API REST de Cadastro de Usuários e Endereços Utilizando Spring Boot e Hibernate

Othon Breener Marques Da Silva

29 de maio de 2021

Ultimamente os serviços web são feitos utilizando as API's REST, isso se deve a facilidade de comunicação entre aplicações e a segurança dos dados fornecidos pelo usuário, já que o API REST separa as aplicações back-end e front-end. Por utilizar o formato JSON a API REST é capaz de se comunicar com aplicações de várias plataformas como android, terminal e web, uma vez que o front-end pode processar esses dados e exibir o resultado de acordo com o dispositivo utilizado. Para entendermos melhor o conceito de API REST é preciso saber o que cada palavra dessa frase significa, assim:

- API (*Application Programming Interface*) é um conjunto de rotinas e padrões pré estabelecidos que permite a comunicação entre aplicações, isso é feito através dos métodos de requisição HTTP. HTTP significa Protocolo de Transferência de Dados (*Hypertext Transfer Protocol*), o protocolo é baseado em um conjunto de regras e convenções que definem a troca de dados entre cliente e servidor. Os principais métodos de requisição são:

  1. POST: Utilizado para criar dados. Se criado com sucesso, o código HTTP 201 (criado) é retornado.
  2. GET: Utilizado para leitura ou recuperação de dados no host. Se bem sucedido, ele retornará uma representação em JSON ou XML com o status HTTP 200 (OK). Em casos de erros, retornará o código 404 (Não Encontrado) ou 400 (Pedido Ruim).

  existe mais duas requisições importantes o PUT e DELETE, usados respectivamente, para atualizar e deletar um recurso, mas para nossa aplicação vamos focar apenas no POST e GET.

- REST (*Representational State Transfer*) é conjunto de recomendações consideradas na criação de aplicativos e serviços web, ele funciona em cima do HTTP sendo uma das maneiras de utilizar o HTTP. As principais recomendações do REST são:

  1. Interação cliente-servidor: Separar a interface do usuário, de forma que as aplicações entre o cliente e o servidor sejam separadas.

2. Sem estado: As solicitações de um cliente devem conter todas as informações necessárias, sem depender de nenhum dado armazenado no servidor.
3. Cache: Um par de solicitação - resposta pode ser marcado como armazenamento em cache, evitando chamadas recorrentes ao servidor.
4. Interface Uniforme: Qualquer serviço respeitando a arquitetura REST dever ser compreensível sem seu desenvolvedor.

Um serviço escrito obedecendo às recomendações REST é chamado de RESTful.

Com isso, o nome API REST quer dizer que vai ser utilizado uma API para acessar aplicações back-end, baseados na arquitetura REST.

1 O Spring

O Spring é uma estrutura de desenvolvimento de aplicativos web que pode ser usada para aplicativos desktop e linha de comando, podemos pensar no Spring como um framework dos frameworks, ele gerencia um conjunto de frameworks visando fornecer um padrão as funcionalidades dos aplicativos.

Framework é um software utilizado em todas as linguagens de programação, este foi criado para simplificar e automatizar o processo de criação de códigos evitando a repetição mecânica. Por exemplo, diferentes redes sociais possuem semelhantes processos ao cadastrar uma nova conta, onde são solicitados dados como nome, e-mail, etc. Pensando nisso foi criado um software universal reutilizável visando facilitar o desenvolvimento de aplicativos.

O Spring é baseado no conceito de Inversão de Controle (IoC - *Inversion of Control*) e Injeção de Dependências (DI - *Dependency Injection*). IoC significa que a estrutura está responsável por controlar o fluxo geral do programa, ou seja, o IoC dá liberdade para um elemento (contêiner) criar e controlar um objeto sem a necessidade do programador definir tal tarefa. A DI é apenas uma forma de definir o IoC, em suma injetamos uma dependência em uma classe de forma que esta passa a ter o controle sobre a outra.

2 Objetivo

O intuito desse texto é criar uma aplicação de API REST utilizando o Java como linguagem de programação com o auxilio do Spring Boot e Hibernate. Nossa aplicação se resume em criar um cadastro de usuários, onde deve ser recebido como atributos os seguintes dados:

- Nome;
- E-mail;
- CPF;
- Data de Nascimento;

Além destes dados básicos, é necessário associar o usuário à um cadastro de endereços contendo os seguintes itens obrigatórios:

- Logradouro, Número e Complemento;
- Bairro, Cidade e Estado;
- CEP;

Para tal, é precisar fazer uma validação de dados de forma que os campos não sejam nulos ou preenchidos em branco. O endereço de e-mail deve conter o '@' e o CPF (Cadastro de Pessoa Física) deve seguir as regras da receita federal, para tais validações utilizaremos o Bean Validation mas ainda chegaremos lá.

3 Utilizando o Spring Initializr para Criar o Projeto

Dentro do Spring há um framework chamado Spring Boot, este framework define um conjunto de configurações padrões de forma que o desenvolvedor possa criar um aplicativo baseado em Spring e simplesmente executar. O Spring fornece uma interface para iniciar seu projeto, através desta é escolhido o tipo de projeto, no nosso caso Maven, e alguns dados principais como as versões tanto do Java como do Spring Boot.

O Spring Initializr oferece uma grande variedade de dependências, dentre as disponíveis utilizaremos as seguintes:

- Spring Data JPA: Java Persistence API (JPA) é uma ferramenta persistência de dados, ele é o responsável por conectar banco de dados relacionais e modelos orientados a objetos. O Sping Data fornece a implementação do JPA através do framework Hibernate, o qual fica responsável por criar os bancos de dados.
- Spring Web: É o responsável por criar o servidor Tomcat, utilizando o Spring MVC é possível receber as requisições HTTP. O framework busca a classe responsável por tratar as requisições e entrega a ela os dados enviados pelo browser.
- Validation: Fornece um conjunto de anotações providas do bean validation e do validador hibernate, as quais são responsáveis por validar os atributos da API.
- H2 Database: Oferece um banco de dados relacional com armazenamento em memória.

4 Iniciando a API

Uma vez que sabemos o que é uma API REST e conhecemos as ferramentas utilizadas para sua criação, está na hora de colocar a mão na massa e começar a construir o código fonte. Ao criar um projeto com o Spring Initializr é gerado um arquivo zip com o projeto em Maven, partiremos dele.

Baseado no objetivo apresentado na seção 2, podemos pensar no programa como uma classe usuário onde é definido os atributos necessários e uma segunda classe endereços, a classe usuários fica da seguinte forma:

```
public class User {

    private String nome;
    private String email;
    private String cpf;
    private String dataNascimento;

    public String getNome() {
        return nome;
    }

    public String getDataNascimento() {
        return dataNascimento;
    }

    public String getEmail() {
        return email;
    }

    public String getCpf() {
        return cpf;
    }

}
```

De forma similar criamos a classe endereços:

```
public class AdressRegister {

    private String logradouro;
    private Integer numero;
    private String complemento;
    private String bairro;
    private String estado;
    private String CEP;
    private User titular;

    public int getNumero() {
        return numero;
    }

    public String getLogradouro() {
        return logradouro;
    }
```

```
18
19      public String getComplemento() {
20          return complemento;
21      }
22
23      public String getBairro() {
24          return bairro;
25      }
26
27      public String getEstado() {
28          return estado;
29      }
30
31      public String getCEP() {
32          return CEP;
33      }
34
35  }
```

Bem, possuímos duas classes onde uma é o cadastro de usuários e a outra o cadastrado de endereço, as duas estão relacionadas pelo atributo titular. O próximo passo é definir como é este relacionamento e como atribuir esses dados em um banco de dados, para isto utilizaremos as anotações. Anotações são um tipo de metadados que fornecem informações sobre o programa, elas podem ser utilizadas para marcar classes, métodos, variáveis, entre outros.

As marcações são feitas utilizando o símbolo '@'. Há várias anotações disponíveis, falaremos delas a medida que formos utilizando. Para representar dados em um banco de dados de objetos java, precisamos de duas anotações relacionadas ao JPA e Hibernate, sendo elas:

- @Entity: A entidade é uma classe POJO (Plain Old Java Object), ou seja ela representa uma classe simples, não devemos relacionar tal classe à uma interface ou classes pré-especificadas. Uma classe marcada com @Entity representa uma tabela no banco de dados, enquanto os campos da entidade representam as colunas da tabela.
- @Id: Para poder identificar uma instância de entidade específica dentro do banco de dados precisamos marca-la com uma chave primária utilizando a anotação @Id, está representará uma coluna no banco de dados.
- @GeneratedValue: Está anotação define como os Id's são gerados automaticamente, onde o hibernate identifica o padrão que está sendo utilizado no projeto.
- @OneToMany e @ManyToOne: Estas definem o tipo de relacionamento entre duas classes, falaremos delas quando a aplicação no código for mostrada.
- @Column: Esta anotação é utilizada para definir um atributo como uma coluna, dentro da anotação coluna ainda é possível passar parâmetros. Utilizaremos os seguintes:
  - Nullable: define se o valor do atributo pode ser nulo.

– Unique: define se o atributo pode ou não ter registros iguais.

Agora, o código fica:

```
@Entity
public class User {

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

    @Column(nullable = false)
    private String nome;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false, unique = true)
    private String cpf;

    @Column(nullable = false)
    private String dataNascimento;

    @OneToMany(mappedBy = "titular")
    private List<AddressRegister> address = new ArrayList<>();

    public User(String nome, String email, String cpf, String
       dataNascimento) {
        this.nome = nome;
        this.email = email;
        this.cpf = cpf;
        this.dataNascimento = dataNascimento;
    }

    @Deprecated
    public User() {
    }

    public Long getId() {
        return id;
    }

    public List<AddressRegister> getAddress() {
        return address;
    }

     ..... getters ......

```

```
44
45 }
```

Já a classe de endereços:

```
1
2  @Entity
3  public class AdressRegister {
4
5      @Id
6      @GeneratedValue(strategy = GenerationType.IDENTITY)
7      private Long id;
8
9      private String logradouro;
10     private Integer numero;
11     private String complemento;
12     private String bairro;
13     private String estado;
14     private String CEP;
15
16     @ManytoOne
17     private User titular;
18
19
20
21     public AddressRegister(String logradouro, Integer numero, String
           complemento, String bairro, String estado, String CEP, User
          titular) {
22         this.logradouro = logradouro;
23         this.numero = numero;
24         this.complemento = complemento;
25         this.bairro = bairro;
26         this.estado = estado;
27         this.CEP = CEP;
28         this.titular = titular;
29     }
30
31     @Deprecated
32     public AddressRegister() {
33     }
34
35      ..... getters .....
36
37 }
```

Uma vez introduzido os relacionamentos vamos explorar com um pouco mais de cuidado o que o @OneToMany e @ManyToOne representa. Na classe de usuários, introduzimos o @OneToMany isso nos diz que o atributo titular possui um relacionamento de um para

muitos, ou seja um usuário pode ter mais de um endereço associado ao seu cpf e e-mail mas estes devem ser únicos.

Isso significa que podemos nos referir à entidade endereços como um tipo de ArrayList relacionado ao titular, para isso utilizamos o mappedBy como parâmetro no relacionamento @OneToMany. Já na classe de endereços utilizamos o @ManyToOne, podemos entender este relacionamento pela tradução literal muitos endereços pode pertencer a um usuário. Além desses relacionamentos nota-se a presença de dois construtores na classe de endereço e dois na classe usuários, falaremos deles agora.

Um construtor funciona como uma rotina de inicialização que é chamada sempre que um objeto é criado, forçando o usuário a passar argumentos para o objeto assim que ele é criado. O construtor também é útil para evitar a criação de diversos métodos set. O construtor vazio com o nome das classes anotado com @Deprecated diz ao programa que este elemento está obsoleto e não dever ser utilizado, evitando assim que o programa inicia com parâmetros vazios.

Com isso, nossas classes de usuários e de endereço estão prontas, mas nossa API ainda não. Até agora criamos nossas classes e usamos as anotações @Entity e @Id para mapear as classes como tabelas em um banco de dados, o próximo passo é criar uma interface UserRepository e AddressRepository que terão acesso aos métodos CRUD (Creat Read Update Delete) que ainda serão implementados. Implementamos os repositórios da seguinte forma:

```
1 public interface UserRepository extends JpaRepository<User,Long> {
2     boolean existsByEmail(String email);
3     boolean existsByCpf(String cpf);
4 }
```

```
1 public interface AddressRegisterRepository extends JpaRepository<
    AddressRegister,Long> {
2     List<AddressRegister> findByTitularId(Long id);
3 }
```

Como parâmetro foi passado as classes principais User e AddressRegister assim como seus tipos de identificador, em ambos os casos Long. Observe que na interface do usuário criamos dois boolean que recebem como atributo o email e o cpf, esses boolean serão uteis para verificar se tanto o email quanto o cpf cadastrado ainda não existem no banco de dados. A interface evita ter que criar uma nova classe responsável por acessar e manipular o banco de dados.

Aplicativos web se comunicam com o servidor usando a API, mencionamos isso no começo do texto, como sabemos essa comunicação é feita utilizando os métodos de requisição HTTP. Criaremos uma nova classe que ficará responsável por fazer essa comunicação entre o cliente e o servidor, dentro dessa classe utilizaremos os métodos POST, GET e suas anotações correspondentes @PostMapping e @GetMapping. As classes tanto para o usuário quanto para o endereço ficam:

```java
@RestController
@RequestMapping("users")
public class UserController {
    private final UserRepository repository;

    public UserController(UserRepository repository) {
        this.repository = repository;
    }

    @PostMapping
    public ResponseEntity<?>cadastrarUsuario(@RequestBody @Valid
       UserRequest usuario) {
        if (repository.existsByEmail(usuario.getEmail()) ||
           repository.existsByCpf(usuario.getCpf()) ) {
            return ResponseEntity.badRequest().build();

        }
        User salvo = repository.save(usuario.paraModelo());
        return ResponseEntity.status(HttpStatus.CREATED).body(new
           UserResponse(salvo));
    }

    @GetMapping
    public ResponseEntity<List<UserResponse>> mostrar() {
        return ResponseEntity.ok(repository.findAll().stream().map(
           UserResponse::new).collect(Collectors.toList()));
    }
}
```

```java
@RestController

public class AddressRegisterController {
    private final UserRepository userRepository;

    private final AddressRegisterRepository
       addressRegisterRepository;

    public AddressRegisterController(UserRepository userRepository,
       AddressRegisterRepository addressRegisterRepository) {
        this.userRepository = userRepository;
        this.addressRegisterRepository = addressRegisterRepository;
    }

    @PostMapping("enderecos")
    public ResponseEntity<AddressRegisterResponse> cadastrarEndereco
       (@RequestBody @Valid AddressRegisterRequest
       addressRegisterRequest) {
```

```
15          AddressRegister salvo = addressRegisterRepository.save(
                addressRegisterRequest.paraModelo(userRepository));
16
17          return ResponseEntity.status(HttpStatus.CREATED).body(new
                AddressRegisterResponse(salvo));
18      }
19
20      @GetMapping("users/{idUsuario}/enderecos")
21      public ResponseEntity<List<AddressRegisterResponse>>
            listarEnderecosUsuario(@PathVariable Long idUsuario) {
22          List<AddressRegisterResponse> enderecos =
                addressRegisterRepository.findByTitularId(idUsuario).
                stream().map(AddressRegisterResponse::new).collect(
                Collectors.toList());
23          return ResponseEntity.ok(enderecos);
24      }
25  }
```

Esses dois trechos de código possuem muita informação, então vamos por partes. Primeiro a anotação @RestCoontroller que está sempre localizada no começo da classe, ela declara que uma classe fornecerá os URL's solicitados para acessar os métodos REST. Dentro das classes nomeadas controller há as anotações @PostMapping e @GetMapping, caso você não lembre o POST é utilizado para criar dados e o GET para ler ou recuperar dados. Na classe que recebe o POST criamos uma declaração condicional, onde é verificado se os atributos email e cpf já existem ou não no banco de dados, ele só prossegue se ambos forem novos dados.

Note que, utilizamos duas novas anotações dentro do @PostMapping e @GetMapping, sendo elas:

- @ResquestBody: Converte a entrada JSON no objeto de entrada do método.
- @Valid: Diz ao Spring Boot que deve executar as validações do Hibernate validator no corpo da solicitação de acordo com as anotações especificadas.
- @PathVariable: Apartir dela é possível obter uma variável que esta no endereço do end-point (Path).

Observe também que estamos utilizando uma classe chamada ResponseEntity a qual nos permite manipular os dados HTTP da resposta. Fora isso, note que não estamos utilizando a classe principal User e AddressRegister mas sim uma classe auxiliar com o nome Response na frente, antes de falarmos sobre ela vamos introduzir uma nova classe responsável por receber e validar os dados de entrada antes desses dados serem enviados para o repositório e para a classe principal.

A classe Request é importante para manter os dados do usuário protegidos e evitar a entrada de dados inválidos no banco de dados. Assim, nossas classes são implementadas da seguinte forma:

```java
public class UserRequest {

    @NotBlank
    private String nome;

    @Email
    @NotBlank
    private String email;

    @CPF
    @NotBlank
    private String cpf;

    @NotBlank
    private String dataNascimento;


    public User paraModelo() {
        return new User(this.nome, this.email, this.cpf, this.
           dataNascimento);
    }

    .... getters .....
}
```

```java
public class AddressRegisterRequest {
    @NotNull
    private Long idTitular;

    @NotBlank
    private String logradouro;

    @NotNull
    private Integer numero;

    @NotBlank
    private String complemento;

    @NotBlank
    private String bairro;

    @NotBlank
    private String estado;

    @NotBlank
    private String CEP;

```

```
23      public AddressRegister paraModelo(UserRepository repository) {
24          User titular = repository.getById(idTitular);
25          return new AddressRegister(this.logradouro, this.numero,
                this.complemento, this.bairro, this.estado, this.CEP,
                titular);
26      }
27
28      .... getters ......
29  }
```

Novamente introduzimos novas anotações relacionadas à validações efetuadas pelo Hibernate Validator:

- @NotNull: Declara que um campo não pode ser nulo, mas pode ser de comprimento 0, como .
- @NotBlank: Declara que o valor não pode estar vazio.
- @Email: Verifica se o campo anotado contém o símbolo '@'.
- @CPF: Verifica se o campo anotado está de acordo com as regras da receita federal.

Essas anotações mostram quanto o Hibernate é um framework poderoso e útil, imagine quanto trabalho repetitivo teríamos para criar métodos que fizessem essas validações. Bem, agora podemos voltar para a classe response mencionada anteriormente, esta classe tem o intuito de proteger os dados do usuário evitando que estes sejam exibidos ao fazer um GET. A implementação foi feita da seguinte forma:

```
1   public class UserResponse {
2
3       private String nome;
4
5       private String email;
6
7       private String dataNascimento;
8
9       UserResponse(User user) {
10          this.nome = user.getNome();
11          this.email = user.getEmail();
12          this.dataNascimento = user.getDataNascimento();
13      }
14
15      ... getters ...
16  }
```

```
1   public class AddressRegisterResponse {
2
3       private String logradouro;
4
5       private Integer numero;
```

```
6
7       private String complemento;
8
9       private String bairro;
10
11      private String estado;
12
13      private String CEP;
14
15      AddressRegisterResponse(AddressRegister addressRegister) {
16          this.logradouro = addressRegister.getLogradouro();
17          this.numero = addressRegister.getNumero();
18          this.complemento = addressRegister.getComplemento();
19          this.bairro = addressRegister.getBairro();
20          this.estado = addressRegister.getEstado();
21          this.CEP = addressRegister.getCEP();
22      }
23          .... getters .....
24
25  }
```

Uma vez que implementamos todas essas classes, podemos finalmente rodar o projeto e verificar as resposta HTTP's. Lembrando que, ao criar um dados com sucesso através do POST devemos receber o código 201. Ao ler um dado corretamente o código 200, em caso de erros o código sera o 400 referente à um pedido ruim. Isso será feito utilizando o software de licença gratuita Insomnia. Logo, vamos cadastrar um novo usuário e verificar as respostas HTTP's obtidas:

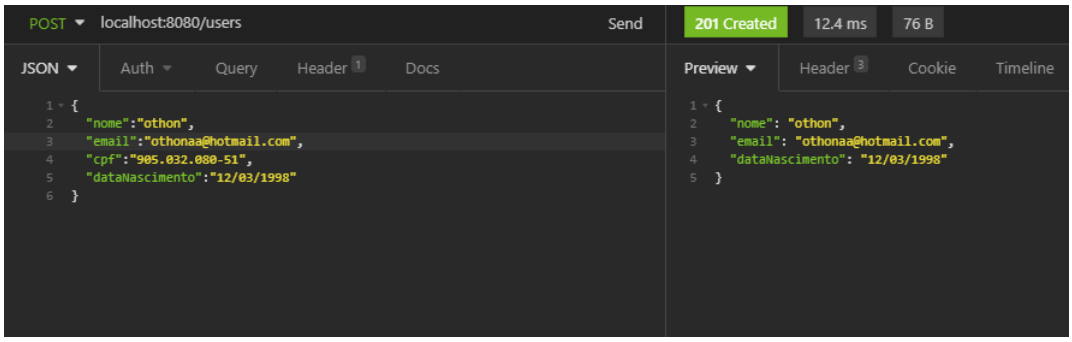

Figura 1: Cadastro de usuário através do Insomnia, com Status 201.

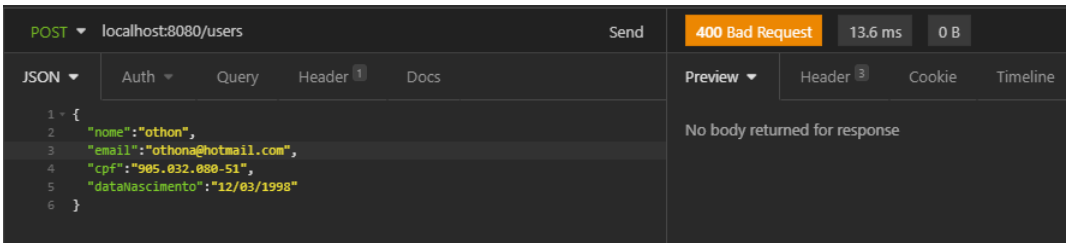

Figura 2: Cadastro invalido utilizando dados únicos repetidos, Status 400.

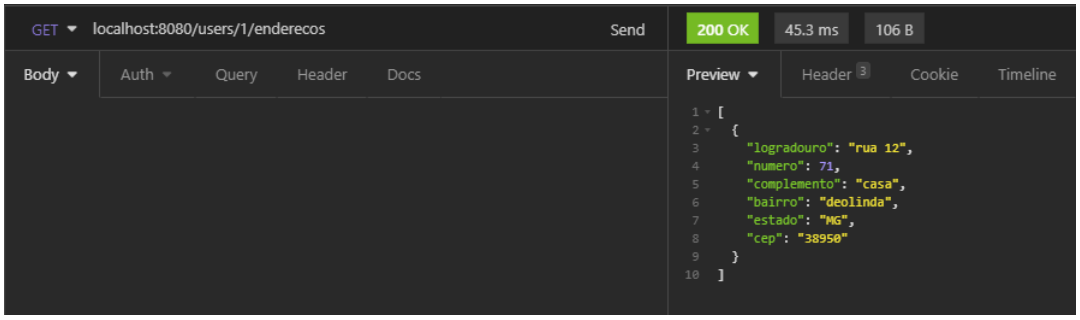

Figura 3: Consulta de endereços relacionados a um usuário correta, Status 200.

5 Conclusão

Neste post mostramos como criar uma API REST utilizando o framework Spring, no qual as dependências Spring Web, Spring Data JPA e Spring Validation mostraram ser frameworks poderosos para simplificar o trabalho mecânico do programador e permitir que o mesmo se concentre na lógica e objetivo da API . Na medida que as implementações são feitas é possível acompanhar as vantagens das dependências utilizadas, evitando várias vezes o erro humano durante a criação de métodos adicionais e repetitivos para realizar tarefas de validação ou criação de banco de dados.