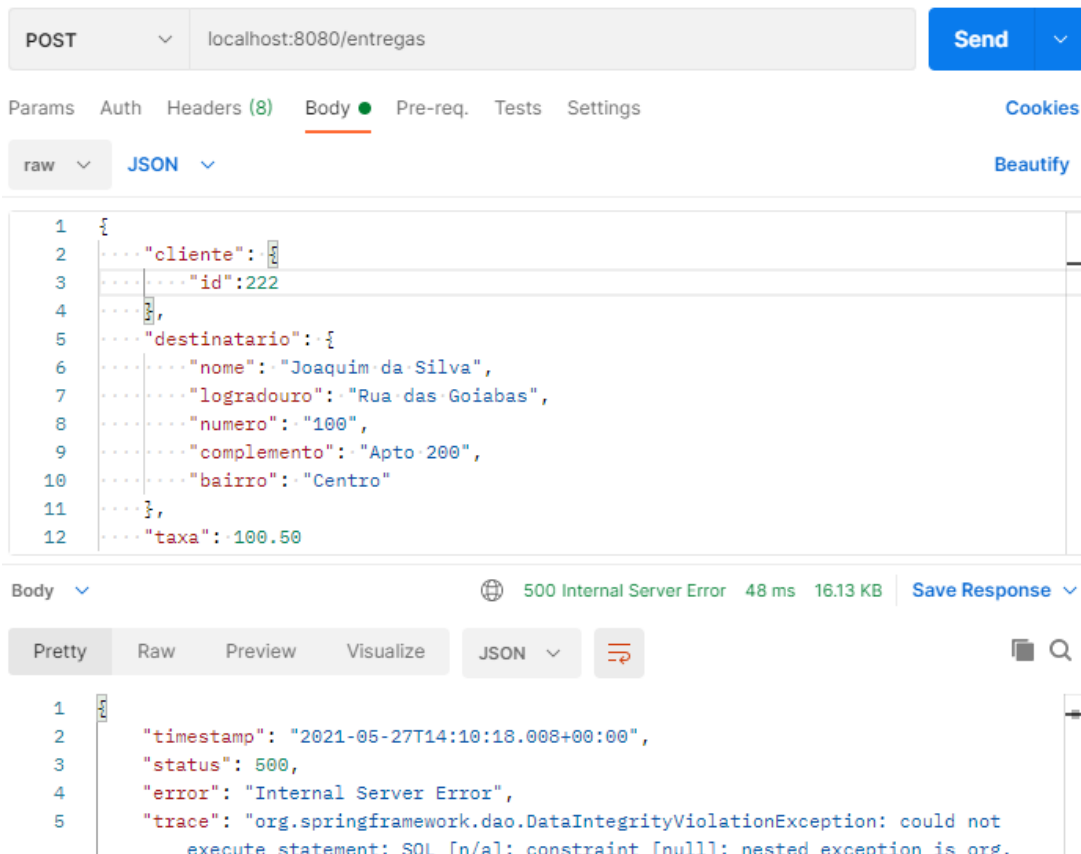


Figura 69 - Erro HTTP 500 baseado na referência de um cliente com Id inexistente.

Vamos tratar esse erro com o nosso `NegocioException` (que abriga as exceções acionadas por regras de negócio) para que, ao invés do usuário receber um código 500 de erro interno de servidor, ele receba um erro 400 explicando o que ele fez de errado.

Nesse caso, não utilizamos HTTP 404 pois o recurso entregas existe, o que não existe é uma resposta para um cliente com Id inexistente.

Nosso `SolicitacaoClienteService` ficou dessa forma:

```
@AllArgsConstructor
@Service
public class SolicitacaoEntregaService {

    private EntregaRepository entregaRepository;
    private ClienteRepository clienteRepository;

    @Transactional // será totalmente revertida no BD caso falhe
    public Entrega solicitar(Entrega entrega) {

        Cliente cliente =
            clienteRepository.findById(entrega.getCliente().getId())
                .orElseThrow(() -> new NegocioException("Cliente
                não encontrado."));

        entrega.setCliente(cliente);
        entrega.setStatus(StatusEntrega.PENDENTE);
        entrega.setDataPedido(LocalDate.now());

        return entregaRepository.save(entrega);
    }
}
```

Dentro da função `Entrega`, injetamos um objeto da classe `Cliente` para realizar a busca pelo id do cliente em nosso banco de dados. Se um cliente com esse id não for encontrado, retornamos uma exceção com a mensagem “Cliente não encontrado.”. Se o cliente for encontrado, o carregamos em nossa resposta ao método `solicitar()`. Vejamos isso nas figuras abaixo:

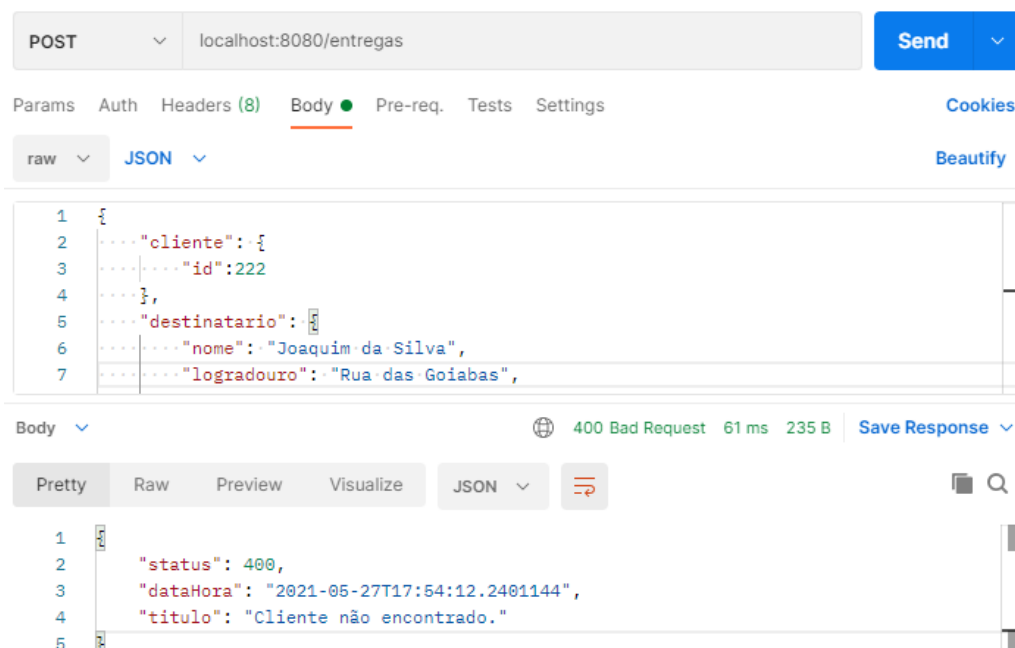


Figura 70 - Nova exceção mostrada após passagem de cliente com id inexistente.

POST localhost:8080/entregas Send

Params Auth Headers (8) **Body** Pre-req. Tests Settings Cookies

raw JSON Beautify

```

1 {
2   .... "cliente": {
3     .... "id": 3
4   },
5   .... "destinatario": {
6     .... "nome": "Joaquim da Silva",
7     .... "logradouro": "Rua das Goiabas",

```

Body 201 Created 22 ms 506 B Save Response

Pretty Raw Preview Visualize JSON 🔍

```

1 {
2   "id": 3,
3   "cliente": {
4     "id": 3,
5     "nome": "Fernando",
6     "email": "fernando@algaworks.com",
7     "telefone": "34 97778-9990"
8   },
9   "destinatario": {
10    "nome": "Joaquim da Silva",
11    "logradouro": "Rua das Goiabas",
12    "numero": "100",
13    "complemento": "Apto 200",
14    "bairro": "Centro"
15  },
16  "taxa": 100.50,
17  "status": "PENDENTE",
18  "dataPedido": "2021-05-27T17:56:49.4409637",
19  "dataFinalizacao": null
20 }

```

Figura 71 - Resposta de POST request bem-sucedida. Agora, os dados do cliente referentes ao id informado são exibidos.

Considerando que podemos querer buscar clientes em outros momentos e a partir de outros serviços da nossa API, vamos refatorar essa busca de Cliente, movendo-a de SolicitacaoEntregaService para CatalogoClienteService. Dessa forma, removemos a injeção da dependência Cliente do nosso SolicitacaoEntregaService e passamos a injetar o CatalogoClienteService.

Nossa SolicitacaoEntregaService ficou assim:

```

@AllArgsConstructor
@Service
public class SolicitacaoEntregaService {

    private EntregaRepository entregaRepository;
    private CatalogoClienteService catalogoClienteService;

    @Transactional // será totalmente revertida no BD caso falhe
    public Entrega solicitar(Entrega entrega) {

```

```

        Cliente cliente =
catalogoClienteService.buscar(entrega.getCliente().getId());

        entrega.setCliente(cliente);
        entrega.setStatus(StatusEntrega.PENDENTE);
        entrega.setDataPedido(LocalDate.now());

        return entregaRepository.save(entrega);
    }
}

```

E nosso CatalogoClienteService recebeu a nova função buscar():

```

public Cliente buscar(Long clienteId) {
    return clienteRepository.findById(clienteId)
        .orElseThrow(() -> new NegocioException("Cliente
não encontrado."));
}

```

Uma preocupação de que devemos ter com a segurança da aplicação é que, no momento, o consumidor da nossa API consegue enviar qualquer parâmetro dos nossos Domain Models como entrada. Isso pode não só afetar a integridade dos nossos dados, como também pode comprometer a segurança deles.

Até então estávamos cadastrando todas as nossas entregas sem data de finalização, mas imagine que o consumidor da nossa API descobriu que nosso Domain Model Entrega possui essa propriedade. Vejamos o que acontece se criarmos uma requisição passando uma dataFinalizacao como parâmetro:

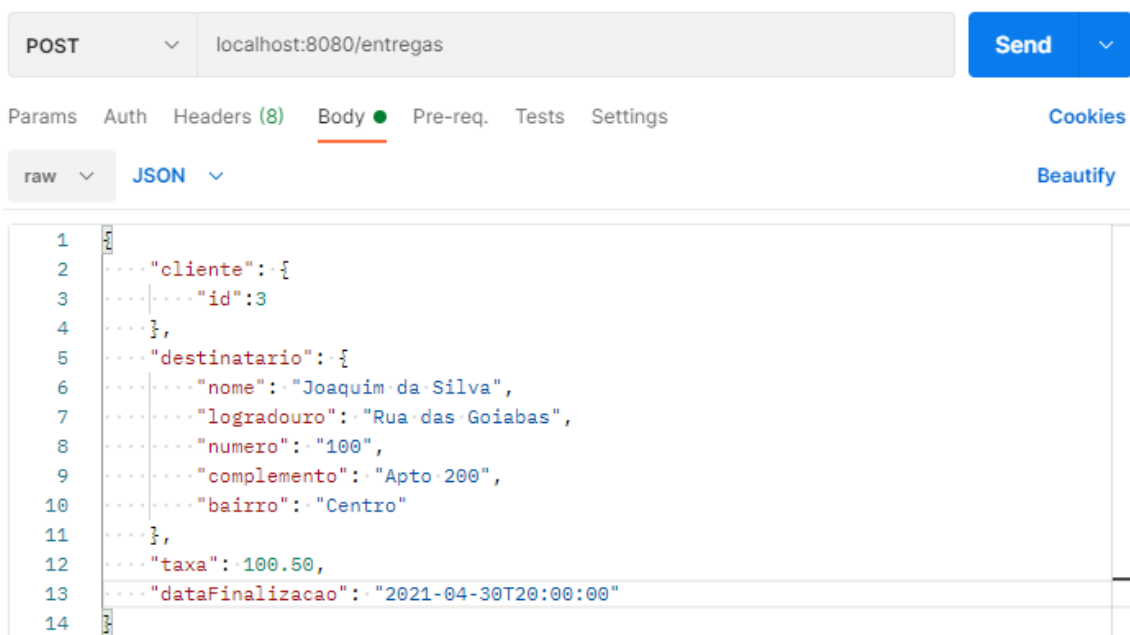


Figura 72 - POST request passando um parâmetro indesejado no corpo da requisição.

```
Body 201 Created 43 ms 523 B Save Response

Pretty Raw Preview Visualize JSON

1  {
2    "id": 4,
3    "cliente": {
4      "id": 3,
5      "nome": "Fernando",
6      "email": "fernando@algaworks.com",
7      "telefone": "34 97778-9990"
8    },
9    "destinatario": {
10     "nome": "Joaquim da Silva",
11     "logradouro": "Rua das Goiabas",
12     "numero": "100",
13     "complemento": "Apto 200",
14     "bairro": "Centro"
15   },
16   "taxa": 100.50,
17   "status": "PENDENTE",
18   "dataPedido": "2021-05-27T18:16:12.2617808",
19   "dataFinalizacao": "2021-04-30T20:00:00"
20 }
```

Figura 73 - Resposta à requisição feita na figura 72. Repare que foi criada uma Entrega com data de finalização informada pelo consumidor da API, o que não é um comportamento desejado.

Para evitar esse comportamento proveniente de POST requests, podemos adicionar a anotação `@JsonProperty(access = Access.READ_ONLY)` em cada propriedade das nossas classes. A princípio, estamos tendo esse problema com a classe `Entrega`, então vejamos como ela fica com essas anotações:

```
@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Entrega {

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

    @ManyToOne
    private Cliente cliente;

    @Embedded
    private Destinatario destinatario;
    private BigDecimal taxa;

    @JsonProperty(access = Access.READ_ONLY)
    @Enumerated(EnumType.STRING)
    private StatusEntrega status;

    @JsonProperty(access = Access.READ_ONLY)
    private LocalDateTime dataPedido;

    @JsonProperty(access = Access.READ_ONLY)
    private LocalDateTime dataFinalizacao;
```

}

Vamos testar nossa última requisição novamente:

The screenshot shows a REST client interface. At the top, a POST request is configured to `localhost:8080/entregas`. The request body is in JSON format, containing nested objects for `cliente` and `destinatario`, along with `taxa` and `dataFinalizacao`. The response status is 201 Created, with a response time of 31 ms and a body size of 506 B. The response body is also in JSON format, containing nested objects for `cliente` and `destinatario`, along with `taxa`, `status`, `dataPedido`, and `dataFinalizacao`.

```
POST localhost:8080/entregas

{
  "cliente": {
    "id": 3
  },
  "destinatario": {
    "nome": "Joaquim da Silva",
    "logradouro": "Rua das Goiabas",
    "numero": "100",
    "complemento": "Apto 200",
    "bairro": "Centro"
  },
  "taxa": 100.50,
  "dataFinalizacao": "2021-04-30T20:00:00"
}
```

201 Created 31 ms 506 B Save Response

```
{
  "id": 5,
  "cliente": {
    "id": 3,
    "nome": "Fernando",
    "email": "fernando@algaworks.com",
    "telefone": "34 97778-9990"
  },
  "destinatario": {
    "nome": "Joaquim da Silva",
    "logradouro": "Rua das Goiabas",
    "numero": "100",
    "complemento": "Apto 200",
    "bairro": "Centro"
  },
  "taxa": 100.50,
  "status": "PENDENTE",
  "dataPedido": "2021-05-27T18:25:54.3156676",
  "dataFinalizacao": null
}
```

Figura 74 - Restrição de propriedade feita com sucesso. Mesmo que o consumidor da API passe o parâmetro `dataFinalizacao`, nossa entidade não o reconhece.

### 3.2. Validação em cascata e Validation Groups

No momento, se tentarmos criar requisições sem taxa ou sem cliente, acionamos exceções que nos retornam erros HTTP 500 (Internal Server Error), e não queremos isso. Precisamos criar maneiras de validar esses campos em nossa aplicação. Para isso, adicionamos a anotação `@Valid`

ao argumento da nossa requisição no EntregaController e @NotNull nas propriedades taxa e cliente em nossa entidade Entrega.

```
public class EntregaController {

    private SolicitacaoEntregaService solicitacaoEntregaService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Entrega solicitar(@Valid @RequestBody Entrega entrega) {
        return solicitacaoEntregaService.solicitar(entrega);
    }

}

@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Entrega {

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

    @NotNull
    @ManyToOne
    private Cliente cliente;

    @Embedded
    private Destinatario destinatario;

    @NotNull
    private BigDecimal taxa;

    @JsonProperty(access = Access.READ_ONLY)
    @Enumerated(EnumType.STRING)
    private StatusEntrega status;

    @JsonProperty(access = Access.READ_ONLY)
    private LocalDateTime dataPedido;

    @JsonProperty(access = Access.READ_ONLY)
    private LocalDateTime dataFinalizacao;

}
```

Feito isso, nossa requisição já aciona o ApiExceptionHandler que criamos anteriormente e já recebemos mensagens de erro 400.



POST localhost:8080/entregas

Params Auth Headers (8) Body ● Pre-req. Tests Settings Cookies

raw JSON Beautify

```

1 {
2   "destinatario": {
3     "nome": "Joaquim da Silva",
4     "logradouro": "Rua das Goiabas",
5     "numero": "100",
6     "complemento": "Apto 200",
7     "bairro": "Centro"
8   }
9 }

```

Body 400 Bad Request 6 ms 406 B Save Response

Pretty Raw Preview Visualize JSON

```

1 {
2   "status": 400,
3   "dataHora": "2021-05-27T20:27:56.9021249",
4   "titulo": "Um ou mais campos estão inválidos. Faça o preenchimento correto e tente novamente.",
5   "campos": [
6     {
7       "nome": "taxa",
8       "mensagem": "não deve ser nulo"
9     },
10    {
11      "nome": "cliente",
12      "mensagem": "não deve ser nulo"
13    }
14  ]
15 }

```

Figura 75 - POST request de Entrega sem cliente e sem taxa. Nossa resposta explica o motivo de forma limpa ao consumidor da API.

Vamos alterar o retorno para NotNull em nosso src/main/resources/messages.properties:

NotBlank=é obrigatório.

NotNull=é obrigatório.

Size=deve ter no mínimo {2} e no máximo {1} caracteres.

Email=deve ser um e-mail válido.

Alterações na resposta após repetir a última requisição:

```

"campos": [
  {
    "nome": "taxa",
    "mensagem": "é obrigatório."
  },
  {
    "nome": "cliente",
    "mensagem": "é obrigatório."
  }
]

```

Figura 76 - Tratamento de validação NotNull no messages.properties.

Certo... Mas agora, e se passarmos um cliente vazio, sem id?

POST localhost:8080/entregas

Params Auth Headers (8) Body Pre-req. Tests Settings Cookies

raw JSON Beautify

```
1 {
2   "cliente": {
3   },
4 },
5 "destinatario": {
6   "nome": "Joaquim da Silva",
7   "logradouro": "Rua das Goiabas",
8   "numero": "100",
9   "complemento": "Apto 200",
10  "bairro": "Centro"
11 },
12 "taxa": 100.50
13 }
```

Body 500 Internal Server Error 25 ms 11.85 KB Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "timestamp": "2021-05-27T23:35:30.536+00:00",
3   "status": 500,
4   "error": "Internal Server Error",
5   "trace": "org.springframework.dao.InvalidDataAccessApiUsageException: The given
            id must not be null!; nested exception is java.lang.
            IllegalArgumentException: The given id must not be null!\r\n\tat org.
            springframework.orm.jpa.EntityManagerFactoryUtils.
            convertJpaAccessExceptionIfPossible(EntityManagerFactoryUtils.java:374)"
}
```

Figura 77 - Resposta de POST request de Entrega com cliente sem id.

Então é só colocar o @NotNull lá no id do cliente, né?

```
public class Cliente {
    @NotNull
    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

Não.

Body 500 Internal Server Error 21 ms 11.85 KB Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "timestamp": "2021-05-27T23:37:07.912+00:00",
3   "status": 500,
4   "error": "Internal Server Error",
5   "trace": "org.springframework.dao.InvalidDataAccessApiUsageException: The given
            id must not be null!; nested exception is java.lang.
            IllegalArgumentException: The given id must not be null!\r\n\tat org.
            springframework.orm.jpa.EntityManagerFactoryUtils.
            convertJpaAccessExceptionIfPossible(EntityManagerFactoryUtils.java:374)"
}
```

Figura 78 - Resposta de POST request de Entrega com cliente sem id. @NotNull no id da classe Cliente não adiantou.

Para validarmos os dados da propriedade cliente que temos dentro de entrega, a anotamos com `@Valid` ao invés de anotarmos o id da própria classe Cliente.

```
public class Entrega {  
  
    @EqualsAndHashCode.Include  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    @Valid  
    @NotNull  
    @ManyToOne  
    private Cliente cliente;  
  
    ...  
}
```

Vamos retestar nossa última requisição.

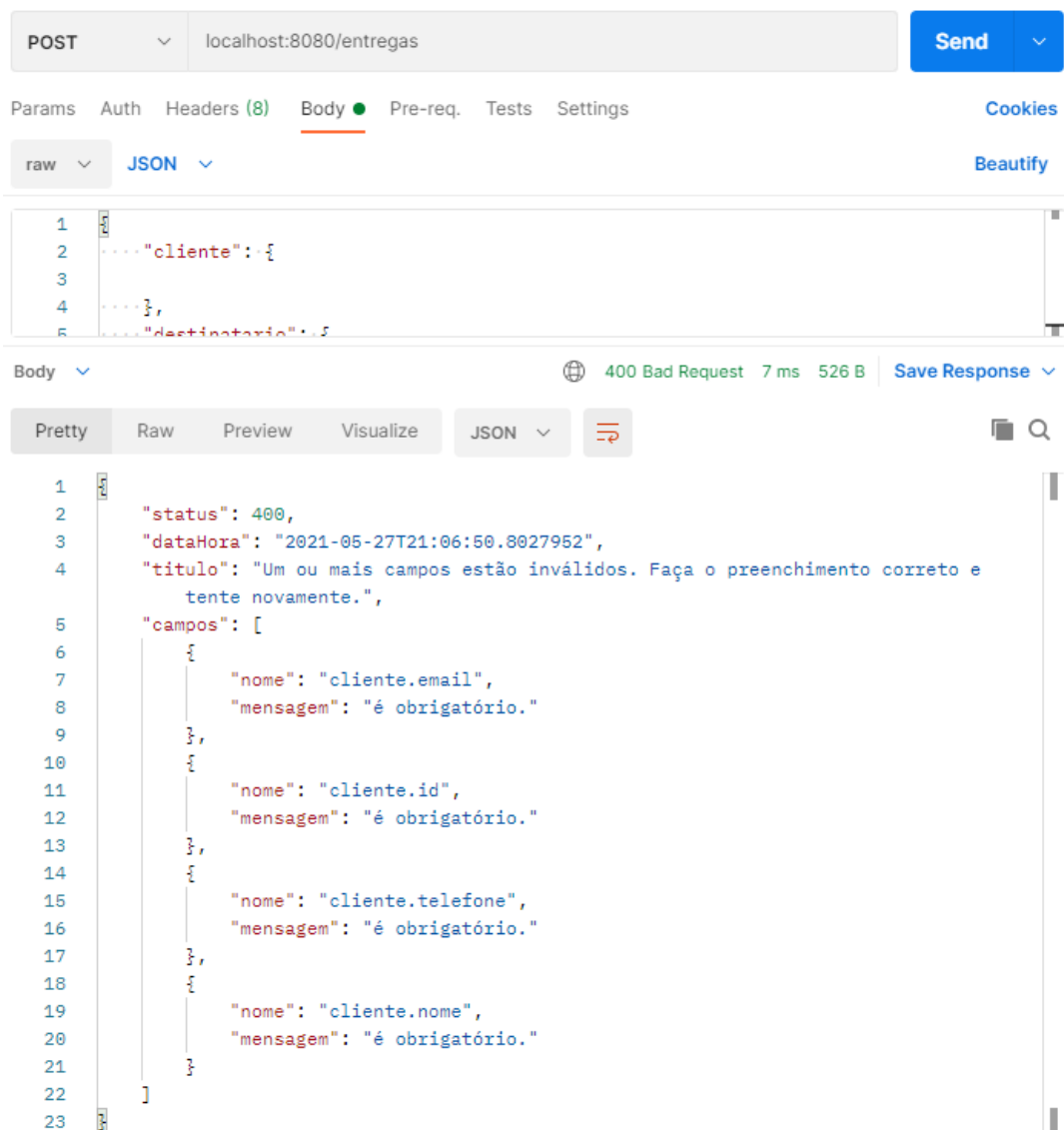


Figura 79 - POST request de Entrega sem cliente id após anotar o cliente da Entrega com `@Valid`.

Ainda não é isso que queremos. Estamos apenas referenciando um cliente pelo id, não queremos passar como parâmetro todas as suas informações. Ao mesmo tempo, não podemos remover todas as anotações de validações já feitas na classe Cliente apenas para conseguirmos executar nossa última POST request. Inclusive, se deixarmos a anotação `@NotNull` no id da classe Cliente, não conseguiremos cadastrar novos clientes, pois ele está exigindo um id para o cadastro (o que não faz sentido). E agora?

Vamos usar Validation Groups. Por padrão, todas as anotações de validação possuem um grupo chamado Default, implicitamente declarado. Por exemplo:

```
@NotNull é a mesma coisa que @NotNull(groups = Default.class)
```

```
@NotBlank é a mesma coisa que @NotBlank(groups = Default.class)
```

Default é uma interface de marcação de grupos, e não possui nenhuma lógica implementada dentro dela.

O que podemos fazer é criar nossas próprias interfaces de marcação para diferenciarmos propriedades a serem validadas em diferentes grupos. Em `src/main/java/com.algaworks.algalog.domain`, criamos nossa interface `ValidationGroups`:

- PACKAGE -> `com.algaworks.algalog.domain`
- NAME -> `ValidationGroups`

Dentro do `ValidationGroups`, criamos nossa interface para lidar com o `Clienteld`:

```
public interface ValidationGroups {  
  
    public interface Clienteld { }  
  
}
```

E anotamos o id da nossa classe Cliente para referenciar a nova marcação da anotação pela interface `Clienteld` em `ValidationGroups`.

```
public class Cliente {  
  
    @NotNull(groups = ValidationGroups.Clienteld.class)  
    @EqualsAndHashCode.Include  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    ...  
  
}
```

O problema é que isso ainda não funciona. Já conseguimos cadastrar clientes novamente.

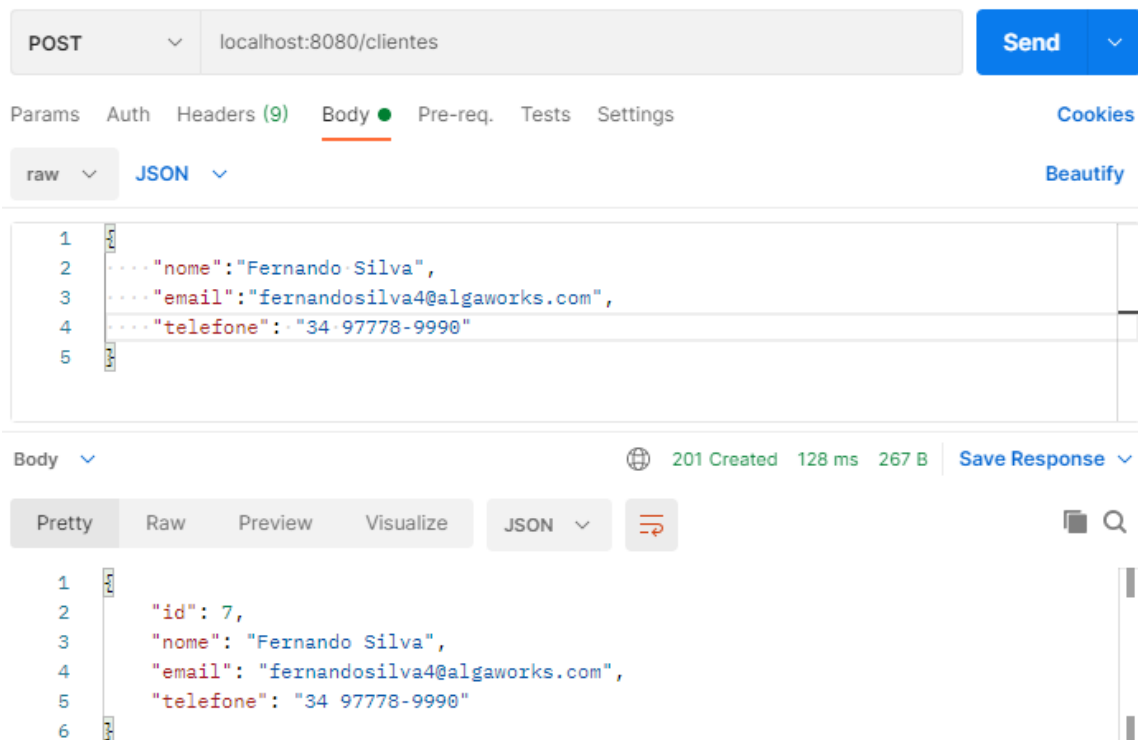


Figura 80 - POST request para cadastro de cliente após a alteração do Validation Group no id da classe Cliente.

Mas agora a validação do id da propriedade Cliente em Entrega deixou de funcionar.

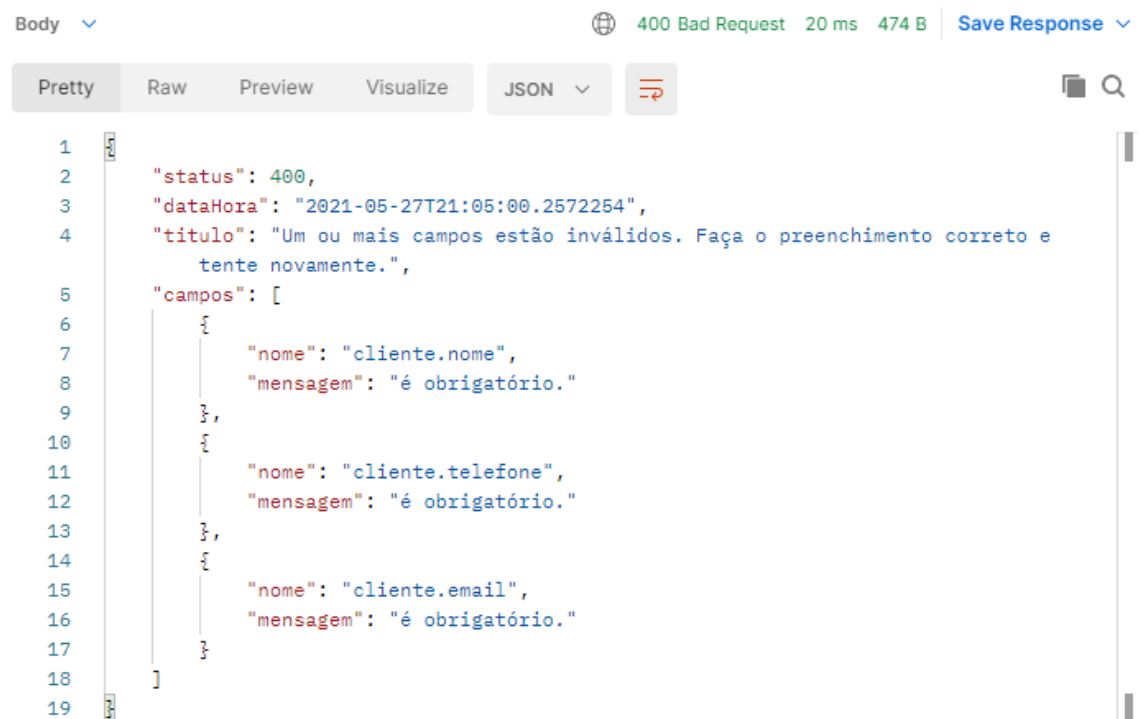


Figura 81 - Resposta à POST request de Entrega com cliente sem id.

Para isso, utilizamos uma nova anotação, o `@ConvertGroup`, que altera o grupo utilizado em uma validação de um para outro. No nosso caso, queremos alterar do grupo Default para o grupo Clienteld. **MUITO CUIDADO NA HORA DE IMPORTAR O Default.Class. Se ele for importado do pacote errado, o @ConvertGroup não vai funcionar. É PRA IMPORTAR DO import javax.validation.groups.Default;** Vamos conferir esse código em nossas classes:

```

public class Entrega {

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

    @Valid
    @ConvertGroup(from = Default.class, to =
ValidationGroups.ClientId.class)
    @NotNull
    @ManyToOne
    private Cliente cliente;

    ...
}

```

Na classe Entrega, dizemos que queremos que a validação seja feita utilizando o grupo ValidationGroups.ClientId.

```

@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Getter
@Setter
@Entity
public class Cliente {

    @NotNull(groups = ValidationGroups.ClientId.class)
    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank
    @Size(max = 60)
    private String nome;

    ...
}

```

Já na classe Cliente, só validamos o id utilizando o grupo ValidationGroups.ClientId. Todas as outras propriedades são validadas através do grupo Default.

Na prática, isso significa que, ao cadastrarmos um novo Cliente, não informaremos um grupo de validação, tornando implícito o uso da validação pelo grupo Default e, assim, descartando a validação do id.

Por outro lado, quando cadastrarmos uma Entrega, validaremos sua propriedade cliente com o grupo de validação ClientId, que marca apenas o id (ou desmarca-o do grupo Default) e descarta a validação de todas as outras propriedades, implicitamente validadas pelo grupo Default.

Vamos começar a testar o resultado das nossas requisições após a separação e conversão dos ValidationGroups. Primeiro, cadastraremos um novo cliente:

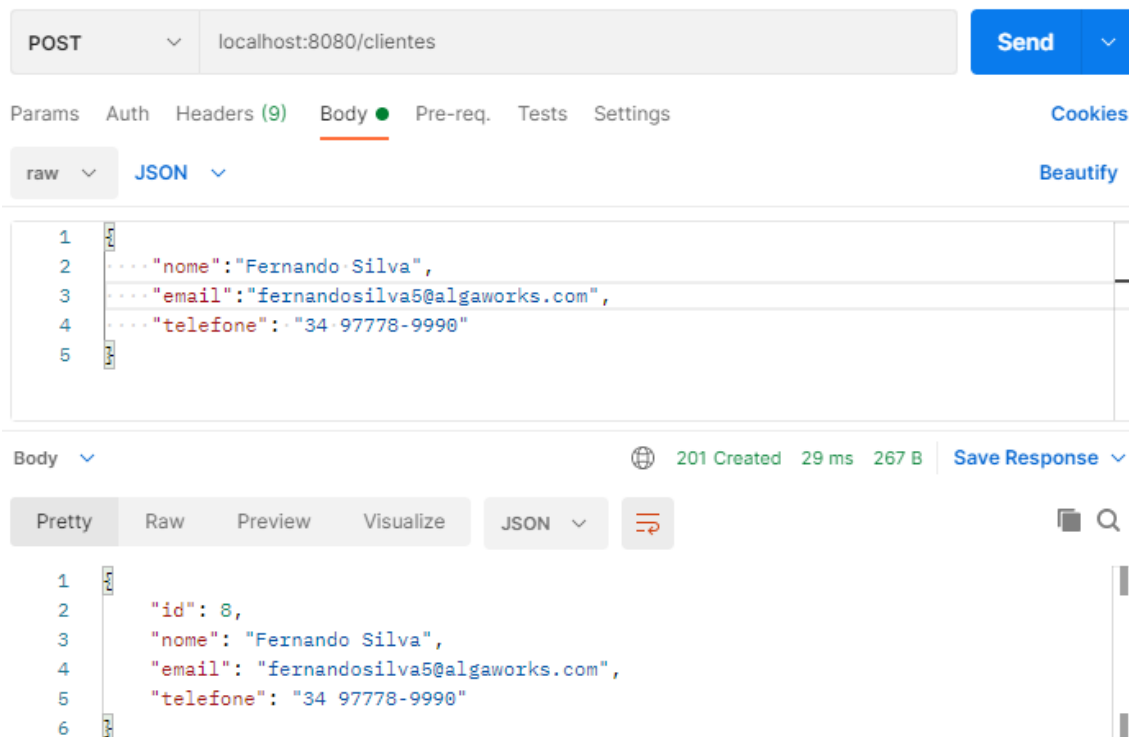


Figura 82 - Cadastro de cliente após implementação do ValidationGroup ClientId. Requisição bem-sucedida.

Se não especificarmos os campos do Cliente, nossa validação Default segue funcionando.

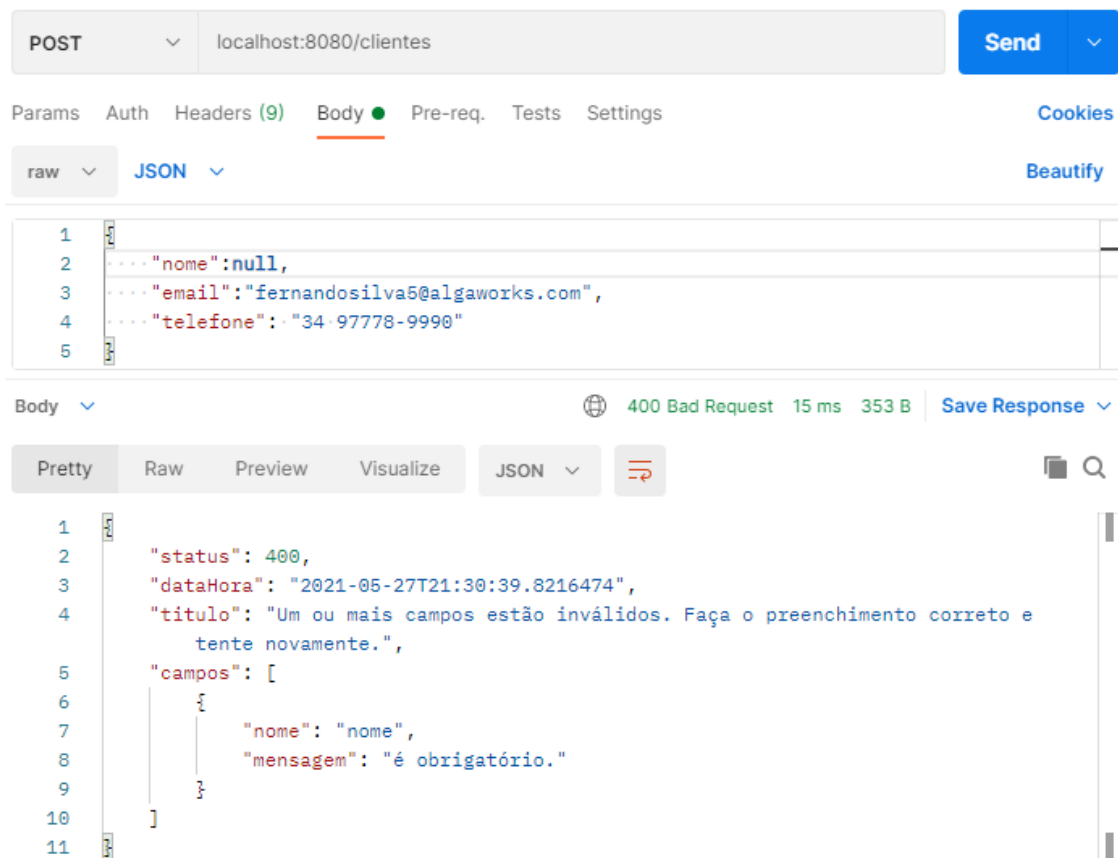
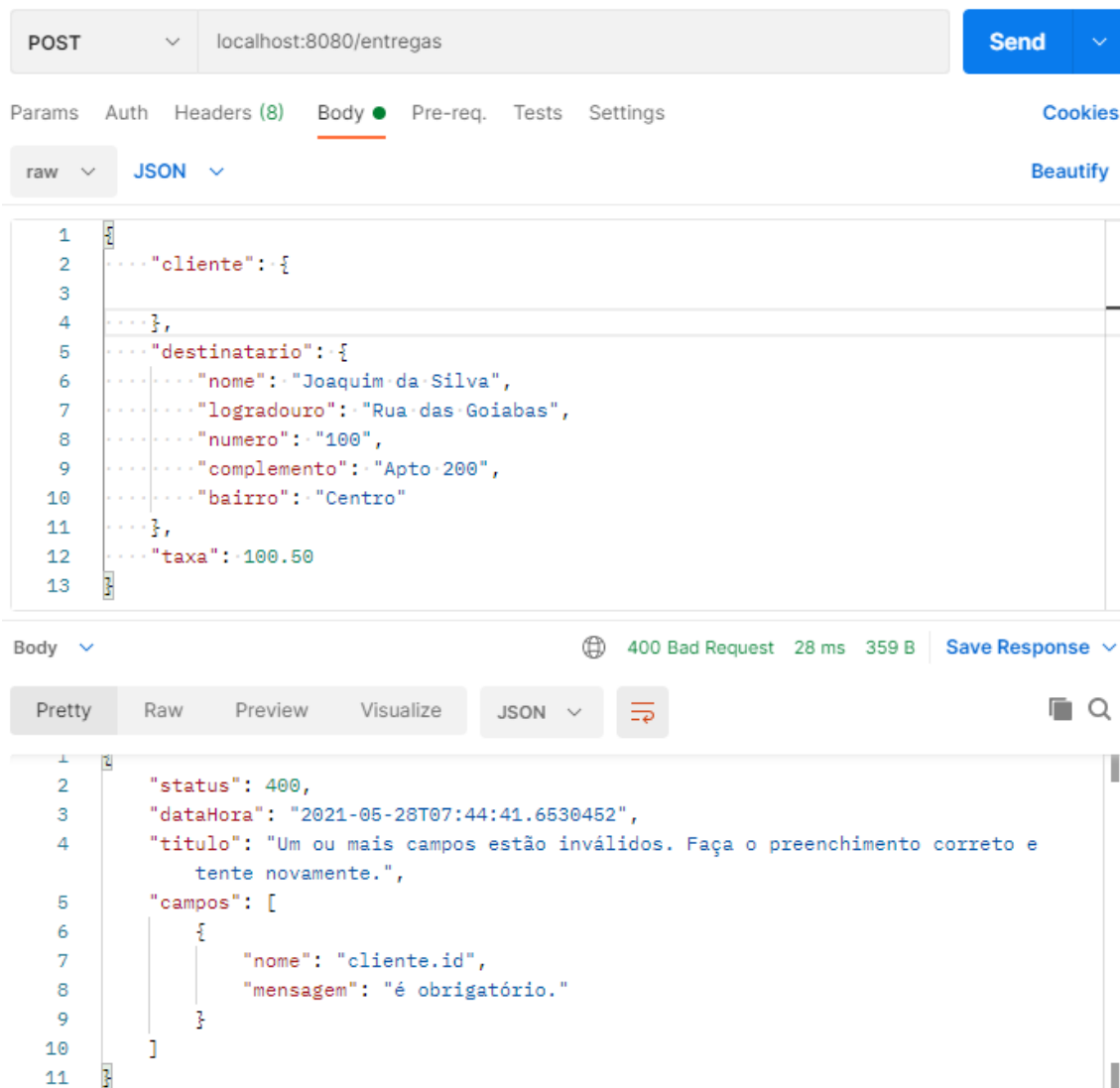


Figura 83 - Tratamento de exceção de nome vazio após criação de ValidationGroup. Tudo funcionando.

Agora, se fizermos uma última solicitação de entrega com o cliente.id vazio, vejamos o que acontece:



POST localhost:8080/entregas

Send

Params Auth Headers (8) Body ● Pre-req. Tests Settings Cookies

raw JSON Beautify

```
1 {
2   "cliente": {
3   },
4 },
5 "destinatario": {
6   "nome": "Joaquim da Silva",
7   "logradouro": "Rua das Goiabas",
8   "numero": "100",
9   "complemento": "Apto 200",
10  "bairro": "Centro"
11 },
12 "taxa": 100.50
13 }
```

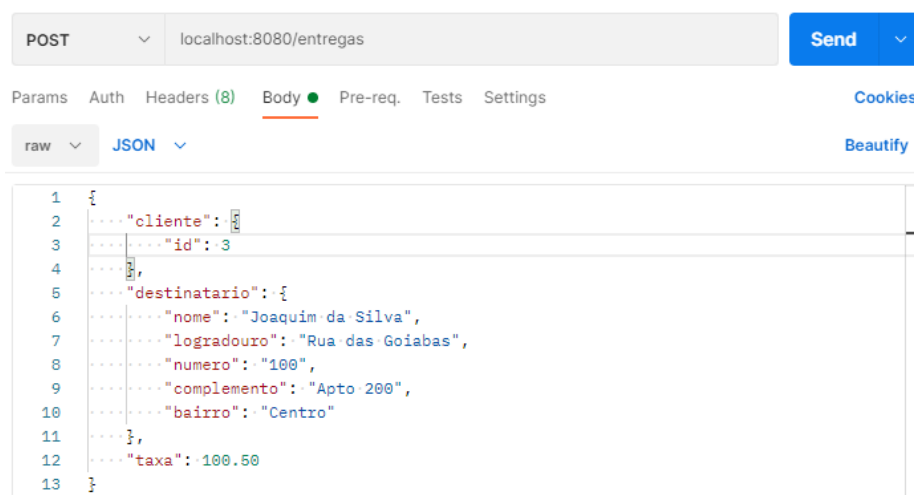
Body 400 Bad Request 28 ms 359 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "status": 400,
3   "dataHora": "2021-05-28T07:44:41.6530452",
4   "titulo": "Um ou mais campos estão inválidos. Faça o preenchimento correto e
5   tente novamente.",
6   "campos": [
7     {
8       "nome": "cliente.id",
9       "mensagem": "é obrigatório."
10    }
11  ]
12 }
```

Figura 84 - Validação condicional por ValidationGroups para o cliente.id funcionando.

E, agora, se inserirmos um id correto, ele efetua a solicitação da Entrega corretamente.



POST localhost:8080/entregas

Send

Params Auth Headers (8) Body ● Pre-req. Tests Settings Cookies

raw JSON Beautify

```
1 {
2   "cliente": {
3     "id": 3
4   },
5   "destinatario": {
6     "nome": "Joaquim da Silva",
7     "logradouro": "Rua das Goiabas",
8     "numero": "100",
9     "complemento": "Apto 200",
10    "bairro": "Centro"
11  },
12  "taxa": 100.50
13 }
```

Figura 85 - POST request de solicitação de entrega após criação de ValidationGroups.



```
1 {
2   "id": 6,
3   "cliente": {
4     "id": 3,
5     "nome": "Fernando",
6     "email": "fernando@algaworks.com",
7     "telefone": "34 97778-9990"
8   },
9   "destinatario": {
10    "nome": "Joaquim da Silva",
11    "logradouro": "Rua das Goiabas",
12    "numero": "100",
13    "complemento": "Apto 200",
14    "bairro": "Centro"
15  },
16  "taxa": 100.50,
17  "status": "PENDENTE",
18  "dataPedido": "2021-05-28T07:52:23.8549862",
19  "dataFinalizacao": null
20 }
```

Figura 86 - Corpo da resposta da POST request feita na Figura 85.

Agora, vamos finalizar os nossos métodos HTTP do EntregaController, implementando os métodos listar() e buscar() como fizemos com o Cliente:

```
@AllArgsConstructor
@RestController
@RequestMapping("/entregas")
public class EntregaController {

    private EntregaRepository entregaRepository;
    private SolicitacaoEntregaService solicitacaoEntregaService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Entrega solicitar(@Valid @RequestBody Entrega entrega) {
        return solicitacaoEntregaService.solicitar(entrega);
    }

    @GetMapping
    public List<Entrega> listar() {
        return entregaRepository.findAll();
    }

    @GetMapping("/{entregaId}")
    public ResponseEntity<Entrega> buscar(@PathVariable Long entregaId) {
        return entregaRepository.findById(entregaId)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }
}
```

Além disso, vamos validar as outras propriedades da nossa classe Entrega: adicionamos @Valid e @NotNull a destinatário e @NotBlank em cada propriedade da classe Destinatario, olha só:

```

@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Entrega {

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

    @Valid
    @ConvertGroup(from = Default.class, to =
ValidationGroups.ClienteId.class)
    @NotNull
    @ManyToOne
    private Cliente cliente;

    @Valid
    @NotNull
    @Embedded
    private Destinatario destinatario;

    ...
}

@Getter
@Setter
@Embeddable // embutível, será embutida na classe Entrega
public class Destinatario {

    @NotBlank
    @Column(name = "destinatario_nome")
    private String nome;

    @NotBlank
    @Column(name = "destinatario_logradouro")
    private String logradouro;

    @NotBlank
    @Column(name = "destinatario_numero")
    private String numero;

    @NotBlank
    @Column(name = "destinatario_complemento")
    private String complemento;

    @NotBlank
    @Column(name = "destinatario_bairro")
    private String bairro;

}

```

Dessa forma, completamos a criação e validação do nosso endpoint Entrega, pelo menos até novos ajustes serem feitos.

### 3.3. Boas práticas para trabalhar com data hora

Por padrão, utilizamos o padrão internacional ISO8601 como configuração padrão de data e hora, que é o recomendado para REST APIs.

O problema é que nossa API, até então, mostra as datas e horas sem nenhum offset em relação ao UTC 00, ou seja, sem nenhuma identificação de fuso horário, como mostrado abaixo:

```
"dataPedido": "2021-05-28T07:52:23.8549862"
```

O horário de Brasília, por exemplo, está em UTC -03. É uma boa prática não somente utilizarmos o ISO8601, como também adicionar o offset na representação do horário. Podemos resolver isso alterando as nossas propriedades LocalDateTime, que não possui offset, para OffsetDateTime.

Foram alterados os arquivos:

- Entrega,
- SolicitacaoEntregaService,
- Problema,
- ApiExceptionHandler.

Onde havia LocalDateTime, agora há OffsetDateTime, e as importações desses pacotes também foram substituídas.

Agora, nossos objetos DateTime possuem o offset:

```
"dataPedido": "2021-05-28T07:52:24-03:00"
```

### 3.4. Isolando Domain Model do Representation Model

Quando fazemos uma GET request em um recurso único de /entregas, temos uma representação do recurso, que pode ser em JSON, XML ou qualquer outro tipo.

Considerando que a nossa requisição é feita por essa função:

```
@GetMapping("/{entregaId}")
public ResponseEntity<Entrega> buscar(@PathVariable Long entregaId) {
    return entregaRepository.findById(entregaId)
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
}
```

Entendemos que um objeto da classe Entrega é o modelo de representação do recurso, ou Resource Representation Model: é a classe que modela o que deve ser retornado na API.

A classe Entrega também é uma classe de outra camada da nossa aplicação, a camada de Domínio (domain). Entrega é uma entidade do nosso domain.model. Ou seja, estamos compartilhando a mesma classe tanto com o modelo de domínio como como o modelo de representação, e isso pode não ser bom.

Podemos, por exemplo, ter alguma propriedade da entidade que não queremos expor na API e que precisará ser configurada manualmente. Alguém pode estar trabalhando na camada domain e adiciona uma nova propriedade.

Assim que adicionamos uma nova propriedade em nosso domain.model (nossa Classe Entrega), o representation model da nossa API terá sua representação alterada imediatamente, possivelmente expondo uma propriedade que contenha dados sensíveis.

Há, portanto, dois casos em que possivelmente as coisas dão errado:

1. Quebramos nossa API para consumidores que já utilizavam antes da adição da nova propriedade
2. Queremos expor uma representação do objeto diferente do domain.model.

Quanto mais o nosso domain.model cresce, mais difícil fica de alterar a composição do nosso representation model. Por isso, o ideal seria isolar o domain.model do representation model. Para isso, utilizamos classes de transferências de dados usando o padrão DTO (Data Transfer Object).

DTOs são bastante utilizados para transportes de dados entre diferentes camadas de sistemas, diferentes módulos de sistemas distribuídos ou para serialização de dados. DTOs agrupam uma ou mais propriedades de uma classe ou mais classes em uma classe DTO mais simples, apenas com as propriedades necessárias.

Por isso, criamos uma classe DTO para servir como modelo de representação do recurso Entrega no pacote API. Em src/main/java/com.algaworks.algalog, criamos nosso EntregaModel:

- PACKAGE -> com.algaworks.algalog.api.model
- NAME -> EntregaModel

```
@Getter
@Setter
public class EntregaModel {

    private Long id;
    private String nomeCliente;
    private DestinatarioModel destinatario;
    private BigDecimal taxa;
    private StatusEntrega status;
    private OffsetDateTime dataPedido;
    private OffsetDateTime dataFinalizacao;

}
```

Nomeamos a classe como EntregaModel para ficar curto, mas poderia ser EntregaRepresentationModel, ou qualquer outro nome que indique que estamos tratando do modelo de representação. De qualquer forma, visto que essa classe está dentro do pacote .api, subentende-se que estamos falando do modelo de representação.

Repare que temos uma propriedade chamada DestinatarioModel, pois faremos a mesma separação para o domain.model e o representation model da classe Destinatario.

```
@Getter
@Setter
public class DestinatarioModel {

    private String nome;
    private String logradouro;
    private String numero;
    private String complemento;

}
```

```

        private String bairro;
    }

```

Agora, em nosso EntregaController, ao invés de retornar uma Entrega (entidade, vinculada ao domain.model), vamos retornar um EntregaModel, que é o nosso novo representation model. Começaremos pelo método buscar():

```

    @GetMapping("/{entregaId}")
    public ResponseEntity<EntregaModel> buscar(@PathVariable Long
entregaId) {
        return entregaRepository.findById(entregaId).map(entrega -> {
            EntregaModel entregaModel = new EntregaModel();
            entregaModel.setId(entrega.getId());

            entregaModel.setNomeCliente(entrega.getCliente().getNome());
            entregaModel.setDestinatario(new DestinatarioModel());

            entregaModel.getDestinatario().setNome(entrega.getDestinatario().getNo
me());

            entregaModel.getDestinatario().setLogradouro(entrega.getDestinatario()
.getLogradouro());

            entregaModel.getDestinatario().setNumero(entrega.getDestinatario().get
Numero());

            entregaModel.getDestinatario().setComplemento(entrega.getDestinatario(
).getComplemento());

            entregaModel.getDestinatario().setBairro(entrega.getDestinatario().get
Bairro());

            entregaModel.setTaxa(entrega.getTaxa());
            entregaModel.setStatus(entrega.getStatus());
            entregaModel.setDataPedido(entrega.getDataPedido());

            entregaModel.setDataFinalizacao(entrega.getDataFinalizacao());

            return ResponseEntity.ok(entregaModel);
        })
        .orElse(ResponseEntity.notFound().build());
    }

```

Nosso código ficou bem extenso, então vamos explicar:

Estamos transferindo as propriedades do objeto Entrega para o nosso EntregaModel manualmente, por meio de uma lambda function. Repare que ainda instanciamos um objeto Destinatario e que, agora, para modificarmos suas propriedades, realizamos uma combinação de getDestinatario.setPropriedade().

Ao final, quando testamos nosso EntregaModel por meio de uma GET request, verificamos que ele já funciona:

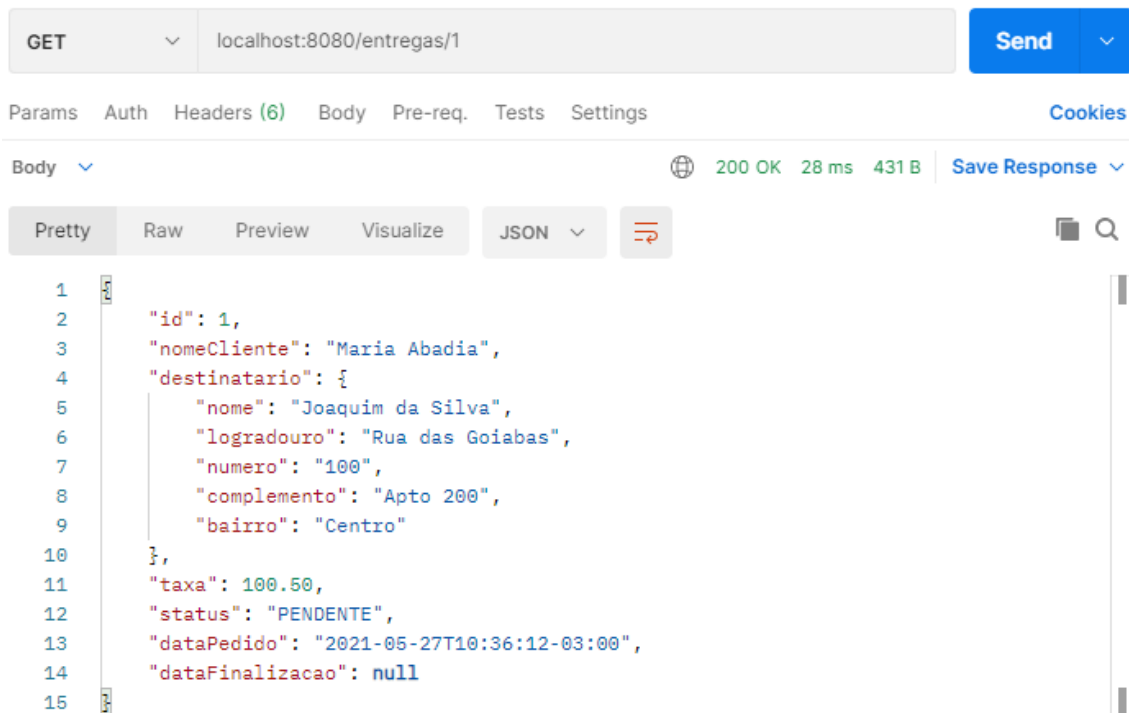


Figura 87 - Resposta a uma GET request modificada para retornar um objeto da classe EntregaModel.

### 3.5. Simplificando a transformação de objetos com ModelMapper

A separação do Domain Model para o Representation Model gerou muito código boilerplate, e em sistemas mais robustos, seria um trabalho muito repetitivo.

Felizmente, há uma biblioteca que facilita esse trabalho para nós: o ModelMapper. Para utilizá-la em nosso projeto, vamos adicioná-la como uma dependência do Maven no nosso arquivo pom.xml.

```
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>2.4.2</version>
</dependency>
```

Após incluirmos o código abaixo em nosso pom.xml, podemos começar a utilizar a biblioteca.

Até então estávamos fazendo a conversão do Domain Model para o Representation Model em nosso EntregaController. Iniciaremos nossa mudança injetando a dependência ModelMapper nele.

Feito isso, podemos substituir todo aquele código de atribuição das propriedades de Entrega para o EntregaModel com esse código aqui:

```
EntregaModel entregaModel = modelMapper.map(entrega, EntregaModel.class);
```

Destrinchando:

- Criamos o objeto entregaModel, da classe EntregaModel
- Ele recebe o retorno do método modelMapper.map()

- O primeiro argumento desse método (entrega) é um objeto da classe original que queremos converter
- O segundo argumento (EntregaModel.class) é a classe para qual queremos converter o objeto do primeiro argumento.

Nosso método, então, ficou assim:

```
public ResponseEntity<EntregaModel> buscar(@PathVariable Long
entregaId) {
    return entregaRepository.findById(entregaId)
        .map(entrega -> {
            EntregaModel entregaModel =
modelMapper.map(entrega, EntregaModel.class);
            return ResponseEntity.ok(entregaModel);
        })
        .orElse(ResponseEntity.notFound().build());
}
```

Mas será que funciona?

```
*****
APPLICATION FAILED TO START
*****
```

Description:

Parameter 2 of constructor in com.algaworks.algalog.api.controller.EntregaController required a bean of type

Action:

Consider defining a bean of type 'org.modelmapper.ModelMapper' in your configuration.

Figura 88 - Inicializar a API utilizando o ModelMapper na injeção de dependências não funcionou.

Não. O que acontece é que o ModelMapper é uma biblioteca de terceiros criada para uso geral no Java, ela não é um componente ou dependência do ecossistema Spring e, por isso, não é reconhecida como um Spring Bean. Para resolver esse problema, precisamos criar um arquivo de configurações, que intermedeie o contato entre o Spring e essa biblioteca externa.

Em src/main/java/com.algaworks.algalog, criamos nossa classe ModelMapperConfig:

- PACKAGE -> com.algaworks.algalog.common
- NAME -> ModelMapperConfig

@Configuration

```
public class ModelMapperConfig {

    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }

}
```

A anotação @Configuration marca o que queremos que se comporte como um componente do Spring e, assim, possa ser injetado em outras classes. Vamos sair um pouco da nossa API para explorarmos melhor os conceitos de Injeção de Dependências (DI) e Inversão de Controle (IoC):

Normalmente, para utilizarmos classes dentro de outras classes, precisamos instanciar objetos de uma classe dentro da outra. O problema disso é que geramos uma **dependência direta** entre classes, nos rendendo um **código fortemente acoplado**.

Quanto mais acoplado o código, mais difícil de refatorar, prover manutenção e testar esse código. Uma maneira de lidar com esse problema é **gerar código desacoplado** é utilizando injeções de dependência. Nas injeções de dependência, ao invés de instanciarmos um objeto da classe B dentro da classe A, passamos um objeto da classe B como parâmetro de A. Vejamos os exemplos a seguir:

```
public class A {
    B b;
}

public class B {

    public void helloWorld() {
        System.out.println("Hello World!");
    }

}
```

Para utilizar o método helloWorld() em objetos da classe A, preciso que eles tenham uma propriedade B. Se eu tentar executar o método a.helloWorld(), ele não funcionará. Vamos criar um objeto da classe B para atribuí-lo como propriedade A, então.

```
A a = new A();
a.helloWorld(); // não funciona!

B b = new B();
b.helloWorld();

a.setB(b);

a.getB().helloWorld(); // funciona!
```

Percebeu que temos que instanciar um objeto da classe B dentro de A caso queiramos utilizar o método helloWorld() da classe B, e que fazemos isso manualmente?

Pois é. Isso é um código altamente acoplado devido a uma dependência direta.

Imagina refatorar o código de um sistema grande e altamente acoplado: um pesadelo.

Para lidar com isso, o Spring facilita nossa vida com a Inversão de Controle (IoC) e com as injeções de dependências. Uma injeção de dependência, no nosso caso, significa que passaríamos um objeto da classe B como parâmetro na criação do nosso objeto A. Ou seja, todo o código ali acima mudaria para isso:

```
A a = new A(b);
a.getB().helloWorld();
```

Certo, mas resta uma dúvida... De onde surge esse objeto b passado por parâmetro? É aí que entra o princípio da IoC: o Spring instancia esse objeto para nós. Dessa forma, eliminamos a dependência direta na implementação de um objeto b para utilizarmos o método helloWorld() em objetos da classe A, deixando nosso código mais limpo, fácil de testar e manutenível.

O Spring possui vários componentes internos que podem ser injetados automaticamente de forma individual por meio da anotação como @Autowired ou todos de uma vez por meio da anotação @AllArgsConstructor.



Para os componentes externos, entretanto, precisamos criar a classe Configuration, anotada com @Configuration, e anotar cada um desses componentes com @Bean.

Voltando para a nossa API, vamos testar se o ModelMapper funcionou.

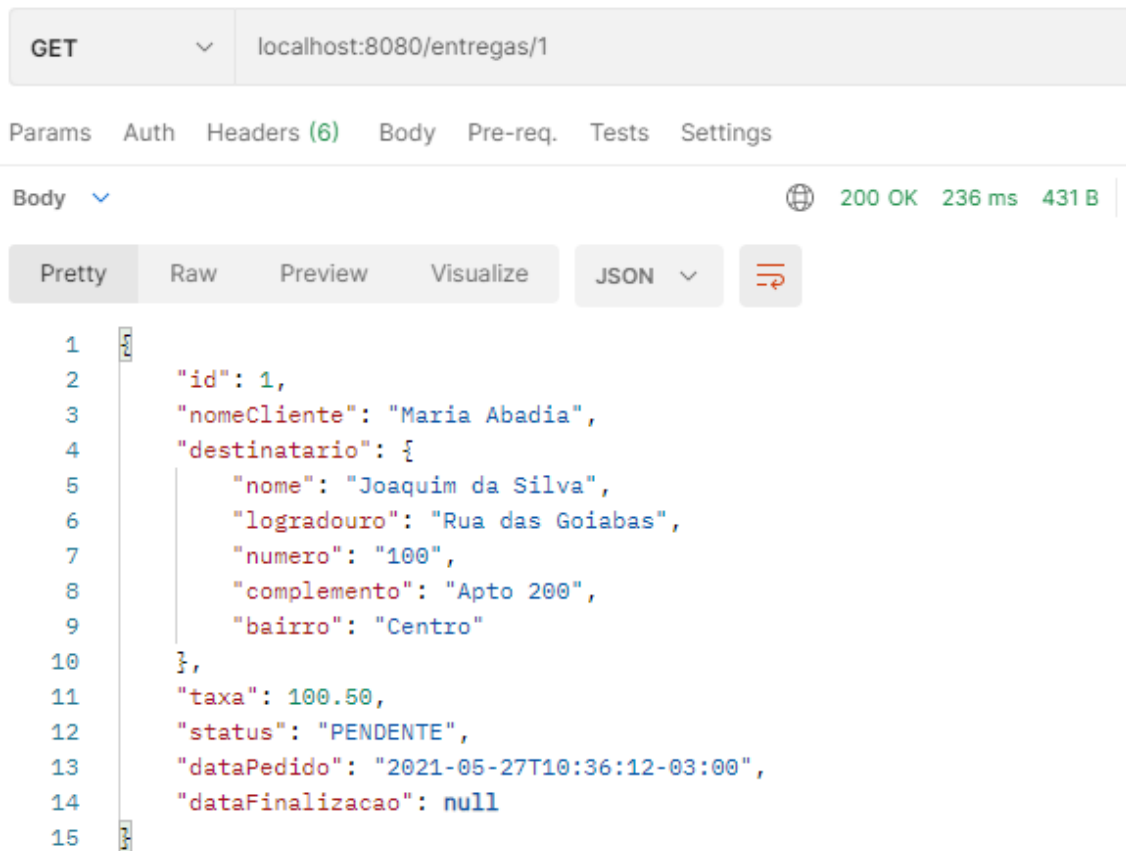


Figura 89 - GET request da entrega 1 após uso do ModelMapper. A resposta veio representada pelo novo modelo de representação.

Como o ModelMapper faz isso? Como ele conseguiu entender que nomeCliente é a propriedade nome dentro da propriedade cliente da entrega? O ModelMapper tem estratégias de correspondência de propriedades e normalmente consegue fazer o match mesmo que ela esteja em uma estrutura diferente.

A documentação do ModelMapper está disponível em seu site e, lá, é possível consultar o funcionamento de cada estratégia.

Como de costume, não queremos poluir o nosso Controller com operações que não sejam relacionadas ao manuseio de requisições HTTP. Por isso, vamos criar uma nova classe para lidar com essa transformação de Models. Em `src/main/java/com.algaworks.algalog`, criamos nossa classe ModelMapperConfig:

- PACKAGE -> `com.algaworks.algalog.api.assembler`
- NAME -> `EntregaAssembler`

```
@AllArgsConstructor
```

```
@Component
```

```
public class EntregaAssembler {
```

```
    private ModelMapper modelMapper;
```

```

    public EntregaModel toModel(Entrega entrega) {
        return modelMapper.map(entrega, EntregaModel.class);
    }
}

```

Anotamos o EntregaAssembler como @Component, marcando-o para detecção como componente do Spring e injetamos o ModelMapper dentro do EntregaAssembler.

```

public class EntregaAssembler {

    private ModelMapper modelMapper;

    public EntregaModel toModel(Entrega entrega) {
        return modelMapper.map(entrega, EntregaModel.class);
    }
}

```

Agora, refatoramos a função buscar() em nosso Controller para que ela retorne uma ResponseEntity de EntregaModel:

```

@GetMapping("/{entregaId}")
public ResponseEntity<EntregaModel> buscar(@PathVariable Long
entregaId) {
    return entregaRepository.findById(entregaId)
        .map(entrega ->
ResponseEntity.ok(entregaAssembler.toModel(entrega)))
        .orElse(ResponseEntity.notFound().build());
}

```

Além disso, queremos que o retorno da nossa POST request também seja representada pelo representation model, portanto, o refatoramos também:

```

@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public EntregaModel solicitar(@Valid @RequestBody Entrega entrega) {
    return
entregaAssembler.toModel(solicitacaoEntregaService.solicitar(entrega));
}

```

Por fim, precisamos refatorar o método listar(), que retorna uma lista de todas as entregas. Nesse caso, ao invés de passar um recurso único para conversão, estamos passando uma coleção de recursos, e precisamos implementar um novo método em nosso EntregaAssembler para lidar com isso:

```

public List<EntregaModel> toCollectionModel(List<Entrega> entregas) {
    return entregas.stream()
        .map(this::toModel)
        .collect(Collectors.toList());
}

```

Destrinchando:

- Na função toCollectionModel
  - Recebemos entregas, uma List<Entrega> como parâmetro
  - Retornamos uma List<EntregaModel>
- Geramos uma .stream() (fluxo de dados) de entregas.

- Cada item desse fluxo será convertido pelo método `toModel` do `EntregaAssembler`
- Ao final, todos os itens serão recolhidos em uma coleção e incluídos em uma `List<EntregaModel>` pelo método `.collect(Collectors.toList())`.

Vamos testar os nossos métodos agora:

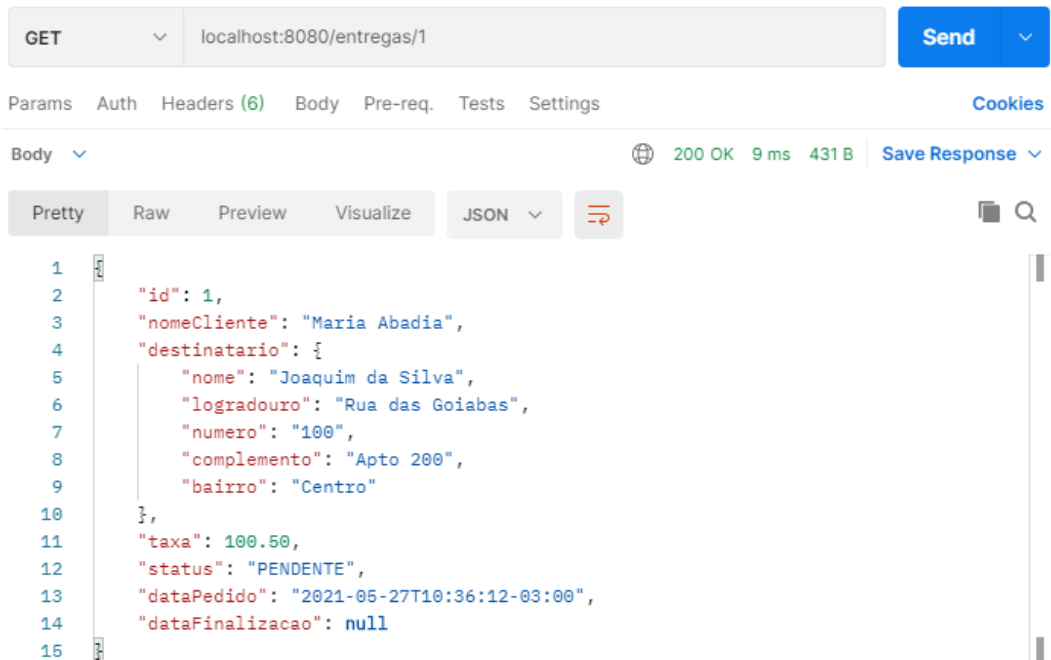


Figura 90 - GET request de entrega única. ModelMapper e EntregaAssembler funcionando.

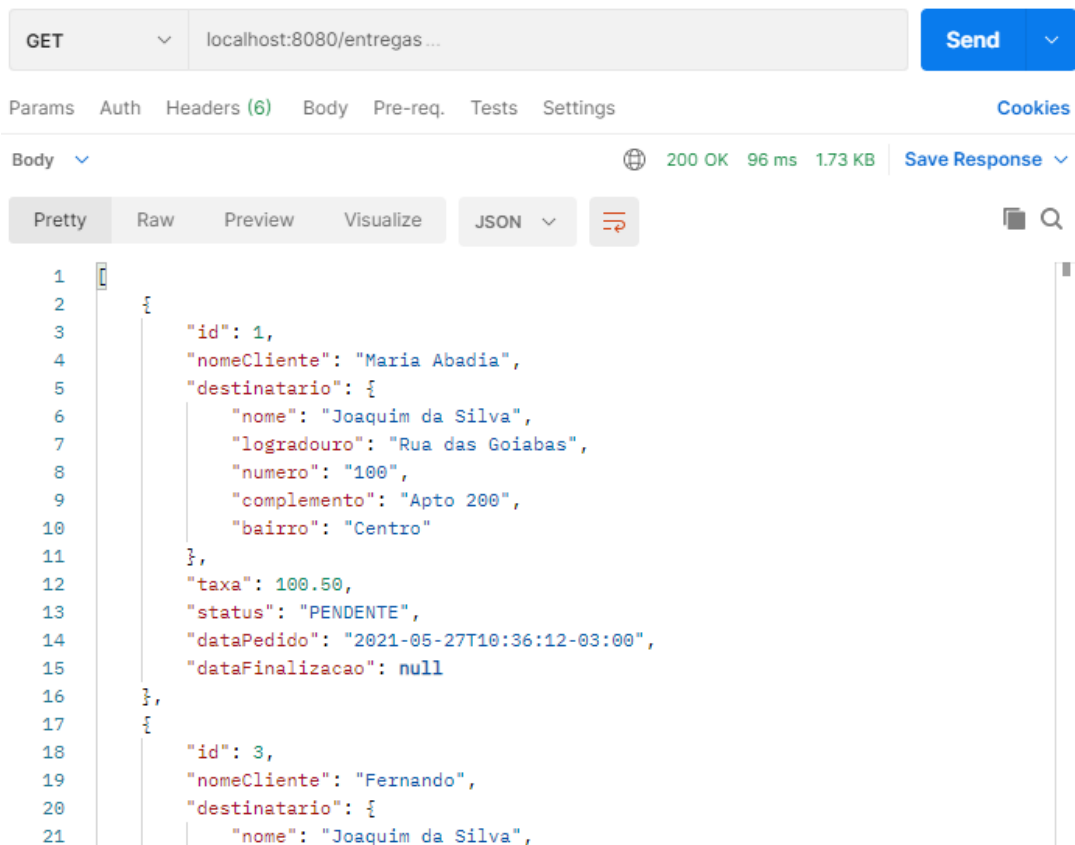


Figura 91 - GET request da listagem de entregas. ModelMapper e EntregaAssembler funcionando. Repare que recebemos o `nomeCliente` em vez de todas as informações do cliente.

POST localhost:8080/entregas...

Params Auth Headers (8) Body ● Pre-req. Tests Settings Cookies

raw JSON Beautify

```

1 {
2   .... "cliente": {
3     .... "id": 3
4   },
5   .... "destinatario": {
6     .... "nome": "Joaquim da Silva",
7     .... "logradouro": "Rua das Goiabas",
8     .... "numero": "100",
9     .... "complemento": "Apto 200",
10    .... "bairro": "Centro"
11  },
12  .... "taxa": 100.50
13 }

```

Body 201 Created 111 ms 440 B Save Response

Pretty Raw Preview Visualize JSON

```

1 {
2   "id": 7,
3   "nomeCliente": "Fernando",
4   "destinatario": {
5     "nome": "Joaquim da Silva",
6     "logradouro": "Rua das Goiabas",
7     "numero": "100",
8     "complemento": "Apto 200",
9     "bairro": "Centro"
10  },
11  "taxa": 100.50,
12  "status": "PENDENTE",
13  "dataPedido": "2021-05-29T11:05:09.3866298-03:00",
14  "dataFinalizacao": null
15 }

```

Figura 92 - POST request pós ModelMapper e com aplicação do EntregaAssembler. Recebemos nomeCliente.

Feito isso, podemos customizar livremente tanto o Domain Model quanto o Representation Model sem que um automaticamente influencie no outro. Vamos supor que queremos agora que, ao invés do nomeCliente, nosso Representation Model tenha uma propriedade cliente com id e nome. Como implementar?

Podemos criar um Representation Model em com.algaworks.algalog.api.model para cliente:

```

@Getter
@Setter
public class ClienteResumoModel {

    private Long id;
    private String nome;
}

```

E em nosso EntregaModel, referenciamos o ClienteResumoModel:

```

@Getter
@Setter
public class EntregaModel {

    private Long id;
    //private String nomeCliente;
    private ClienteResumoModel cliente;
    private DestinatarioModel destinatario;
    private BigDecimal taxa;
    private StatusEntrega status;
    private OffsetDateTime dataPedido;
    private OffsetDateTime dataFinalizacao;

}

```

Testando uma requisição:

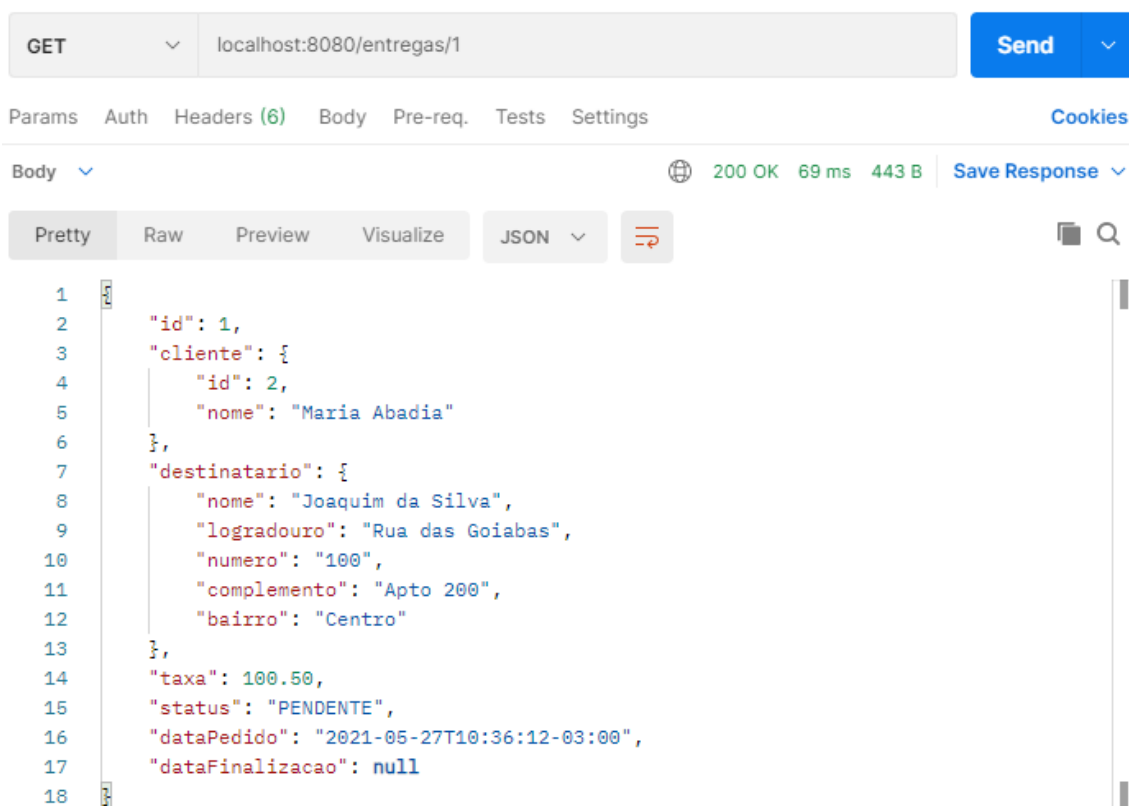


Figura 93 - Nova representação de cliente dentro de uma entrega.

Veja como foi fácil alterar a representação da nossa resposta sem sequer mexermos na classe `Cliente`. Muito bom!

Em relação aos nossos Models, ainda há algo que pode ser melhorado. Já temos dois Models: um para entrada/representação da entidade no nosso banco de dados (Domain Model), e outro para representação da resposta da API (Representation Model).

Acontece que se ainda dividimos o mesmo Model para entrada de dados e representação da entidade do BD. Isso já se mostrou um problema, quando vimos que o consumidor da API conseguia passar parâmetros sensíveis no corpo da requisição, como `status`, `dataPedido` e `dataFinalizacao`. Resolvemos o problema adicionando as anotações `@JsonProperty(access = Access.READ_ONLY)`, mas será que isso é resolver mesmo?

E se precisarmos adicionar mais propriedades que não queremos que o consumidor da API não deve poder manipular no input, precisaremos anotar cada uma delas como `@JsonProperty(access = Access.READ_ONLY)`? Isso não é tão prático.

O que podemos fazer é criar um terceiro Model, dedicado ao input de dados do consumidor da API. Em `src/main/java/com.algaworks.algalog.api`, criamos nossa classe `EntregaInput`:

- PACKAGE -> `com.algaworks.algalog.api.model.input`
- NAME -> `EntregaInput`

```
@Getter
@Setter
public class EntregaInput {

    @Valid
    @NotNull
    private ClienteIdInput cliente;

    @Valid
    @NotNull
    private DestinatarioInput destinatario;

    @NotNull
    private BigDecimal taxa;

}
```

A classe `EntregaInput` é bem mais limpa do que `Entrega` e `EntregaModel`, e só contém as propriedades que serão fornecidas pelo consumidor da API. Vejamos as classes `Input` derivadas que também criamos, `ClienteIdInput` e `DestinatarioInput`:

```
@Getter
@Setter
public class ClienteIdInput {

    @NotNull
    private Long id;

}

@Getter
@Setter
public class DestinatarioInput {

    @NotBlank
    private String nome;

    @NotBlank
    private String logradouro;

    @NotBlank
    private String numero;

    @NotBlank
    private String complemento;

    @NotBlank
    private String bairro;

}
```

```
}
```

Tanto o `ClientInput` quanto o `DestinatarioInput` também só possuem as propriedades condizentes com o que o consumidor da API deve fornecer no corpo da sua request.

Para aproveitarmos as vantagens trazidas pelo nosso modelo de Input, o `EntregaInput`, se faz necessário criar mais um método de conversão de tipos, agora convertendo o objeto da classe `EntregaInput` em um objeto da classe `Entrega`.

```
public Entrega toEntity(EntregaInput entregaInput) {  
    return modelMapper.map(entregaInput, Entrega.class);  
}
```

E o nosso `EntregaController`, como fica agora?

```
@PostMapping  
@ResponseStatus(HttpStatus.CREATED)  
public EntregaModel solicitar(@Valid @RequestBody EntregaInput  
entregaInput) {  
    Entrega novaEntrega = entregaAssembler.toEntity(entregaInput);  
    Entrega entregaSolicitada =  
solicitacaoEntregaService.solicitar(novaEntrega);  
    return entregaAssembler.toModel(entregaSolicitada);  
}
```

Com a chegada da nova classe `EntregaInput`, só modificamos o método POST, que é o método que receberá um corpo em sua requisição.

- Recebemos um objeto da classe `EntregaInput` como parâmetro.
- O convertemos para um objeto da classe `Entrega` e realizamos a transação no banco de dados.
- Convertemos o objeto da classe `Entrega` para um objeto `EntregaModel` e retornamos na resposta da API.

Depois disso tudo, vamos criar uma POST request e percebemos que, de acordo com a figura 94, está tudo funcionando. Ótimo!

Uma dúvida que pode surgir é se precisamos validar os dados tanto no `InputModel` quanto no `Domain Model`, e isso depende.

Caso a manipulação de dados seja feita apenas pela API, a validação na entrada é suficiente.

Entretanto, se nossos dados são manipulados de alguma outra forma, a validação no `Domain Model` pode ser importante para manter a consistência de dados antes que eles sejam persistidos em nosso banco de dados.

POST localhost:8080/entregas ... Send

Params Auth Headers (8) **Body** Pre-req. Tests Settings Cookies

raw **JSON** Beautify

```

1 {
2   ... "cliente": {
3     ... "id": 3
4   },
5   ... "destinatario": {
6     ... "nome": "Joaquim da Silva",
7     ... "logradouro": "Rua das Goiabas",
8     ... "numero": "100",
9     ... "complemento": "Apto 200",
10    ... "bairro": "Centro"
11  },
12  ... "taxa": 100.50
13 }

```

Body 201 Created 114 ms 452 B Save Response

Pretty Raw Preview Visualize **JSON** 🔍

```

1 {
2   "id": 8,
3   "cliente": {
4     "id": 3,
5     "nome": "Fernando"
6   },
7   "destinatario": {
8     "nome": "Joaquim da Silva",
9     "logradouro": "Rua das Goiabas",
10    "numero": "100",
11    "complemento": "Apto 200",
12    "bairro": "Centro"
13  },
14   "taxa": 100.50,
15   "status": "PENDENTE",
16   "dataPedido": "2021-05-29T11:53:09.2445416-03:00",
17   "dataFinalizacao": null
18 }

```

Figura 94 - POST request após todas as conversões entre Models. Tudo funcionando.

Considerando que a nossa API é autocontida, e que não há nenhum manuseio externo de dados, podemos enxugar a validação no nosso Domain Model Entrega:

```

@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Entrega {

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

```



```

@ManyToOne
private Cliente cliente;

@Embedded
private Destinatario destinatario;

private BigDecimal taxa;

@Enumerated(EnumType.STRING)
private StatusEntrega status;

private OffsetDateTime dataPedido;

private OffsetDateTime dataFinalizacao;
}

```

E aí, podemos até remover o ValidationGroup da Entrega e do Cliente, que ficaria assim:

```

@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Getter
@Setter
@Entity
public class Cliente {

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

    @NotBlank
    @Size(max = 60)
    private String nome;

    @NotBlank
    @Email
    @Size(max = 255)
    private String email;

    @NotBlank
    @Size(max = 20)
    @Column(name="fone")
    private String telefone;
}

```

Lembrando que a nossa classe Cliente ainda está sendo utilizada como Domain e Representation Model. O ideal seria criar um Representation Model para Cliente também, e, no geral, para qualquer classe do sistema, mas evitaremos esse trabalho repetitivo nesta aula.

### 3.6. Implementando sub recursos

Agora, vamos implementar o endpoint /ocorrencias. Uma entrega pode ter de 0 a n ocorrências, e cada ocorrência está vinculada a apenas uma entrega. Ocorrências de uma entrega são registros de entregas mal sucedidas ou bem-sucedidas.

Em src/main/java/com.algaworks.algalog.domain.model, criamos nossa classe Ocorrencia:

- PACKAGE -> com.algaworks.algalog.domain.model
- NAME -> Ocorrencia

```
@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Ocorrencia {

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

    @ManyToOne
    private Entrega entrega;

    private String descricao;
    private OffsetDateTime dataRegistro;

}
```

Nesse caso, precisamos criar o mapeamento bidirecional, anotando a nossa classe Entrega com uma nova anotação, @OneToMany.

```
@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Entrega {

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

    @ManyToOne
    private Cliente cliente;

    @Embedded
    private Destinatario destinatario;

    private BigDecimal taxa;

    @OneToMany(mappedBy = "entrega")
    private List<Ocorrencia> ocorrencias = new ArrayList<>();

    @Enumerated(EnumType.STRING)
    private StatusEntrega status;

    private OffsetDateTime dataPedido;

    private OffsetDateTime dataFinalizacao;

}
```

Na classe Entrega, criamos uma lista de ocorrências e a anotamos com o @OneToMany. O argumento mappedBy = “entrega” aponta para a propriedade que contém o @ManyToOne no outro objeto mapeado pela relação bidirecional. Nesse caso, a propriedade entrega.

Agora, vamos criar nossa tabela ocorrencia no banco de dados. Criamos nosso DDL diretamente no MySQLWorkbench para verificar se ele está correto antes de criar a migration.

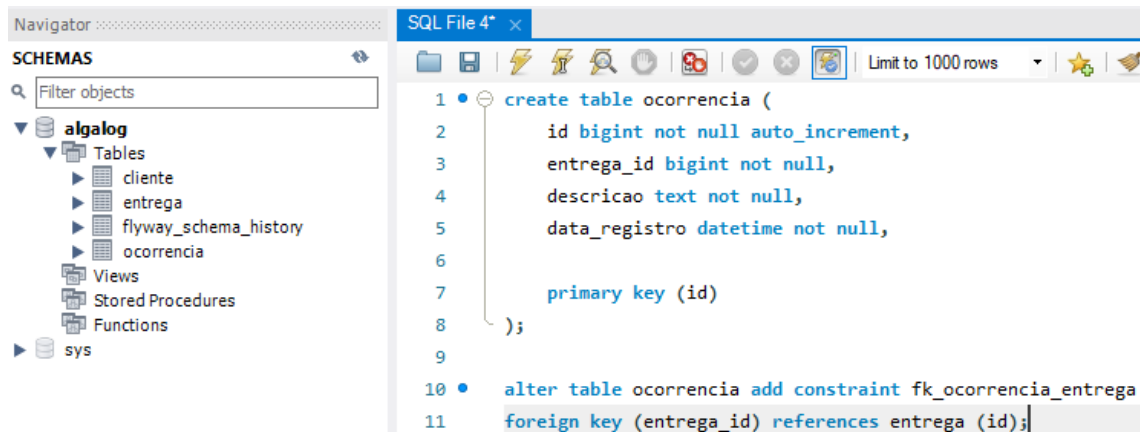


Figura 95 - Criação bem-sucedida da tabela ocorrencia.

Uma vez criada a tabela, podemos derrubá-la e transferir o código de criação dela para o nosso arquivo de migration. Sendo assim, movemos esse código para o arquivo criado em src/main/resources/db, V004\_\_cria-tabela-ocorrencia, e rodamos nosso servidor para vermos que a tabela foi criada e seu registro foi adicionado no flyway\_schema\_history.

```
1 • SELECT * FROM algalog.flyway_schema_history;
```

installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success
1	001	cria-tabela-cliente	SQL	V001__cria-tabela-cliente.sql	-1506170770	root	2021-05-19 19:24:18	24	1
2	002	renomeia-coluna-telefone	SQL	V002__renomeia-coluna-telefone.sql	392309845	root	2021-05-19 19:33:10	26	1
3	003	cria-tabela-entrega	SQL	V003__cria-tabela-entrega.sql	1041010867	root	2021-05-27 10:06:57	81	1
4	004	cria-tabela-ocorrencia	SQL	V004__cria-tabela-ocorrencia.sql	462146062	root	2021-05-29 15:21:33	86	1

Figura 96 - Tabela flyway\_schema\_history após a criação da última migration, a V004.

O próximo passo é criar nossa classe Service para o registro de ocorrências de entrega.

```

@AllArgsConstructor
@Service
public class RegistroOcorrenciaService {

    private EntregaRepository entregaRepository;

    @Transactional
    public Ocorrencia registrar(Long entregaId, String descricao) {
        Entrega entrega = entregaRepository.findById(entregaId)
            .orElseThrow(() -> new NegocioException("Entrega
não encontrada."));

        return entrega.adicionarOcorrencia(descricao);
    }
}

```

Destrinchando:

- Registraremos uma ocorrência que contém um corpo e que está vinculada a uma entrega pelo id da entrega
  - Verificamos se tal entrega existe
  - Se não existir, retornamos uma exceção
- Se a entrega existir, adicionamos uma ocorrência a ela.

Criaremos, agora, o método `adicionarOcorrencia()` dentro do nosso objeto `Entrega`.

```
public Ocorrencia adicionarOcorrencia(String descricao) {
    Ocorrencia ocorrencia = new Ocorrencia();
    ocorrencia.setDescricao(descricao);
    ocorrencia.setDataRegistro(OffsetDateTime.now());
    ocorrencia.setEntrega(this);

    this.getOcorrencias().add(ocorrencia);

    return ocorrencia;
}
```

Esse método:

- Recebe a descrição (corpo) da ocorrência
- Instancia um novo objeto da classe `Ocorrencia` e preenche o objeto com:
  - a descrição recebida como parâmetro
  - coloca um timestamp da hora atual como `dataRegistro`
  - se vincula à entrega atual.
- Adiciona a nova `Ocorrencia` à entrega atual.

**Algo que pode ser reparado é que não há uma classe `OcorrenciaRepository` para salvar as ocorrências. Por quê?** O Jakarta Persistence automaticamente salva as operações quando temos uma transação aberta. Ou seja, quando abrimos a transação para buscar a entrega, as alterações feitas durante a execução do método são salvas automaticamente.

**E quais foram as alterações feitas durante a execução do método?** A criação de novas ocorrências. Já que estamos criando-as e salvando-as dentro da nossa entrega, o JPA automaticamente povoa a tabela `ocorrencia` também, justamente devido ao relacionamento entre elas.

Podemos nos adiantar e separar o método de procurar entregas criados ali em cima em outra classe `Service`, por comodidade e reusabilidade:

```
@AllArgsConstructor
@Service
public class BuscaEntregaService {

    private EntregaRepository entregaRepository;

    public Entrega buscar(Long entregaId) {
        return entregaRepository.findById(entregaId)
            .orElseThrow(() -> new NegocioException("Entrega
não encontrada."));
    }

}
```

E o nosso `RegistroOcorrenciaService` refatorado fica assim:

```

@AllArgsConstructor
@Service
public class RegistroOcorrenciaService {

    private BuscaEntregaService buscaEntregaService;

    @Transactional
    public Ocorrencia registrar(Long entregaId, String descricao) {
        Entrega entrega = buscaEntregaService.buscar(entregaId);

        return entrega.adicionarOcorrencia(descricao);
    }
}

```

Podemos definir ocorrências como um próprio endpoint, ou podemos defini-las como sub recurso do endpoint entregas. Optaremos por definir ocorrências como sub recurso de entregas, pois há uma relação de dependência fortíssima entre elas: não existem ocorrências se não existirem entregas.

Vamos definir nossos novos Models (Input e Representation) para Ocorrencia agora.

```

@Getter
@Setter
public class OcorrenciaModel {

    private Long id;
    private String descricao;
    private OffsetDateTime dataRegistro;

}

@Getter
@Setter
public class OcorrenciaInput {

    @NotBlank
    private String descricao;

}

```

E criar o OcorrenciaController:

```

@AllArgsConstructor
@RestController
@RequestMapping("/entregas/{entregaId}/ocorrencias")
public class OcorrenciaController {

    private RegistroOcorrenciaService registroOcorrenciaService;
    private OcorrenciaAssembler ocorrenciaAssembler;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public OcorrenciaModel registrar(@PathVariable Long entregaId,
                                     @Valid @RequestBody OcorrenciaInput ocorrenciaInput) {

        Ocorrencia ocorrenciaRegistrada =
registroOcorrenciaService.registrar(entregaId,
ocorrenciaInput.getDescricao());
    }
}

```

```

        return ocorrenciaAssembler.toModel(ocorrenciaRegistrada);
    }
}

```

A criação do Controller é igual a dos Controllers anteriores, a diferença é que, agora, nossa URI, “/entregas/{entregaId}/ocorrencias” é marcada como um sub recurso de entregas.

Serializamos o corpo da requisição em um objeto do tipo OcorrenciaInput com a anotação @RequestBody, validamos esse corpo com a anotação @Valid e recebemos o entregaId pela @PathVariable.

Utilizamos nosso registroOcorrenciaService para buscar a entrega por Id por meio do buscaEntregaService e, em seguida, registrar a Ocorrencia em nosso BD e retornar um objeto da classe Ocorrencia.

Então, a Ocorrencia registrada é convertida pelo ocorrenciaAssembler, de Ocorrencia para OcorrenciaModel:

```

@AllArgsConstructor
@Component
public class OcorrenciaAssembler {

    private IMapper modelMapper;

    public OcorrenciaModel toModel(Ocorrencia ocorrencia) {

        return modelMapper.map(ocorrencia, OcorrenciaModel.class);
    }
}

```

Para, assim, ser retornada a OcorrenciaModel como resposta da API. Vamos testar:

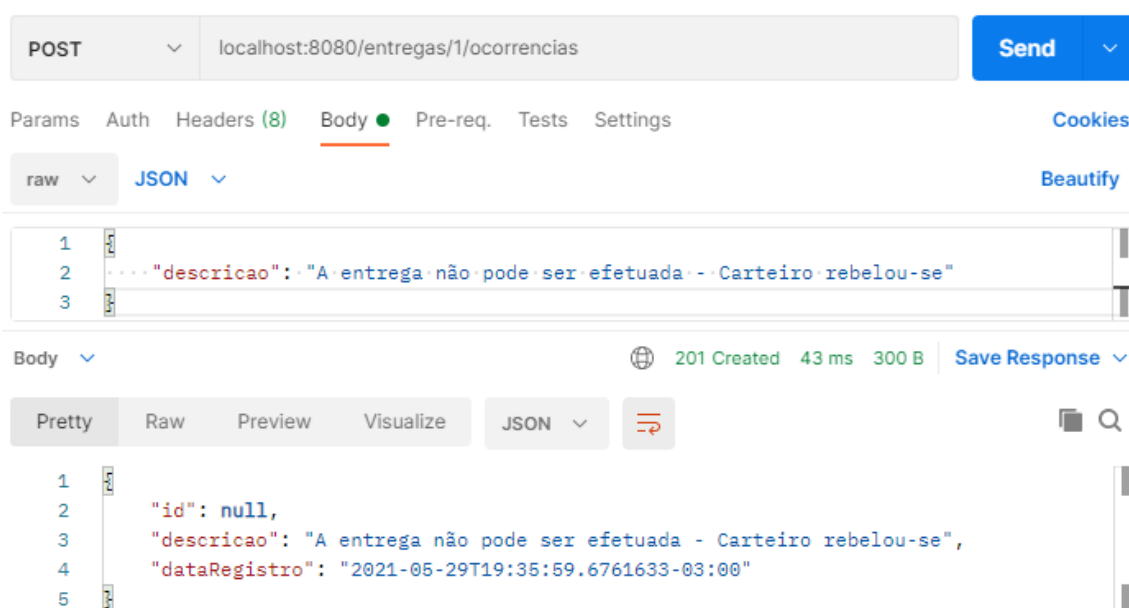


Figura 97 - Falha ao salvar um registro de ocorrência. Por quê?

Acontece que o Spring JPA ainda não consegue salvar itens por cascadeamento dentro de uma mesma transação, como mencionamos. Ainda precisamos adicionar uma anotação no relacionamento das entregas com as ocorrências.

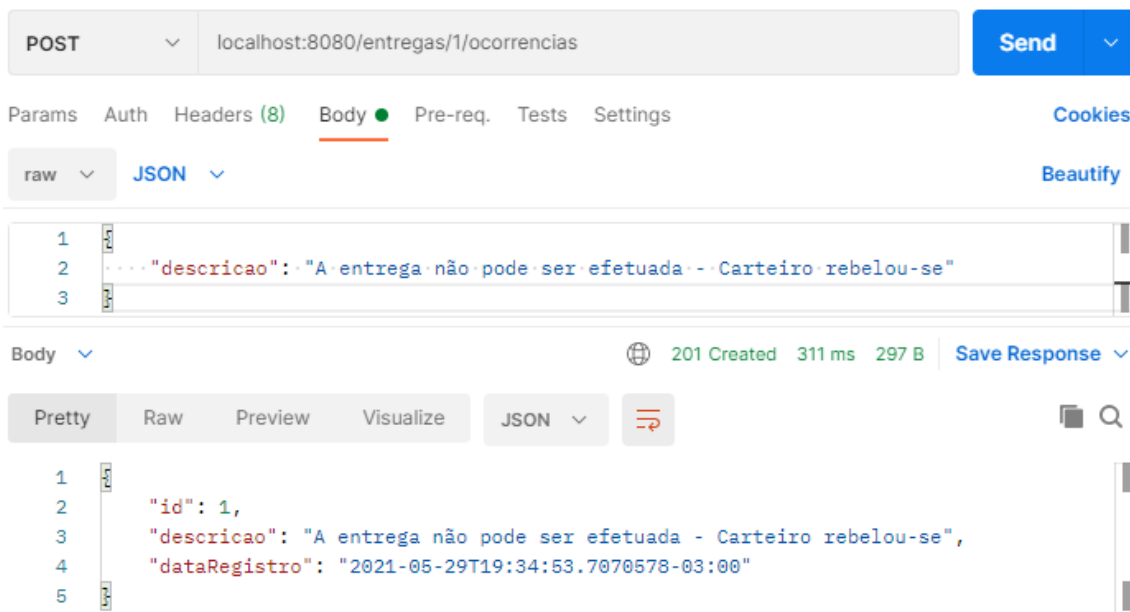
Em nossa classe Entrega alteramos

```
@OneToMany(mappedBy = "entrega")
private List<Ocorrencia> ocorrencias = new ArrayList<>();
```

Para

```
@OneToMany(mappedBy = "entrega", cascade = CascadeType.ALL)
private List<Ocorrencia> ocorrencias = new ArrayList<>();
```

Vamos retestar a nossa POST request:



POST localhost:8080/entregas/1/ocorrencias

Params Auth Headers (8) Body Pre-req. Tests Settings Cookies

raw JSON Beautify

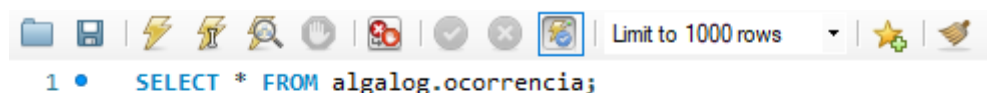
```
1 {
2   "descricao": "A entrega não pode ser efetuada - Carteiro rebelou-se"
3 }
```

Body 201 Created 311 ms 297 B Save Response

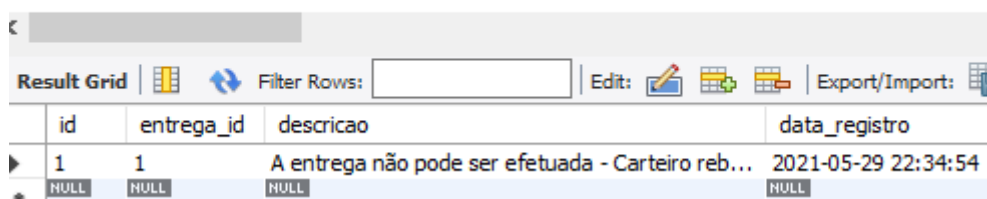
Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "descricao": "A entrega não pode ser efetuada - Carteiro rebelou-se",
4   "dataRegistro": "2021-05-29T19:34:53.7070578-03:00"
5 }
```

Figura 98 - Registro efetuado em cascata. Adicionamos uma ocorrência pela entrega e ela foi salva na tabela ocorrência.



1 • SELECT \* FROM algalog.ocorrencia;



	id	entrega_id	descricao	data_registro
▶	1	1	A entrega não pode ser efetuada - Carteiro reb...	2021-05-29 22:34:54
*	NULL	NULL	NULL	NULL

Figura 99 - ocorrencia salva no banco de dados.

E se tentarmos criar uma ocorrência para uma entrega inexistente?

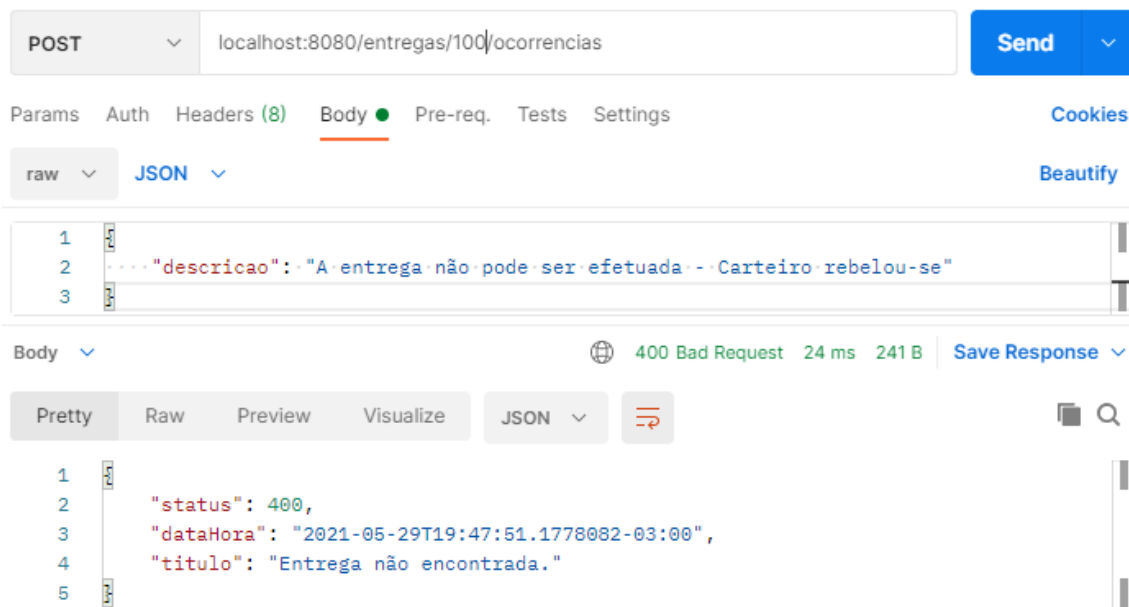


Figura 100 - POST request com entregald inexistente.

Nesse caso, o recurso entrega não existe. A melhor prática aqui, é retornar um erro 404 NOT FOUND ao invés de um erro 400 BAD REQUEST.

O erro 400 BAD REQUEST é proveniente da nossa classe `NegocioException`, e não queremos mudá-la. Ao invés de adicionar uma condição de uso à nossa `NegocioException`, vamos criar uma outra classe de Exception mais específica, a `EntidadeNaoEncontradaException`:

```
public class EntidadeNaoEncontradaException extends NegocioException{

    private static final long serialVersionUID = 1L;

    public EntidadeNaoEncontradaException(String message) {
        super(message);
        // TODO Auto-generated constructor stub
    }

}
```

Essa nova Exception irá estender e construir uma `NegocioException` e utilizá-la para fins mais específicos. Alteramos, agora, nosso `BuscaEntregaService`:

```
@AllArgsConstructor
@Service
public class BuscaEntregaService {

    private EntregaRepository entregaRepository;

    public Entrega buscar(Long entregaId) {
        return entregaRepository.findById(entregaId)
            .orElseThrow(() -> new
EntidadeNaoEncontradaException("Entrega não encontrada."));
    }

}
```



Sendo que isso ainda não muda nada também. A lógica por trás do `NegocioException` está em nosso `ApiExceptionHandler`, e é lá que iremos criar a lógica por trás da nova exceção, `EntidadeNaoEncontradaException`.

```
@ExceptionHandler(NegocioException.class)
public ResponseEntity<Object> handleNegocio(NegocioException ex,
WebRequest request) {

    HttpStatus status = HttpStatus.BAD_REQUEST;

    Problema problema = new Problema();
    problema.setStatus(status.value());
    problema.setDataHora(OffsetDateTime.now());
    problema.setTitulo(ex.getMessage());

    return handleExceptionInternal(ex, problema, new HttpHeaders(),
status, request);
}

@ExceptionHandler(EntidadeNaoEncontradaException.class)
public ResponseEntity<Object>
handleEntidadeNaoEncontrada(EntidadeNaoEncontradaException ex, WebRequest
request) {

    HttpStatus status = HttpStatus.NOT_FOUND;

    Problema problema = new Problema();
    problema.setStatus(status.value());
    problema.setDataHora(OffsetDateTime.now());
    problema.setTitulo(ex.getMessage());

    return handleExceptionInternal(ex, problema, new HttpHeaders(),
status, request);
}
```

Basicamente, nós duplicamos a `NegocioException` e adaptamos para `EntidadeNaoEncontrada`. Além de mudarmos o nome da exceção, alteramos também o status HTTP retornado, de `BAD_REQUEST` para `NOT_FOUND`.

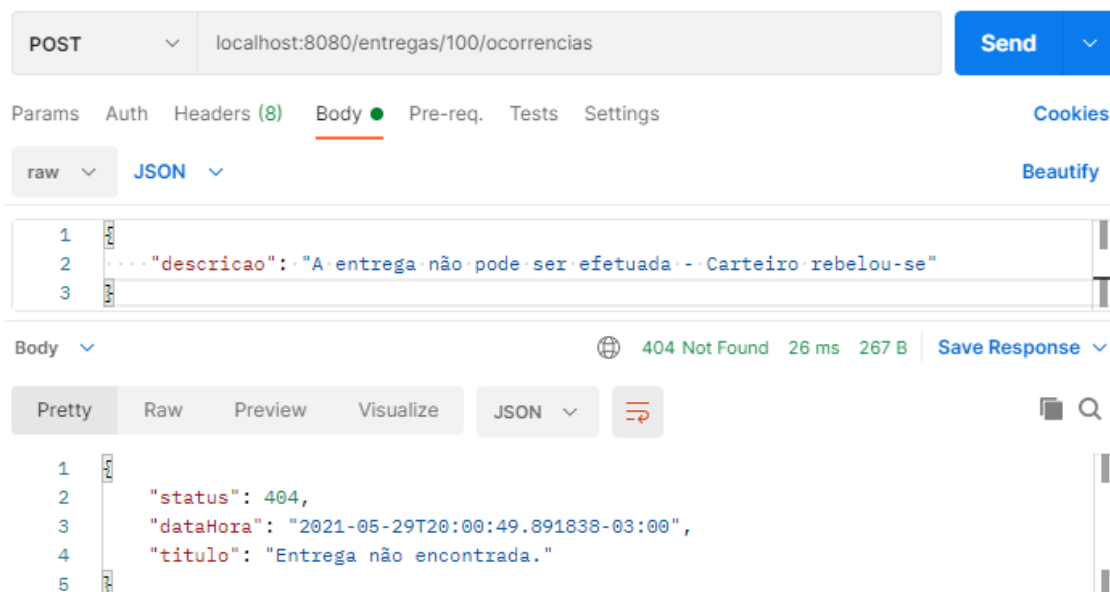


Figura 101 - Resposta de POST request com entregald inexistente. Agora, retornamos HTTP 404.

Agora, vamos criar um método de listagem de ocorrências para uma determinada entrega.

```
@GetMapping
public List<OcorrenciaModel> listar(@PathVariable Long entregaId) {
    Entrega entrega = buscaEntregaService.buscar(entregaId);

    return
    ocorrenciaAssembler.toCollectionModel(entrega.getOcorrencias());
}
```

O objeto entrega encontrado pelo entregaId possui uma List<Ocorrencia>, entretanto, queremos retornar uma List<OcorrenciaModel>. Por isso, precisamos criar o método de conversão de listas em nosso OcorrenciaAssembler.

```
public List<OcorrenciaModel> toCollectionModel(List<Ocorrencia>
ocorrencias) {
    return
    ocorrencias.stream().map(this::toModel).collect(Collectors.toList());
}
```

E agora injetamos o OcorrenciaAssembler no OcorrenciaController:

```
private OcorrenciaAssembler ocorrenciaAssembler;
```

Feito tudo isso, agora conseguimos listar todas as ocorrências de uma entrega:



Figura 102 - GET request para listagem de ocorrências de uma entrega válida.

E se buscarmos uma entrega inexistente para listar suas ocorrências, recebemos um HTTP 404.

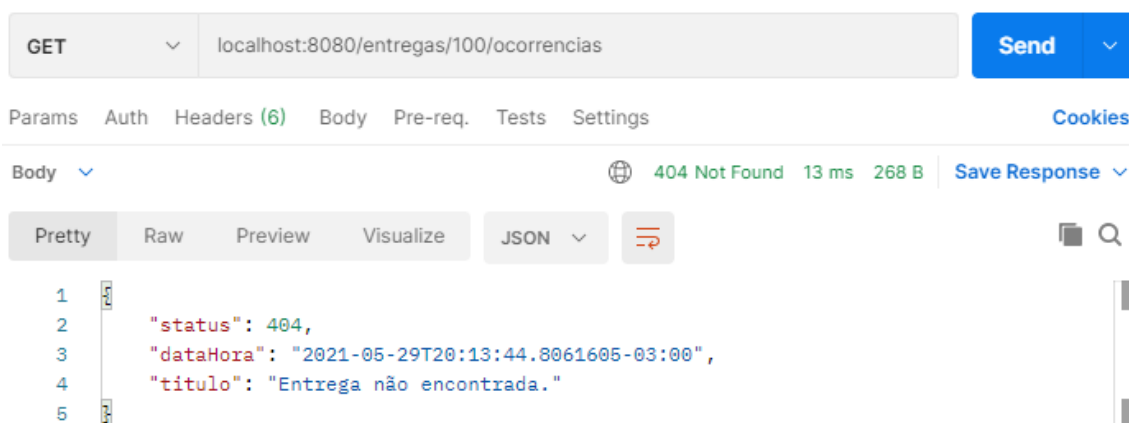


Figura 103 - GET request de listagem de ocorrências de uma entrega inexistente.

### 3.7. Implementando ação não CRUD

Por fim, vamos implementar o processo de negócio de finalizar uma entrega: uma ação não CRUD.

Mas o Update faz parte do CRUD, não?

Queremos fazer a transição de status de entrega de PENDENTE para FINALIZADA.

Para isso, vamos criar mais um Service, o FinalizacaoEntregaService:

```
@AllArgsConstructor
@Service
public class FinalizacaoEntregaService {

    private EntregaRepository entregaRepository;
    private BuscaEntregaService buscaEntregaService;

    @Transactional
    public void finalizar(Long entregaId) {
        Entrega entrega = buscaEntregaService.buscar(entregaId);

        entrega.finalizar();

        entregaRepository.save(entrega);
    }
}
```

A princípio, queremos que o método entrega.finalizar() altere o status de PENDENTE para FINALIZADA, e podemos fazer isso nesse Service. Porém, é importante e saudável possuir alguns métodos de classe dentro de nossa própria entidade, de modo a lhe conferir alguma responsabilidade além de ser apenas uma estrutura de dados, e de tornar o código mais legível (e até reaproveitável) em outras seções da aplicação.

```
public void finalizar() {
    if (naoPodeSerFinalizada()) {
        throw new NegocioException("Entrega não pode ser
finalizada");
    }

    setStatus(StatusEntrega.FINALIZADA);
    setDataFinalizacao(OffsetDateTime.now());
}

public boolean podeSerFinalizada() {
    return StatusEntrega.PENDENTE.equals(getStatus());
}

public boolean naoPodeSerFinalizada() {
    return !podeSerFinalizada();
}
```

Perceba o foco em legibilidade de código. Esses 3 métodos poderiam ser fundidos em um único método menos legível, mas considerando que estamos atuando na nossa entidade, é uma boa prática manter subdividir a lógica de maneira legível. Além disso, todos esses métodos podem ser aproveitados onde há um objeto da classe Entrega sendo instanciado.

Finalmente, vamos modelar essa operação na nossa API. Mas espera, como faremos isso? Até então, estávamos utilizando os verbos HTTP em nosso CRUD, mas “finalizar” não se encaixa em nenhum deles.

O segredo é não ficar preso apenas às entidades do Domain Model. Não precisamos que todos os recursos da API possuam uma representação no Domain Model: podemos modelar conceitos abstratos de negócio como recursos da API, mesmo sem ter uma entidade ou uma tabela no banco de dados.

Poderíamos utilizar um PUT ou PATCH para alterar apenas o status da entrega. Teoricamente o verbo PATCH seria mais adequado para nós, que só queremos alterar uma propriedade da entrega, porém o método PATCH é bem mais limitado que o PUT, pois a requisição precisa de mais dados para executar o processo de negócio. Por isso, usaremos o verbo PUT.

```
@PutMapping("/{entregaId}/finalizacao")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void finalizar(@PathVariable Long entregaId) {
    finalizacaoEntregaService.finalizar(entregaId);
}
```

Nessa implementação, não estamos retornando nada, portanto usamos NO CONTENT 204.

Usamos PUT em vez de POST, pois o verbo PUT é idempotente. PUT requests sucessivas não alterarão mais nada em nosso banco de dados, enquanto POST requests podem causar várias alterações caso efetuadas em sucessão.

Bom, agora, vamos finalizar a entrega 1. A figura a seguir mostra a entrega 1 e seu status atual.

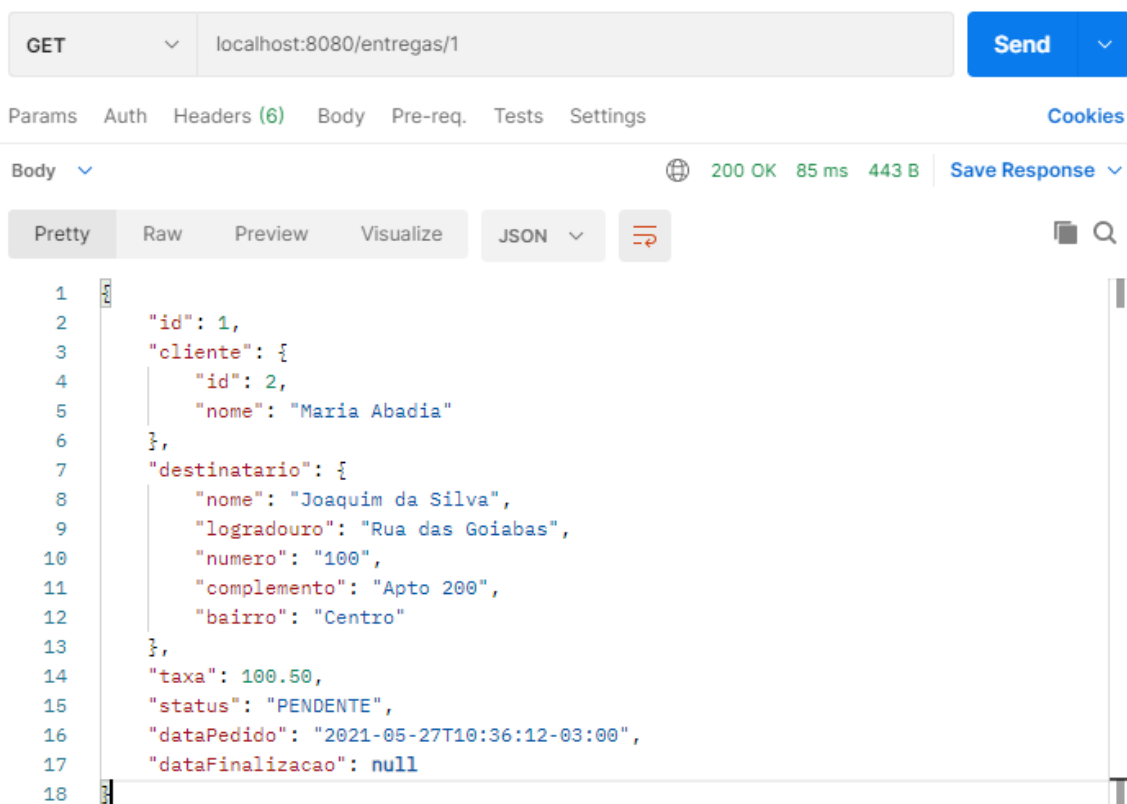


Figura 104 - Entrega 1. Atualmente, possui o status PENDENTE.

Agora, vamos finalizar a entrega em nossa API:

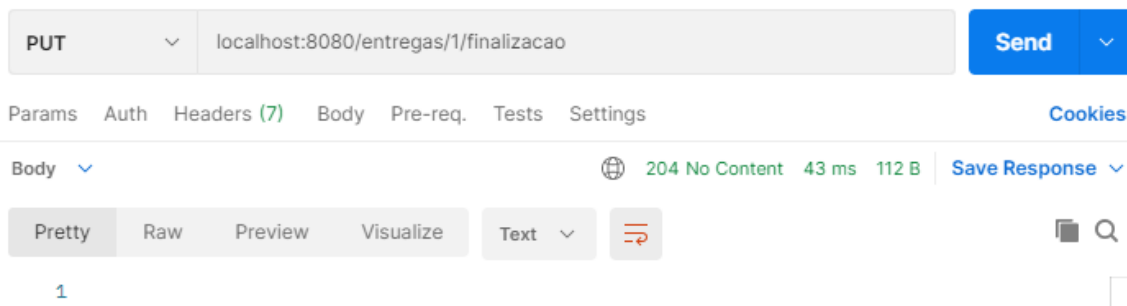


Figura 105 - Finalização da entrega 1 por PUT request feita com sucesso.

O corpo da resposta está vazio, o que é esperado. Sabemos que a entrega foi finalizada devido ao status 204 NO CONTENT.

Se repetirmos a mesma requisição:

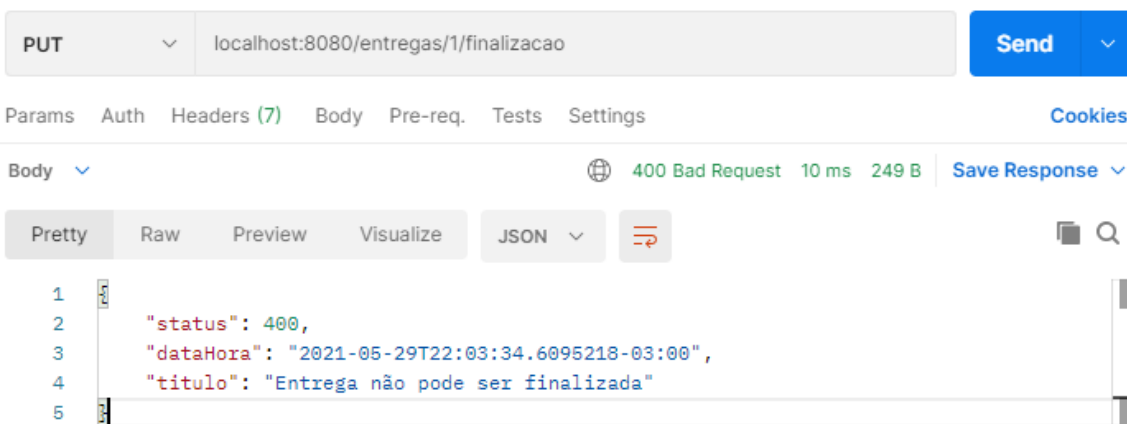


Figura 106 - Exceção acionada ao tentar finalizar uma entrega já finalizada.

Se tentarmos finalizar uma entrega que não existe, nosso ExceptionHandler também entra em ação:

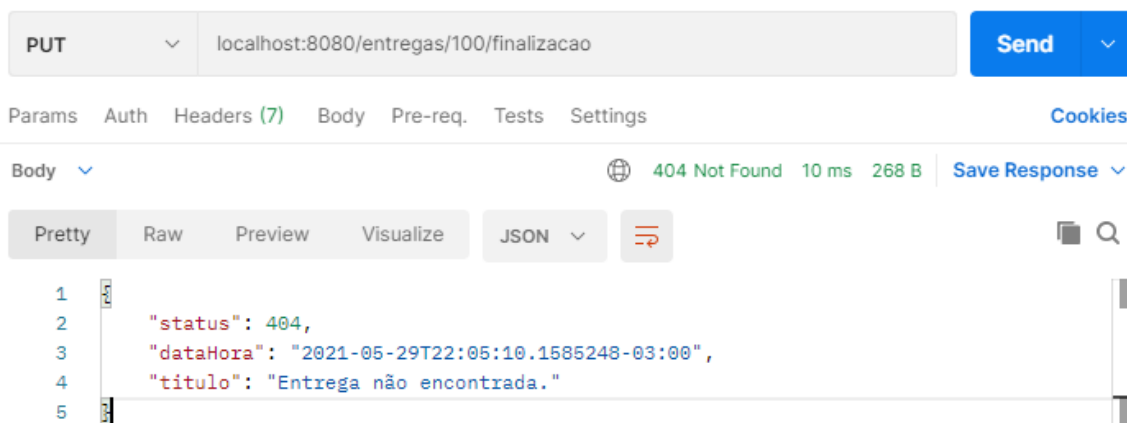


Figura 107 - Exceção acionada ao tentar finalizar uma entrega inexistente.

Finalmente, se conferirmos a entrega 1 agora, podemos ver se ela foi finalizada ou não:

GET localhost:8080/entregas/1 Send

Params Auth Headers (6) Body Pre-req. Tests Settings Cookies

Body 200 OK 13 ms 468 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "cliente": {
4     "id": 2,
5     "nome": "Maria Abadia"
6   },
7   "destinatario": {
8     "nome": "Joaquim da Silva",
9     "logradouro": "Rua das Goiabas",
10    "numero": "100",
11    "complemento": "Apto 200",
12    "bairro": "Centro"
13  },
14  "taxa": 100.50,
15  "status": "FINALIZADA",
16  "dataPedido": "2021-05-27T10:36:12-03:00",
17  "dataFinalizacao": "2021-05-29T22:01:29-03:00"
18 }
```

Figura 108 - Entrega finalizada com sucesso.