Criando uma API REST com o Spring Boot

Neste post vamos criar uma API Rest usando a plataforma Spring com Java, essa API será responsável pelo cadastro de pessoas na primeira fase de um processo de abertura de conta bancaria. Teremos uma breve introdução sobre API REST, Java e Spring e depois iremos ao passo a passo de como criar essa aplicação utilizando essas tecnologias. Vamos lá!

O que é uma API REST?

Uma **API** *(Application Programming Interface)* é uma interface, uma ponte que conecta diversos sistemas, aplicações que estão ou podem estar em contato em um determinado momento ou mesmo com bastante frequência na rede de computadores (web).

É de extrema importância, pois permite que aplicativos sejam criados, sistemas se comuniquem, por exemplo para que uma aplicação Spring seja criada utilizando o Java, é necessário que aplicações e sistemas estejam integrados para disponibilizar recursos para que esta nova aplicação seja criada, como em nosso caso que veremos neste post como uma API Rest com Spring Boot foi implantada e implementada.

Ao navegar no site de hotéis e escolher um destino para uma viagem, a interface do hotel se comunica com a interface da Google que disponibiliza o mapa indicando o caminho exato da viagem, há, portanto, uma comunicação graças as APIs que foram conectadas e acessaram a interface uma da outra e isso ocorre com frequência na rede.

Por que usar Java?

Se tratando de linguagem de programação, onde são desenvolvidos os códigos que representam as aplicações, o Java é uma das linguagens mais acessadas e utilizadas no mundo da tecnologia e tem “*n*” maneiras de desenvolver aplicações, pois encontramos uma gama de recursos disponibilizados nesta linguagem de programação.

Além disso o Java é uma linguagem que traz a característica de programação orientada a objetos, onde uma classe traz consigo suas variáveis e seus comportamentos/métodos, levando em consideração os aspectos da vida real para dentro do ambiente de desenvolvimento com a linguagem.

Por ser versátil e rodar em diversos sistemas operacionais é bastante utilizada pelas empresas de tecnologia, por profissionais e estudantes entre instituições de ensino entre outros.

Plataforma Spring

O Spring é uma plataforma opensource (comunidade desenvolvedora de aplicações) de código aberto, é propícia para o desenvolvimento de aplicações que dependem de recursos fora do programa para criar, desenvolver, manter, atualizar seus produtos por meio de conjunto de bibliotecas que trazem componentes prontos, que otimizam o tempo e deixam os projetos com uma estética e performance satisfatórios.

Assim como sua tradução, do termo em inglês, spring = primavera, tem como característica prover vida através de recursos que somente esta estação tem, porque das estações é a mais equilibrada, florida, com clima temperado, fazendo com que intempérie de outras estações não afetem de maneira negativa o trabalho dos desenvolvedores. Na nossa aplicação utilizaremos os seguintes componentes:

- **Spring Boot:** é um módulo Spring que auxilia na configuração e facilita as aplicações por meio de um conjunto de configurações iniciais, é um start para os desenvolvedores que os deixa livre para codificar não perdendo tempo com configurações de (métodos, classes) que ajudam a implementar de maneira mais ágil uma aplicação.
- **Spring Data e JPA (Hibernate):** faz a comunicação com o banco de dados, através JPA Hibernate que funciona como um agente desta comunicação.
- **Spring MVC (WEB):** O Spring MVC, faz a divisão das responsabilidades dos dados das aplicações nas camadas MVC, nesse caso utilizaremos ele como um servidor de aplicação WEB, nesse caso não existira a camada View.

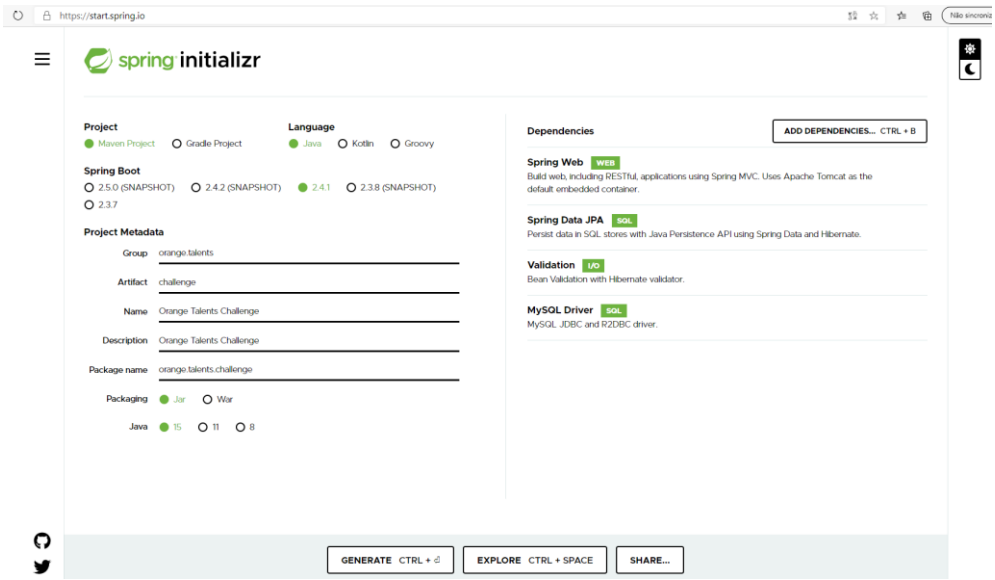
Banco de Dados Relacional com MySQL

Os bancos de dados são uma coleção de dados organizados e gerenciados por aplicativos associados a esses bancos de dados, geralmente numa estrutura como blocos que armazenam informações que serão utilizadas por demanda, com frequência ou não por usuários que querem acessar esses dados como uma página web hospedada em um determinado banco de dados etc.

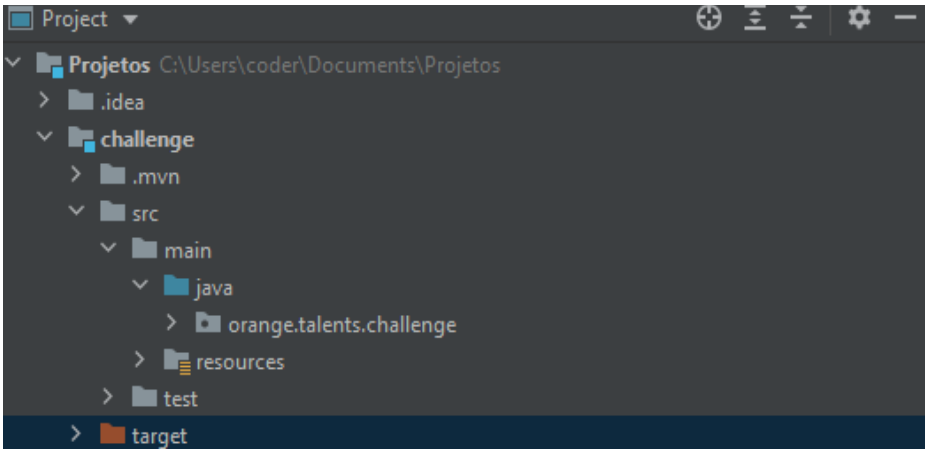
O MySQL é um sistema de gerenciamento de banco de dados, que utiliza a linguagem SQL como interface. É atualmente um dos sistemas de gerenciamento de bancos de dados mais populares da Oracle Corporation, com mais de 10 milhões de instalações pelo mundo.

O Projeto: Criando a aplicação

Existem algumas formas de criar uma aplicação Spring Boot, nesse tutorial iremos criar utilizando o Spring Initialzr que pode ser acessado em https://start.spring.io/. Configure o projeto utilizando *Maven*, *Spring Boot 2.4.1 (versão estável), Java JDK 15* e com as dependências *"Spring Web", "Spring Data JPA", "Validation"* e *"MySQL Driver"*.



Após concluir o download, descompacte o projeto e abra na sua IDE de preferência (nesse tutorial utilizaremos a **IDE IntelliJ)**. Ao abrir, aplicação ficará com a estrutura abaixo:



Na raiz do projeto podemos ver um arquivo chamado **pom.xml**, arquivo que por sua vez é uma unidade fundamental de trabalho em Maven. O **POM (Project Object Model)** contém informações sobre o projeto e vários detalhes de configuração usados pela Maven para construir e manter o projeto.

Veja abaixo como ficou esse arquivo:

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

   <properties>
      <java.version>15</java.version>
   </properties>

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

      <dependency>
         <groupId>mysql</groupId>
         <artifactId>mysql-connector-java</artifactId>
         <scope>runtime</scope>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId>
         <scope>test</scope>
      </dependency>

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

```
         <artifactId>hibernate-validator</artifactId>
         <version>6.0.11.Final</version>
      </dependency>
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-validator-annotation-
processor</artifactId>
         <version>6.0.11.Final</version>
      </dependency>
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-web</artifactId>
         <version>5.3.2</version>
      </dependency>
   </dependencies>

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

</project>
```

Também na raiz do projeto podemos ver o arquivo **OrangeTalentsChallengeApplication**, esse arquivo contém a classe que abriga o nosso método main:

```
package orange.talents.challenge;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class OrangeTalentsChallengeApplication {

   public static void main(String[] args) {
      SpringApplication.run(OrangeTalentsChallengeApplication.class,
args);
   }

}
```

Essa classe traz em seu nome a explicação clara sobre o que se trata o projeto, isso deixa mais claro e objetivo o conteúdo do projeto para o leitor e o programador.

Configurando o banco de dados.

Para o banco de dados, utilizaremos o servidor **MySQL CE**. Como usaremos apenas para fins didáticos, configuraremos o banco de dados com as configurações básicas, com a senha "**1234**" no usuário "**root**". É necessário que ele esteja rodando na porta **3306** para que tudo ocorra como esperado.

Esta configuração deve ser definida no arquivo **application.properties**:

```
spring.datasource.url=jdbc:mysql://localhost:3306/mysql?useTimezone=tr
ue&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=1234

spring.jpa.hibernate.ddl-auto=create

server.error.include-message=always
```

Note que configuramos *spring.jpa.hibernate.ddl-auto* como *create* e *server.error.include-message* como *always*, que nos ajudam com a criação automática das tabelas no banco de dados e na tratativa de exceções respectivamente.

Criando a Model (Entity e Repository)

Para iniciar o desenvolvimento do projeto, criaremos o nosso Model. Crie o pacote **orange.talents.challenge.model**, e dentro dele defina a seguinte entidade:

```
package orange.talents.challenge.model;

import com.fasterxml.jackson.annotation.JsonFormat;
import org.hibernate.validator.constraints.br.CPF;
import org.springframework.format.annotation.DateTimeFormat;

import javax.persistence.*;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import java.time.LocalDate;

@Entity
@Table(name = "pessoa")
public class PessoaModel {
```

```java
    @Id
    @Column(name = "cpf")
    @NotNull(message = "O CPF não pode ser vazio")
    @CPF(message = "CPF Inválido")
    private String cpf;

    @Column(name = "email")
    @NotNull(message = "O E-mail não pode ser vazio")
    @Email(message = "E-mail Inválido")
    private String email;

    @Column(name = "nome")
    @NotNull(message = "O Nome não pode ser vazio")
    @Size(min=2, message= "Nome precisa ter no mínimo 2 caracteres")
    private String nome;

    @Column(name = "dt_nascimento")
    @NotNull(message = "A data de nascimento não pode ser vazia")
    @DateTimeFormat(pattern = "dd/MM/yyyy")
    @JsonFormat(pattern = "dd/MM/yyyy")
    private LocalDate nascimento;

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public LocalDate getNascimento() {
        return nascimento;
    }

    public void setNascimento(LocalDate nascimento) {
        this.nascimento = nascimento;
    }
}
```

Como podemos notar, nossa model utiliza algumas anotações:

- **@Entity:** informa que essa classe Java é uma entidade.
- **@Table:** informa qual tabela no banco de dados essa entidade se relaciona.
- **@Id:** informa que essa propriedade é a chave primária.
- **@Column:** informa qual campo no banco de dados essa propriedade deve ser armazenada.

As outras anotações servem para validar as propriedades de acordo com a regra de negócio, você pode ver mais sobre elas acessando a documentação do **JPA/Hibernate Validador**.

Após criar a entidade, iramos criar nosso repositório no pacote **orange.talents.challenge.repository,** como no código a seguir:

```
package orange.talents.challenge.repository;

import orange.talents.challenge.model.PessoaModel;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

public interface PessoaRepository extends CrudRepository<PessoaModel,
String> {

    @Query(value =
            "SELECT " +
            "   CAST(" +
            "       IF(COUNT(*) > 0, " +
            "           'true', " +
            "           'false'" +
            "       ) " +
            "   AS JSON) " +
            "FROM pessoa WHERE email = :email",
            nativeQuery = true)
    boolean existsByEmail(@Param("email") String email);

}
```

Note que para o repositório foi necessário estender a interface **CrudRepository** do Spring Data. Esta interface possui métodos para as operações padrão de um CRUD.

Criamos também o método **existsByEmail,** ele verifica se existe algum registro com um E-mail especifico e retorna essa resposta como booleano, assim podemos proteger nosso banco de dados de E-mails duplicados. Para fazer isso, utilizamos a anotação **@Query** que nos possibilita adicionar uma query SQL personalizada para esse método, onde selecionamos e contamos quantos registros com esse E-mail existem, e após isso fazemos um CAST para booleano.

Criando o Controller

Neste tópico vamos definir o controller da nossa aplicação, criaremos em **orange.talents.challenge.controller**, segue código:

```
package orange.talents.challenge.controller;

import orange.talents.challenge.model.PessoaModel;
import orange.talents.challenge.repository.PessoaRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.server.ResponseStatusException;

import javax.validation.Valid;

@RestController
@RequestMapping(path = "/pessoa")
public class PessoaController {
    @Autowired
    private PessoaRepository pessoaRepository;

    @GetMapping(path = "/")
    public Iterable<PessoaModel> obtemTodos() {
        return this.pessoaRepository.findAll();
    }

    @GetMapping(path = "/{cpf}")
    public PessoaModel obtemPorCPF(@PathVariable String cpf) {
        boolean existsCpf = this.pessoaRepository.existsById(cpf);

        if(existsCpf) {
            return this.pessoaRepository.findById(cpf).get();
        } else {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND,
"Pessoa não encontrada!");
        }
    }

    @PostMapping(path = "/")
```

```
    public ResponseEntity<PessoaModel> salvar(@Valid @RequestBody
PessoaModel pessoa) {
        boolean existsCpf =
this.pessoaRepository.existsById(pessoa.getCpf());
        boolean existsEmail =
this.pessoaRepository.existsByEmail(pessoa.getEmail());

        if(existsCpf) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
"CPF ja existe no banco de dados!");
        } else if (existsEmail) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
"E-mail ja existe no banco de dados!");
        } else {
            PessoaModel p = this.pessoaRepository.save(pessoa);
            return new ResponseEntity(p, HttpStatus.CREATED);
        }
    }

    @DeleteMapping(path = "/{cpf}")
    public void deletarPeloCPF(@PathVariable("cpf") String cpf) {
        boolean existsCpf = this.pessoaRepository.existsById(cpf);

        if(existsCpf) {
            this.pessoaRepository.deleteById(cpf);
        } else {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND,
"Pessoa não encontrada!");
        }
    }
}
```

Na classe acima, é importante destacar alguns detalhes:

- **@RestController:** permite definir um controller com características REST;
- **@Autowired:** delega ao Spring Boot a injeção dessa dependência nesse objeto;
- **@PostMapping:** é usado para mapear solicitações HTTP POST.
- **@GetMapping:** é usado para mapear solicitações HTTP GET.
- **@DeleteMapping:** é usado para mapear solicitações HTTP DELETE.
- **@PathVariable:** indica que o valor da variável virá de uma informação da rota;
- **@RequestBody:** indica que o valor do objeto virá do corpo da requisição;
- **@Valid:** indica que os dados recebidos devem ser validados.

Para evitar duplicidade de **CPF** e **E-mail,** no recurso de criação de pessoa (POST), verificamos se já existe algum registro com o mesmo CPF (Id / Primary Key) ou E-mail enviado no corpo da requisição, caso exista, retornamos um “Bad Request” com a mensagem referente ao dado duplicado, caso não exista, prosseguimos com a criação do respectivo dado.

Com isso nossa API REST dispõe de algumas rotas, que nos permitem a criação, visualização e exclusão de pessoas no nosso banco de dados.

Tratando Dados Inválidos (NotValidExceptions)

Para tratar as exceções de dados inválidos, vamos criar o pacote **orange.talents.challenge.exception**, dentro dele criaremos duas classes, uma será o modelo de resposta de erros de validação, e a outra será a classe que realmente trata o erro de validação (NotValidException).

Segue o código da classe **ErrorResponse**:

```
package orange.talents.challenge.exception;

public class ErrorResponse {
    private Integer status;
    private String error;
    private String message;

    public ErrorResponse(Integer status, String error, String message)
{
        super();
        this.status = status;
        this.error = error;
        this.message = message;
    }

    public Integer getStatus() {
        return status;
    }

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

    public String getError() {
        return error;
    }

    public void setError(String error) {
```

```
        this.error = error;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

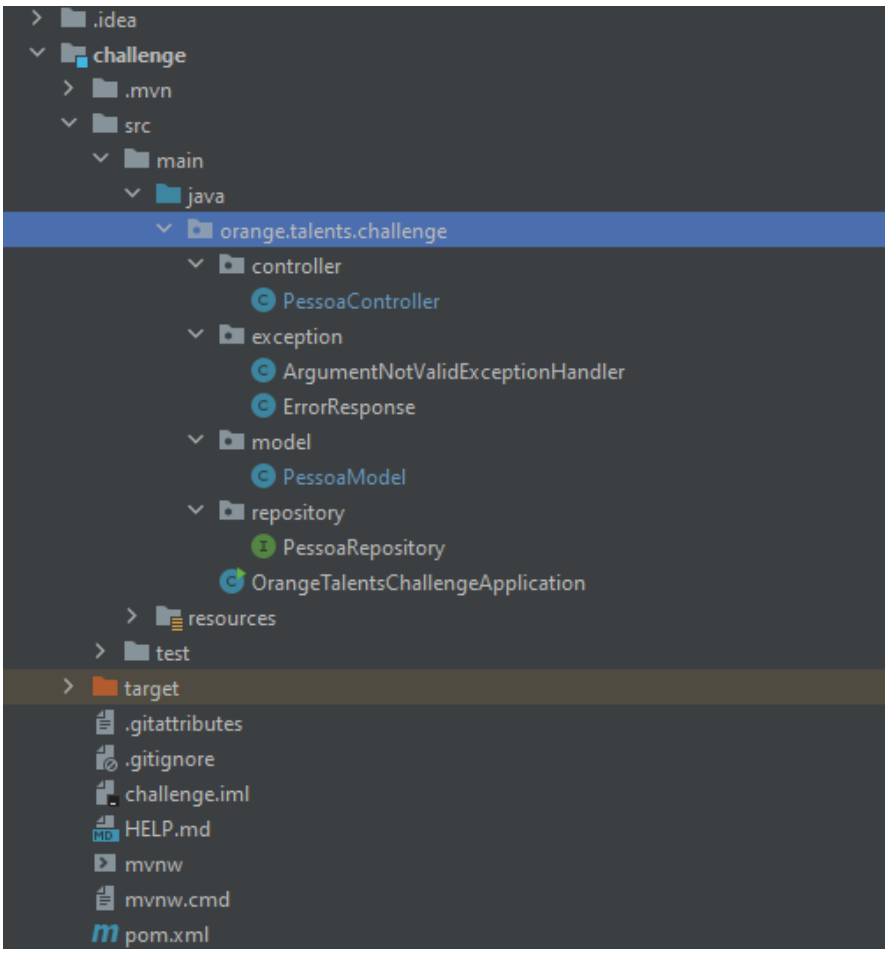
Segue código da classe **ArgumentNotValidExceptionHandler:**

```
package orange.talents.challenge.exception;

import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.context.request.WebRequest;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityEx
ceptionHandler;

@ControllerAdvice
public class ArgumentNotValidExceptionHandler extends
ResponseEntityExceptionHandler
{
    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex,
            HttpHeaders headers,
            HttpStatus status,
            WebRequest request
    ) {
        String message = ex.getBindingResult()
                .getAllErrors()
                .get(0)
                .getDefaultMessage();
        ErrorResponse error = new ErrorResponse(status.value(),
status.name(), message);
        return new ResponseEntity(error, status);
    }
}
```

Com isso quando um erro de validação for lançado pelo JPA/Hibernate Validator, nós o trataremos e responderemos para o *client* com o status e mensagem adequados.

Estrutura de pacotes final

Após todas as configurações e implementações, a estrutura de pacotes deve ficar como na imagem a seguir:



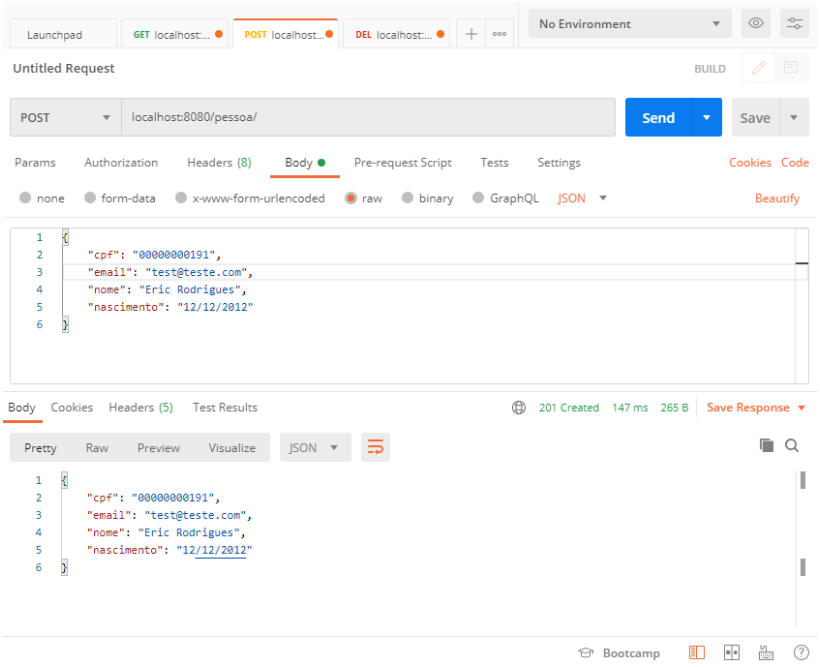
Com a API REST implementada, passaremos agora ao teste.

Testando nossa API com Postman

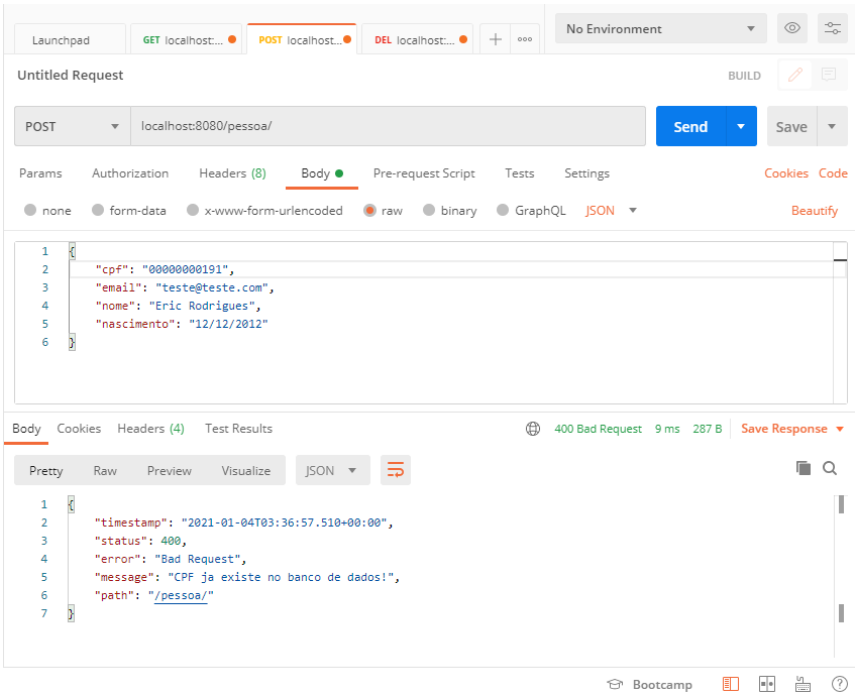
Para testarmos os recursos da nossa API REST, vamos utilizar o Postman. Essa ferramenta oferece suporte à documentação e possui também em seu ambiente a testagem de APIs e requisições. Testaremos e analisaremos as respostas dos testes que submetemos, assim podemos garantir o bom funcionamento da aplicação criada.

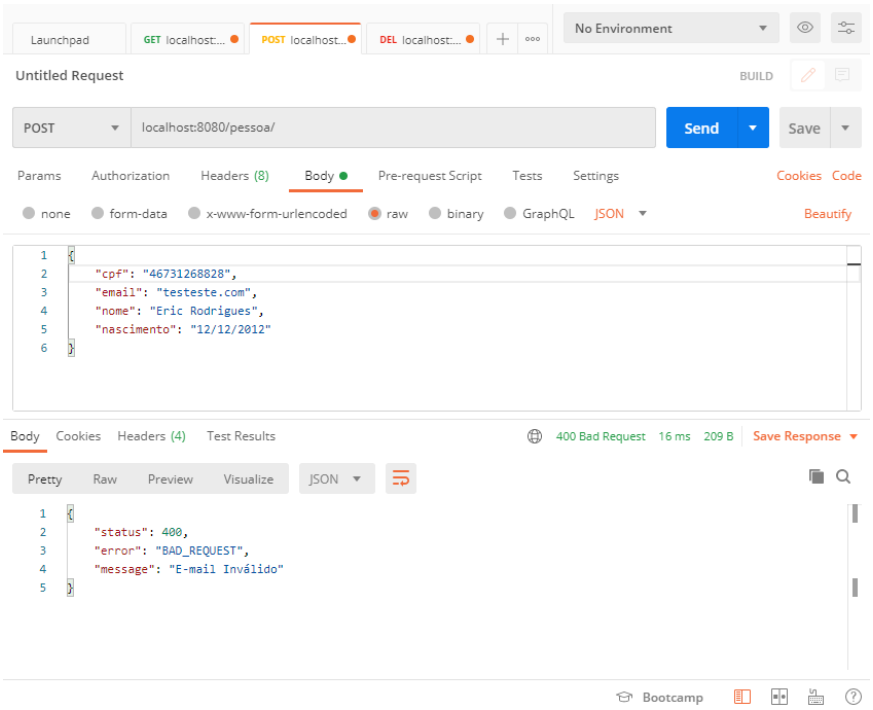
Inserindo uma pessoa (POST)

Caso todos os dados estejam definidos no JSON corretamente e não haja CPF ou E-mail duplicado, como pede o contrato da API de criação de pessoa, teremos uma resposta de sucesso, como na imagem a seguir:



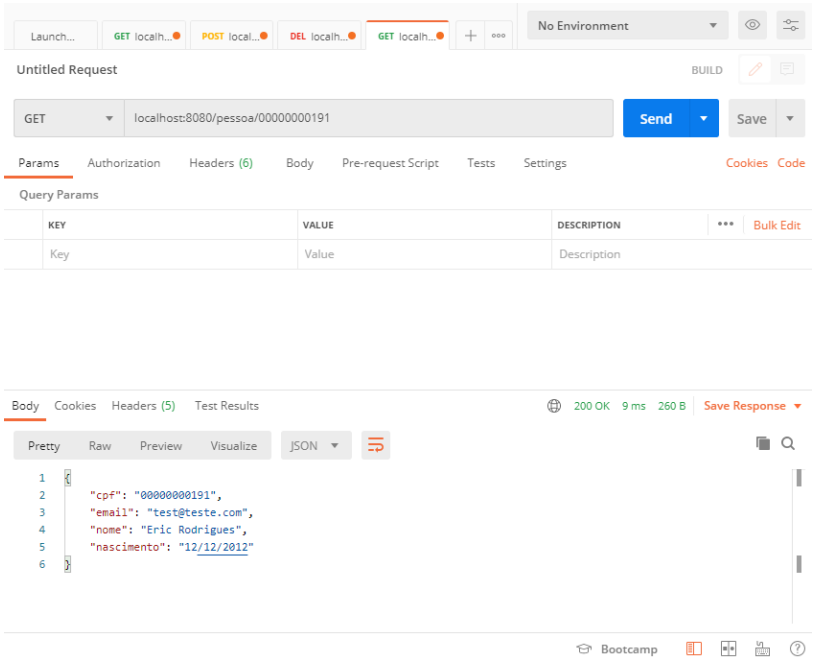
Caso alguma das regras anteriores seja violada, teremos uma resposta de erro, como nas imagens a seguir:



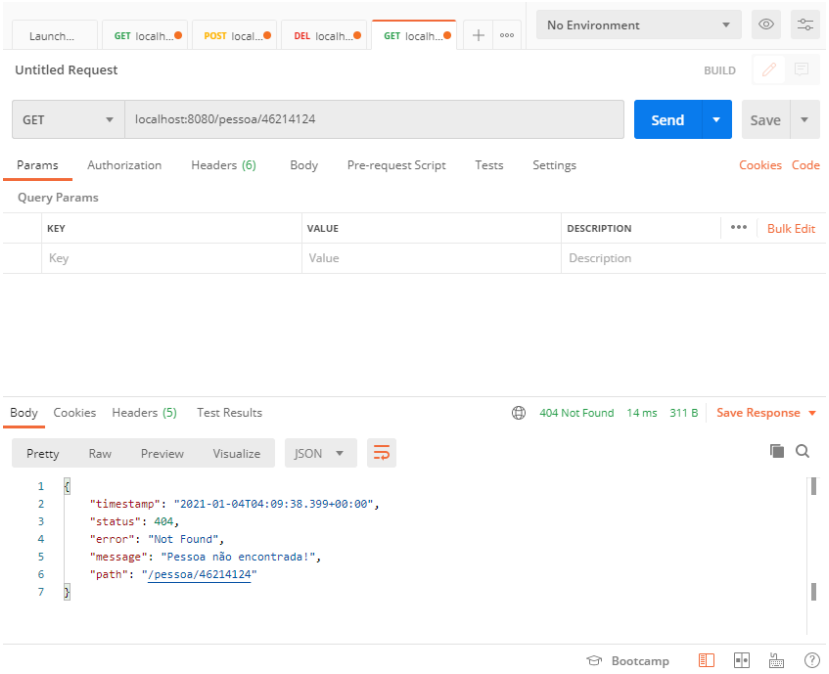


Obtendo uma pessoa (GET)

Caso exista um registro com o CPF passado via parâmetro de caminho, retorna uma resposta de sucesso com os dados dessa pessoa, como na imagem a seguir:

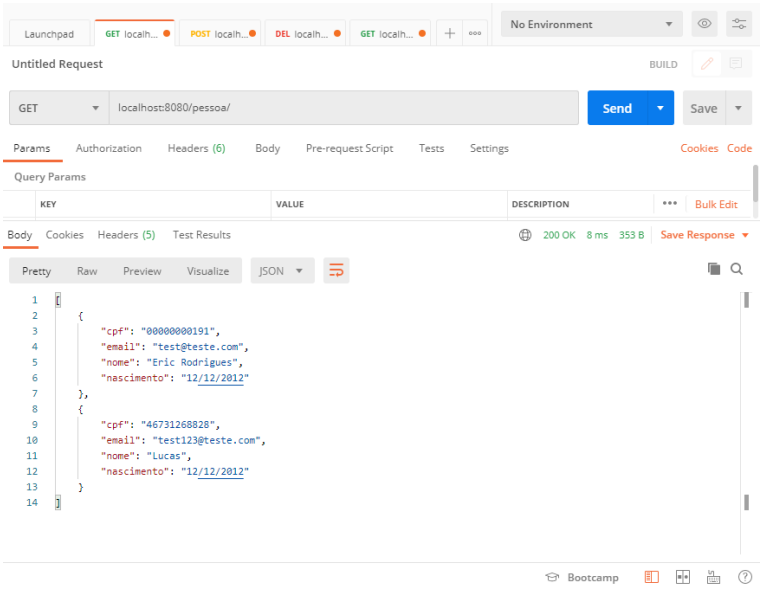


Caso não exista, retornará “Not Found”, como na imagem a seguir:



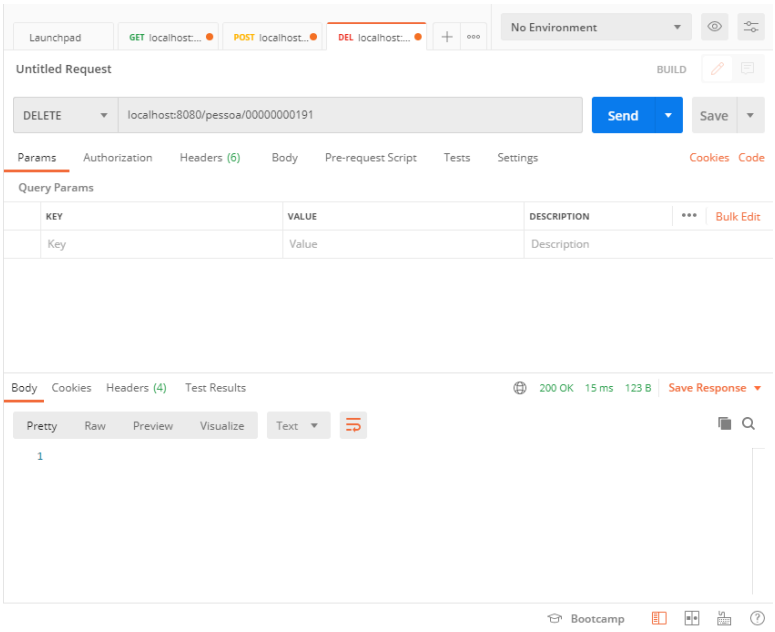
Obtendo todas as pessoas (GET)

Caso exista um registro com o CPF passado via parâmetro de caminho, retorna uma resposta de sucesso com os dados dessa pessoa, como na imagem a seguir:

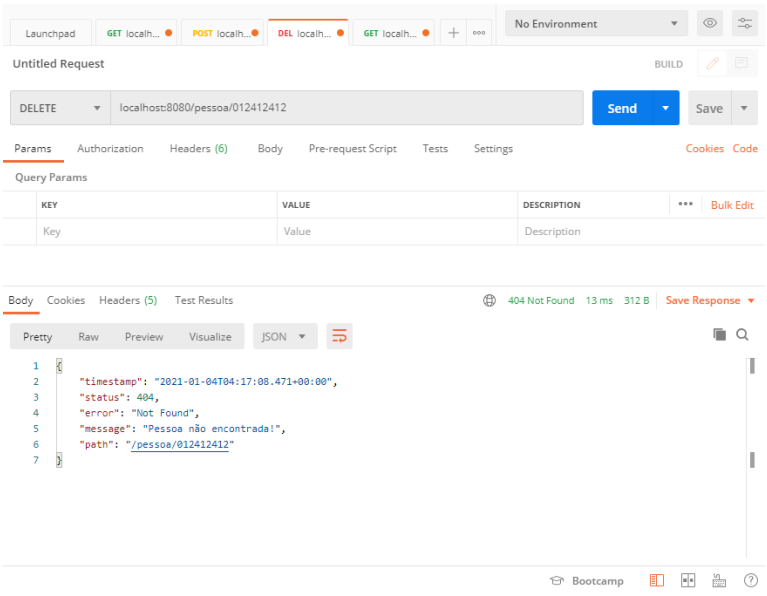


Deletando uma pessoa (DELETE)

Caso exista um registro com o CPF passado via parâmetro de caminho, deleta essa pessoa e retorna uma resposta de sucesso, como na imagem a seguir:



Caso não exista, retornará “Not Found”, como na imagem a seguir:



Conclusão

O processo de criação e implementação de uma API REST com Spring Boot propicia uma boa aprendizagem, porque a experiência de criar uma aplicação com um ecossistema e demais ferramentas para incrementar a aplicação faz com que você desfrute dos recursos que a plataforma Spring oferece e, possibilita a interação do desenvolvedor configurar algumas ações como métodos, criação de pastas, pacotes, classes de acordo com a especificidade do projeto. Nesta plataforma cabe estudar os recursos que o ecossistema oferece e saber utilizá-lo conforme a circunstância e a preferência da aplicação.

Éric de Souza Rodrigues Fonseca
proericrodrigues@gmail.com
Github: Orange Talents Challenge (github.com)