

API REST para controle de veículos

autora: Giovanna Severo de Souza

Esses dias recebi um desafio do programa Orange-Talents, o desafio consiste na execução de um projeto de controle de veículos utilizando Spring como principal tecnologia. Então é isso hoje iremos implementar um API REST.



Antes de iniciar a implementação da aplicação é importante primeiro definirmos as tecnologias que iremos utilizar no projeto.

A linguagem de programação que utilizaremos será Java juntamente com o módulo Spring boot. Java é uma linguagem de programação de propósito geral e uma de suas maiores vantagens é o suporte para diversas aplicações. O uso de um framework como o spring facilita o processo de desenvolvimento, de forma a encapsular diversas configurações trazendo os benefícios de design patterns. O Spring boot é um módulo que permite a criação de microsserviços possibilitando o desenvolvimento de sistemas independentes, ampliando a utilização de tecnologia em uma aplicação.



Para o mapeamento objeto relacional incluiremos a especificação do JPA juntamente com Hibernate a partir do Spring Data JPA, de forma a facilitar e otimizar a persistência de dados no nosso projeto. Por fim, o banco de dados que utilizaremos será o PostgreSQL.



Definido as principais tecnologias vamos a implementação, para explicar o passo a passo vamos definir as seguintes etapas do desenvolvimento :

1. Criação e configuração do projeto
2. Mapeamento de Entidades
3. Criação de Services
4. Criação dos Endpoints

Criação e configuração do projeto

Para criação do projeto iremos utilizar o spring initializr, que facilita a inicialização e inclusão das dependências, sendo necessário somente selecionar o projeto, a linguagem e a versão do spring boot, no nosso caso iremos utilizar o Maven Project, Java 8 e spring boot 2.5.1.

The image is a screenshot of the Spring Initializr web application. It has a dark theme. At the top, there's a logo with a green leaf and the text 'spring initializr'. Below the logo, there are several sections: 'Project' with radio buttons for 'Maven Project' (selected) and 'Gradle Project'; 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for versions '2.6.0 (SNAPSHOT)', '2.5.2 (SNAPSHOT)', '2.5.1' (selected), and '2.4.8 (SNAPSHOT)', plus '2.4.7' and '2.3.12'; 'Project Metadata' with input fields for 'Group' (com.oragetalents), 'Artifact' (controleveiculos), 'Name' (controleveiculos), 'Description' (Projeto para controle de veiculos), and 'Package name' (com.oragetalents.controleveiculos); and 'Packaging' with radio buttons for 'Jar' (selected) and 'War'. At the bottom, there are radio buttons for 'Java' and versions '16', '11', and '8' (selected).

As dependências utilizadas serão spring-boot-starter-web, spring-boot-starter-data-jpa e postgresql e são declaradas no arquivo pom.xml. Ao gerar o projeto será feito o download do pacote jar e então só fazemos a importação para a IDE desejada, no nosso o Eclipse.

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

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
</dependency>
```

Devemos também fazer a criação do nosso banco de dados e incluir os dados de acesso no arquivo application.properties. Para fazer a criação no postgres é só rodar os comandos abaixo no terminal psql.

```
postgres=# create database controleveiculos
postgres=# create user orange with encrypted password 'banco123';
```

Mapeamento de Entidades

O projeto consiste no controle de veículos de usuários, dessa forma teremos duas entidades principais, Usuário e Veículo, que se relacionam a partir de um mapeamento one to many, nesse caso cada usuário poderá ter vários veículos e cada veículo está associado a um único usuário. Para realizar o mapeamento foram criadas duas classes de entidades que representam as tabelas criadas no banco de dados.

```
20 @Entity
21 @Table(name="usuario", schema = "clientes")
22 public class Usuario {
23     @Id
24     @Column(name="id_usuario")
25     @GeneratedValue(strategy = GenerationType.IDENTITY)
26     private Long id;
27
28     @Column(name="nome")
29     private String nome;
30
31     @Column(name="email", unique=true)
32     @NotNull
33     private String email;
34
35     @Column(name="cpf", unique=true)
36     @NotNull
37     private String cpf;
38
39     @Column(name="data_nascimento")
40     private Date dataNascimento;
41
42     @OneToMany
43     private Set<Veiculo> veiculos;
44
45 }
```

```

20 @Entity
21 @Table(name="veiculo",schema="clientes")
22 public class Veiculo {
23
24     @Id
25     @Column(name="id_veiculo")
26     @GeneratedValue(strategy = GenerationType.IDENTITY)
27     private Long id;
28
29     @Column(name="marca")
30     private String marca;
31
32     @Column(name="modelo")
33     private String modelo;
34
35     @Column(name="ano")
36     private Integer ano;
37
38     @Column(name="valor")
39     private String valor;
40
41     @ManyToOne
42     @JoinColumn(name="id_usuario",nullable=false)
43     private Usuario usuario;
44
45     @Transient
46     private boolean rodizioAtivo;
47
48     @Transient
49     private String diaRodizio;

```

Para definição da classe como uma entidade é necessário adicionar as annotations do pacote javax.persistence

- @Entity para indicar que se trata de um entidade
- @Table para referenciar o nome da tabela como está no banco
- @Column para o nome da coluna em cada atributo da classe
- @Id para indicar o atributo que representa a chave primária
- @GeneratedValue para definir a estratégia de geração de valor do id
- @ManyToOne para indicar o tipo de relação na chave estrangeira nos Veículos
- @OneToMany para indicar o tipo de relação no Usuário
- @JoinColumn para indicar o atributo que representa a chave estrangeira
- @Transient foi utilizado para representar um atributo que não é persistido no banco

No caso do domínio de usuário os campos email e cpf devem ser únicos e seu preenchimento é obrigatório, dessa forma foram indicados como @NotNull e Unique.

É importante lembrar que devem ser gerados os getters e setters de cada atributo para leitura e escrita das variáveis, assim como os construtores para criação do objeto.

Realizado a criação das entidades já temos nossas tabelas mapeadas no projeto java e já conseguimos criar os objetos para realização de operações, porém ainda não conseguimos consultar nem persistir dados no banco, para isso é necessário a criação dos Repositories que serão responsáveis por prover uma interface para as regras de negócio permitindo a obtenção e persistência dos objetos no banco de dados.

Como os campos email e cpf do usuário devem ser únicos foram criadas duas funções para verificar a existência do valor passado na requisição, essas funções executam uma query que derivam do nome do método, sendo assim o método existsByEmail e existsByCpf

verificam se já existe um registro no banco para o valor passado na função, esses métodos serão usados mais a frente na validação dos dados no método de salvar usuário.

```
7 public interface UsuarioRepository extends JpaRepository<Usuario, Long>{
8     boolean existsByEmail(String email);
9
10    boolean existsByCpf(String cpf);
11
12 }
```

Para os veículos foi criado um método que retorna todos os veículos associados a um usuário, a partir da execução da query declarada com @Query.

```
11 public interface VeiculoRepository extends JpaRepository<Veiculo, Long>{
12     @Query(value= "SELECT v FROM Veiculo v INNER JOIN v.usuario u where u.id =:id")
13     Set<Veiculo> findAllVeiculosOfUsuario(@Param("id") Long id);
14 }
15
```

Criação de Services

Agora que já possuímos nosso modelo de negócio definido, com nossas entidades mapeadas podemos implementar as regras de negócio, para isso realizaremos a criação das nossas classes de services.

Para as classes de Service adicionamos a notação @Service , e implementamos os métodos definidos na interface, devemos criar uma variável do nosso repository para termos acesso aos métodos que fazem as consultas e persistências e assim criar as validações para as regras de negócio.

Primeiro definimos a interface com as funções que serão implementadas na classe service, para o service de usuário foram criados os métodos de salvarUsuario, validarEmail, validarCpf e findById.

```
5 public interface UsuarioService {
6
7     Usuario salvarUsuario(Usuario usuario);
8
9     void validarEmail(String email);
10
11    void validarCpf(String cpf);
12
13    Usuario findById(Long id);
14 }
```

O método de salvarUsuario verifica se as regras de negócio são atendidas antes de chamar o método que irá persistir o registro de usuário no banco, dessa forma devemos chamar os métodos de validarEmail e validarCpf, pois precisamos verificar se já não existem outros usuários com essas informações de email e cpf. Com isso esses métodos chamam as funções declaradas no nosso repository de usuário que nos retorna essa

informação e assim caso exista podemos disparar o Exception criado para Regra de negócio com a mensagem adequada do erro.

O método findById é responsável por validar a existência de um usuário no banco de dados dessa forma caso o nosso repository não retorne nenhum registro disparamos um Exception do tipo Not Found.

```
14 @Service
15 public class UsuarioServiceImpl implements UsuarioService {
16
17     private UsuarioRepository repository;
18
19     @Autowired
20     public UsuarioServiceImpl(UsuarioRepository repository) {
21         super();
22         this.repository = repository;
23     }
24
25     @Override
26     @Transactional
27     public Usuario salvarUsuario(Usuario usuario) {
28         validarEmail(usuario.getEmail());
29         validarCpf(usuario.getCpf());
30         return repository.save(usuario);
31     }
32
33     @Override
34     public void validarEmail(String email) {
35         if(repository.existsByEmail(email))
36             throw new RegraNegocioException("Já existe um usuário cadastrado para o email informado");
37     }
38
39     @Override
40     public void validarCpf(String cpf) {
41         if(repository.existsByCpf(cpf))
42             throw new RegraNegocioException("Já existe um usuário cadastrado para o cpf informado");
43     }
44
45     @Override
46     public Usuario findById(Long id) {
47         if(repository.findById(id).isPresent())
48             return repository.findById(id).get();
49         else
50             throw new RegistroNaoEncontradoException("Usuário não cadastrado na base de dados");
51     }
52
53 }
```

Já para o service de veículos foram criados quatro métodos, salvarVeiculo, buscaVeiculosUsuario, buscaFipe, buscaMarca, buscaModelo e buscaAno. Para os veículos foi definido que o campo valor deve ser preenchido de acordo com as informações da tabela FIPE, para recuperar essas informações iremos consultar uma API externa, essa consulta é feita através dos métodos buscaFipe, buscaMarca, buscaModelo e buscaAno, para recuperar o valor do veículo desejado é necessário executar uma série de chamadas uma vez que precisamos do código da marca, modelo e ano do veículo para termos acesso ao valor. Para consultar a API da FIPE podemos utilizar o RestTemplate uma aplicação fornecida pelo Spring que permite a abstração de web client, assim criamos uma classe para mapear cada endpoint que iremos fazer as requisições e usamos a função getForEntity passando o endpoint e a classe que criamos, lembrando que é preciso fazer a criação de uma classe RestTemplateConfiguration do tipo @Configuration e construtor do tipo @Bean para termos acesso às funcionalidades .

Uma vez que conseguimos consultar as informações necessárias na tabela FIPE podemos criar as validações dentro do método de salvarVeiculo, para caso de qualquer uma das informações passadas não retornar um veículo existente na FIPE sejam lançados os Exceptions de regra de negócio com a mensagem adequada.

Por último o método `buscaVeiculosUsuarios` chama a função `findAllVeiculosOfUsuario` do nosso repository de veículos que retorna a lista de veículos cadastrados para o usuário informado e preenche as informações do dia de rodízio de acordo com o último dígito do ano do veículo, e informação do dia de rodízio ativo que informa se o dia de hoje é o dia de rodízio daquele veículo.

```
13 public interface VeiculoService {
14     Veiculo salvarVeiculo(Veiculo veiculo);
15
16     VeiculoFipe buscaFipe(String codModelo, String codMarca, String ano);
17
18     VeiculoMarca[] buscaMarca();
19
20     VeiculoModelo[] buscaModelo(String codMarca);
21
22     VeiculoAno [] buscaAno(String codModelo,String codMarca);
23
24     Set<Veiculo> buscaVeiculosUsuario(Usuario usuario);
25
26 }
27
28
```

```
23 @Service
24 public class VeiculoServiceImpl implements VeiculoService{
25
26     private VeiculoRepository repository;
27
28     @Autowired
29     private RestTemplate restTemplate;
30
31
32     public VeiculoServiceImpl(RestTemplate restTemplate, VeiculoRepository repository) {
33         this.restTemplate = restTemplate;
34         this.repository = repository;
35     }
36
37     @Override
38     public Veiculo salvarVeiculo(Veiculo veiculo) {
39         VeiculoMarca[] veiculosMarcas = buscaMarca();
40         VeiculoMarca marca = (VeiculoMarca) Arrays.stream(veiculosMarcas)
41             .filter(veiculoMarca -> veiculo.getMarca().equals(veiculoMarca.getNome()))
42             .findAny()
43             .orElse(null);
44
45         if(marca == null)
46             throw new RegraNegocioException("Não foi possível encontrar as informações de veículo para a marca informada");
47
48         VeiculoModelo[] veiculoModelos = buscaModelo(marca.getCodigo());
49         VeiculoModelo modelo = (VeiculoModelo) Arrays.stream(veiculoModelos)
50             .filter(veiculoModelo -> veiculo.getModelo().equals(veiculoModelo.getNome()))
51             .findAny()
52             .orElse(null);
53
54         if(modelo == null)
55             throw new RegraNegocioException("Não foi possível encontrar as informações de veículo para o modelo informado");
56
57         VeiculoAno[] veiculoAnos = buscaAno(modelo.getCodigo(), marca.getCodigo());
58         VeiculoAno ano = (VeiculoAno) Arrays.stream(veiculoAnos)
59             .filter(veiculoAno -> veiculo.getAno().toString().equals(veiculoAno.getNome().substring(0, veiculoAno.getNome().indexOf(" "))))
60             .findAny()
61             .orElse(null);
62
63         if(ano == null)
64             throw new RegraNegocioException("Não foi possível encontrar as informações de veículo para o ano informado");
65
66         VeiculoFipe veiculoFipe = buscaFipe(modelo.getCodigo(), marca.getCodigo(),ano.getCodigo().toString());
67
68         veiculo.setMarca(veiculoFipe.getMarca());
69         veiculo.setModelo(veiculoFipe.getModelo());
70         veiculo.setAno(veiculoFipe.getAnoModelo());
71         veiculo.setValor(veiculoFipe.getValor());
72
73         return repository.save(veiculo);
74     }
75
76 }
```



```

76 @Override
77 public VeiculoFipe buscaFipe(String codModelo, String codMarca, String ano) {
78     ResponseEntity<VeiculoFipe> responseEntity = this.restTemplate.getForEntity("https://parallelum.com.br/fipe/api/v1/carros/marcas/" + codMarca + "/modelos/" + codModelo + "/anos/" + ano, VeiculoFipe.class);
79     VeiculoFipe veiculo = responseEntity.getBody();
80     return veiculo;
81 }
82
83 @Override
84 public VeiculoMarca[] buscaMarca() {
85     ResponseEntity<VeiculoMarca[]> responseEntity = this.restTemplate.getForEntity("https://parallelum.com.br/fipe/api/v1/carros/marcas", VeiculoMarca[].class);
86     VeiculoMarca[] veiculosMarcas = responseEntity.getBody();
87     return veiculosMarcas;
88 }
89
90 @Override
91 public VeiculoModelo[] buscaModelo(String codMarca) {
92     ResponseEntity<ObjetoModelo> responseEntity = this.restTemplate.getForEntity("https://parallelum.com.br/fipe/api/v1/carros/marcas/" + codMarca + "/modelos", ObjetoModelo.class);
93     ObjetoModelo objetoModelos = responseEntity.getBody();
94     VeiculoModelo[] veiculoModelos = objetoModelos.getModelo();
95     return veiculoModelos;
96 }
97
98 @Override
99 public VeiculoAno[] buscaAno(String codModelo, String codMarca) {
100     ResponseEntity<VeiculoAno[]> responseEntity = this.restTemplate.getForEntity("https://parallelum.com.br/fipe/api/v1/carros/marcas/" + codMarca + "/modelos/" + codModelo + "/anos", VeiculoAno[].class);
101     VeiculoAno[] veiculoAnos = responseEntity.getBody();
102     return veiculoAnos;
103 }
104
105 @Override
106 public Set<Veiculo> buscaVeiculosUsuario(Usuario usuario) {
107     Set<Veiculo> veiculos = repository.findAllVeiculosOfUsuario(usuario.getId());
108     for (Veiculo veiculo : veiculos) {
109         int numeroAno = veiculo.getAno() % 10;
110         switch (numeroAno) {
111             case 0 :
112             case 1:
113                 veiculo.setDiaRodizio("Segunda-feira");
114                 break;
115
116             case 2:
117             case 3:
118                 veiculo.setDiaRodizio("Terça-feira");
119                 break;
120
121             case 4:
122             case 5:
123                 veiculo.setDiaRodizio("Quarta-feira");
124                 break;
125
126             case 6:
127             case 7:
128                 veiculo.setDiaRodizio("Quinta-feira");
129                 break;
130
131             case 8:
132             case 9:
133                 veiculo.setDiaRodizio("Sexta-feira");
134                 break;
135         }
136
137         String diaDaSemana = new DateFormatSymbols().getWeekdays()[Calendar.getInstance().get(Calendar.DAY_OF_WEEK)];
138         if(veiculo.getDiaRodizio().equals(diaDaSemana))
139             veiculo.setRodizioAtivo(true);
140         else
141             veiculo.setRodizioAtivo(false);
142     }
143     return veiculos;
144 }

```

Criação dos Endpoints

Por fim, criamos a nossa camada de Controllers, onde serão declarados os nossos Endpoints Rest. Nessa camada são manipuladas as requisições Http. Desta forma receberemos um objeto do tipo Json e transformamos em um objeto java.

Assim implementamos uma classe DTO (data transfer object) que faz esse mapeando da informação recebida. Criamos um DTO para mapear cada entidade.

```

5 public class UsuarioDTO {
6     String nome;
7     String email;
8     String cpf;
9     Date dataNascimento;
10 }

```



```

4 public class VeiculoDTO {
5
6     private Long usuarioId;
7
8     private String marca;
9
10    private String modelo;
11
12    private Integer ano;
13

```

Nas nossas classes de controller devemos adicionar as annotations `@RestController` para indicar que o controller é do tipo Rest e a annotation `@RequestMapping` em que declaramos a uri das requisições que serão criadas neste controller.

Na classe `UsuarioController` devemos então criar dois métodos do tipo `ResponseEntity`, o método salvar que será do tipo `Post` e por isso devemos adicionar a annotation `@PostMapping`, nesse método criamos um objeto do tipo `Usuario` passando no construtor os parâmetros do nosso DTO, que recebemos na função com a annotation `@RequestBody` para indicar que está mapeado de acordo com o corpo da nossa requisição que nesse caso é um objeto `Json`. Com isso podemos chamar o método `salvarUsuario` do `UsuarioService` passando as informações recebidas na requisição e caso passe em todas as validação retornamos `HttpStatus` do tipo `201 CREATED`, como resposta de sucesso e caso contrário retornamos um `badRequest 400` com a mensagem do erro.

O segundo método do `UsuarioController` será uma requisição do tipo `GET`, para isso adicionamos a annotation `@GetMapping` informando o parâmetro de busca, assim podemos então chamar o método `findById` do `UsuarioService` e o método `buscaVeiculosUsuario` do `VeiculoService`, e caso a consulta ocorra com sucesso retornamos o `HttpStatus` do tipo `202 ACCEPTED`, da mesma forma caso contrário retornamos um `badRequest`.

```

23 @RestController
24 @RequestMapping("/api/usuario")
25 public class UsuarioController {
26
27     private UsuarioService service;
28     private VeiculoService veiculoService;
29
30     public UsuarioController(UsuarioService service, VeiculoService veiculoService){
31         this.service = service;
32         this.veiculoService = veiculoService;
33     }
34
35     @PostMapping
36     public ResponseEntity salvar(@RequestBody UsuarioDTO dto) {
37         Usuario usuario = new Usuario(dto.getNome(), dto.getEmail(), dto.getCpf(), dto.getDataNascimento());
38         try {
39             service.salvarUsuario(usuario);
40             return new ResponseEntity(usuario, HttpStatus.CREATED);
41         } catch (RegraNegocioException e) {
42             return ResponseEntity.badRequest().body(e.getMessage());
43         }
44     }
45
46     @GetMapping(value="{id}")
47     public ResponseEntity ListaUsuarioUnico(@PathVariable("id") Long id) {
48         Usuario usuario = null;
49         try {
50             usuario = service.findById(id);
51             usuario.setVeiculos(veiculoService.buscaVeiculosUsuario(usuario));
52             return new ResponseEntity(usuario, HttpStatus.ACCEPTED);
53         } catch (RegraNegocioException e) {
54             return ResponseEntity.badRequest().body(e.getMessage());
55         }
56     }
57 }

```

Para o VeiculoController criaremos somente o método de salvar seguindo o mesmo padrão utilizado no UsuarioController.

```
20 @RestController
21 @RequestMapping("/api/veiculo")
22 public class VeiculoController {
23     private VeiculoService service;
24     private UsuarioService usuarioService;
25
26     public VeiculoController(VeiculoService service, UsuarioService usuarioService) {
27         this.service = service;
28         this.usuarioService = usuarioService;
29     }
30
31     @PostMapping
32     public ResponseEntity salvar(@RequestBody VeiculoDTO dto) {
33         Usuario usuario = usuarioService.findById(dto.getUsuarioId());
34         Veiculo veiculo = new Veiculo(dto.getMarca(), dto.getModelo(), dto.getAno(), usuario);
35         try {
36             service.salvarVeiculo(veiculo);
37             return new ResponseEntity(veiculo, HttpStatus.CREATED);
38         } catch (RegraNegocioException e) {
39             return ResponseEntity.badRequest().body(e.getMessage());
40         }
41     }
42 }
```

O projeto desenvolvido está disponível no github através do link:

<https://github.com/giovanna96/API-REST-Controle-Veiculos>