

Gerenciando o rodízio de vagas para veículos utilizando Spring e Java

Victor Emanuel - 24/05/2021

Suponha que queremos gerenciar quais pessoas podem pôr seus veículos numa determinada garagem. Não há vagas para todo mundo, então as pessoas fazem um rodízio de automóveis. Dependendo do dia da semana, determinados carros podem usar a garagem e outros não.

Precisaremos cadastrar cada um dos usuário e seus veículos em nosso sistema. Para decidir quais dias um veículo pode ficar estacionado, utilizaremos apenas o último dígito do ano do modelo do carro, assim:

Final 0 ou 1: segunda-feira

Final 2 ou 3: terça-feira

Final 4 ou 5: quarta-feira

Final 6 ou 7: quinta-feira

Final 8 ou 9: sexta-feira

A API que faremos deverá listar todos os veículos de um usuário mostrando detalhes como modelo, ano, marca, valor e outros dois campos referentes ao uso das vagas: um informando seu dia de rodízio e um booleano indicando se pode usar uma vaga naquele dia em que a consulta foi feita.

Para exemplificar, suponha que hoje é segunda-feira, o carro é da marca Fiat, modelo Uno do ano de 2001, ou seja, seu rodízio será às segundas-feiras e o atributo de rodízio ativo será TRUE.

Tecnologias

Java, Kotlin ou Groovy

O Spring Framework nos dá 3 opções de linguagem para trabalhar, todas elas muito boas. Então, como escolher se, na prática, qualquer uma delas serve para resolver nosso problema? Geralmente, uma linguagem é escolhida em termos da performance, tempo de uso, quantidade de programadores disponíveis no mercado e até mesmo na quantidade material de suporte encontrado na internet... como este artigo!

Devido ao caráter introdutório deste texto, escolhi a linguagem que mais se adequa aos itens anteriores, a prova viva de que uma tecnologia não é ruim só porque é feita há muito tempo, o bom e velho Java. E já que o Spring permite, usaremos a versão LTS mais recente, o Java 11, para aproveitar o que há de mais atual na linguagem.

Hibernate e H2 Database

Usarei o Hibernate em conjunto com banco de dados H2, duas tecnologias amplamente usadas em projetos Springs. O Hibernate será importante para tornar a manipulação com o banco menos trabalhosa e mais amigável ao público iniciante. Isso porque ele fornece várias das funções mais comuns já implementadas e também nos poupa de escrever comandos SQL para tarefas simples. Porém, sua maior vantagem está no fato

de ser uma camada que abstrai o acesso aos dados, podendo trabalhar em conjunto com vários SGBDs e permitindo a eventual migração de um para outro.

O H2 foi escolhido por ser um dos banco de dados mais simples de ser utilizado no Spring Boot. Como consequência da simplicidade, é uma boa escolha quando estamos começando com Spring ou só estamos criando o protótipo de uma ideia. Ele não exige configuração alguma, basta apenas adicioná-lo às dependências do projeto, e é o mais leve das opções disponíveis. Mas calma, o H2 por padrão guarda seus dados na memória RAM, ou seja, eles não serão persistidos entre execuções do programa, tudo é apagado ao desligar o backend. Mesmo assim, serve para prototipar nossa aplicação, afinal, podemos migrar para outro banco posteriormente.

Spring Cloud Feign

Nosso backend precisa se comunicar com uma API REST. Uma forma bem prática de se fazer isso é usando o projeto Spring Cloud Feign. Ele nos permite criar as requisições por meio da definição de funções numa interface, mais ou menos como fazemos com os `@Repository`. A diferença é que os dados vêm de outra aplicação web ao invés do banco de dados local. Outra funcionalidade do Feign é que ele pode construir objetos de classes locais a partir dos JSONs recebidos, desde que tenham os mesmos campos.

Por que não usar Lombok

Optei por **não** usar Lombok neste projeto por um único motivo. Ele é uma ferramenta muito útil para esconder o código boilerplate dos Getters, Setters e construtores, mas esta vantagem na redução do código visível pode se tornar um problema para quem está iniciando sua carreira.

Ao ler códigos de minha autoria e de terceiros, vi que em alguns casos as anotações eram usadas indevidamente. Notei que sempre tive preferência por usar o `@Data` por resolver na maioria dos casos, mas sem perceber que, às vezes, estava gerando construtores e outras funções que nunca eram usadas, além de métodos acessores em variáveis que não precisavam (ou sequer deviam) ser acessadas.

Usar o Lombok *corretamente* exige assumir a responsabilidade de usar anotações que gerem exatamente o código necessário, o que requer um conhecimento aprofundado na sintaxe da ferramenta. Infelizmente, alguns programadores inadvertidamente usam-o sem saber exatamente o que está adicionando ao código. Resolvi assumir essa responsabilidade apenas futuramente, talvez quando estiver mais seguro na linguagem.

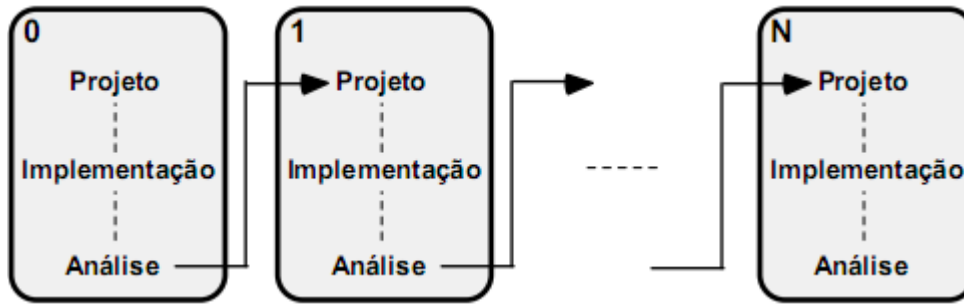
Implementação

Modelo de desenvolvimento

Quem tem experiência profissional na área tecnológica sabe que, ao lidar com clientes, não é bom deixá-lo esperando tanto tempo até ver seu produto na versão final. E, ao trabalhar com projetos ambiciosos, não tem jeito, é impossível fazer tudo de uma vez. Na nossa área, precisamos saber dividir o problema em problemas menores, tal qual faz o computador executando o merge sort.

Com o problema em vista, procurei iniciar pela sua funcionalidade principal, aquela que exigisse o mínimo possível de implementação, para iniciar e, a partir disso, ir trabalhando em cima adicionando uma funcionalidade de cada vez até que atenda as especificações desejadas. Este é o modelo **iterativo e incremental**. Cada incremento vai adicionando ao sistema novas capacidades funcionais, até a obtenção do

sistema final. É como executar vários “miniprojetos” onde cada um adiciona novas funcionalidades no programa até que o mesmo esteja completo.



Primeiramente, notei que nada poderia ser feito sem cadastrar usuários nem veículos. Por tanto, deveria começar com um dos dois. O cadastro de veículo exige que consultemos dados numa API externa, o que requer um esforço a mais, logo foi melhor começar pelo cadastro do usuário. Isso caracterizou uma iteração do projeto. Com a iteração 1, testada e pronta, o passo seguinte foi programar o cadastro do veículo, ainda sem os detalhes vindos da FIPE. Eis outra iteração. Abaixo deixo em detalhes o que foi feito em cada iteração:

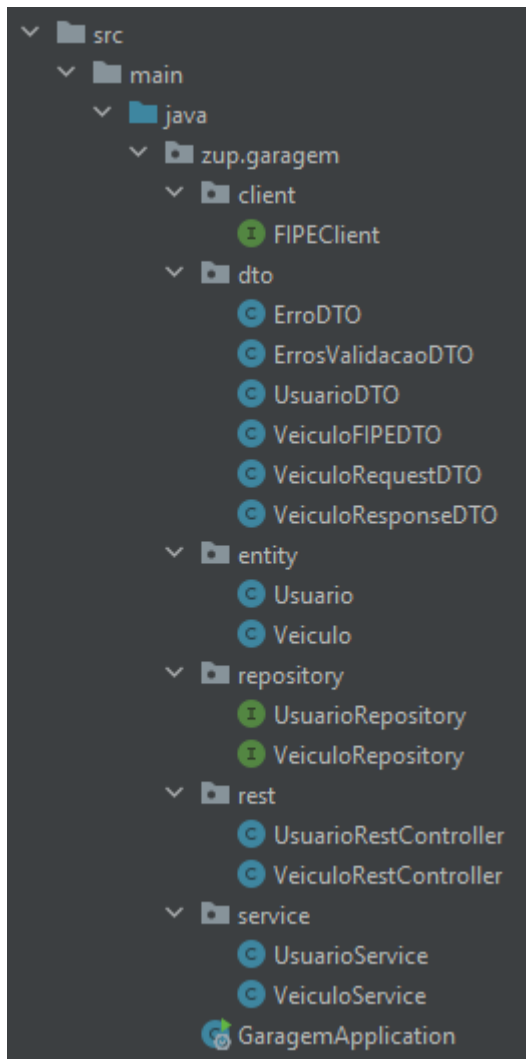
- Iteração 1: Cadastro do cliente com validação
- Iteração 2: Tratamento dos erros de validação do cliente
- Iteração 3: Cadastro e tratamento de erros do veículo sem dados da FIPE
- Iteração 4: Implementação do FeignClient e obtenção dos dados da FIPE no cadastro do veículo
- Iteração 5: Tratamento de erro em caso de falha na comunicação com a API da FIPE
- Iteração 6: Associação de veículo cadastrado no usuário
- Iteração 7: Implementa cálculo do dia do rodízio no DTO de resposta do veículo
- Iteração 8: Refatorações de código e arquiteturais, para refinar a legibilidade

Tudo isso fará sentido no decorrer da leitura do artigo.

Estrutura de pastas

Os arquivos foram separados nas seguintes pastas:

- **entity**: definições das tabelas do banco de dados;
- **repository**: classes responsáveis por recuperar dados no banco;
- **service**: classes responsáveis por utilizar os repositories e tratar os dados que estes enviam e recebem;
- **rest**: controladores REST. São responsáveis por receber as requisições HTTP, decidir o que fazer com os dados recebidos e o que mandar como resposta. Quando um dado precisam ser salvos ou recuperado do banco, eles usam o service adequado para tal;
- **client**: clientes REST de API externas. Neste projeto, é usado apenas um para consultar o preço de determinados veículos;
- **dto**: objetos de transferência de dados. É onde estão definidos o conteúdo dos objetos recebidos e enviados pela nossa API.



Falarei mais sobre cada uma delas no decorrer do artigo.

Entidades

As duas entidades (ou tabelas) criadas foram **Usuario** e **Veiculo**. No Spring, essas classes são construídas de forma que tenham apenas:

- Definição das colunas;
- Construtor com todas variáveis;
- Construtor vazio;
- Métodos getters.

No Usuário, temos além do ID os campos:

- **String nome**: não nulo;
- **String email**: não nulo e único;
- **String cpf**: não nulo e único;
- **Date dataNascimento**: pode ser nulo.

```
package zup.garagem.entity;  
  
// Imports
```

```
@Entity
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    private String nome;

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

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

    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy")
    private Date dataNascimento;

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

    public Usuario() {
    }

    public Long getId() {
        return id;
    }

    public String getNome() {
        return nome;
    }

    public String getEmail() {
        return email;
    }

    public String getCpf() {
        return cpf;
    }

    public Date getDataNascimento() {
        return dataNascimento;
    }
}
```

Já no veículo, temos além do ID:

- `String marca`: marca do veículo. Não pode ser nulo;

- `String modelo`: modelo do veículo. Não pode ser nulo;
- `String anoModelo`: código do ano que deve ser enviado para API da FIPE. Por exemplo, "2020-1". Não pode ser nulo;
- `String valor`: preço do veículo com o 'R\$' na frente. Pode ser nulo.
- `DayOfWeek diaRodizio`: dia da semana no qual o carro pode ocupar a vaga. Não pode ser nulo.
- `Usuario usuarioDono`: o dono do veículo. Essa coluna na verdade guarda apenas o ID de um usuário existente na tabela correspondente.

```
package zup.garagem.entity;

// Imports

@Entity
public class Veiculo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    private String marca;

    @NotNull
    private String modelo;

    @NotNull
    private String anoModelo;

    private String valor;

    @ManyToOne
    @JoinColumn(name = "usuario_id", nullable = false)
    private Usuario usuarioDono;

    private DayOfWeek diaRodizio;

    public Veiculo() {
    }

    public Veiculo(Long id,
                    String marca,
                    String modelo,
                    String anoModelo,
                    String valor,
                    Usuario usuarioDono,
                    DayOfWeek diaRodizio) {
        this.id = id;
        this.marca = marca;
        this.modelo = modelo;
        this.anoModelo = anoModelo;
        this.valor = valor;
        this.usuarioDono = usuarioDono;
        this.diaRodizio = diaRodizio;
    }
}
```

```
    }

    public Long getId() {
        return id;
    }

    public String getMarca() {
        return marca;
    }

    public String getModelo() {
        return modelo;
    }

    public String getAnoModelo() {
        return anoModelo;
    }

    public String getValor() {
        return valor;
    }

    public Usuario getUsuarioDono() {
        return usuarioDono;
    }

    public DayOfWeek getDiaRodizio() {
        return diaRodizio;
    }
}
```

Aqui, o emprego do `@ManyToOne` serve para mapear vários veículos a um usuário. Para identificar qual anotação usar, eu uso o macete a seguir. Tanto a anotação anterior quanto a `@OneToMany` podem ser interpretadas de forma similar: a primeira palavra se refere à classe à qual pertence e a última palavra à variável que está logo abaixo. Logo, no código acima estou fazendo com que haja "Many Veiculo para One usuarioDono".

Note que não há sentido em persistir no banco o booleano indicando se o carro pode ocupar a vaga no dia de hoje, pois este valor varia todo dia. Decidi por fazer seu cálculo somente se um usuário da API requisitá-lo.

DTOs

Mostrarei exemplos dos DTOs mais interessantes do projeto. A ideia por trás deles é definir quais dados devem ser recebidos na requisição e quais devem ser enviados nas respostas. Eles servem como uma interface de dados para os utilizadores da nossa API, pois nem sempre queremos mandar todos os campos das tabelas ao nosso banco.

É uma boa prática dividir o DTO em duas versões, uma com os campos que devemos receber e outra com os campos que queremos enviar, mas nem sempre isso é necessário. Na nossa API, somente a entidade Veículo precisou ter um `VeiculoRequestDTO` e um `VeiculoResponseDTO`. O primeiro é usado para realizar o cadastro de um novo veículo no banco e conta apenas com os campos de ID, tais como: `marcaId`, `modeloId`,

`anoModelo`, `usuarioId`; O segundo é o objeto retornado na listagem de veículos de um usuário, logo, contém o devido nome da marca, modelo, ano e o valor.

Em geral, os DTOs são classes com o objetivo principal de guardar valores e não devem conter funções significativas, muito menos regras de negócios. Neste projeto, fiz com que eles tivessem apenas dois construtores e os métodos getters. A ausência de setters obriga que eles sejam imutáveis.

```
package zup.garagem.dto;

import com.fasterxml.jackson.annotation.JsonFormat;
import org.hibernate.validator.constraints.Length;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotNull;
import java.util.Date;

public class UsuarioDTO {
    private Long id;

    @NotNull(message = "Nome é obrigatório")
    @Length(min = 3, max = 100, message = "Nome deve ter de 3 a 100 caracteres")
    private String nome;

    @NotNull(message = "Email é obrigatório")
    @Email(message = "Email inválido")
    private String email;

    @NotNull(message = "CPF é obrigatório")
    @Length(min = 11, max = 11, message = "CPF deve ter exatamente 11 dígitos")
    private String cpf;

    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy")
    private Date dataNascimento;

    public UsuarioDTO(Long id, String nome, String email, String cpf, Date dataNascimento) {
        this.id = id;
        this.nome = nome;
        this.email = email;
        this.cpf = cpf;
        this.dataNascimento = dataNascimento;
    }

    public UsuarioDTO() {
    }

    public Long getId() {
        return id;
    }

    public String getNome() {
        return nome;
    }
}
```



```
    }

    public String getEmail() {
        return email;
    }

    public String getCpf() {
        return cpf;
    }

    public Date getDataNascimento() {
        return dataNascimento;
    }
}
```

Notar que é nele que declaramos os critérios usados para fazer a validação dos valores e definimos as mensagens que serão retornadas em caso de inadequação. Para citar exemplos, definimos que o nome do usuário cadastrado deve conter de 3 a 100 caracteres e é obrigatório. Também obrigamos que haja um CPF e que seja fornecido apenas seus 11 dígitos.

Foi criado inclusive um DTO para as mensagens de erros de validação. Alguns dos campos contidos nele, como `timestamp`, `status` e `path`, foram incluídos para imitar a resposta padrão do Spring quando ocorre erro 500.

```
package zup.garagem.dto;

import org.springframework.http.HttpStatus;
import org.springframework.validation.BindingResult;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class ErrosValidacaoDTO {
    private final String timestamp;
    private final Integer status;
    private final List<ErroDTO> errors = new ArrayList<>();
    private final String message;
    private final String path;

    public ErrosValidacaoDTO(BindingResult resultadoValidacao, HttpStatus status,
String message, String path) {
        this.message = message;
        this.path = path;
        this.status = status.value();
        this.timestamp = new Date().toString();

        for (var e : resultadoValidacao.getFieldErrors()) {
            errors.add(new ErroDTO(e.getField(), e.getDefaultMessage()));
        }
    }
}
```

```
// Getters  
}
```

Vemos que o construtor recebe o resultado da validação contendo os erros e estes são adicionados em `List<ErroDTO> errors`. `ErroDTO`, por sua vez, serve para fazer um objeto com o título do erro e uma mensagem associada, como vemos a seguir.

```
package zup.garagem.dto;  
  
public class ErroDTO {  
    private final String titulo;  
    private final String mensagem;  
  
    public ErroDTO(String titulo, String mensagem) {  
        this.titulo = titulo;  
        this.mensagem = mensagem;  
    }  
  
    public String getTitulo() {  
        return titulo;  
    }  
  
    public String getMensagem() {  
        return mensagem;  
    }  
}
```

O objeto JSON gerado pelo `ErrosValidacaoDTO` tem esse formato:

```
{  
  "timestamp": "2021-05-22T19:01:26.349468800-03:00[America/Sao_Paulo]",  
  "status": 400,  
  "errors": [  
    {  
      "titulo": "email",  
      "mensagem": "Email inválido"  
    },  
    {  
      "titulo": "cpf",  
      "mensagem": "CPF deve ter exatamente 11 dígitos"  
    }  
  ],  
  "message": "Erro ao validar usuário",  
  "path": "/usuarios"  
}
```

E a resposta padrão do Spring é:

```
{
  "timestamp": "2021-05-23T00:13:28.531+00:00",
  "status": 500,
  "error": "Internal Server Error",
  "message": "[...]",
  "path": "/veiculos"
}
```

REST controllers

REST controllers (ou controladores REST) são os responsáveis *basicamente* por receberem requisições HTTP de aplicações externas, decidir o que fazer com os dados recebidos e retornar uma resposta adequada para os clientes.

Vamos começar com o mais simples, o `UsuarioRestController`.

```
@RestController
@RequestMapping("/usuarios")
public class UsuarioRestController {
    private final UsuarioService usuarioService;
    private final VeiculoService veiculoService;

    public UsuarioRestController(UsuarioService usuarioService, VeiculoService
veiculoService) {
        this.usuarioService = usuarioService;
        this.veiculoService = veiculoService;
    }

    @GetMapping
    public ResponseEntity<List<UsuarioDTO>> listar() {
        var usuariosDTO = usuarioService.findAllDTO();
        return ResponseEntity.ok(usuariosDTO);
    }

    @PostMapping
    public ResponseEntity<?> criar(@Validated @RequestBody UsuarioDTO u,
BindingResult result) {
        if (result.hasErrors()) {
            ErrosValidacaoDTO erro = new ErrosValidacaoDTO(result,
HttpStatus.BAD_REQUEST, "Erro ao validar usuário", "/usuarios");
            return ResponseEntity.badRequest().body(erro);
        }

        var novoUsuarioDTO = usuarioService.salvar(u);
        return ResponseEntity.status(HttpStatus.CREATED).body(novoUsuarioDTO);
    }

    @GetMapping("/{id}/veiculos")
    public ResponseEntity<?> listarVeiculosDoUsuario(@PathVariable Long id) {
        usuarioService.findById(id); // emite erro se usuário não existe
    }
}
```

```
        List<VeiculoResponseDTO> veiculosDTO =  
        veiculoService.findAllDTODByUsuarioDonoId(id);  
        return ResponseEntity.ok(veiculosDTO);  
    }  
}
```

Foram criadas as funções:

- `listar()`, para listar todos na tabela Usuario;
- `criar()`, para criar novo usuário;
- `listarVeiculosDoUsuario()`, para retornar todos os veículos de um usuário.

A primeira delas, `ResponseEntity<List<UsuarioDTO>> listar()`, é facilmente entendida, usamos o `usuarioService` para obter todos os usuários cadastrados e retornamos dentro de um objeto `ResponseEntity` a lista completa.

A segunda, `ResponseEntity<?> criar()`, recebe do cliente apenas um usuário e gera internamente o resultado da validação definida no `UsuarioDTO`. Se houve erros na validação, o controlador detecta-o, constrói a mensagem de erro e envia-o ao cliente. Caso esteja tudo correto, manda o service salvar o novo usuário. Observação: **salvar** no Spring significa criar, caso não exista, ou atualizar, caso já exista.

Por fim, a `ResponseEntity<?> listarVeiculosDoUsuario()` é a responsável por listar os veículos cadastrados do usuário cujo ID é passado na URL da requisição. A primeira linha serve para verificar se o usuário está no banco. O `findById()` dentro do `usuarioService` retornará o usuário caso exista ou emitirá uma exceção, caso contrário. Como a utilidade dessa linha não fica clara olhando só para ela, optei por deixar um comentário. Outra opção seria criar uma função dentro do service com um nome explicitando sua intenção como, por exemplo, `Boolean usuarioExistente()`. Após essa verificação, prosseguimos para recuperar todos os veículos do usuário em questão e retornamos a lista numa resposta com status 200.

Já no `VeiculoRestController`, temos somente a listagem e a criação.

```
@RestController  
@RequestMapping("/veiculos")  
public class VeiculoRestController {  
  
    private final UsuarioService usuarioService;  
    private final VeiculoService veiculoService;  
    private final FIPEClient fipeClient;  
  
    public VeiculoRestController(UsuarioService usuarioService, FIPEClient  
fipeClient, VeiculoService veiculoService) {  
        this.usuarioService = usuarioService;  
        this.fipeClient = fipeClient;  
        this.veiculoService = veiculoService;  
    }  
  
    @GetMapping  
    public ResponseEntity<List<VeiculoResponseDTO>> listar() {  
        var veiculosDTO = veiculoService.findAllDTOD();  
        return ResponseEntity.ok(veiculosDTO);  
    }  
}
```

```

    }

    @PostMapping
    public ResponseEntity<?> criar(@Validated @RequestBody VeiculoRequestDTO v,
    BindingResult result) {
        if (result.hasErrors()) {
            var erro = new ErrosValidacaoDTO(result, HttpStatus.BAD_REQUEST, "Erro
ao validar veículo", "/veiculos");
            return ResponseEntity.badRequest().body(erro);
        }

        var usuario = usuarioService.findById(v.getUsuarioId());

        VeiculoFIPEDTO veiculoFipeDTO;
        try {
            veiculoFipeDTO = fipeClient.getVeiculo(v.getMarcaId(),
v.getModeloId(), v.getAnoModelo());
        } catch (Exception e){
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Veículo não
encontrado");
        }
        var novoVeiculoDTO = veiculoService.salvarVeiculoFIPE(veiculoFipeDTO,
usuario);
        return ResponseEntity.status(HttpStatus.CREATED).body(novoVeiculoDTO);
    }
}

```

A listagem basicamente segue o mesmo procedimento do correspondente no `UsuarioRestController`, isto é, pede ao `veiculoService` a lista de todos os veículos e retorna seus DTOs.

Na criação, temos um passo a mais, pois precisamos consultar a API da FIPE pelas informações do automóvel. Logo os passos neste caso são os seguinte:

- Validação do `VeiculoRequestDTO`;
- Recupera usuário dono do veículo a partir do ID passado;
- Dentro do try, consulta API da FIPE e constrói um DTO com os dados recebidos;
- De posse do veículo e o usuário, ordena que o `veiculoService` salve o veículo associando-o ao seu dono;
- Retorna o veículo salvo ao cliente.

Services

Neste projeto, os Services são os responsáveis por *servir* os controllers com os dados necessários e as regras por trás do sistema. Enquanto o controlador REST escuta e responde os clientes da aplicação, os Services sabem das regras de negócio e realizam o tratamento de dados para que o controlador possa desempenhar suas atividades. Começemos pelo `UsuarioService`.

Nele, temos funções com propósitos variados. Vemos mais adiante a `toDTO()` e a `toUsuario()`, que convertem objetos de `Usuario` para suas versões DTO e vice-versa. Temos a `existe()` cujo propósito é auto explicativo, procurar se há um usuário com o dado CPF ou E-mail e retornar um booleano indicando se a busca encontrou algo. O interessante sobre ela é que é usada somente em outra função da mesma classe, a

`salvar()`. Naturalmente, ao salvar um novo usuário, é necessário checar se ele tem valores inéditos de CPF e E-mail.

```
package zup.garagem.service;

// Imports

@Controller
public class UsuarioService {
    private final UsuarioRepository usuarioRepository;

    public UsuarioService(UsuarioRepository usuarioRepository) {
        this.usuarioRepository = usuarioRepository;
    }

    public UsuarioDTO salvar(UsuarioDTO usuarioDTO) {
        if (existe(usuarioDTO)) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Já existe um usuário com este CPF ou e-mail");
        }
        return toDTO(usuarioRepository.save(toUsuario(usuarioDTO)));
    }

    public List<UsuarioDTO> findAllDTO() {
        return usuarioRepository
            .findAll()
            .stream()
            .map(u -> toDTO(u))
            .collect(Collectors.toList());
    }

    public Usuario findById(Long id) throws ResponseStatusException {
        Optional<Usuario> u = usuarioRepository.findById(id);
        if (u.isEmpty()) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Usuário não existe");
        }
        return u.get();
    }

    public Boolean existe(UsuarioDTO u) {
        return usuarioRepository.findUsuarioByCpfOrEmail(u.getCpf(), u.getEmail()).isPresent();
    }

    public Usuario toUsuario(UsuarioDTO u) {
        return new Usuario(u.getNome(), u.getEmail(), u.getCpf(), u.getDataNascimento());
    }

    public UsuarioDTO toDTO(Usuario u) {
        return new UsuarioDTO(u.getId(), u.getNome(), u.getEmail(), u.getCpf(),
```

```
u.getDataNascimento());  
    }  
  
}
```

O **VeiculoService** evidencia ainda mais meu método de construção de um Service. Abaixo vemos várias funções que fazem consultas ao banco e retornam os dados convertidos para DTO. Isto porque essas funções foram criadas especificamente para os REST controllers, que trabalham quase sempre com os DTOS.

De fato, procuro sempre criar funções que fazem exatamente o que os seus utilizadores precisam, ao invés de tentar fazer algo genérico mas que não serve diretamente ao propósito no qual foi criado. Um exemplo disso é o **findAllDTO()**, pois é isto que o controller quer, a lista de todos os veículos já convertidos em DTOs. A outra alternativa seria retornar os próprios veículos e deixar a conversão para o controller, ambas abordagens resolvem o mesmo problema e não há problemas em usar qualquer uma delas. Eu resolvi manter o código do controller o menor possível, transferindo-o para o service.

```
package zup.garagem.service;  
  
// Imports  
  
@Service  
public class VeiculoService {  
  
    VeiculoRepository veiculoRepository;  
  
    public VeiculoService(VeiculoRepository veiculoRepository) {  
        this.veiculoRepository = veiculoRepository;  
    }  
  
    public VeiculoResponseDTO salvarVeiculoFIPE(VeiculoFIPEDTO veiculoFipeDTO,  
    Usuario dono) {  
        var veiculo = toVeiculo(veiculoFipeDTO, dono);  
        return toResponseDTO(veiculoRepository.save(veiculo));  
    }  
  
    public Veiculo findById(Long id) {  
        var veiculo = veiculoRepository.findById(id);  
        if (veiculo.isEmpty()) {  
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Veículo com  
não existe");  
        }  
        return veiculo.get();  
    }  
  
    public List<VeiculoResponseDTO> findAllDTO() {  
        return veiculoRepository  
            .findAll()  
            .stream()  
            .map(v -> toResponseDTO(v))  
            .collect(Collectors.toList());  
    }  
}
```

```
public List<VeiculoResponseDTO> findAllDTObByUsuarioDonoId(Long id) {
    return mapToResponseDTO(veiculoRepository.findAllByUsuarioDonoId(id));
}

public VeiculoResponseDTO toResponseDTO(Veiculo v) {
    DayOfWeek diaRodizio = calcDiaRodizio(v.getAnoModelo());
    return new VeiculoResponseDTO(
        v.getId(),
        v.getMarca(),
        v.getModelo(),
        v.getAnoModelo(),
        v.getUsuarioDono().getId(),
        diaRodizio.getDisplayName(TextStyle.FULL, Locale.getDefault()),
        calcRodizioAtivo(diaRodizio)
    );
}

public List<VeiculoResponseDTO> mapToResponseDTO(List<Veiculo> veiculos) {
    List<VeiculoResponseDTO> veiculosDTO = new ArrayList<>();
    for (var veiculo : veiculos) {
        veiculosDTO.add(toResponseDTO(veiculo));
    }

    return veiculosDTO;
}

public Veiculo toVeiculo(VeiculoFIPEDTO v, Usuario u) throws RuntimeException
{
    DayOfWeek diaRodizio = calcDiaRodizio(v.getAnoModelo());
    return new Veiculo(v.getId(), v.getMarca(), v.getModelo(),
v.getAnoModelo(), v.getValor(), u, diaRodizio);
}

Boolean calcRodizioAtivo(DayOfWeek dia) {
    LocalDate hoje = LocalDate.now();
    return hoje.getDayOfWeek() == dia;
}

DayOfWeek calcDiaRodizio(String ano) throws RuntimeException {
    char ultimoDigito = ano.charAt(ano.length() - 1);
    switch (ultimoDigito) {
        case '0':
        case '1':
            return DayOfWeek.MONDAY;
        case '2':
        case '3':
            return DayOfWeek.TUESDAY;
        case '4':
        case '5':
            return DayOfWeek.WEDNESDAY;
        case '6':
        case '7':
            return DayOfWeek.THURSDAY;
    }
}
```



```
        case '8':
        case '9':
            return DayOfWeek.FRIDAY;
        default:
            throw new RuntimeException();
    }
}
```

Repository

Os repositórios são bem simples, pois o Spring nos permite utilizar algumas de funções prontas e outras implementadas automaticamente a partir da declaração da função. A seguir vemos as interfaces usadas tanto para consultar a tabela **Usuario** quanto a **Veiculo**.

Durante a implementação dos Services, vemos quais dados eles precisarão para suprir as necessidades do REST controller. A partir dessa análise, precisamos colocar apenas as declarações das funções na interface que extends de **JpaRepository<T, ID>**. Nós até poderíamos criar nossa própria implementação usando a anotação **@Query**, mas deixamos para o **Hibernate** fazer este trabalho.

Apesar de usarmos o termo JPA (Java Persistence API), que é uma *especificação* e não uma implementação, o Spring usa por padrão a tecnologia Hibernate para realizar consultas no banco, este sim, é uma implementação da especificação JPA.

O código abaixo é equivalente à realização da seguinte consulta: **SELECT u FROM Usuario u WHERE u.cpf = :cpf OR u.email = :email**.

```
@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
    Optional<Usuario> findUsuarioByCpfOrEmail(String cpf, String email);
}
```

Já no repositório do Veículo, buscamos todos os veículos a partir no id de seu dono dessa forma, ou seja, **SELECT v FROM Veiculo v WHERE v.usuarioDono.id = :id**.

```
@Repository
public interface VeiculoRepository extends JpaRepository<Veiculo, Long> {
    List<Veiculo> findAllByUsuarioDonoId(Long id);
}
```

Vale lembrar que só precisamos declarar as funções customizadas para nosso problema. Nós também usamos outras que não aparecem aqui pois já vêm declaradas por padrão na interface **JpaRepository**, como as **findAll()**, **findById()** e **save()**.

Cliente REST

Como dito anteriormente, o uso do `FeignClient` facilitou bastante a aquisição das informações sobre o veículo cadastrado. Abaixo, estão todas as linhas de código escritas para relizar a comunicação com a API da FIPE.

```
package zup.garagem.client;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import zup.garagem.dto.VeiculoFIPEDTO;

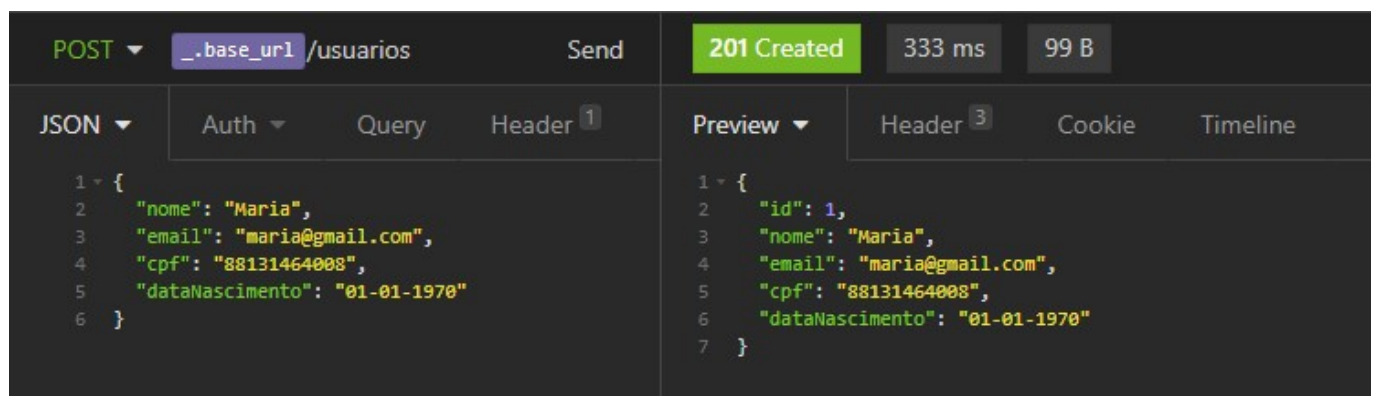
@FeignClient(name = "fipeClient", url =
    "https://parallelum.com.br/fipe/api/v1/carros/")
public interface FIPEClient {
    @GetMapping("marcas/{marcaId}/modelos/{modeloId}/anos/{anoCodigo}")
    VeiculoFIPEDTO getVeiculo(@PathVariable String marcaId,
                              @PathVariable String modeloId,
                              @PathVariable String anoCodigo);
}
```

Funcionamento

Os testes foram realizados usando o programa Insomnia. Separei exemplos de vários casos de usos e tratamentos de erros.

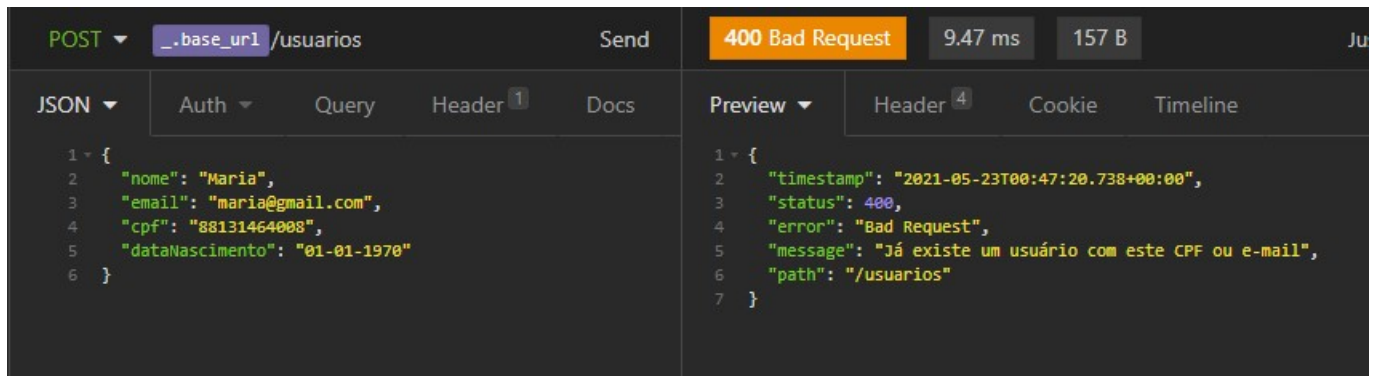
Cadastro de usuário

Usuário com campos corretos:



The screenshot shows the Insomnia API client interface. At the top, a green status bar indicates '201 Created' with a response time of '333 ms' and a body size of '99 B'. The request is a POST to '._base_url /usuarios'. The JSON body of the request is: { "nome": "Maria", "email": "maria@gmail.com", "cpf": "88131464008", "dataNascimento": "01-01-1970" }. The 'Preview' tab on the right shows the response body: { "id": 1, "nome": "Maria", "email": "maria@gmail.com", "cpf": "88131464008", "dataNascimento": "01-01-1970" }.

Usuário com campos corretos, mas já existente:



```
POST .base_url /usuarios Send 400 Bad Request 9.47 ms 157 B
```

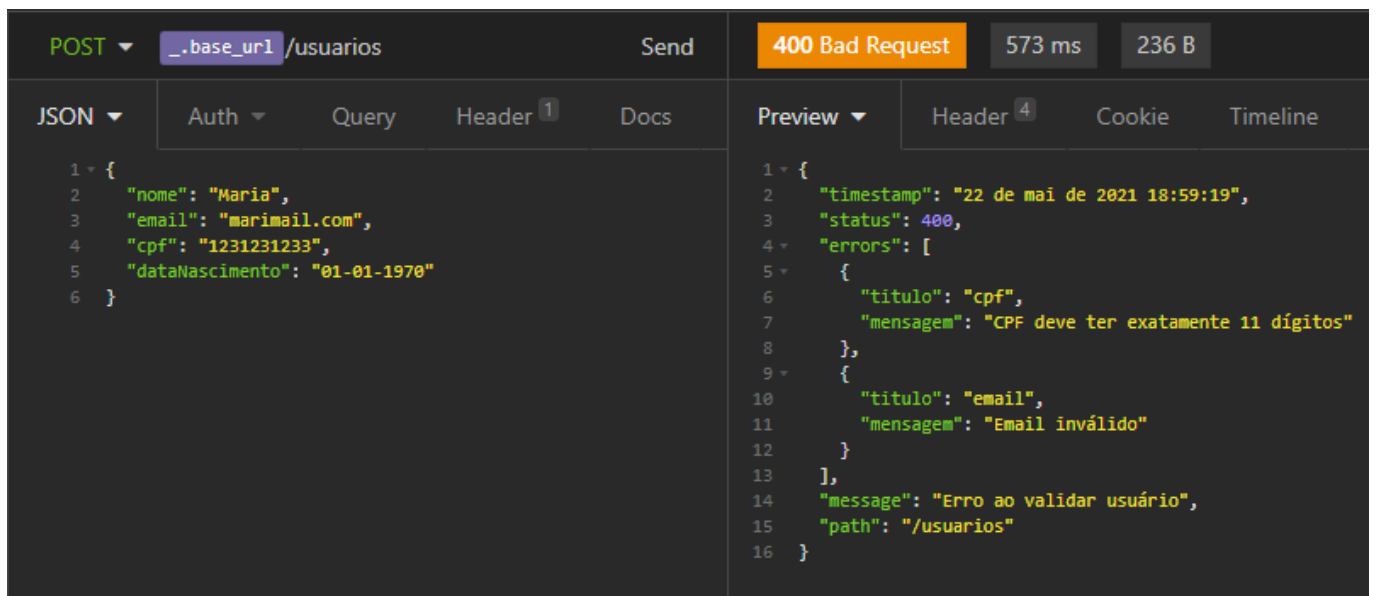
```
JSON Auth Query Header 1 Docs
```

```
1 {
2   "nome": "Maria",
3   "email": "maria@gmail.com",
4   "cpf": "88131464008",
5   "dataNascimento": "01-01-1970"
6 }
```

```
Preview Header 4 Cookie Timeline
```

```
1 {
2   "timestamp": "2021-05-23T00:47:20.738+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "message": "Já existe um usuário com este CPF ou e-mail",
6   "path": "/usuarios"
7 }
```

Abaixo, um exemplo de usuário com e-mail e CPF inválidos. Vale lembrar que aqui a única validação aplicada ao CPF é o tamanho da String, porém, existem bibliotecas Java que conseguem fazer este tratamento para nós.



```
POST .base_url /usuarios Send 400 Bad Request 573 ms 236 B
```

```
JSON Auth Query Header 1 Docs
```

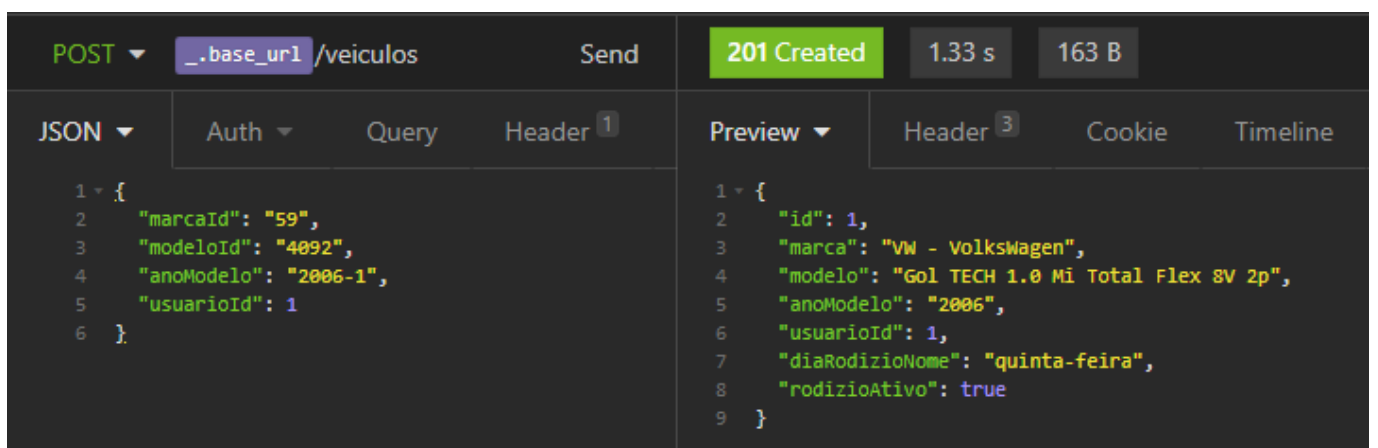
```
1 {
2   "nome": "Maria",
3   "email": "marimail.com",
4   "cpf": "1231231233",
5   "dataNascimento": "01-01-1970"
6 }
```

```
Preview Header 4 Cookie Timeline
```

```
1 {
2   "timestamp": "22 de mai de 2021 18:59:19",
3   "status": 400,
4   "errors": [
5     {
6       "titulo": "cpf",
7       "mensagem": "CPF deve ter exatamente 11 dígitos"
8     },
9     {
10      "titulo": "email",
11      "mensagem": "Email inválido"
12    }
13  ],
14   "message": "Erro ao validar usuário",
15   "path": "/usuarios"
16 }
```

Cadastro de veículo

Veículo e usuário válido:



```
POST .base_url /veiculos Send 201 Created 1.33 s 163 B
```

```
JSON Auth Query Header 1 Docs
```

```
1 {
2   "marcaId": "59",
3   "modeloId": "4092",
4   "anoModelo": "2006-1",
5   "usuarioId": 1
6 }
```

```
Preview Header 3 Cookie Timeline
```

```
1 {
2   "id": 1,
3   "marca": "VW - Volkswagen",
4   "modelo": "Gol TECH 1.0 Mi Total Flex 8V 2p",
5   "anoModelo": "2006",
6   "usuarioId": 1,
7   "diaRodizioNome": "quinta-feira",
8   "rodizioAtivo": true
9 }
```

Veículo inválido, passando IDs nulos:

POST `_.base_url` /veiculos Send **400 Bad Request** 45.6 ms 234 B

JSON Auth Query Header 1 Preview Header 4 Cookie Timeline

```
1 {
2   "marcaId": null,
3   "modeloId": "4092",
4   "anoModelo": null,
5   "usuarioId": 1
6 }
```

```
1 {
2   "timestamp": "22 de mai de 2021 21:32:45",
3   "status": 400,
4   "errors": [
5     {
6       "titulo": "marcaId",
7       "mensagem": "marca é obrigatória"
8     },
9     {
10      "titulo": "anoModelo",
11      "mensagem": "ano é obrigatório"
12    }
13  ],
14   "message": "Erro ao validar veículo",
15   "path": "/veiculos"
16 }
```

Veículo válido, mas usuário inexistente. Requisição realizada antes de cadastrar o primeiro usuário com ID = 1:

POST `_.base_url` /veiculos Send **400 Bad Request** 436 ms 132 B

JSON Auth Query Header 1 Preview Header 4 Cookie Timeline

```
1 {
2   "marcaId": "59",
3   "modeloId": "4092",
4   "anoModelo": "2006",
5   "usuarioId": 1
6 }
```

```
1 {
2   "timestamp": "2021-05-23T00:26:57.627+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "message": "Usuário não existe",
6   "path": "/veiculos"
7 }
```

Abaixo, um exemplo de veículo e usuário válidos, mas o veículo não existe na Tabela FIPE. Notar que todas informações estão corretas com exceção do "2006", que deveria ser "2006-1":

POST `_.base_url` /veiculos Send **404 Not Found** 1.17 s 134 B

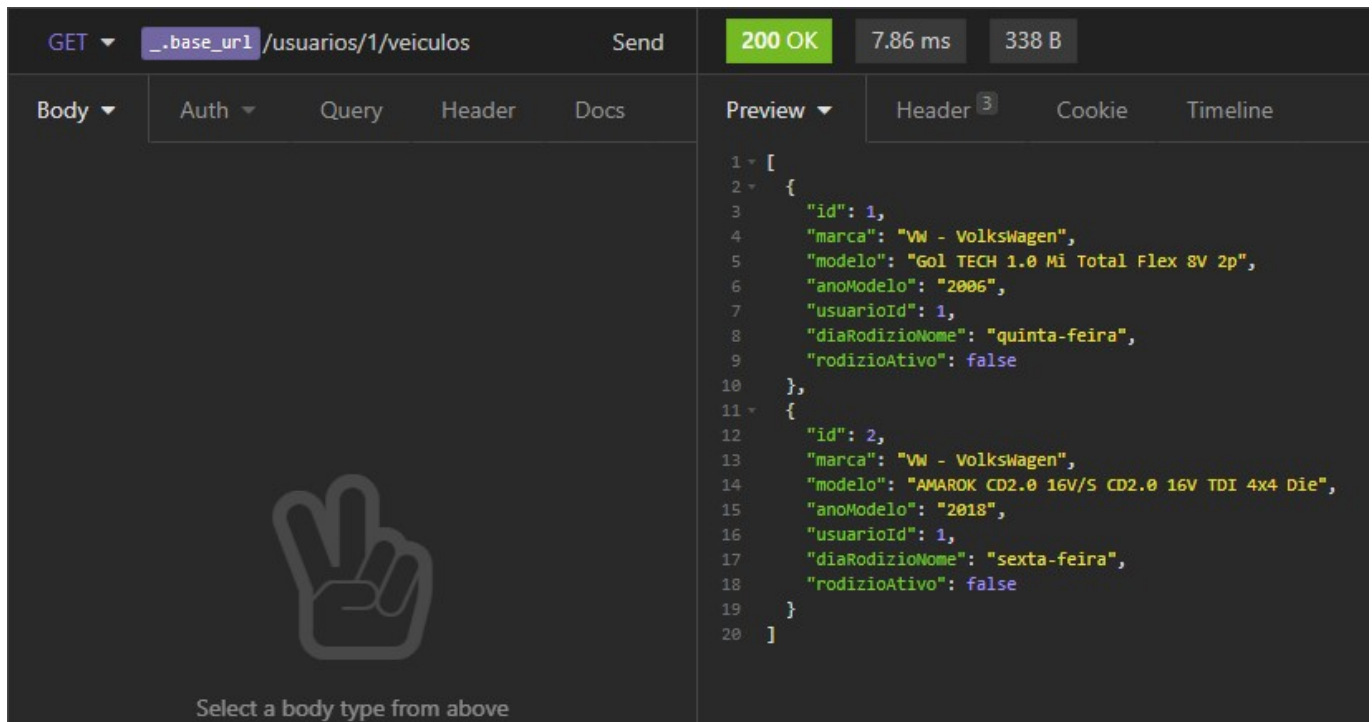
JSON Auth Query Header 1 Preview Header 3 Cookie Timeline

```
1 {
2   "marcaId": "59",
3   "modeloId": "4092",
4   "anoModelo": "2006",
5   "usuarioId": 1
6 }
```

```
1 {
2   "timestamp": "2021-05-23T00:28:31.338+00:00",
3   "status": 404,
4   "error": "Not Found",
5   "message": "Veículo não encontrado",
6   "path": "/veiculos"
7 }
```

Listagem de veículos do usuário

Listagem dos veículos do usuário por meio do end-point `{base_url}/usuario/{id}/veiculos`:



GET ▾ `_.base_url /usuarios/1/veiculos` Send **200 OK** 7.86 ms 338 B

Body ▾ Auth ▾ Query Header Docs Preview ▾ Header 3 Cookie Timeline

Select a body type from above

```
1 [
2   {
3     "id": 1,
4     "marca": "VW - Volkswagen",
5     "modelo": "Gol TECH 1.0 Mi Total Flex 8V 2p",
6     "anoModelo": "2006",
7     "usuarioId": 1,
8     "diaRodizioNome": "quinta-feira",
9     "rodizioAtivo": false
10  },
11  {
12    "id": 2,
13    "marca": "VW - Volkswagen",
14    "modelo": "AMAROK CD2.0 16V/S CD2.0 16V TDI 4x4 Die",
15    "anoModelo": "2018",
16    "usuarioId": 1,
17    "diaRodizioNome": "sexta-feira",
18    "rodizioAtivo": false
19  }
20 ]
```

Como o ID do usuário já é enviado como `@PathVariable`, nada vai dentro do body.

Conclusão

Esta foi minha primeira experiência com desenvolvimento back-end não sendo guiado por algum curso da internet. Pela primeira vez, construí uma API por conta própria a partir das especificações dadas e a sensação é diferente daquela ao concluir um curso no qual os alunos copiam a maioria do código visto nas aulas. Também tive a oportunidade de descobrir o Spring Cloud Feign, que aprendi para fazer este projeto. Claro, poderia ter ficado melhor se aplicasse algumas melhorias percebidas ao longo da escrita deste artigo, mas ainda assim fiquei bem satisfeito com o resultado final.

Eu sei, isso é muita coisa! Mas ninguém precisa aprender logo na primeira leitura. De uma coisa eu tenho certeza: não há como cortar caminho no aprendizado, sempre acharemos que é um processo muito lento. Independentemente do tempo que você precisar, saiba que no final terá valido a pena.

Todo o código está disponível [neste repositório](#).