API REST

Neste post iremos construir do zero uma API Rest que é capaz de fazer o controle de endereços de usuários, utilizando Java e Spring.

RODRIGO TEIXEIRA

Conceitos básicos

Ola, tudo bem?

Neste post iremos aprender a criar do zero uma API REST utilizando Java e Spring.

Porém, antes de começarmos a colocar a mão na massa, precisamos aprender alguns conceitos básicos, que padrão iremos utilizar e que tipo de API REST iremos fazer.

Nesta aplicação iremos utilizar o **Padrão MVC**! E ai você me pergunta, o que é o padrão MVC?

O padrão MVC nada mais é que a estrutura que iremos utilizar para montar nossa API. Esse padrão é dividido em três camadas, a model, a view e a controller. A model é responsável por criar as tabelas e atributos do nosso banco de dados, pelo CRUD(Creat, read, update and delete) e pelas regras de negócio, a view é responsável pelo visual(front-end) e interação com o cliente, e a controller por fazer a comunicação entre a Model e a View. Como iremos trabalhar somente com o back-end nesse post não utilizaremos a View.

Devemos prestar bastante atenção na Model, já que ela tem três funções específicas ela também será dividida e três partes que são:

Entity - Responsável pela tabela e atributos do banco de dados.
Repository - Responsável pelos métodos CRUD
Service - Responsável pelas regras de negócio.

Maturidade de Richardson

O que é o modelo de Maturidade de Richardson?

O modelo de Maturidade de Richardson nada mais é que um modelo que classifica APIs por níveis, de 0 a 3, e esses níveis vão evoluindo conforme formos cumprindo devidos requisitos.
A classificação é a seguinte:

Nível 0 - HTTP

Nesse nível a API se comunica pelos verbos HTTP, porém sem nenhum critério de utilização do verbo ou de rotas.

Nível 1 - HTTP + Recursos

Aqui a API já está roteada e utiliza modelos de recursos.

Nível 2 - HTTP + Recursos + Verbos

No nível 2 a API além de estar roteada utiliza os verbos HTTP de forma correta, PUT para update, POST para inserção, GET para consulta e DELETE para deletar.

Nível 3 - HTTP + Recursos + Verbos + HATEOAS

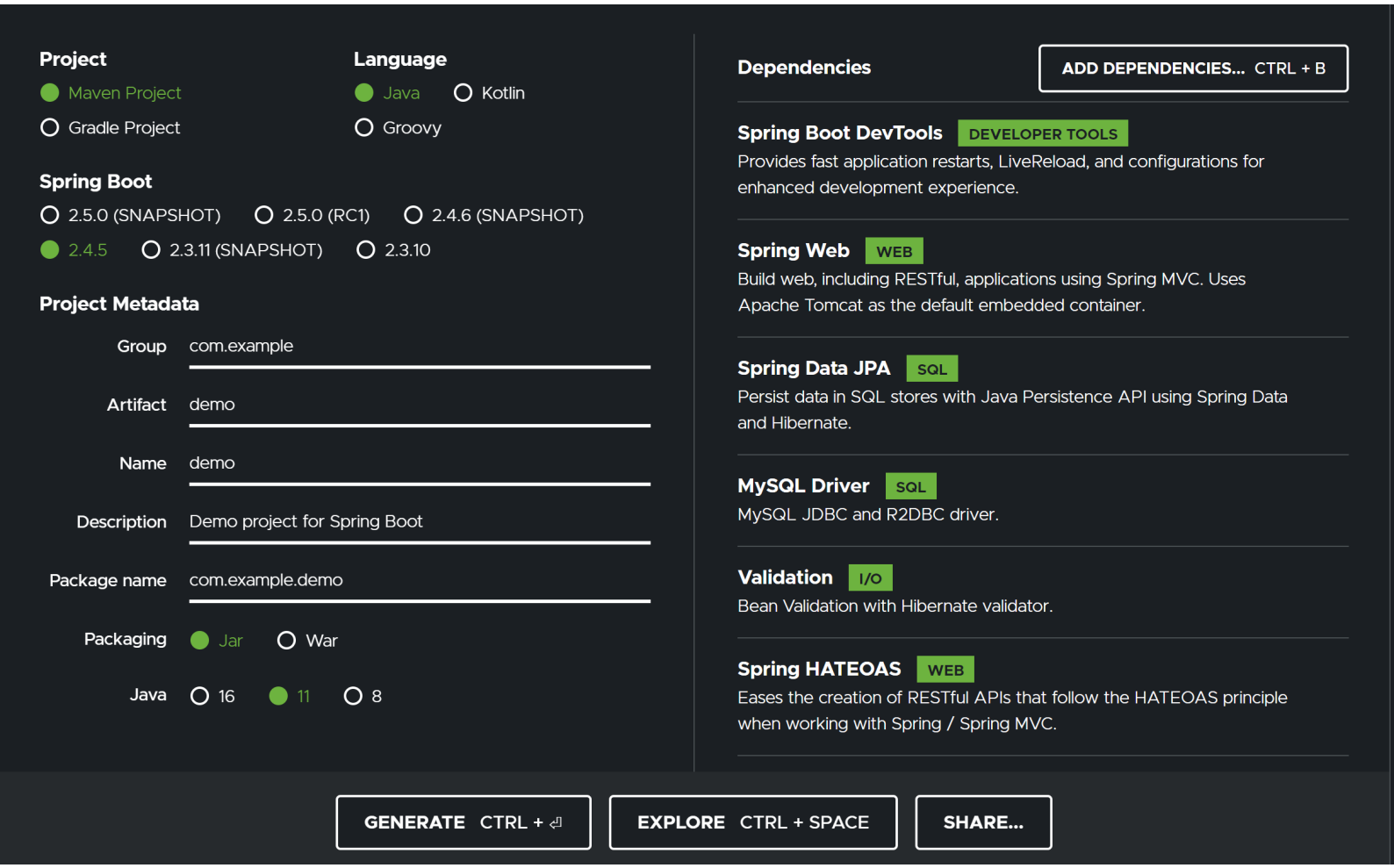
No nível 3 a API atinge todos os requisitos e além de todos os pontos acima citados ela também gera uma lista de rotas com tudo que é possível fazer a partir da chamada principal.

Para esse post iremos criar uma API REST de nível 3 do modelo de maturidade de Richardson.

Mão na massa

Como iniciaremos nossa API Rest?

Para darmos inicio ao nosso projeto o primeiro passo é preciso criar um projeto em Spring e para isso utilizaremos o site *Spring Initializr* (https://start.spring.io/)
Neste site iremos realizar as configurações principais do nosso projeto, definir a versão do Java que utilizaremos, selecionar as dependências, nome entre outros.
Após selecionar as configurações basta clicar em **GENERATE** para baixar o seu projeto.



Mão na massa

Configurações Spring Initilzr

Conforme na imagem acima utilizaremos o projeto maven, versão 2.4.5 do Spring Boot, versão do Java 11 e mais seis dependências.

E porque utilizaremos essas dependências?

Todas elas irão possibilitar e facilitar a criação da nossa aplicação.

Spring Boot Dev Tools - Responsável pelo LiveReload e pelo restart automático e rápido de nossa aplicação, aumentando assim a produtividade.

Spring Web - Necessário para a criação de um web service, contém todas as ferramentas necessárias para esse ambiente.

Spring Data JPA - Responsável pela persistência de dados SQL com o JPA utilizando o Spring Data e Hibernate.

MySQL Driver - Responsável pela conexão do nosso sistema com o banco de dados MySQL.

Validation - Responsável por fazer validação de dados em nosso sistema de uma maneira simplificada.

Spring HATEOAS - Utiliza recursos que facilitam a criação de uma API RESTful.

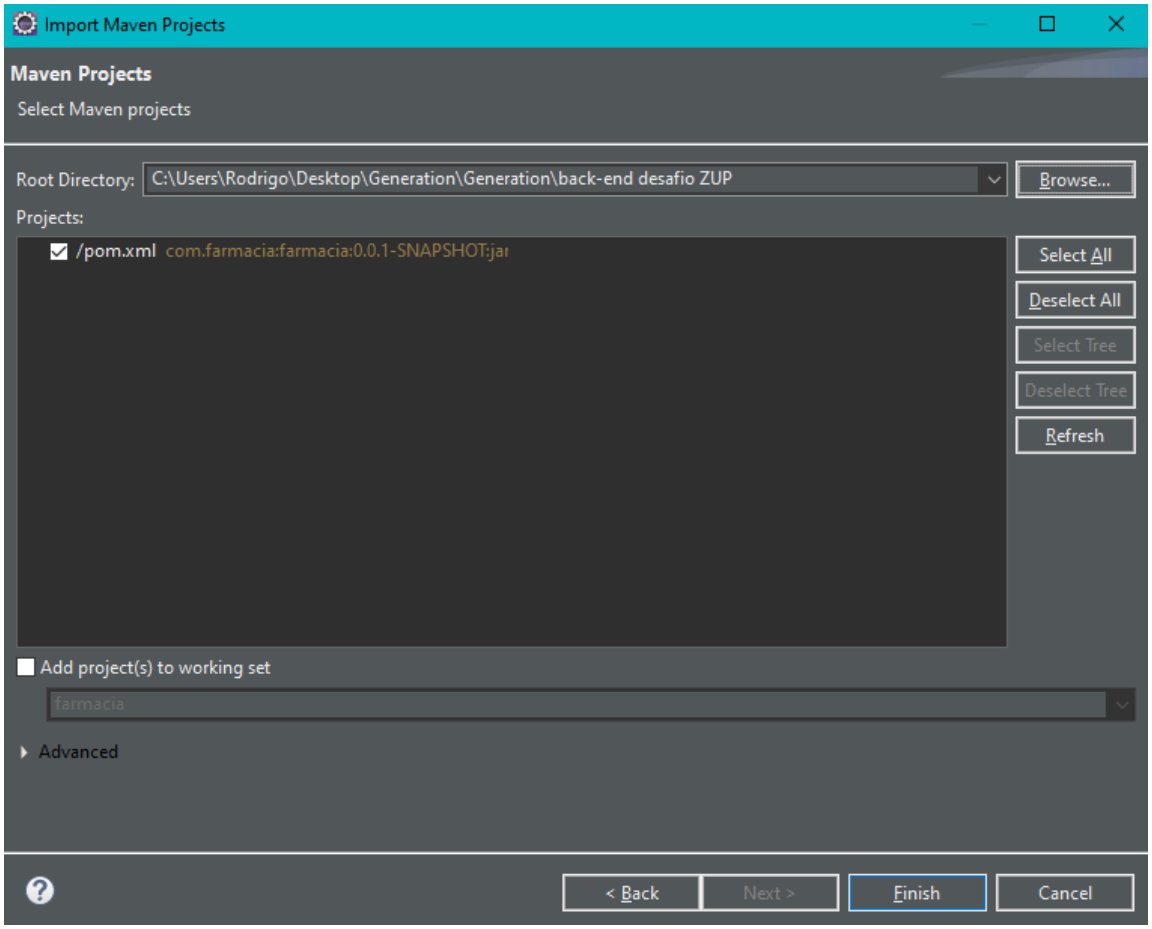
Mão na massa

Inicializando nosso projeto

O primeiro passo é importarmos nosso projeto na nossa IDE de preferencia.

Obs.: caso não tenha uma toma aqui um link para ver as principais que pode trabalhar com Java e Spring
https://www.treinaweb.com.br/blog/principais-ides-para-desenvolvimento-java/

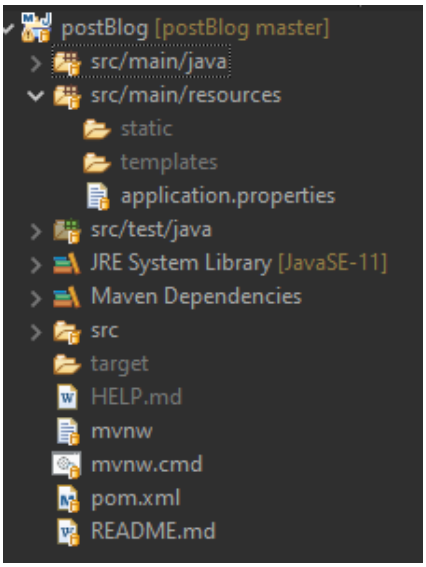
Eu utilizo o Eclipse e para importar é muito simples, basta seguir o seguinte caminho:
FILE => IMPORT => Maven => Existent Project Maven
te levará para essa tela



Mão na massa

Inicializando nosso projeto

a partir daí selecionamos finish e o Maven baixará todas as dependências para começarmos a parte mais esperada, codar!
Assim que todas as dependências forem baixadas iremos configurar nossa conexão com o banco, e isso é feito a partir do *application.properties*



Ao acessar ele estará vazio e será necessário configurar da seguinte forma.

```
1 spring.jpa.hibernate.ddl-auto=update
2 spring.datasource.url=jdbc:mysql://localhost/post_Blog?createDatabaseIfNotExist=true&serverTimezone=UTC&useSS1=false
3 spring.datasource.username=root
4 spring.datasource.password=123456
5 spring.jpa.show-sql=true
```

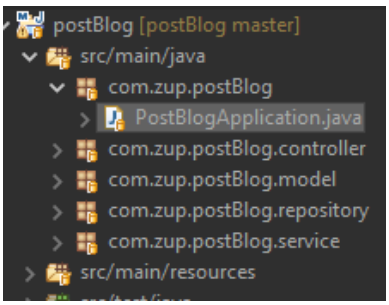
Linha 1 - Autoriza as operações no banco de dados.
Linha 2 - Faz a conexão com o banco local e cria um banco de dados caso não exista, caso queira alterar o nome do banco é só mudar a parte onde está "post_Blog".
Linha 3 - Indica o usuário do seu banco de dados
Linha 4 - Indica a senha do seu banco de dados
Linha 5 - Apresenta no console o as operações que ocorrerem.

Mão na massa

Inicializando nosso projeto

Obs.: Certifique que todos os dados da configuração estão certos, pois qualquer defeito no código pode causar erro no momento da criação do banco de dados.

Para verificarmos se todas as configurações foram feitas certas e para criarmos de uma vez nosso banco de dados iremos rodar nossa aplicação, e como faremos isso?
Iremos na classe Main e inicializaremos a aplicação



Assim que rodarmos a aplicação o banco de dados já será criado no nosso banco de dados.

DICA: A partir desse momento sempre deixe sua aplicação rodando, pois caso de qualquer erro quando salvar uma classe irá aparecer no console, possibilitando que consiga corrigir o erro rápido e não perceba ele só no final da sua aplicação.

Mão na massa

Criando nossa model

Agora que já temos todo o nosso sistema configurado podemos começar a criar nossa model.
Começaremos a criação pelas nossas entidades Usuario e Endereco.

Dica: Como teremos um relacionamento *@OneToMany* e *@ManyToOne* ao salvar sua primeira entidade dará um erro pois não existem as duas tabelas para as Foreing Keys serem mapeadas, mas assim que a segunda entidade for salva o erro sumirá.

Abaixo irei apresentar a criação das duas entidades e em seguida explicar todos os detalhes.

```java
@Entity
@Table(name = "usuario")
public class Usuario extends EntityModel<Usuario> {

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

    @Size(min = 3, max = 100)
    @NotBlank (message = "O campo nome não pode ser vazio!")
    private String nome;

    @Column(unique=false)
    @NotBlank (message = "O campo E-mail não pode ser vazio!")
    @Email
    private String email;

    @Column(unique=true)
    @NotNull (message = "O campo CPF não pode ser vazio!")
    @CPF
    private long cpf;

    @NotNull (message = "O campo Data de nascimento de  não pode ser vazio!")
    @JsonFormat(pattern = "dd/MM/yyyy")
    private LocalDate dataNascimento;

    @OneToMany (mappedBy = "usuario", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JsonIgnoreProperties("usuario")
    private List<Endereco> meusEnderecos = new ArrayList<>();
```

Mão na massa

Criando nossa model

```java
@Entity
@Table (name = "endereco")
public class Endereco extends EntityModel<Endereco> {

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

    @NotBlank (message = "O campo logradouro não pode estar vazio!")
    private String logradouro;

    @NotNull (message = "O campo numero não pode estar vazio!")
    private long numero;


    private String complemento;

    @NotBlank (message = "O campo bairro não pode estar vazio!")
    private String bairro;

    @NotBlank (message = "O campo cidade não pode estar vazio!")
    private String cidade;

    @NotBlank (message = "O campo cidade não pode estar vazio!")
    private String estado;

    @NotNull (message = "O campo CEP não pode estar vazio!")
    private long cep;

    @ManyToOne
    @JsonIgnoreProperties("endereco")
    private Usuario usuario;
```

Primeiramente criamos todos os atributos com seus respectivos tipos porém para podermos criar essas duas entidades precisamos entender as notações que existem dentro delas.

@Entity - Diz que essa classe é uma entidade(uma tabela no BD).

@Id - Diz que o atributo é um Id sendo assim recebe certas características.

@GenerateValue - Gera o valor automático para o Id.

@Column(unique = "true") - Diz que os atributos daquela coluna devem ser únicos.

Mão na massa

Criando nossa model

@Email - Diz que o atributo é um e-mail, e permite fazer a verificação
@CPF - Diz que o atributo é um CPF, e permite fazer a verificação
@NotBlank - Não permite o atributo vir em branco
@OneToMany - Cria uma relação entre tabelas de um para muitos
@ManyToOne - Cria a relação em uma tabela de muitos pra um
Além dessas notações ainda existem outras que podem ser encontradas na documentação:
https://www.baeldung.com/javax-validation

E por ultimo e mais importante não podemos esquecer que nossas classes estenderam de *EntityModel<"Classe">* que permite trabalharmos com HATEOAS adicionando um Link um objeto dos tipos Usuario e Endereco.

Agora que sabemos o que cada notação faz também sabemos o que cada atributo é e quais dados deverão ir dentro dele.

Nesse momento devemos criar os Getters e Setters, muitas pessoas gostam de criar-los automaticamente, porém é aconselhado que sejam criados somente os Getters e Setters necessários para o funcionamento da aplicação.
Assim finalizamos as nossas entidades podemos continuar a criação da nossa model, indo então para a criação dos nossos repositórios.

O repositório ao contrario das entidades não é uma classe e sim uma interface, ele irá estender de *JpaRepository* recebendo todos os seu métodos e por ser uma interface não poderá executar nenhum método, todos os seus recursos serão repassados para a classe service, nosso repositório será necessário somente para conter o CRUD.

Mão na massa

Criando nossa model

Assim como as entidades abaixo irei apresentar o UsuarioRepository e o EnderecoRepository e em seguida explicar todos os detalhes.

UsuarioRepository.java

```java
package com.zup.postBlog.repository;


import java.util.Optional;

@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

    Optional<Usuario> findByCpf(long cpf);

    Optional<Usuario> findByEmail(String email);

}
```

EnderecoRepository.java

```java
package com.zup.postBlog.repository;

import java.util.Optional;

@Repository
public interface EnderecoRepository extends JpaRepository<Endereco, Long> {

    Optional<Endereco> findById(long id);

}
```

Nossa interface assim que criada deve ter obrigatoriamente duas coisas, o *@Repository* para dizer que é um repositório e também estender de *JpaRepository* para herdar todos os metodos CRUD.

Quando temos o primeiro contato com essa interface podemos ficar um pouco confusos com a criação de novos métodos como o *findBy...* porém por herdar *JpaRepository* ela tem a característica de poder criar métodos dessa forma.

Dica: caso queira criar algum outro método existem outros tipos de "*findBy...*" todos são apresentados no link a seguir

https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html

Mão na massa

Criando nossa model

Agora que terminamos nossos repositórios chegamos na última parte da nossa model, o service.

Diferente das nossas entidades e repositórios iremos possuir apenas um service, já que teremos somente dois métodos e os dois serão utilizados pelo usuário.
Conforme anteriormente irei mostrar nosso UsuarioService e em seguida explica-lo

UsuarioService.java

```java
package com.zup.postBlog.service;


import java.util.Optional;

@Service
public class UsuarioService {

    private @Autowired UsuarioRepository repositoryUsuario;
    private @Autowired EnderecoRepository repositoryEndereco;

    public Optional<Usuario> cadastrarUsuario(Usuario novoUsuario) {
        Optional<Usuario> cpfExistente = repositoryUsuario.findByCpf(novoUsuario.getCpf());
        if (cpfExistente.isPresent()) {
            return Optional.empty();
        }
        Optional<Usuario> emailExistente = repositoryUsuario.findByEmail(novoUsuario.getEmail());
        if (emailExistente.isPresent()) {
            return Optional.empty();
        }
        Optional<Usuario> usuarioCadastrado = Optional.ofNullable(repositoryUsuario.save(novoUsuario));
        if (usuarioCadastrado.isPresent()) {
            return usuarioCadastrado;
        } else {
            return Optional.empty();
        }
    }

    public Optional<Endereco> cadastrarEndereco(Endereco novoEndereco, long cpf) {
        Optional<Usuario> usuarioExistente = repositoryUsuario.findByCpf(cpf);
        if (usuarioExistente.isPresent()) {
            novoEndereco.setUsuario(usuarioExistente.get());
            return Optional.ofNullable(repositoryEndereco.save(novoEndereco));
        } else {
            return Optional.empty();
        }
    }
}

```

Mão na massa

Criando nossa model

Assim como nossas outras classes no service a primeira coisa que devemos fazer é colocar o *@Service* para identificar que essa classe é um service.

Em seguida devemos fazer uma injeção de dependências utilizando o *@AutoWired*, isso significa que posso utilizar todos os métodos criados do que eu inserir.

```
private @Autowired UsuarioRepository repositoryUsuario;
private @Autowired EnderecoRepository repositoryEndereco;
```

Em seguida iremos criar dois métodos, primeiro o método cadastrarUsuario e em seguida cadastrarEndereco.

Nos dois métodos eu utilizo o *Optional* como retorno, que me permite criar diversas funções, tanto para retornos como para verificações.
Caso tenha alguma dúvida sobre o *Optional* vou deixar um link aqui embaixo com sua descrição e funções:
https://medium.com/@racc.costa/optional-no-java-8-e-no-java-9-7c52c4b797f1

Voltando para o nosso primeiro método, nós iremos fazer duas verificações:
- Primeira se existe um CPF já cadastrado
- Segunda se existe um Email já cadastrado

Optei por fazer a verificação para no controller conseguir dar uma resposta amigável para o cliente.

Caso o *return* seja um *Optional.empty()* no controller colocaremos para apresentar o erro caso seja um *Optional* com o Usuario dentro o controller apresentara a mensagem de usuário criado.

Mão na massa

Criando nossa model

```
public Optional<Usuario> cadastrarUsuario(Usuario novoUsuario) {
    Optional<Usuario> cpfExistente = repositoryUsuario.findByCpf(novoUsuario.getCpf());
    if (cpfExistente.isPresent()) {
        return Optional.empty();
    }
    Optional<Usuario> emailExistente = repositoryUsuario.findByEmail(novoUsuario.getEmail());
    if (emailExistente.isPresent()) {
        return Optional.empty();
    }
    Optional<Usuario> usuarioCadastrado = Optional.ofNullable(repositoryUsuario.save(novoUsuario));
    if (usuarioCadastrado.isPresent()) {
        return usuarioCadastrado;
    } else {
        return Optional.empty();
    }
```

No nosso segundo método iremos verificar se o usuário é existente e em seguida cadastrar um endereço a lista de endereços do usuário.

```
public Optional<Endereco> cadastrarEndereco(Endereco novoEndereco, long cpf) {
    Optional<Usuario> usuarioExistente = repositoryUsuario.findByCpf(cpf);
    if (usuarioExistente.isPresent()) {
        novoEndereco.setUsuario(usuarioExistente.get());
        return Optional.ofNullable(repositoryEndereco.save(novoEndereco));
    } else {
        return Optional.empty();
    }
}
```

Essa foi a criação da nossa Model completa, começamos criando nossas entidades, depois nossos repositórios e por último nosso service.

A partir desse momento podemos criar nosso controller para fazer a nossa parte de comunicação.

Mão na massa

Criando nosso controller

Para iniciarmos nosso controller assim como em todas as outras classes precisaremos utilizar uma notação para dizer que a classe é um controller, e utilizaremos *@RestController*.
Além dessa notação também utilizaremos outras duas, o *@RequestMapping* que indicará qual a rota inicial para utilização dos verbos HTTP e também o *@CrossOrigin* que permite o acesso de diversas plataformas.

```
@RestController
@RequestMapping("/postBlog/usuario")
@CrossOrigin("*")
public class UsuarioController {
```

Assim como no nosso service também precisamos fazer uma injeção de dependências das classes que iremos utilizar.

```
@Autowired
private UsuarioService services;

@Autowired
private UsuarioRepository repository;
```

Assim que fizermos essas duas etapas já podemos começar a criação dos nossos métodos HTTP.

Mão na massa

Criando nosso controller

O primeiro método que criaremos é o método cadastrar usuário. Por ser um método de inserção iremos utilizar o POST e para dizermos isso utilizamos o *@PostMapping* e em seguida passamos uma rota que será de utilização daquele método.

```
@PostMapping("/novo")
public ResponseEntity<?> cadastrarUsuario(@Valid @RequestBody Usuario novoUsuario) {

        Optional<Usuario> usuarioCriado = services.cadastrarUsuario(novoUsuario);

    if (!usuarioCriado.isEmpty()) {
        Usuario usuario = usuarioCriado.get();
        Link selfLink = new Link("http://localhost:8080/postBlog/usuario/busca/" + usuario.getCpf());
        usuario.add(selfLink);
        return ResponseEntity.status(HttpStatus.CREATED).body(usuario);
    } else {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST)
                .body("Usuario já existente no sistema, tente outro Email ou CPF!");
    }

}
```

Vendo o método é possível ver que utilizamos não só a notação *@PostMapping*, mas também:
@Valid - Verifica se está cumprindo as regras exigidas pelas notações da nossa entidade.
@RequestBody - Diz que retornará o usuário do corpo da mensagem que enviaremos pelo Postman.

Esse método retornará um *ResponseEntity*, me permite dizer qual será seu HttpStatus e ainda apresentar uma informação no *Body*.

Também podemos notar que o usuário criado retornará seu próprio link, aqui temos a utilização do HATEOAS.

Mão na massa

Criando nosso controller

Para o segundo método iremos cadastrar um endereço. Diferente do método anterior esse método receberá dois parâmetros, um para verificar o usuário e outro com o endereço.
Para a verificação do usuário iremos receber seu CPF diretamente na nossa rota, precisando assim utilizar a notação *@PathVariable* e na rota colocar o campo {cpf}.

```
@PostMapping("/{cpf}/novo/endereco")
public ResponseEntity<?> cadastrarEndereco(
        @Valid @RequestBody Endereco novoEndereco,
        @PathVariable(value = "cpf") Long cpf) {

    Optional<Endereco> enderecoCriado = services.cadastrarEndereco(novoEndereco, cpf);
    {
        if (!enderecoCriado.isEmpty()) {
            Endereco endereco = enderecoCriado.get();
            Link selfLink = new Link("http://localhost:8080/postBlog/endereco/" + endereco.getId());
            endereco.add(selfLink);

            Link linkUsuario = new Link("http://localhost:8080/postBlog/usuario/busca/" + endereco.getUsuario().getCpf());
            endereco.getUsuario().add(linkUsuario);

            return ResponseEntity.status(HttpStatus.CREATED).body(endereco);
        } else {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST)
                    .body("O endereço não foi cadastrado, CPF ou Endereco invalidos.");
        }
```

Vemos que aqui também foi utilizado o *Link* para retornar tanto o *selfLink* do endereço como o *selfLink* do usuário que está vinculado com o endereço.

O nosso terceiro método será um método de pesquisa, portando devemos utilizar o verbo HTTP GET, mudando assim a notação que daremos para esse método. A notação será *@GetMapping* com sua rota de pesquisa.

Além disso também iremos observar algo diferente no método, já que um usuário pode ter vários endereços dentro do retorno iremos listar esses endereços e adicionar seus respectivos links para pesquisa.

Mão na massa

Criando nosso controller

```
@GetMapping("/busca/{cpf}")
public ResponseEntity<?> getUsuario(@Valid @PathVariable long cpf) {
    Optional<Usuario> usuario = repository.findByCpf(cpf);
    {
        if (usuario.isEmpty()) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                    .body("CPF pesquisado não se encontra em nosso sistema.");
        } else {
            List<Endereco> listaEndereco = usuario.get().getMeusEnderecos();
            for (Endereco enderecos : listaEndereco) {
                Link selfLink = new Link("http://localhost:8080/postBlog/endereco/" + enderecos.getId());
                enderecos.add(selfLink);
            }

            Link selfLink = new Link("http://localhost:8080/postBlog/usuario/busca" + usuario.get().getCpf());
            usuario.get().add(selfLink);
            return ResponseEntity.status(200).body(repository.findByCpf(cpf));
        }
```

Nesse método mostramos também outra forma de se apresentar um HttpStatus que é utilizando somente seu número.

No inicio do nosso post iriamos criar somente três endpoins porém com o método getUsuario também vimos a necessidade de criar um quarto endpoint para deixar o HATEOAS completo.

Criamos um EnderecoController e adicionamos o seguinte método

```
@GetMapping("/{id}")
public ResponseEntity<?> getEnderecoById(@Valid @PathVariable long id){
    Optional<Endereco> endereco = repository.findById(id);
    if(endereco.isPresent()) {
        Usuario usuario = repository.findById(id).get().getUsuario();
        Link selfLink = new Link("http://localhost:8080/postBlog/usuario/busca/" + usuario.getCpf());
        usuario.add(selfLink);

        List<Endereco> listaEndereco = usuario.getMeusEnderecos();
        for (Endereco enderecos : listaEndereco) {
            Link linkEndereco = new Link("http://localhost:8080/postBlog/endereco/" + enderecos.getId());
            enderecos.add(linkEndereco);
        }
        return ResponseEntity.status(200).body(repository.findById(id).get());
    } else {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Endereco pesquisado não se encontra em nosso sistema.");
    }
}
```

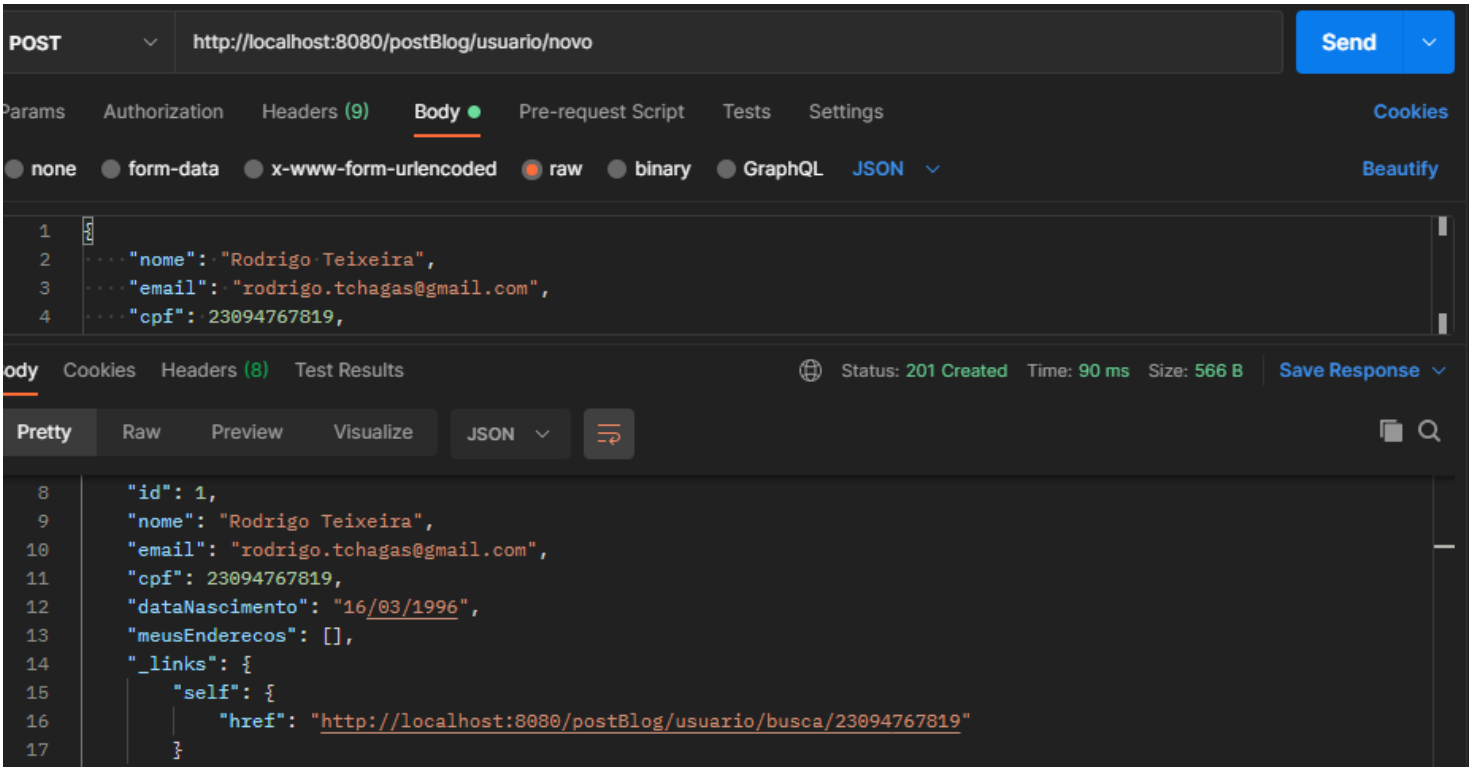
Assim com esse método podemos ter uma navegação completa apenas com o endereço de entrada.

E temos nossa API RESTful pronta para iniciarmos os testes no Postman

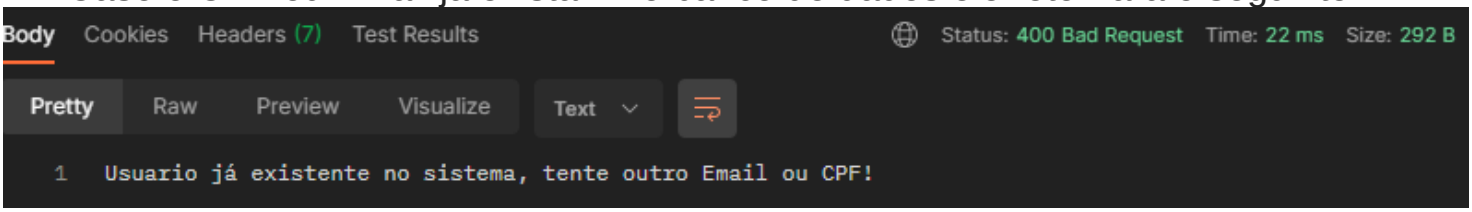
Realizando testes

O primeiro método a ser testado será o cadastrarUsuário, lembrando que se o cadastro estiver correto deve nos retornar o Status 201 caso contrário o Status deve ser 400.

Passando o endereço mapeado no controller com o método correto e o body correto ele nos retorna



Caso o CPF ou Email já existam no banco de dados ele retornará o seguinte.

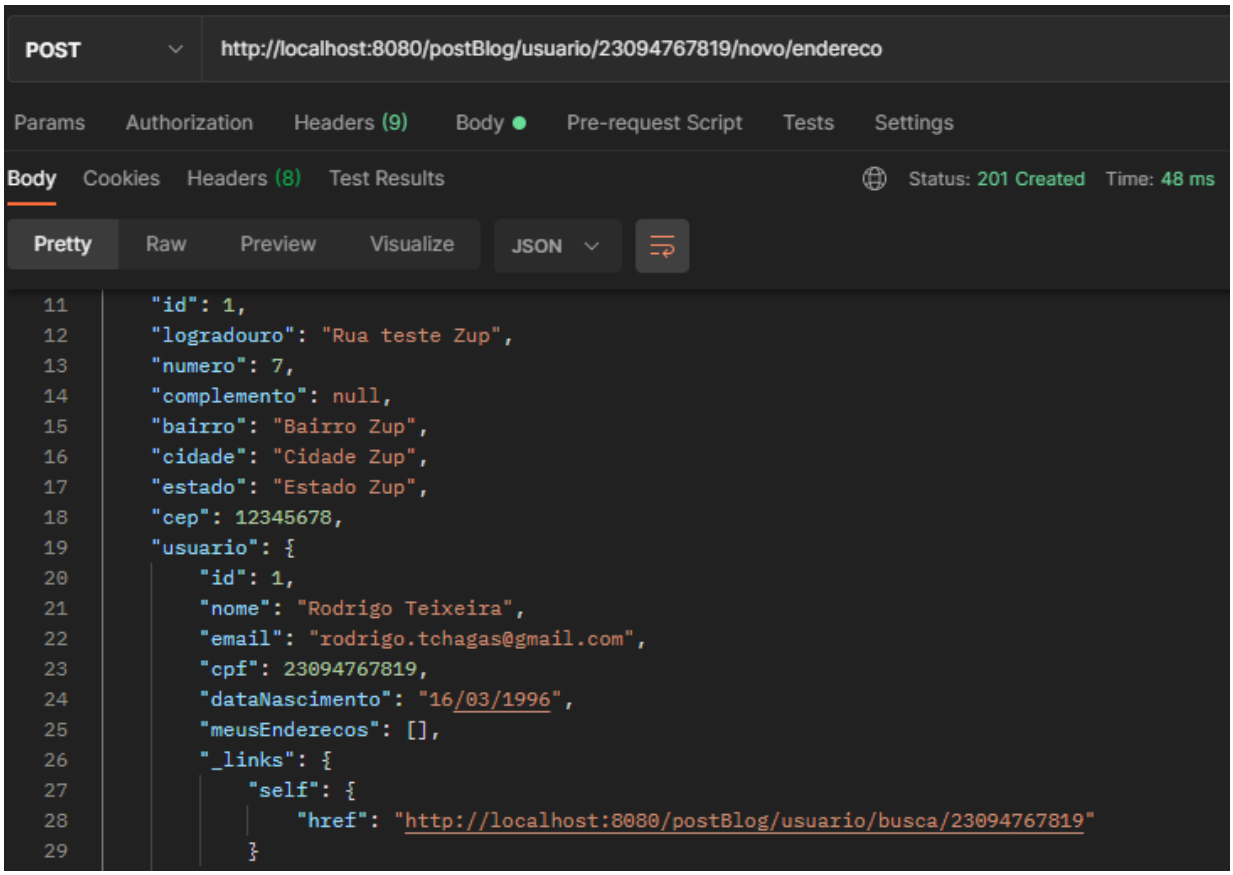


E caso um campo seja incorreto retornará

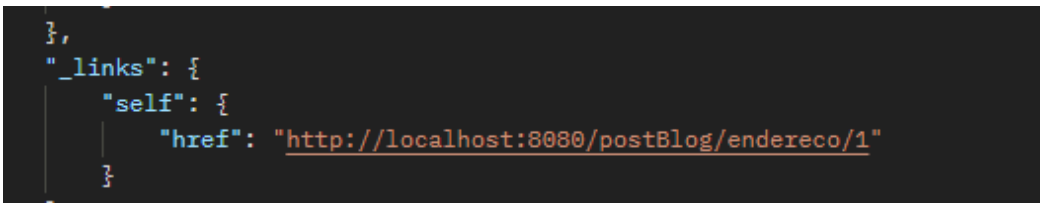
Realizando testes

O segundo método a ser testado será o cadastrarEndereco, lembrando que se o cadastro estiver correto deve nos retornar o Status 201 caso contrário o Status deve ser 400.

Passando o endereço mapeado no controller com o método correto e o body correto ele nos retorna



E mais para baixo também apresentou o link de pesquisa por endereço



Realizando testes

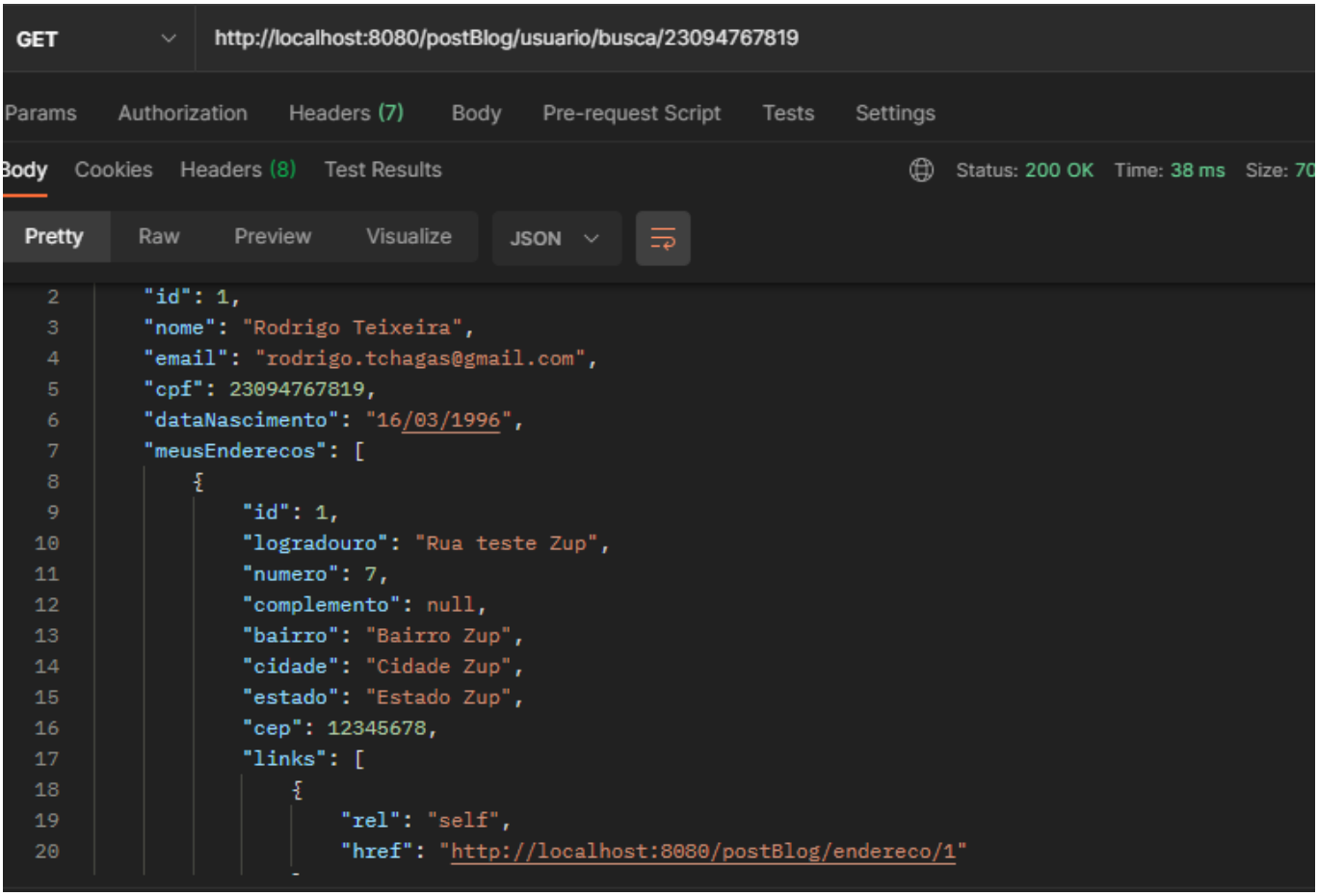
O caso o usuário passado não existisse retornaria

```
O endereço não foi cadastrado, CPF ou Endereco invalidos.
```

Caso algum campo de endereço fosse inválido retornaria

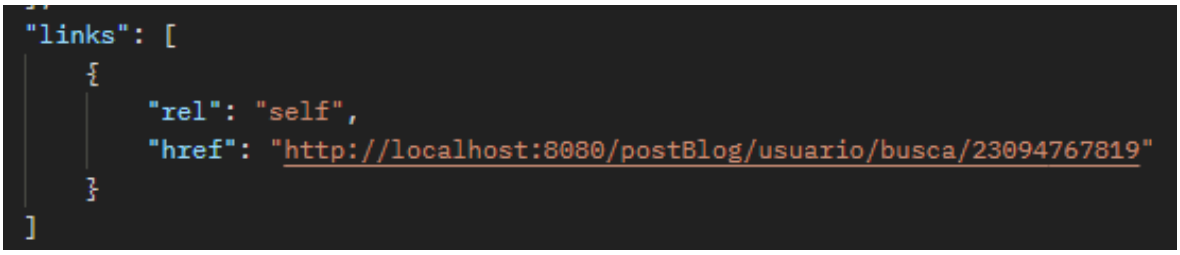
```
"timestamp": "2021-05-05T05:52:05.763+00:00",
"status": 400,
"error": "Bad Request",
"trace": "org.springframework.http.converter.Htt
```

E para nosso último método, getUsuario, quando a busca estiver correta deve nos retornar o Status 200 caso contrário o Status deve ser 404.
Passando o endereço mapeado no controller com o método correto e o body correto ele nos retorna

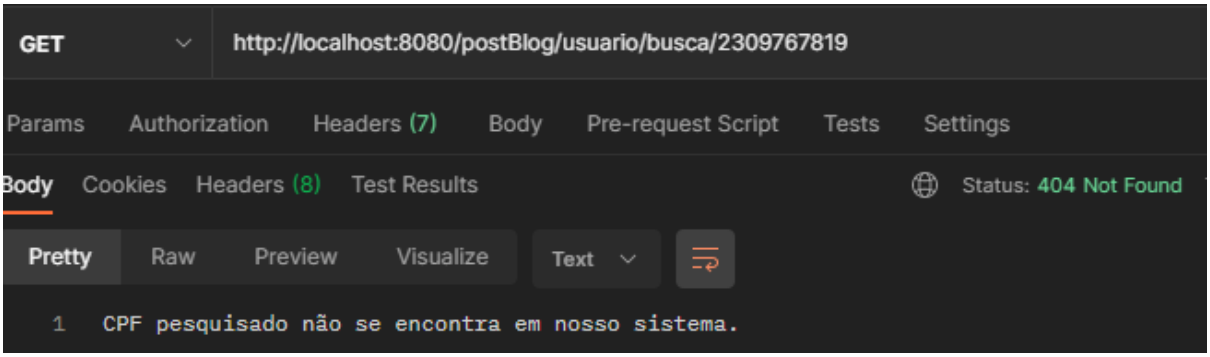


Realizando testes

Além da lista de endereços com suas respectivas rotas ele também apresenta seu próprio link de pesquisa.



Quando a busca não está correta ele nos retorna.



Dessa maneira já com os testes feitos podemos implementar nossa aplicação. O serviço poderia ser hospedado através do Heroku, e consumido por outras aplicações, existem serviços de container que podem auxiliar entre eles recomento o Cloud Run.

Obrigado!

Assim finalizamos nosso aprendizado de hoje, espero que tenham gostado desse post, foi feito com muito carinho, amor e dedicação.

Qualquer dúvida ou sugestões estarei a disposição.

Um abraço e até breve.