

API REST para Controlar Veículos de Usuários

Antes de começar a descrever como será feito o “controle de veículos” conforme título acima, precisamos saber, afinal o que é uma API REST?

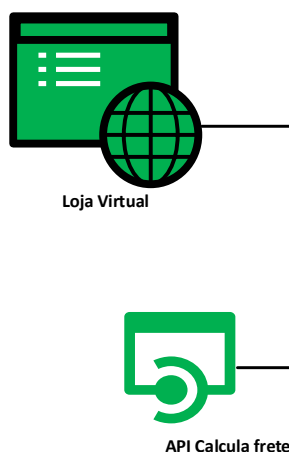
Bom nesse caso é melhor ir por partes “frase antiga”, vamos separar o que é **API** do **REST** e procuramos uma definição para melhor compreendermos.

API vem do termo em inglês “**Application Programming Interface**” nada mais é do que uma comunicação entre sistemas diferentes e a utilização dos seus serviços.

Vamos pensar no seguinte exemplo: temos uma loja virtual e como sabemos eles cobram o frete dos seus produtos para transportar encomendas para os seus destinatários, e do outro lado temos as transportadoras que fazem o serviço de entrega dessas encomendas.

Se notarmos bem existe uma **comunicação entre** a loja virtual e a transportadora, o ponto interessante é que a loja virtual utiliza o serviço da transportadora simples assim.

Trazendo para o universo tecnológico a loja virtual utiliza uma API que faz o cálculo do frete assim como os correios ou uma transportadora, ou seja, a API é quem faz o cálculo do frete(ilustração):



API é uma pequena interface que disponibiliza diversos serviços interagindo com um sistema independente do seu tamanho.

REST vem do termo inglês “Representational State Transfer” seguindo o mesmo exemplo acima onde existe uma comunicação entre uma “API prestando serviço dentro de uma loja virtual”.

Estamos falando do responsável por essa comunicação pois o **REST utiliza o protocolo HTTP** que é usado em larga escala hoje em grande escala seja nos serviços que estão na internet, na nuvem e no seu próprio celular.

Se imaginarmos o seguinte cenário: ao acessar essa loja virtual em um endereço web por exemplo: www.lojavirtual.com.br/carros e aparecer uma mensagem de página não encontrada temos o **REST realizando essa comunicação** e dizendo que essa página não existe ou seja “404 not found”.

O REST possui métodos que são utilizados inclusive na loja online só para adiantar vou passar os principais: **GET**(buscar), **POST**(gravar), **PUT**(atualizar) e **DELETE**(deletar).

Pode ocorrer outra pergunta; quando um usuário comum utiliza esses métodos? irei responder:

- Quando você acessa um site através de um link www.lojavirtual.com.br você obtém uma página HTML aqui ocorreu o **GET**.
- Ao realizar um cadastro enviando os nossos dados através de um formulário em um site ocorreu um **POST**.
- Se tivermos que atualizar esse cadastro nesse site nós realizamos um **PUT**.
- Caso esteja em alguma rede social como Instagram ou Facebook e realizar uma exclusão de alguma foto que não está bonita para você temos um **DELETE**.

“Nessa introdução já podemos perceber que a aplicação API REST de Controle de Veículos para Usuários irá trazer muito dos itens mencionados acima”.

#A API REST foi desenvolvida em:

-Linguagem Java que possui uma grande quantidade de bibliotecas e uma forte integração com Framework Spring e seus serviços. Um ponto interessante é que a linguagem Java trás vários conceitos que servem para outra linguagens como orientação a objetos e estrutura de dados, além de fazer o futuro desenvolvedor entender alguns conceito com TDD(testes) bem como conceitos que envolvem arquitetura de software.

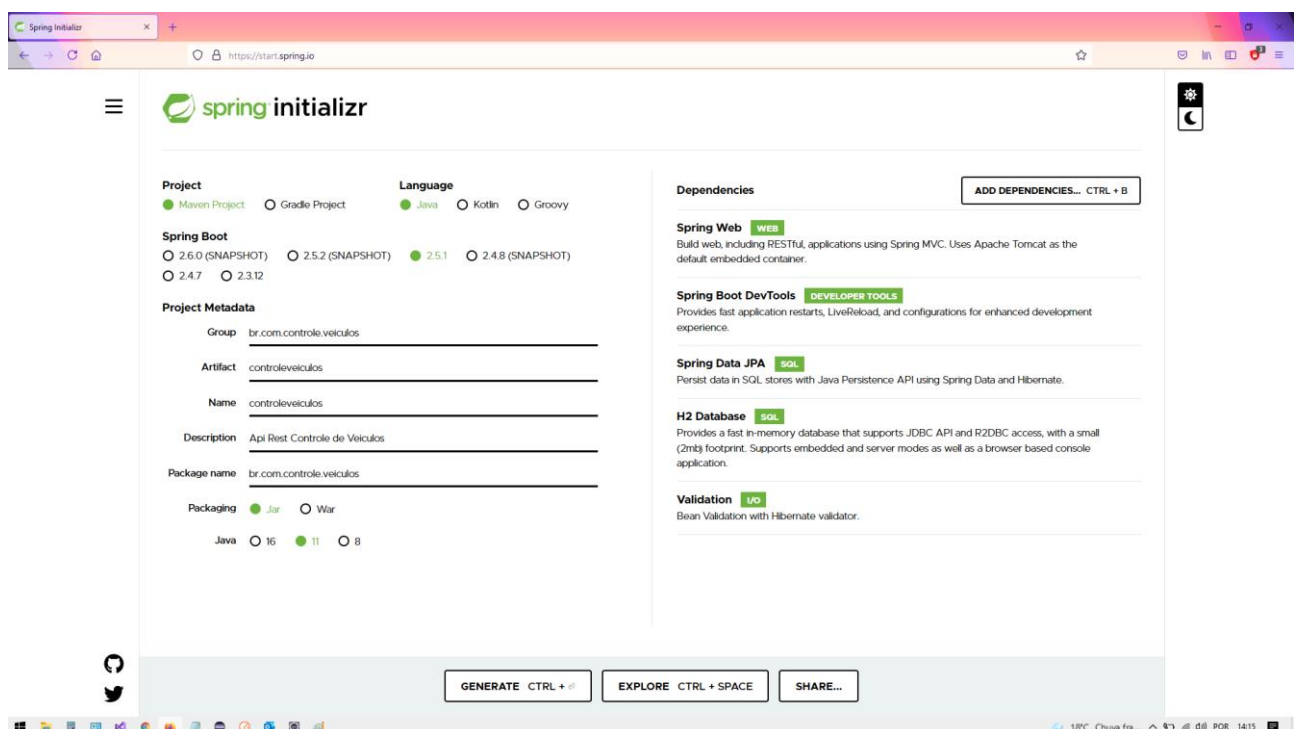
Além de ser multiplataforma que em um período não tão distante abrangia os somente em sistemas operacionais Windows, Linux e macOS, o conceito multiplataforma se estende até nos dias de hoje fortemente, com as aplicações web, computação em nuvem, aplicativos para dispositivos móveis entre outros.

Versão JAVA JDK ZULU 11 LTS

-Para desenvolver com a linguagem Java foi utilizado o **SpringBoot** que são vários frameworks, Esses (frameworks/dependências) são selecionadas e depois baixadas em arquivo.zip com tudo já configurado, integrando depois com a IDE Eclipse, para baixar as configurações de projetos Spring acessar o site: <https://start.spring.io/>

Esses frameworks, ou seja, essas tecnologias serão descritas a seguir:

- Spring Web**
- Spring Boot DevTools**
- Spring Data JPA**
- H2 Database**
- Validation**



Spring Web: Possui embarcado algumas tecnologias que auxiliam no desenvolvimento web entre elas o **Servidor Apache Tomcat** que verifica as solicitações do servidor que são essenciais para nosso projeto pois todas as vezes que compilamos e vamos verificar se **API REST** está disponível é através deste servidor. Também vem junto o padrão **MVC (Model, View, Controle)** que é arquitetura de software de uma aplicação web, essa sigla significa a divisão de responsabilidades com objetivo de facilitar as manutenções futuras.

Spring Boot DevTools: Tem como principal objetivo nos auxiliares enquanto estamos desenvolvendo. Ao realizar uma alteração ou inclusão de alguma classe no projeto e salvar a mesma, não precisamos parar o que estamos fazendo para iniciar o Servidor Apache Tomcat o mesmo ocorrerá automaticamente, com isso nos proporciona maior agilidade na reinicialização de aplicativos.

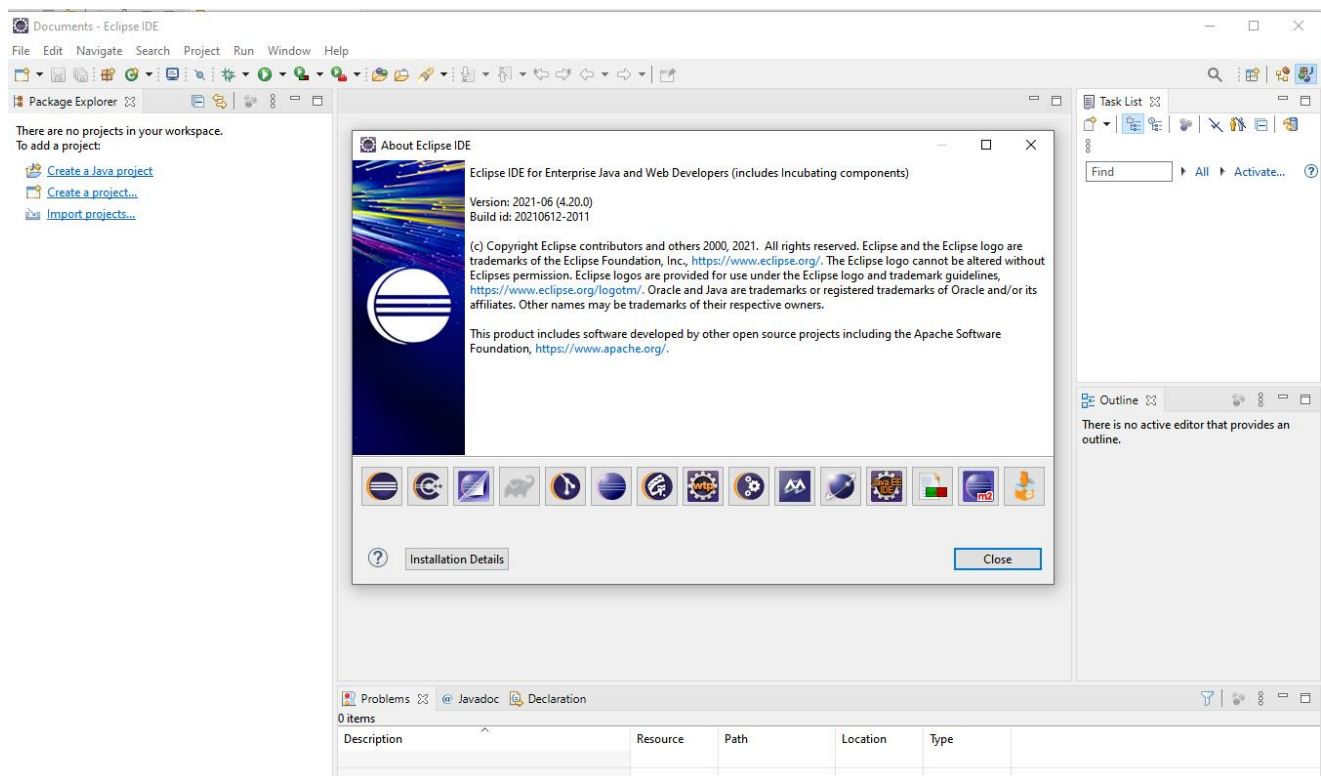
Spring Data JPA: Faz um mapeamento objeto-relacional em uma camada de persistência; exemplo disso no próprio projeto da API REST Controle de Veículos que veremos, temos um objetivo em que um usuário pode ter muitos carros note que aqui entra o conceito de um mapeamento relacional, ou seja 1:N, para exemplificar em uma classe.java declaramos um atributo: **private Long id**, mais em um banco de dados este **id** que colocamos como atributo é uma **chave estrangeira/Foreign Key** o mapeamento faz essa relação.

Outro item que veremos é assinaturas de métodos passando informações em nossas classes.java como se fosse em um banco de dados, exemplo: Em uma classe Repository, temos um atributo: **Carro findCarroByModelo (String modelo)**; é equivalente a clausula where em uma consulta SQL.

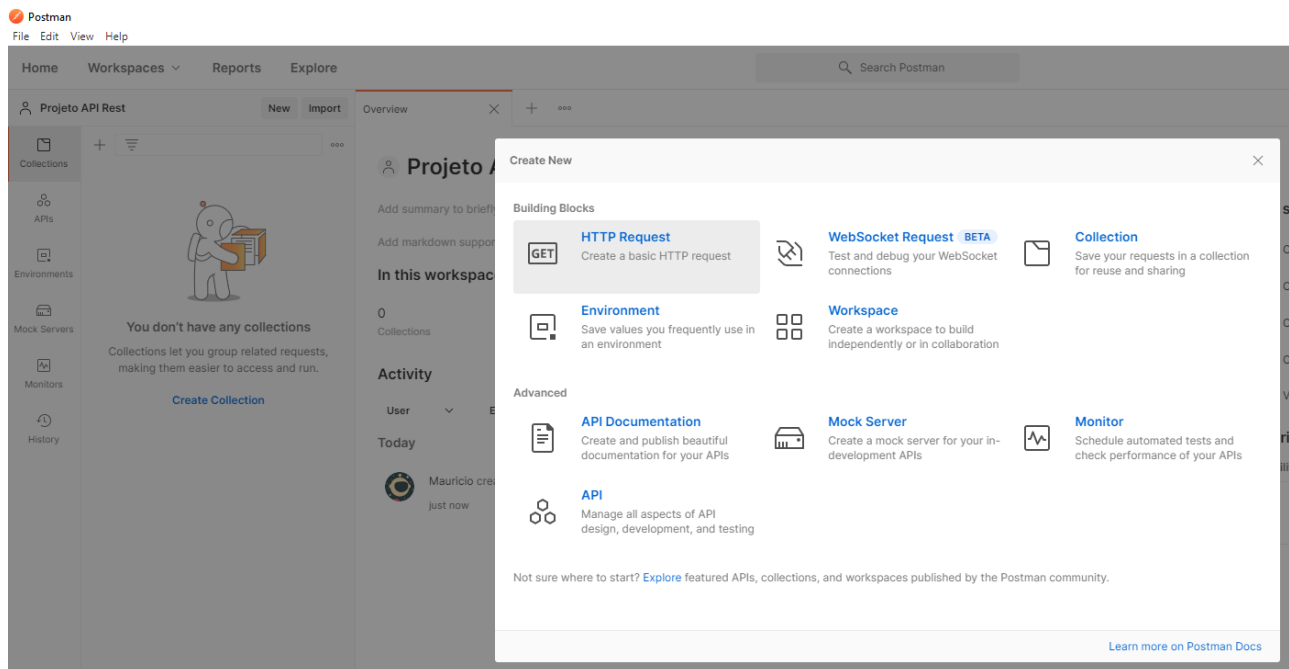
H2 Database: É um banco de dados que fica alocado na memória, sendo muito útil em ambiente para validar a API REST e seus métodos; tem como vantagem uma instalação totalmente simples. O **H2 Database** não se compara a um banco de produção como MySQL, PostgreSQL e Oracle, afinal como os recursos são utilizados enquanto está na memória todas as vezes que o **Spring Boot DevTools** reiniciar o serviço os dados com as informações no banco de dados são excluídas.

Validation: Trata de validar os dados que os usuários irão informar, na verdade o termo correto não seria uma tradução direta “Validação” e sim **restrição**, afinal ao colocar uma **restrição** do tipo **@NotNull** estamos dizendo que esse atributo no banco de dados não pode ficar nulo.

-IDE para Desenvolvimento Eclipse: Utilizada para desenvolvimento em aplicações JAVA e demais linguagens multiplataforma. A IDE Eclipse tem integração com o Spring Boot e suas dependências assim como o JAVA JDK.



-Postman: Ferramenta para testar as APIs mostrando quais resultados são retornados no caso os Status Http, é necessário um simples cadastrado antes de baixar o Postman, segue o link: <https://www.postman.com/>



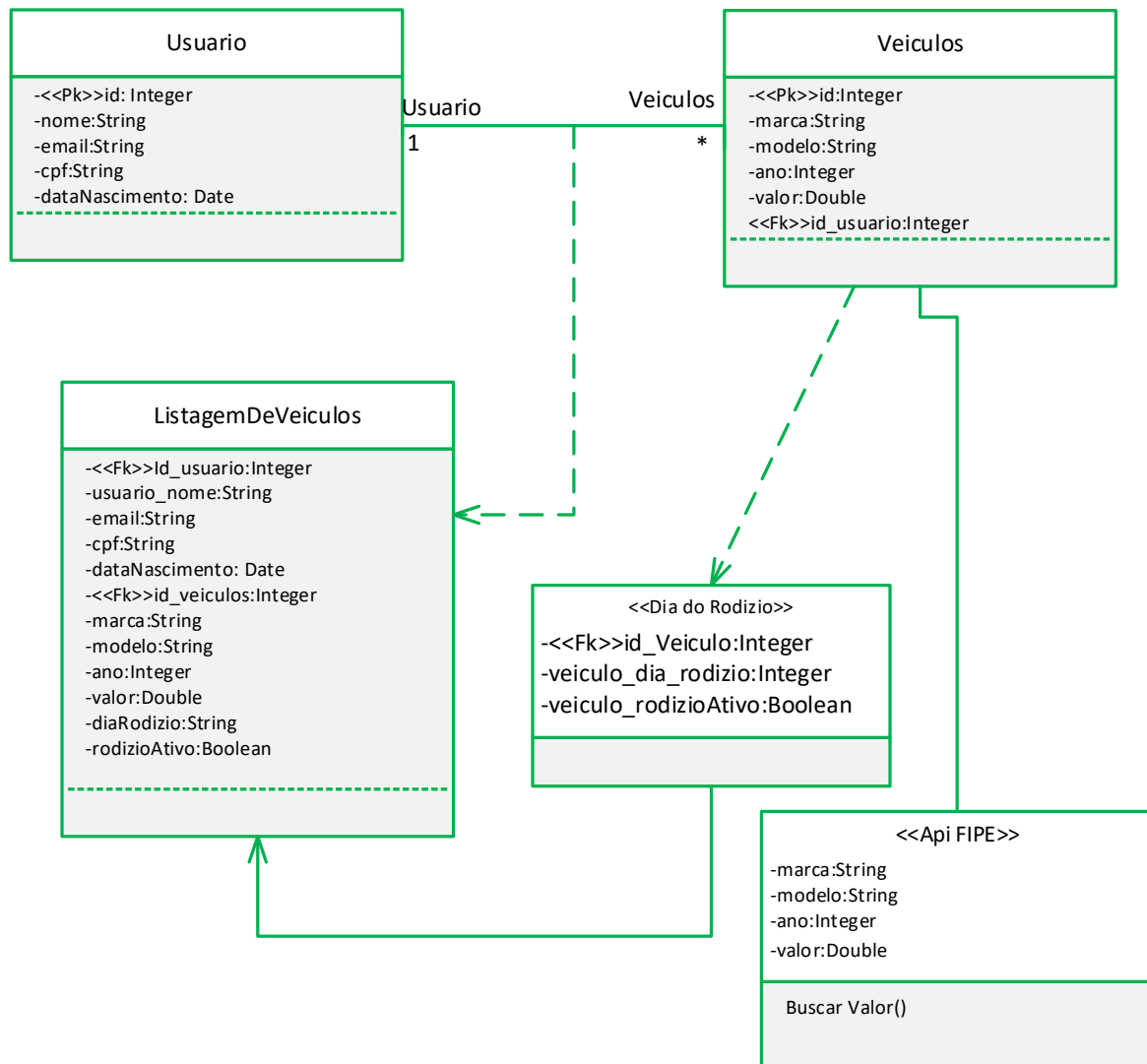
#Modelo de desenvolvimento da API Rest:

Modelo Conceitual: A partir deste modelo conceitual é gerado uma base relacional.

-Um usuário pode ter vários veículos cadastrados em seu nome, o cadastro de veículos só ocorre se o usuário cadastra-lo ou não, pois o usuário pode cadastrar seus dados mais não necessariamente precisa ter um veículo cadastrado.

-Nesse relacionamento entre um usuário e um veículo cadastrado, geram uma Listagem de veículos cadastrados por um usuário.

-Note que no momento desta Listagem dois novos atributos são adicionados a entidade Veículos que são (dia do Rodizio e Rodizio ativo (sim ou não)).



EndPoints que estão nesta API REST:

1-EndPoint: /CadastroUsuario

Dados: nome, email, cpf, dataNascimento

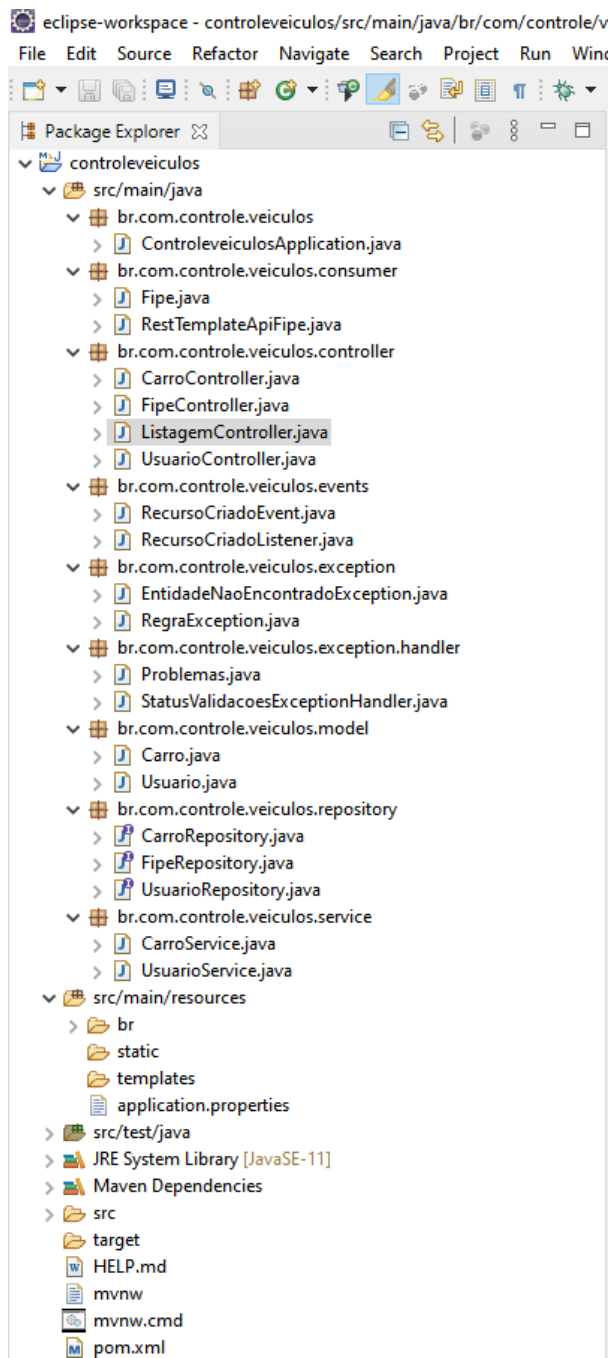
2-EndPoint: /CadastroVeiculos

Dados: marca, modelo, ano, valor

3-EndPoitn: /ListagemVeiculos

Dados: usuário: nome, email, cpf, dataNascimento, veículos: marca, modelo, ano, valor, diaRodizio, rodizioAtivo.

Classes criadas neste projeto:



-Classe Usuario e Classe Carro:

Através destas classes estaremos modelando o banco de dados H2, fazendo a sua representação do banco de dados. Neste caso ambas as classe utilizaram a anotação **@Entity** dizendo que essas classes são entidades no banco de dados, outra característica é a anotação **@Id** referente a chave primaria no banco de dados, assim o Hibernate sabe que é o identificador deste modelo relacional, sendo administrada pelo banco de dados no caso H2 através da anotação **@GeneratedValue(strategy = GenerationType.IDENTITY)**.

As anotações:

@NotBlank: Define que o campo não pode ficar null

@Column(nullable = false, length = 50): Define a coluna no banco de dados e o seu tamanho.

Na classe Usuário: A anotação **@OneToMany**: faz o mapeamento da regra do sistema que diz: um usuário pode ter muitos carros, significa que existe uma relação de 1:N.

Tem um mapeamento do tipo **CascadeType.ALL** dizendo que no momento de excluir um usuário os seus carros serão excluídos. No nível de Hierarquia todos os que tiverem associados ao usuário para baixo, no caso os carros serão excluídos **orphanRemoval = true**.

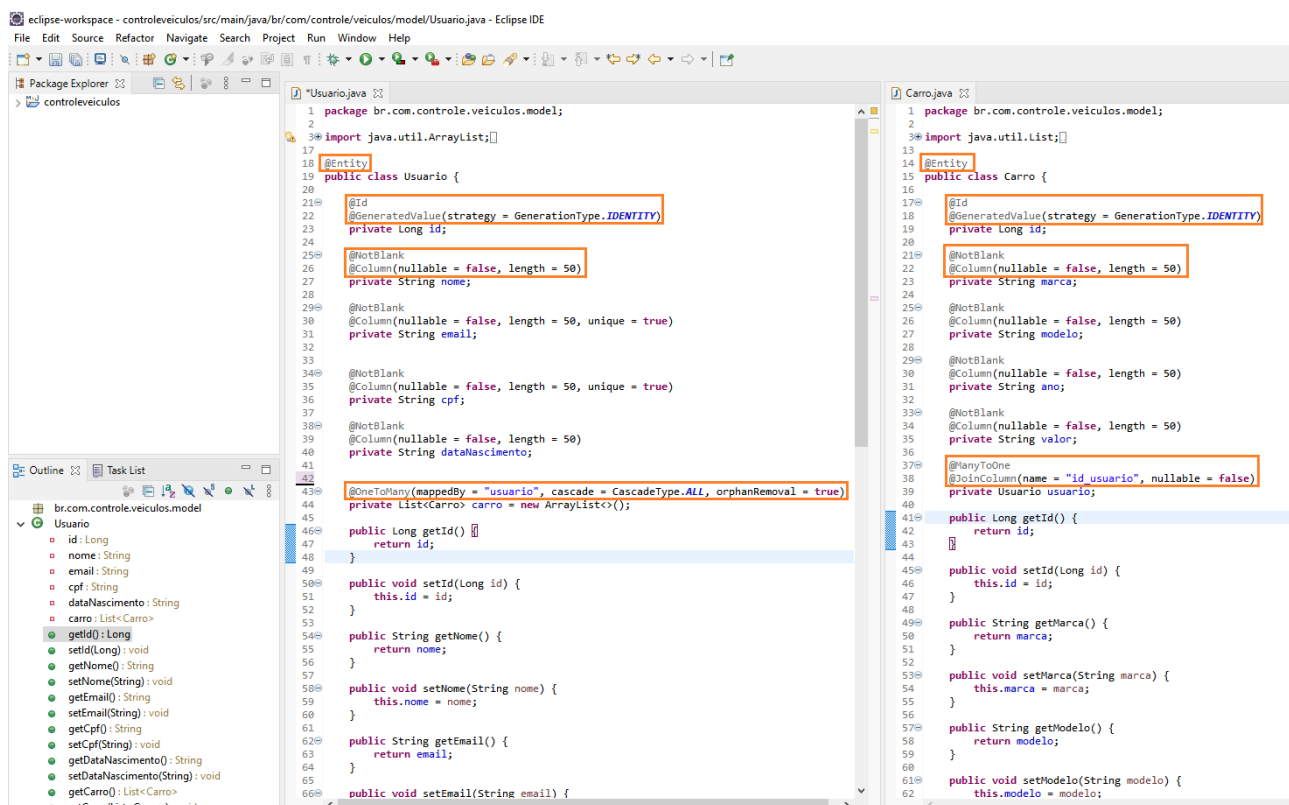
Se por um lado temos uma relação de 1:N também teremos um caminho de volta que é N:1, esse mapeamento de ida e volta.

Para criar o vínculo entre as entidades/tabelas é utilizado outra anotação **@JoinColumn** criando uma chave estrangeira na tabela do carro.

Essa volta está definida na classe carro com a anotação **@ManyToOne**: Muitos carros para um usuário.

Ao definir os atributos das classes é realizado os **Getters e Setters** que serão utilizados para ter um vinculo com o banco de dados uma vez que podemos realizar os métodos de alteração, exclusão, inserção, e atualizar as informações.

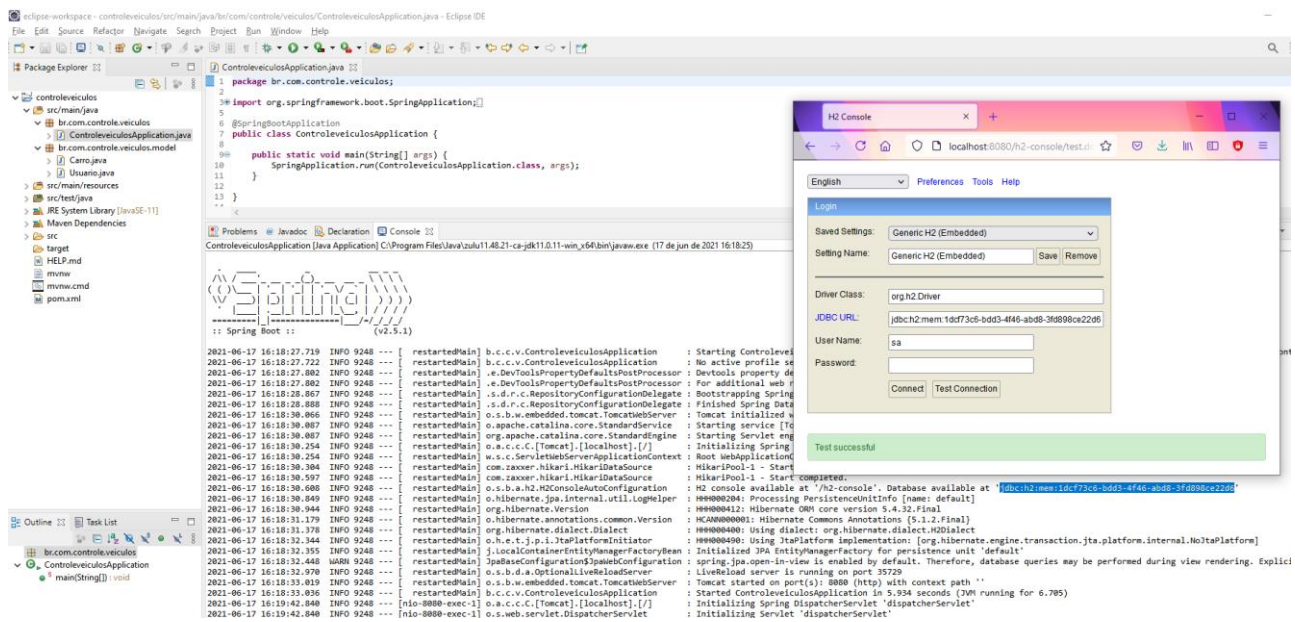
Obs: Para melhores práticas no desenvolvimento de software se realmente não for utilizar algum **Getter ou Setter** melhor não gera-los pois acaba tendo uma quantidade de código que não terá funcionalidade alguma.



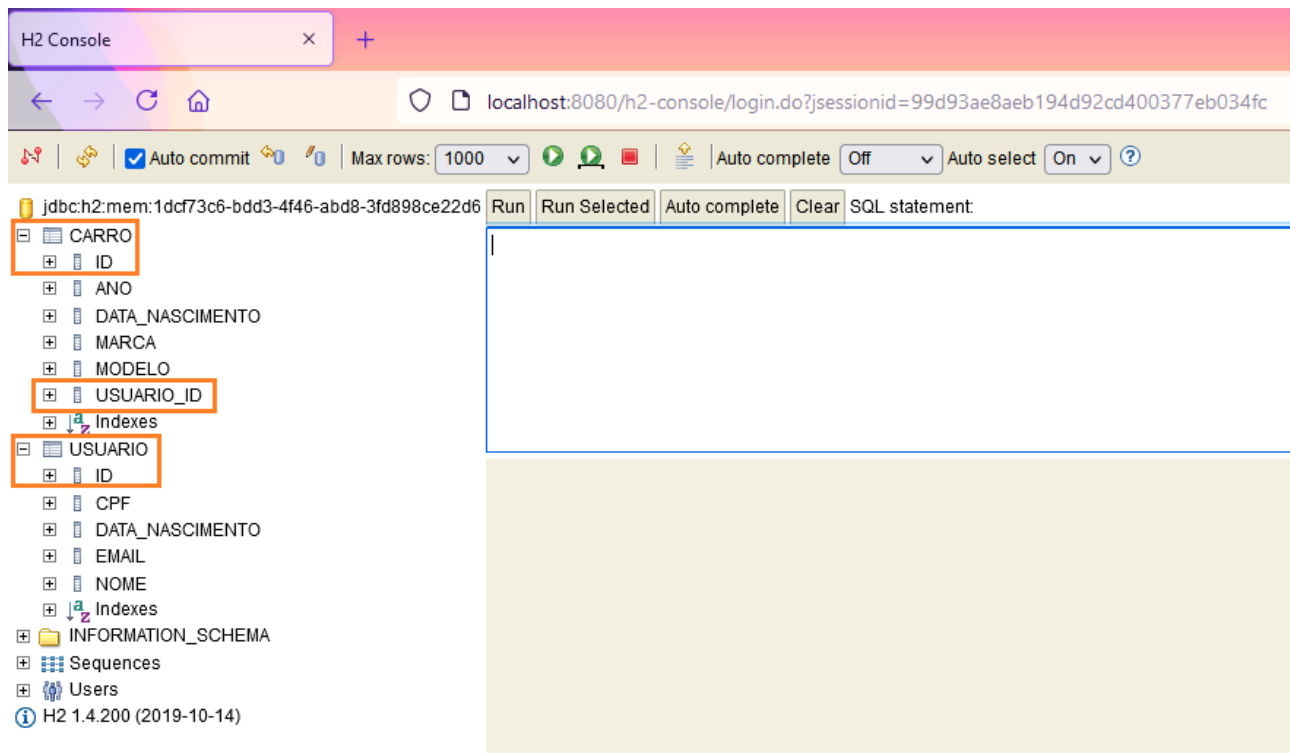
```
1 package br.com.controle.veiculos.model;
2
3 import java.util.ArrayList;
4
5 @Entity
6 public class Usuario {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private Long id;
11
12    @NotBlank
13    @Column(nullable = false, length = 50)
14    private String nome;
15
16    @NotBlank
17    @Column(nullable = false, length = 50, unique = true)
18    private String email;
19
20    @NotBlank
21    @Column(nullable = false, length = 50, unique = true)
22    private String cpf;
23
24    @NotBlank
25    @Column(nullable = false, length = 50)
26    private String dataNascimento;
27
28    @OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL, orphanRemoval = true)
29    private List<Carro> carro = new ArrayList<>();
30
31    public Long getId() {
32        return id;
33    }
34
35    public void setId(Long id) {
36        this.id = id;
37    }
38
39    public String getNome() {
40        return nome;
41    }
42
43    public void setNome(String nome) {
44        this.nome = nome;
45    }
46
47    public String getEmail() {
48        return email;
49    }
50
51    public void setEmail(String email) {
52        this.email = email;
53    }
54
55    public String getCPF() {
56        return cpf;
57    }
58
59    public void setCPF(String cpf) {
60        this.cpf = cpf;
61    }
62
63    public String getDataNascimento() {
64        return dataNascimento;
65    }
66
67    public void setDataNascimento(String dataNascimento) {
68        this.dataNascimento = dataNascimento;
69    }
70
71    public List<Carro> getCarro() {
72        return carro;
73    }
74
75    public void setCarro(List<Carro> carro) {
76        this.carro = carro;
77    }
78
79 }
```

```
1 package br.com.controle.veiculos.model;
2
3 import java.util.List;
4
5 @Entity
6 public class Carro {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private Long id;
11
12    @NotBlank
13    @Column(nullable = false, length = 50)
14    private String marca;
15
16    @NotBlank
17    @Column(nullable = false, length = 50)
18    private String modelo;
19
20    @NotBlank
21    @Column(nullable = false, length = 50)
22    private String ano;
23
24    @NotBlank
25    @Column(nullable = false, length = 50)
26    private String valor;
27
28    @ManyToOne
29    @JoinColumn(name = "id_usuario", nullable = false)
30    private Usuario usuario;
31
32    public Long getId() {
33        return id;
34    }
35
36    public void setId(Long id) {
37        this.id = id;
38    }
39
40    public String getMarca() {
41        return marca;
42    }
43
44    public void setMarca(String marca) {
45        this.marca = marca;
46    }
47
48    public String getModelo() {
49        return modelo;
50    }
51
52    public void setModelo(String modelo) {
53        this.modelo = modelo;
54    }
55
56    public String getValor() {
57        return valor;
58    }
59
60    public void setValor(String valor) {
61        this.valor = valor;
62    }
63
64    public Usuario getUsuario() {
65        return usuario;
66    }
67
68    public void setUsuario(Usuario usuario) {
69        this.usuario = usuario;
70    }
71
72 }
```


Após criarmos nossas classes vamos inicializar o Spring Boot e acessar o banco de dados H2e verificar se as tabelas foram criadas.



Agora que acessamos o banco de dados notamos que foram criadas as 2 tabelas com as chaves primarias e a chave estrangeira quando foram realizadas as anotações: `@Entity`, `@Id`, `@JoinColumn`, e os relacionamentos `@ManyToOne` e `@OneToMany`.

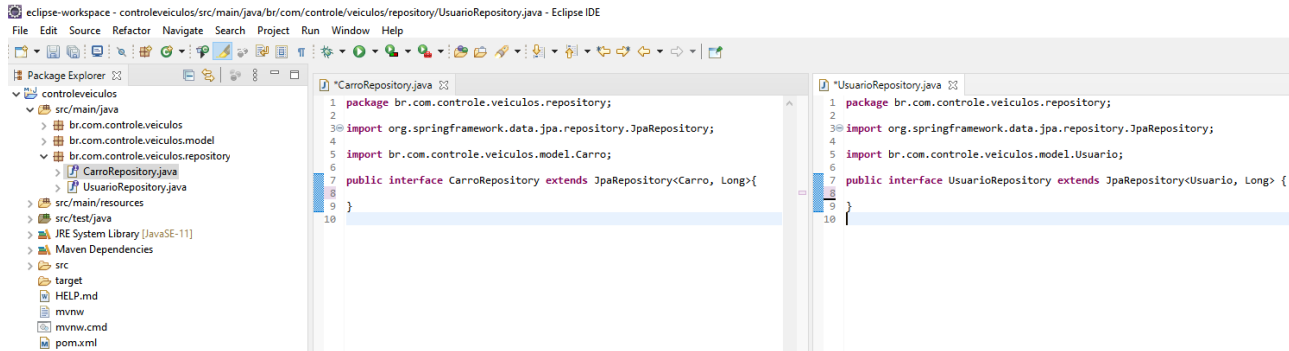


-Classe CarroRepository e Classe UsuarioRepository

Após concluída esta primeira etapa vamos criar agora a nossas interfaces (**CarroRepository** e **UsuarioRepository**) que serão responsáveis pela persistência com o banco de dados como inserir, pesquisar e buscar informações. Estas interfaces herdaram de **JpaRepository** onde devemos dizer com qual objeto ira persistir no banco de dados e qual o seu tipo de identificação, melhor dizendo qual o tipo do seu ID.

No nosso caso nosso ID definido nas classe Usuario e Carro são **Long(int)**, porém nem todo ID de um banco de dados é do tipo Inteiro como por exemplo o banco de dados MongoDB tem o seu ID o tipo String. Após criar as **classes CarroRepository e UsuarioRepository** podemos fazer algumas manipulações do tipo findAll, findById entre outras.

Para exemplificar se dentro desta classe colocarmos o seguinte código: Usuario findByNome(String name); isso seria equivalente a um Select (select * from usuario where nome = "Felipe") no nosso banco de dados buscando pelo nome.

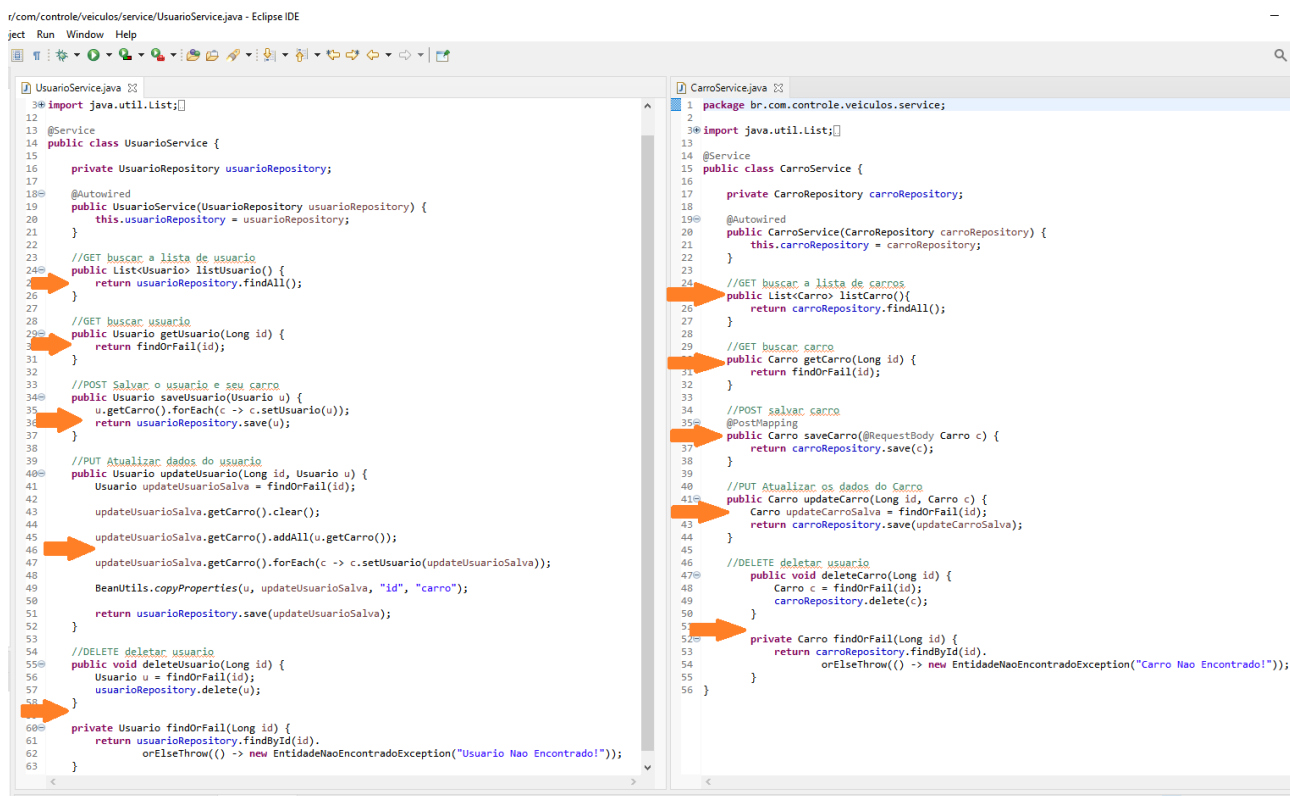


-Classe CarroService e UsuarioService

Nestas classes estão contidas as regras que serão realizadas no sistema API REST como os métodos (GET, POST, PUT, DELETE). Estas classes possui uma anotação **@Service** e terão acesso as respectivas classes **UsuarioRepository e CarroRepository**.

Mais afinal por que fazer várias anotações com "@Algum nome"? Como estamos trabalhando com o Spring, no momento em que ele é inicializado todas as classes que possuem essas anotações são lidas para que possamos trabalhar com injeção de dependência no caso destas classes temos **@Autowired**.

"Para que a classe UsuarioService venha a ser utilizada é necessário passar uma outra Classe no caso UsuarioRepository, caso não venha passar a classe UsuarioService não ira funcionar. A classe UsuarioService é totalmente dependente de uma outra, por isso se fala em injeção de dependência."

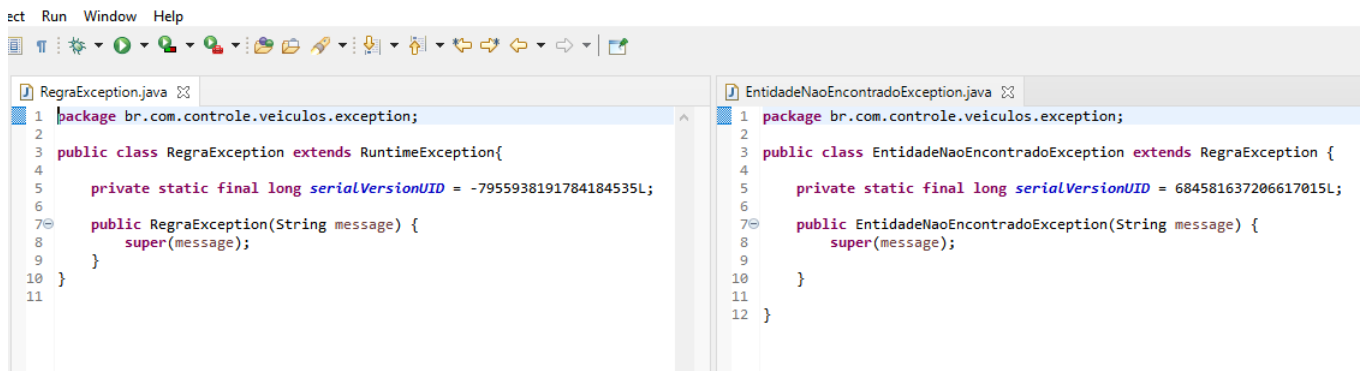


No método PUT como não sabemos o que pode ser alterado antes de salvar a atualização dos dados do Usuario, primeiramente os dados do usuário serão limpos, depois todos os dados que estão na tela serão adicionados e depois serão salvos.

Classe RegraException e EntidadeNaoEncontradaException

A classe **EntidadeNaoEncontradaException** juntamente com a **RegraException** auxiliam as classes acima no momento em que ocorrer uma exceção como por exemplo: Deletar um carro que não existe no cadastro, passando uma mensagem para o usuário de “Carro não encontrado” ou “Usuario não encontrado”.

/com/controle/veiculos/exception/RegraException.java - Eclipse IDE



Classe CarroController e Classe UsuarioController

Nestas classes estão declaradas os nossos **endPoints** que serão utilizados no Postman para verificar se as comunicações estão ocorrendo corretamente.

Os **endPoints** são declarados através de uma anotação **@RequestMapping** dentro dele deve estar descrito qual o seu caminho no caso:

@RequestMapping(path = "/CadastroUsuario", produces = MediaType.APPLICATION_JSON_VALUE).

@RequestMapping(path = "/CadastroCarro", produces = MediaType.APPLICATION_JSON_VALUE).

Ambos irão produzir um formato **JSON** que será visualizado no Postman.

A anotação **@RestController** é utilizado para que o Spring identifique que essa classe faz as comunicações de requisição via protocolo HTTP (**REST**).

Como abordado nos primeiros parágrafos acima com exemplo da loja virtual, estamos vendo os métodos: **GET**(buscar), **POST**(gravar), **PUT**(atualizar) e **DELETE**(deletar), presentes nestas classes e seus Status HTTP que são o retorno se as requisições foram ou não atendidas, mais adiante iremos ver como se comportam cada um destes métodos no Postman.

```
1 UsuarioController.java
2 package br.com.controle.veiculos.controller;
3
4 import java.util.List;
5
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.http.HttpStatus;
8 import org.springframework.http.ResponseEntity;
9 import org.springframework.web.bind.annotation.GetMapping;
10 import org.springframework.web.bind.annotation.PathVariable;
11 import org.springframework.web.bind.annotation.PostMapping;
12 import org.springframework.web.bind.annotation.RequestBody;
13 import org.springframework.web.bind.annotation.RequestMapping;
14 import org.springframework.web.bind.annotation.RestController;
15
16 import br.com.controle.veiculos.model.Usuario;
17 import br.com.controle.veiculos.service.UsuarioService;
18
19 @RestController
20 @RequestMapping("/CadastroUsuario")
21 public class UsuarioController {
22
23     private UsuarioService usuarioService;
24
25     @Autowired
26     public UsuarioController(UsuarioService usuarioService) {
27         this.usuarioService = usuarioService;
28     }
29
30     @GetMapping
31     public List<Usuario> listUsuario(){
32         return usuarioService.listUsuario();
33     }
34
35     @GetMapping("/{id}")
36     public ResponseEntity<Usuario> getIdUsuario(@PathVariable(name = "id") Long id){
37         return ResponseEntity.ok(usuarioService.getIdUsuario(id));
38     }
39
40     @PostMapping
41     public ResponseEntity<Usuario> saveUsuario(@RequestBody Usuario usuario){
42         Usuario u = usuarioService.saveUsuario(usuario);
43         return ResponseEntity.status(HttpStatus.CREATED).body(u);
44     }
45 }

1 CarroController.java
2 package br.com.controle.veiculos.controller;
3
4 import java.util.List;
5
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.http.HttpStatus;
8 import org.springframework.http.ResponseEntity;
9 import org.springframework.web.bind.annotation.GetMapping;
10 import org.springframework.web.bind.annotation.PathVariable;
11 import org.springframework.web.bind.annotation.PostMapping;
12 import org.springframework.web.bind.annotation.RequestBody;
13 import org.springframework.web.bind.annotation.RequestMapping;
14 import org.springframework.web.bind.annotation.RestController;
15
16 import br.com.controle.veiculos.model.Carro;
17 import br.com.controle.veiculos.service.CarroService;
18
19 @RestController
20 @RequestMapping("/CadastroVeiculos")
21 public class CarroController {
22
23     private CarroService carroService;
24
25     @Autowired
26     public CarroController(CarroService carroService) {
27         this.carroService = carroService;
28     }
29
30     @GetMapping
31     public List<Carro> listCarro(){
32         return carroService.listCarro();
33     }
34
35     @GetMapping("/{id}")
36     public ResponseEntity<Carro> getIdCarro(@PathVariable(name = "id") Long id){
37         return ResponseEntity.ok(carroService.getIdCarro(id));
38     }
39
40     @PostMapping
41     public ResponseEntity<Carro> saveCarro(@RequestBody Carro carro){
42         Carro c = carroService.saveCarro(carro);
43         return ResponseEntity.status(HttpStatus.CREATED).body(c);
44     }
45 }
46 }
```

Classe Problemas e Classe StatusValidacoesExceptionHandler

A classe **Problemas** tem a função que trata os possíveis problemas reportando os atributos nela definidos em conjunto com a classe **StatusValidacoesExceptionHandler** que faz a função de monitorar essas exceções e voltar uma mensagem para tela informado para o usuário o que está ocorrendo.

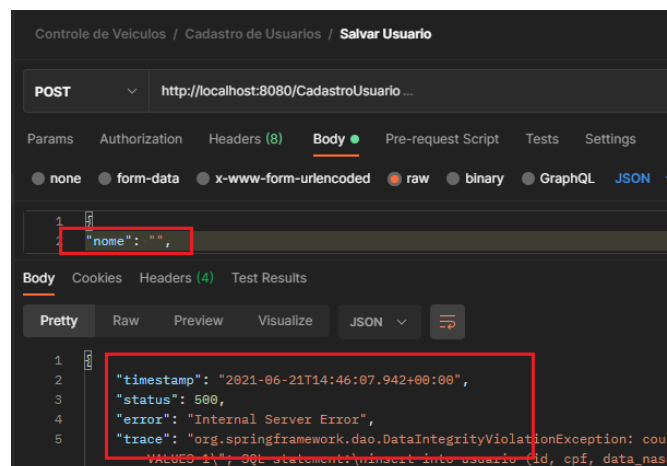
A anotação **@ControllerAdvice** na classe **StatusValidacoesExceptionHandler** tem a função de ficar monitorando nossa aplicação caso ocorra alguma exceção à regra ira retornar os Status Http da requisição.

```
1 Problemas.java
2 package br.com.controle.veiculos.exception.handler;
3
4 import java.time.OffsetDateTime;
5
6 @Include(include = "NOW")
7 public class Problemas {
8
9     private Integer status;
10     private OffsetDateTime datahora;
11     private String titulo;
12     private List<Campo> campos;
13
14     public static class Campo {
15         private String campo;
16         private String mensagem;
17
18         public Campo(String campo, String mensagem) {
19             super();
20             this.campo = campo;
21             this.mensagem = mensagem;
22         }
23
24         public String getCampo() {
25             return campo;
26         }
27
28         public void setCampo(String campo) {
29             this.campo = campo;
30         }
31
32         public String getMensagem() {
33             return mensagem;
34         }
35
36         public void setMensagem(String mensagem) {
37             this.mensagem = mensagem;
38         }
39     }
40
41     public Integer getStatus() {
42         return status;
43     }
44
45     public void setStatus(Integer status) {
46         this.status = status;
47     }
48
49     public OffsetDateTime getDatahora() {
50         return datahora;
51     }
52
53     public void setDatahora(OffsetDateTime datahora) {
54         this.datahora = datahora;
55     }
56 }

1 StatusValidacoesExceptionHandler.java
2 package br.com.controle.veiculos.exception.handler;
3
4 import java.time.OffsetDateTime;
5
6 @ControllerAdvice
7 public class StatusValidacoesExceptionHandler extends ResponseEntityExceptionHandler {
8
9     @Autowired
10     private MessageSource messageSource;
11
12     @ExceptionHandler({EntidadeNaoEncontradaException.class})
13     public ResponseEntity<Object> handleEntidadeNaoEncontrada(RegraException ex, WebRequest req) {
14         HttpStatus status = HttpStatus.NOT_FOUND;
15         return handleExceptionInternal(ex, getProblem(ex, status), new HttpHeaders(), status, req);
16     }
17
18     @ExceptionHandler({RegraException.class})
19     public ResponseEntity<Object> handleRegraException(RegraException ex, WebRequest req) {
20         HttpStatus status = HttpStatus.BAD_REQUEST;
21         return handleExceptionInternal(ex, getProblem(ex, status), new HttpHeaders(), status, req);
22     }
23
24     @Override
25     protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
26         HttpHeaders headers, HttpStatus status, WebRequest request) {
27         List<Campo> campos = new ArrayList<Problemas.Campo>();
28
29         for (ObjectError error : ex.getBindingResult().getAllErrors()) {
30             String campo = ((FieldError) error).getField();
31             String mensagem = messageSource.getMessage(error, LocaleContextHolder.getLocale());
32             campos.add(new Problemas.Campo(campo, mensagem));
33         }
34
35         Problemas problemas = new Problemas();
36         problemas.setStatus(status.value());
37         problemas.setTitulo("Um ou mais campos estão inválidos por gentileza faça o preenchimento correto e tente novamente");
38         problemas.setDatahora(OffsetDateTime.now());
39         problemas.setCampos(campos);
40
41         return super.handleExceptionInternal(ex, problemas, headers, status, request);
42     }
43
44     private Problemas getProblem(RegraException ex, HttpStatus httpStatus) {
45         Problemas problemas = new Problemas();
46         problemas.setStatus(httpStatus.value());
47         problemas.setTitulo(ex.getMessage());
48         problemas.setDatahora(OffsetDateTime.now());
49     }
50 }
```

Note que há uma relação no atributo List<Campo> entre as classes, porém na **classe StatusValidacoesExceptionHandler** é feita uma função que varre todos os campos da nossa aplicação para ver se existe algum exceção e retornar.

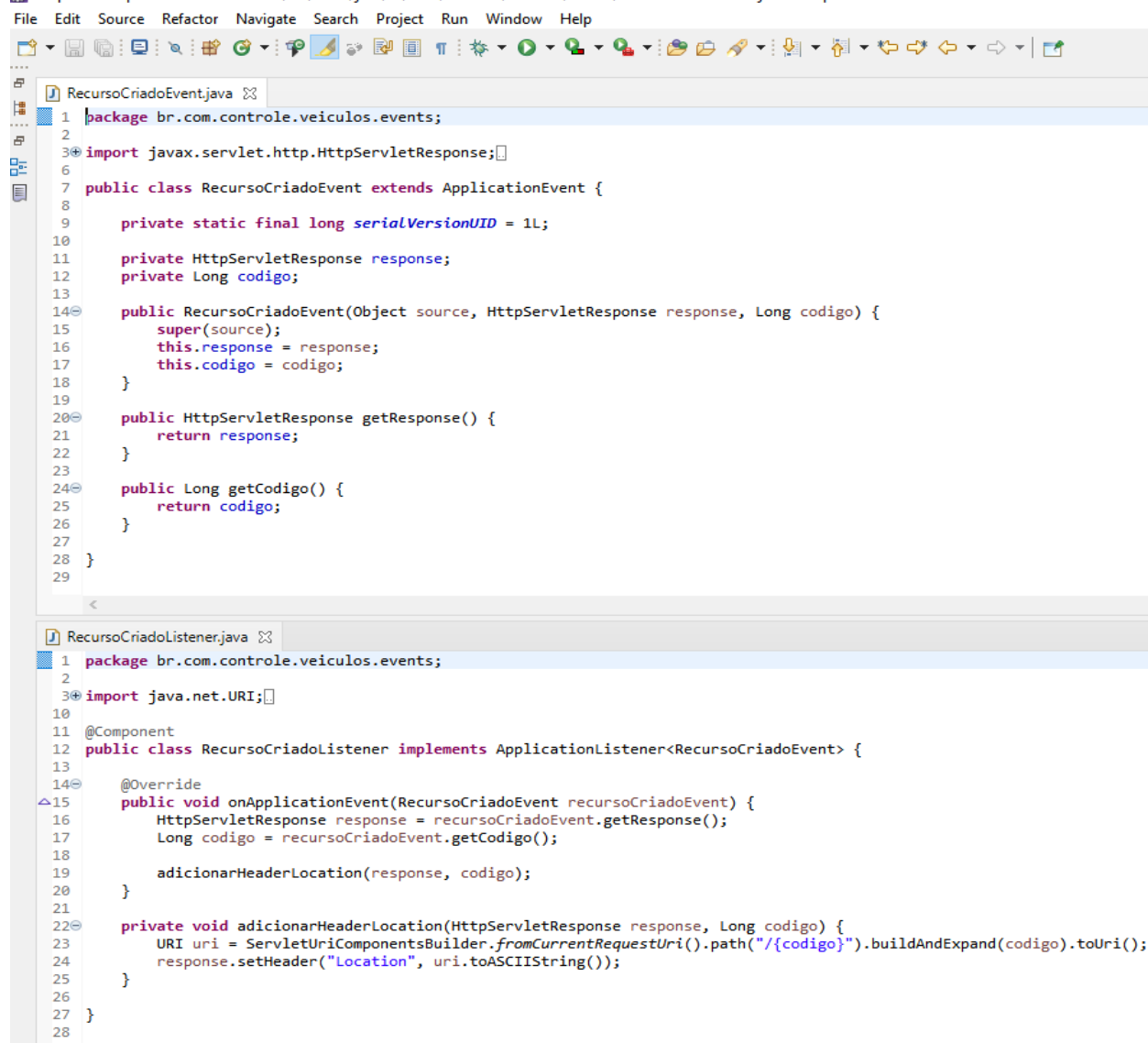
Para se ter uma ideia a mensagem de erro que normalmente ocorre quando não são realizados os tratamentos, note que o campo nome não foi preenchido e algo parecido com esta tela:



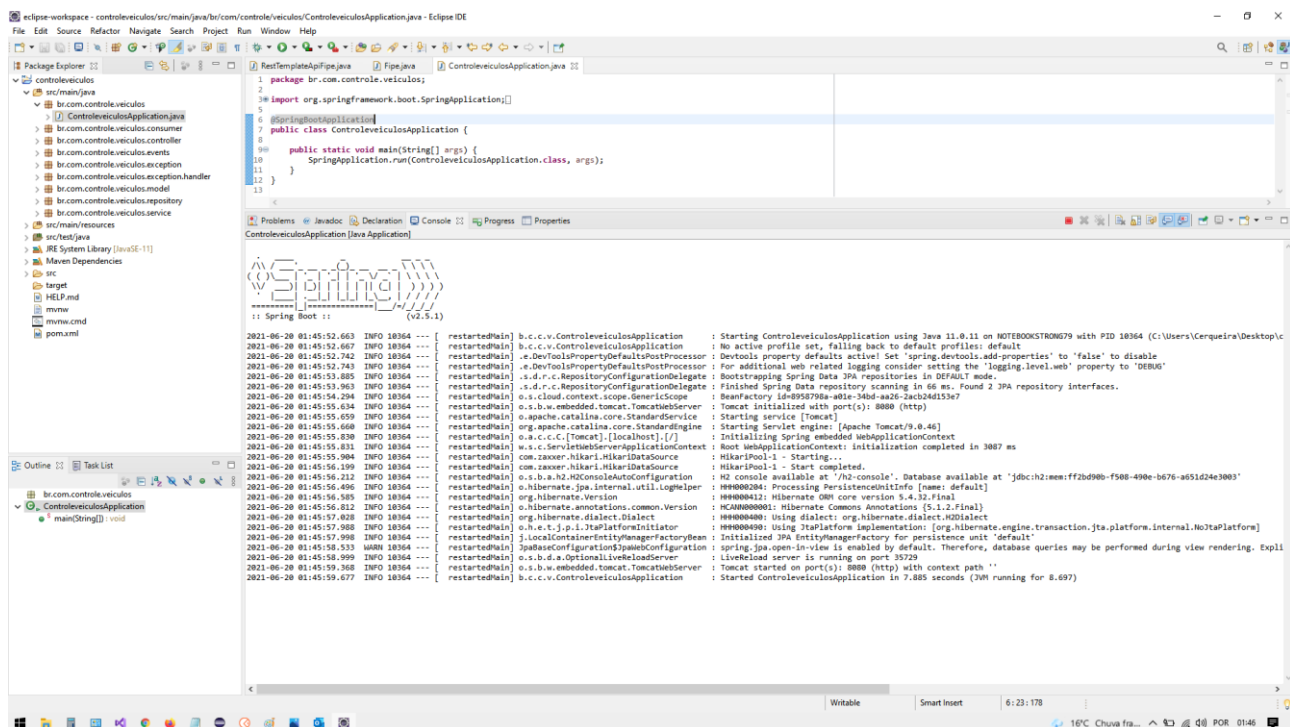
Classe RecursoCriadoListener e Classe RecursoCriadoEvent

A classe RecursoCriadoListener fica ouvindo(monitorando) o que está ocorrendo e repassa para a classe RecursoCriadoEvent que irá passar a referencia direto no nosso Header passando uma **URI** de qual local o recuso foi criado, exemplo: Location: **http://localhost:8080/CadastroUsuario/1**, detalhe todas essas informações serão vistas no Postman.

eclipse-workspace - controleveiculos/src/main/java/br/com/controle/veiculos/events/RecursoCriadoEvent.java - Eclipse IDE

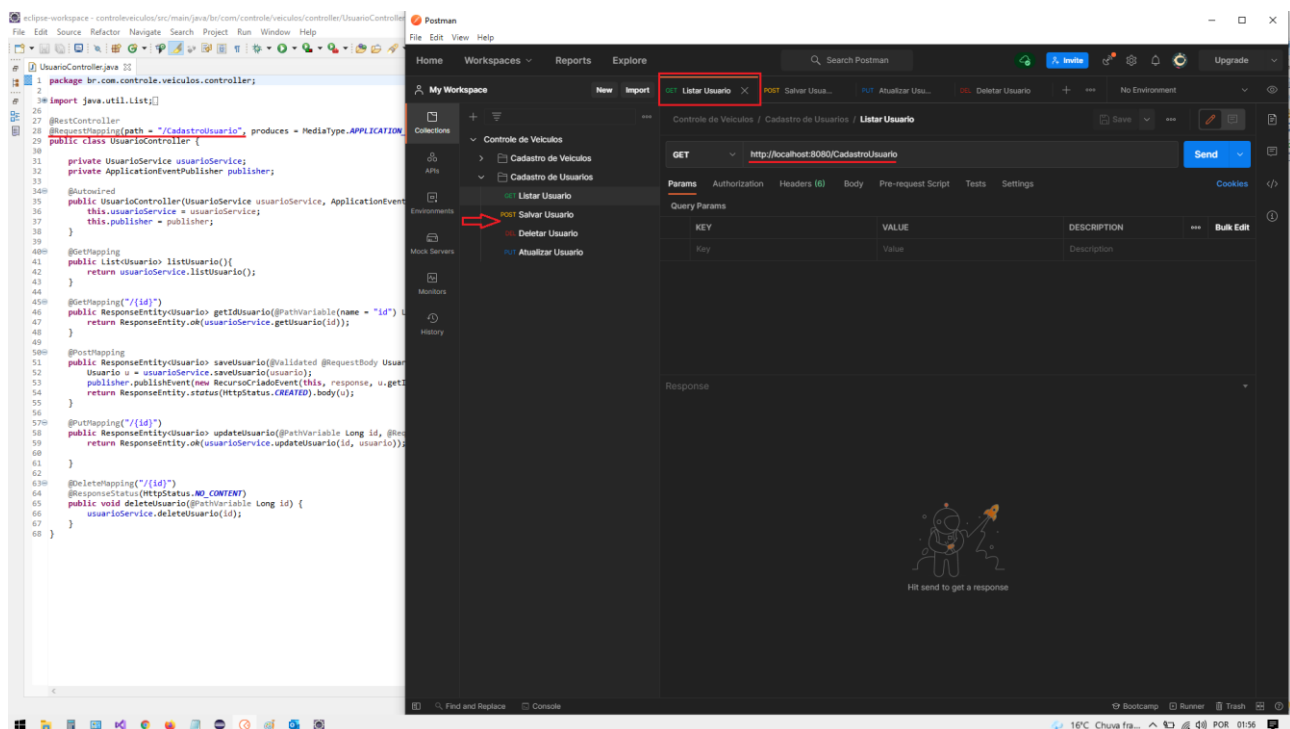


INICIALIZANDO NOSSA APLICAÇÃO SPRING BOOT E ACESSANDO Á ATRAVES DO POSTMAN:

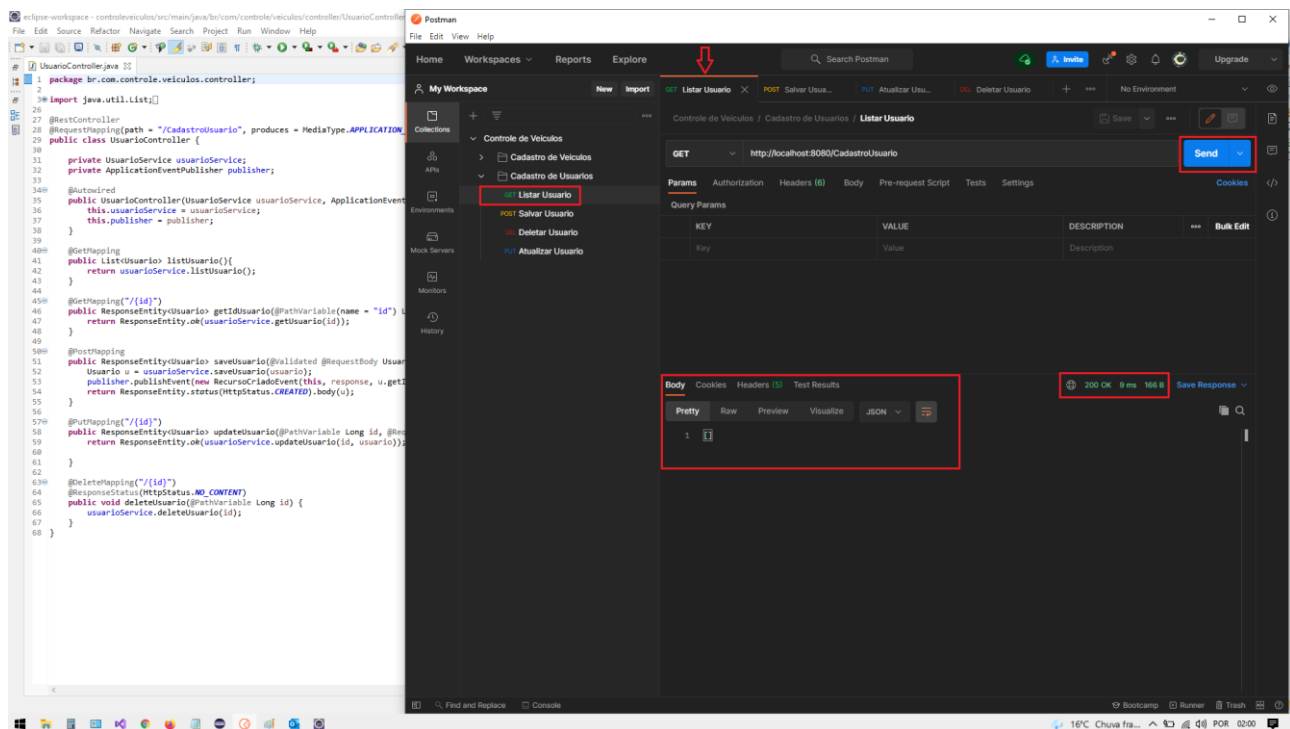


Note bem, onde nosso endPoint está sendo referenciado conforme definido na classe **UsuarioController**, <http://localhost:8080/CadastroUsuario>.

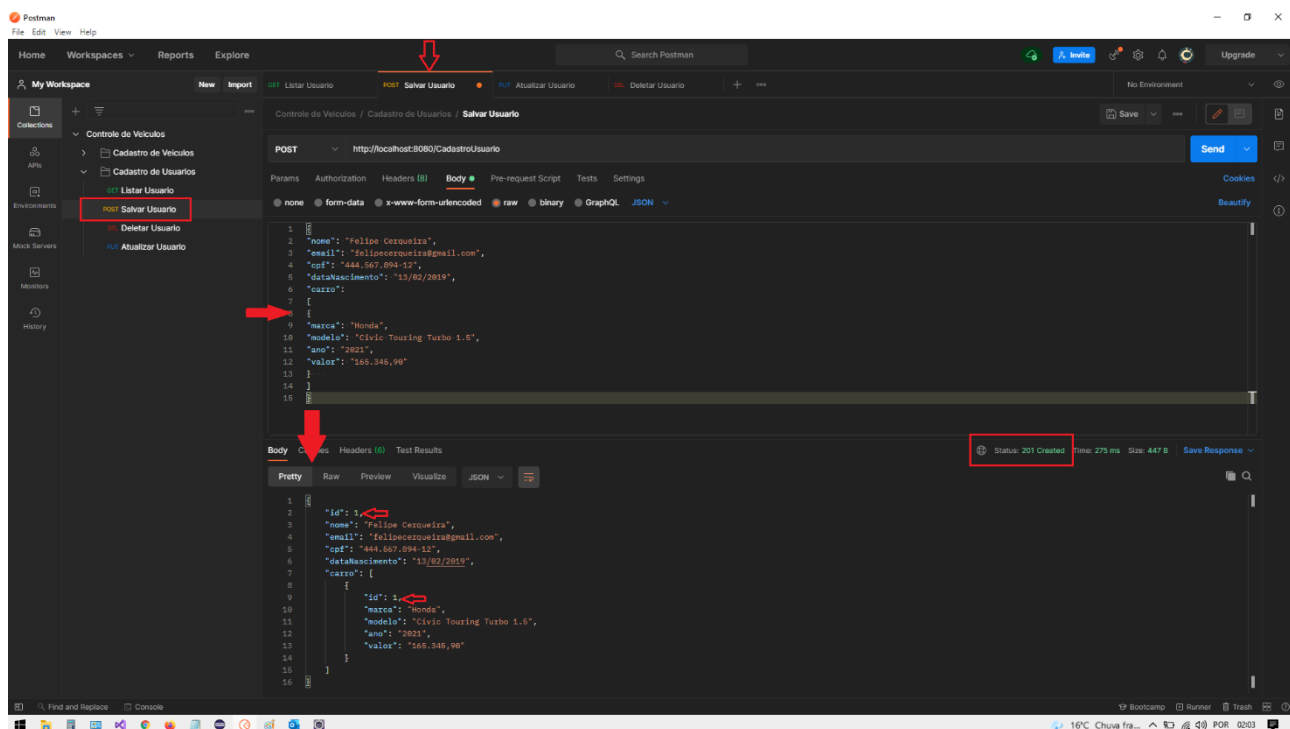
Os métodos **GET**(buscar), **POST**(gravar), **PUT**(atualizar) e **DELETE**(deletar) estão sendo acessado conforme imagem abaixo:



Ao clicar em enviar (Send) estou buscando uma lista de usuários, porém como não existe nenhum usuário cadastrado no sistema está **retornando corretamente** uma lista vazia e o **Status Http é 200**.



Adicionando um usuário no sistema com o seu carro e salvando esta informação no banco de dados, ao salvar as informações corretamente o **Status Http é 201**.



Verificando as informações salvas nas tabelas criadas no banco de dados H2.

H2 Console

localhost:8080/h2-console/login.do?jsessionid=c7a2864c4602a6098752dd5c8d959479

Auto commit ☒ Max rows: 1000 Auto complete Off Auto select On

jdbc:h2:mem:7a22a01d-fee5-42

- CARRO
- USUARIO
- INFORMATION_SCHEMA
- Sequences
- Users
- H2 1.4.200 (2019-10-14)

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM CARRO

SELECT * FROM CARRO;

ID	ANO	MARCA	MODELO	VALOR	ID_USUARIO
1	2021	Honda	Civic Touring Turbo 1.5	165.345,90	1

(1 row, 4 ms)

Edit

H2 Console

localhost:8080/h2-console/login.do?jsessionid=c7a2864c4602a6098752dd5c8d959479

Auto commit ☒ Max rows: 1000 Auto complete Off Auto select On

jdbc:h2:mem:7a22a01d-fee5-42

- CARRO
- USUARIO
- INFORMATION_SCHEMA
- Sequences
- Users
- H2 1.4.200 (2019-10-14)

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM USUARIO

SELECT * FROM USUARIO;

ID	CPF	DATA_NASCIMENTO	EMAIL	NOME
1	444.567.894-12	13/02/2019	felipecerqueira@gmail.com	Felipe Cerqueira

(1 row, 3 ms)

Edit

H2 Console

localhost:8080/h2-console/login.do?jsessionid=c7a2864c4602a6098752dd5c8d959479

Auto commit ☒ Max rows: 1000 Auto complete Off Auto select On

jdbc:h2:mem:7a22a01d-fee5-42

- CARRO
 - ID
 - ANO
 - MARCA
 - MODELO
 - VALOR
 - ID_USUARIO
- USUARIO
 - ID
 - CPF
 - DATA_NASCIMENTO
 - EMAIL
 - NOME
- INFORMATION_SCHEMA
- Sequences
- Users
- H2 1.4.200 (2019-10-14)

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM USUARIO
INNER JOIN CARRO

SELECT * FROM USUARIO
INNER JOIN CARRO;

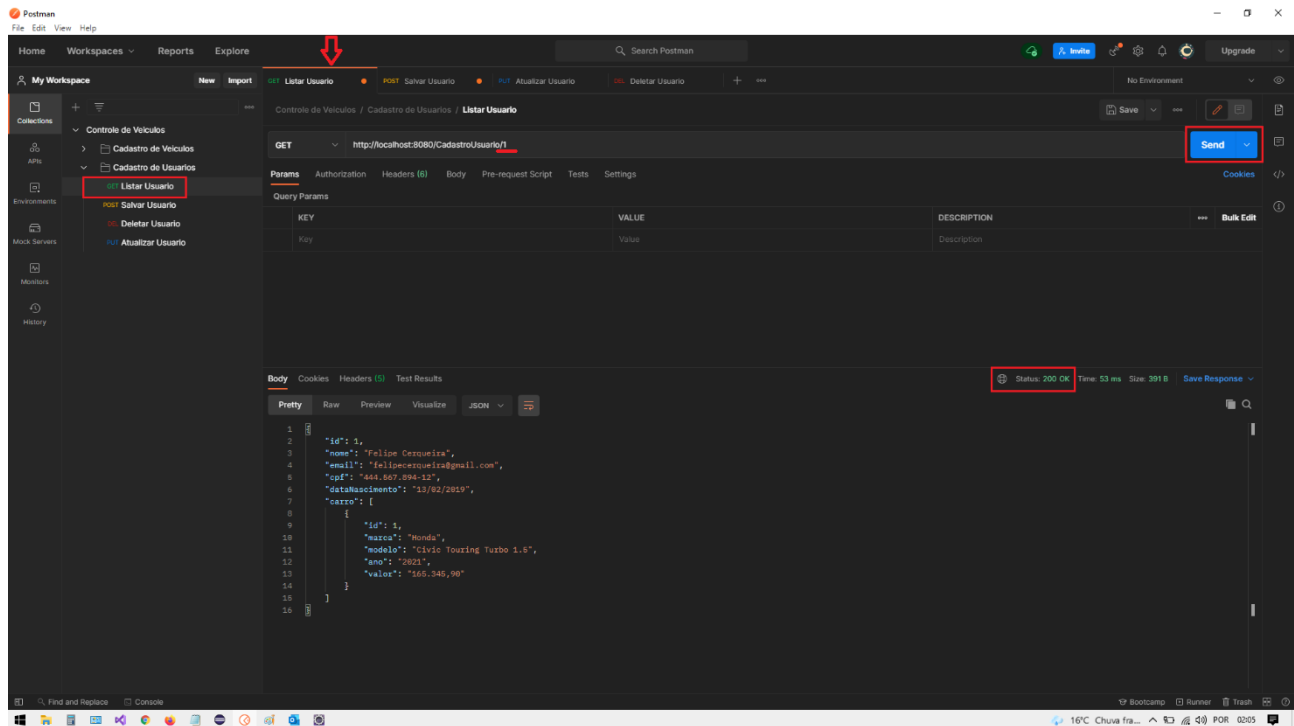
ID	CPF	DATA_NASCIMENTO	EMAIL	NOME	ID	ANO	MARCA	MODELO	VALOR	ID_USUARIO
1	444.567.894-12	13/02/2019	felipecerqueira@gmail.com	Felipe Cerqueira	1	2021	Honda	Civic Touring Turbo 1.5	165.345,90	1

(1 row, 1 ms)

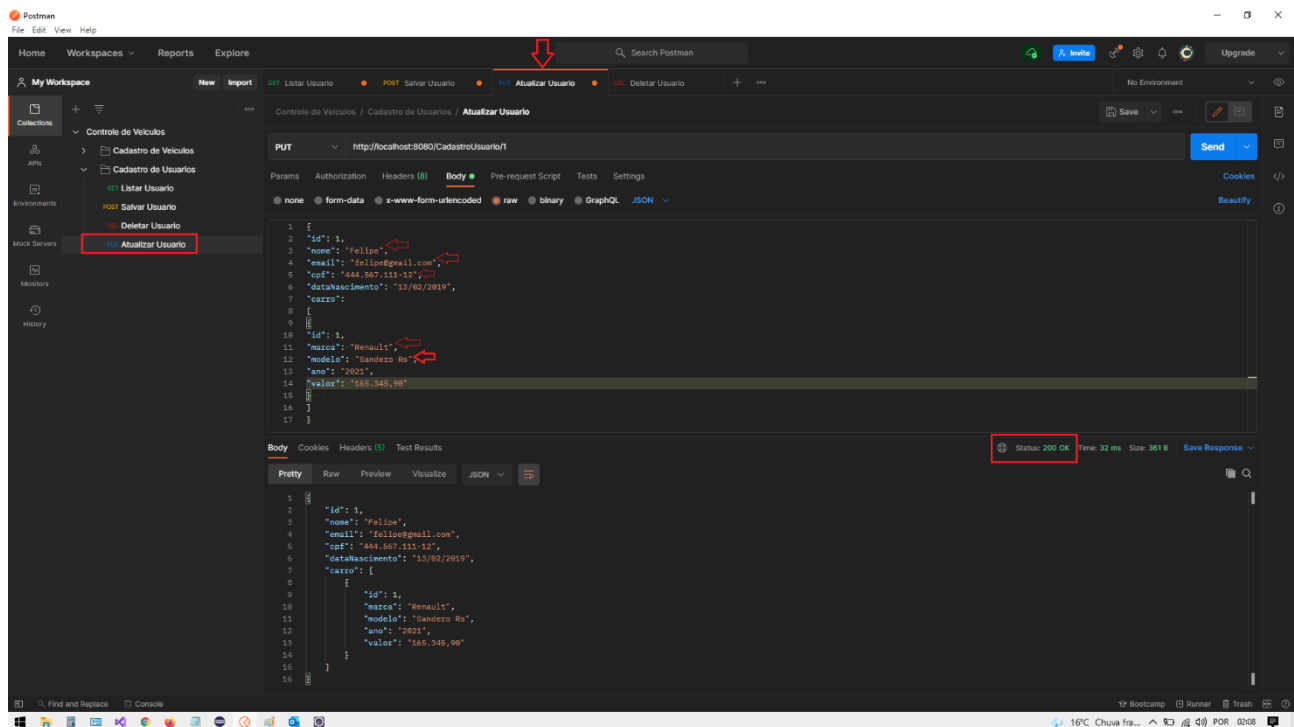
Como as informações estão salvas no banco de dados, vamos buscar esse usuário cadastrado novamente na Lista de UsuarioCadastrados através do GET, no primeiro momento em que está lista realmente não havia nada e agora retorna os dados salvos.

Porem com detalhe como estamos realizando uma busca definimos que será feita pelo seu id por isso no final da URI tem o numero correspondente no caso 1, exemplo:

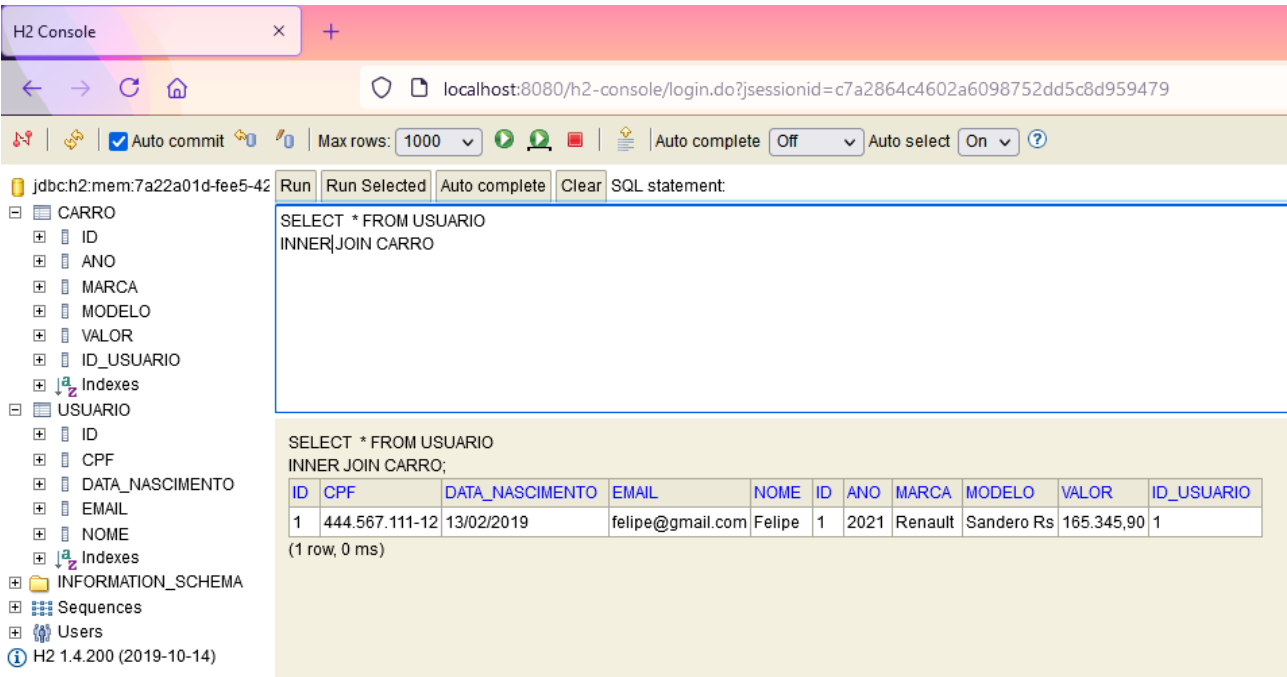
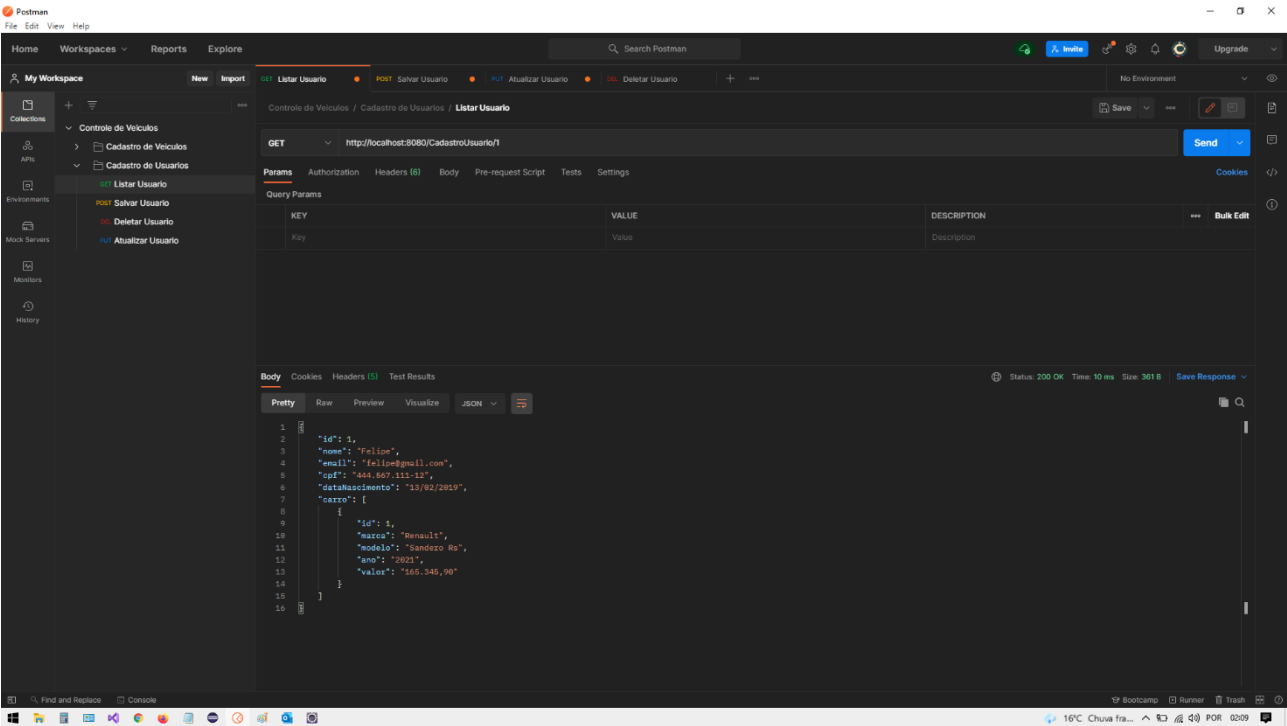
http://localhost:8080/CadastroUsuario/1.



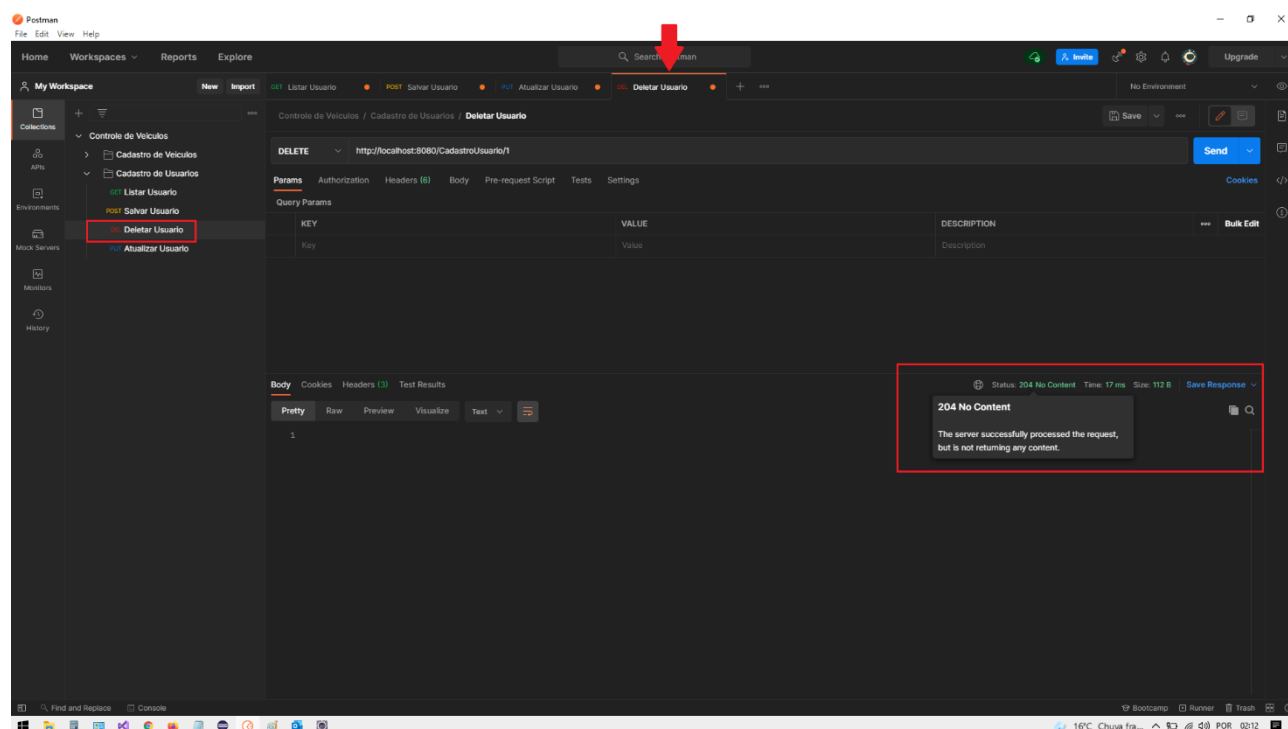
Atualizando as informações dos dados inseridos e depois pesquisando as mesmas ao listar usuários cadastrados com os respectivos carros.



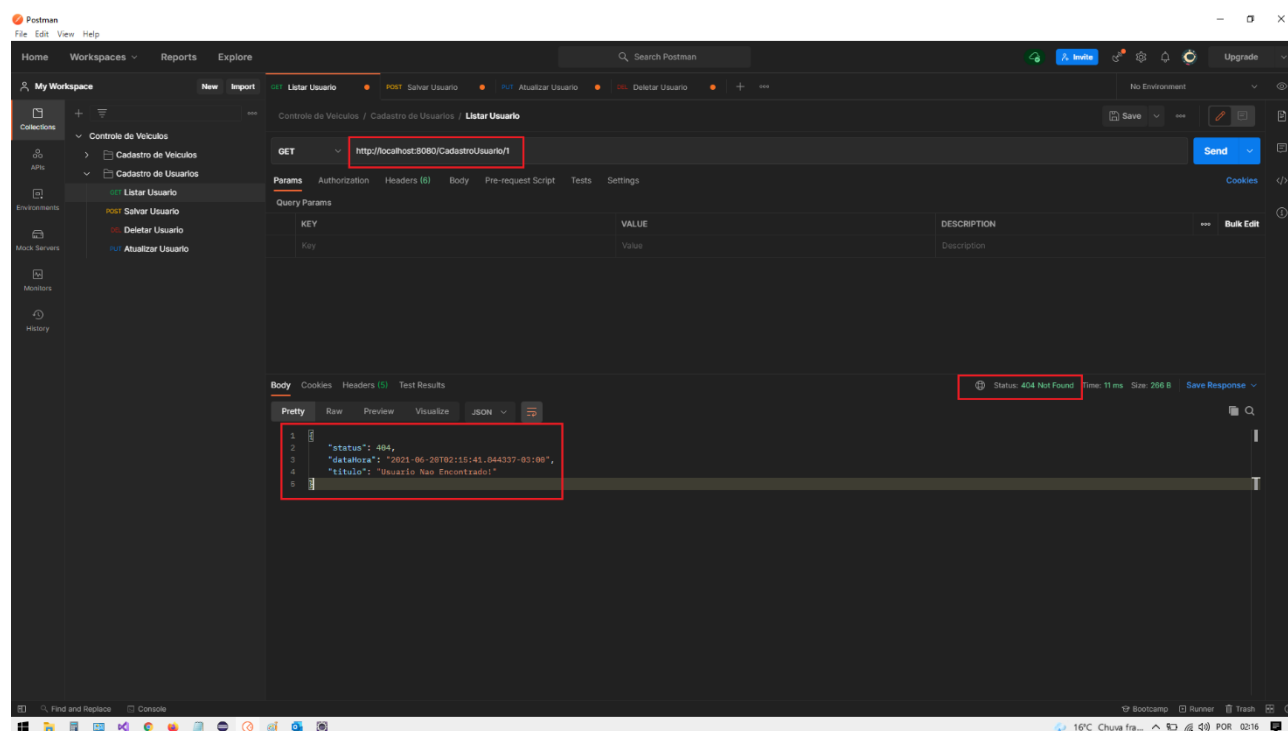
Os dados foram atualizados corretamente tanto na busca pelo usuário quanto no banco de dados.



Agora iremos excluir o cadastro que foi realizado e a informação do retorno está correta **Status** **Http 204**.



E se tentarmos realizar a busca novamente iremos receber uma mensagem mais amigável ao usuário, uma vez que não existe o usuário com o ID 1.



E no banco de dados não consta mais nenhuma informação:

The screenshot shows the H2 Console interface. On the left, a tree view displays the database schema with tables: CARRO (ID, ANO, MARCA, MODELO, VALOR, ID_USUARIO), USUARIO (ID, CPF, DATA_NASCIMENTO, EMAIL, NOME), and INFORMATION_SCHEMA. The main area shows a SQL statement: `SELECT * FROM USUARIO INNER JOIN CARRO`. Below the statement, a table of results is shown with columns: ID, CPF, DATA_NASCIMENTO, EMAIL, NOME, ID, ANO, MARCA, MODELO, VALOR, ID_USUARIO. The status indicates "(no rows, 1 ms)".

Agora estaremos verificando a obrigatoriedade dos campos, caso deixemos algumas informações em branco. Nos dados do usuário os campos: **cpf** e **data de nascimento** não foram preenchidos, nas informações do carro os campos: **marca** e **modelo** também não foram preenchidos. O sistema deve passar uma mensagem amigável dos campos e sua respectiva obrigatoriedade.

The screenshot shows the Postman interface. A POST request is sent to `http://localhost:8080/CadastroUsuario`. The request body is a JSON object with the following fields: `nome`, `email`, `cpf`, `dataNascimento`, `carro` (an array of objects for `marca`, `modelo`, `ano`, and `valor`). The response status is 400 Bad Request. The response body is a JSON object with the following fields: `status`, `dataHora`, `titulo`, and `campos` (an array of objects for `carro[0].modelo`, `carro[0].marca`, `dataNascimento`, and `cpf`). The response body is highlighted with red boxes.

Após a criação dos cadastros foram feitas as classes, referente ao consumo do serviço API FIPE.

```
eclipse-workspace - controleveiculos/src/main/java/br/com/controle/veiculos/consumer/Fipe.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Fipe.java
1 package br.com.controle.veiculos.consumer;
2
3 import javax.persistence.Id;
4
5 public class Fipe {
6
7     @Id
8     private Long id;
9
10    private String marcas;
11    private String modelos;
12    private String anos;
13    private String valor;
14
15    public String getMarcas() {
16        return marcas;
17    }
18    public void setMarcas(String marcas) {
19        this.marcas = marcas;
20    }
21    public String getModelos() {
22        return modelos;
23    }
24    public void setModelos(String modelos) {
25        this.modelos = modelos;
26    }
27    public String getAnos() {
28        return anos;
29    }
30    public void setAnos(String anos) {
31        this.anos = anos;
32    }
33    public String getValor() {
34        return valor;
35    }
36    public void setValor(String valor) {
37        this.valor = valor;
38    }
39
40    @Override
41    public int hashCode() {
42        final int prime = 31;
43        int result = 1;
44        result = prime * result + ((id == null) ? 0 : id.hashCode());
45        return result;
46    }
47    @Override
48    public boolean equals(Object obj) {
49        if (this == obj)
50            return true;
```

```
eclipse-workspace - controleveiculos/src/main/java/br/com/controle/veiculos/repository/FipeRepository.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer
controleveiculos
src/main/java
br.com.controle.veiculos
ControleveiculosApplication.java
br.com.controle.veiculos.consumer
Fipe.java
RestTemplateApiFipe.java
br.com.controle.veiculos.controller
CarroController.java
FipeController.java
ListagemController.java
UsuarioController.java
br.com.controle.veiculos.events
RecursoCriadoEvent.java
RecursoCriadoListener.java
br.com.controle.veiculos.exception
EntidadeNaoEncontradoException.java
RegraException.java
br.com.controle.veiculos.exception.handler
Problemas.java
StatusValidacoesExceptionHandler.java
br.com.controle.veiculos.model
Carro.java
Usuario.java
br.com.controle.veiculos.repository
CarroRepository.java
FipeRepository.java
UsuarioRepository.java
br.com.controle.veiculos.service
CarroService.java
UsuarioService.java

FipeController.java
1 package br.com.controle.veiculos.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5 @RestController
6 @RequestMapping("https://parallelum.com.br/fipe/api/v1/carros/marcas")
7 public class FipeController {
8
9     @Autowired
10    FipeRepository fipeRepository;
11
12    @GetMapping()
13    public Fipe getFipe() {
14        return fipeRepository.getByFipe("https://parallelum.com.br/fipe/api/v1/carros/marcas");
15    }
16 }

FipeRepository.java
1 package br.com.controle.veiculos.repository;
2
3 import org.springframework.cloud.openfeign.FeignClient;
4
5 @Repository
6 @FeignClient(value = "jplaceholder", url = "https://jsonplaceholder.typicode.com/")
7 public interface FipeRepository extends JpaRepository<Fipe, Long> {
8
9     public Fipe getByFipe(String valor);
10 }

FipeRepository.java
1 package br.com.controle.veiculos.repository;
2
3 import org.springframework.cloud.openfeign.FeignClient;
4
5 @Repository
6 @FeignClient(value = "jplaceholder", url = "https://jsonplaceholder.typicode.com/")
7 public interface FipeRepository extends JpaRepository<Fipe, Long> {
8
9     public Fipe getByFipe(String valor);
10 }
```

E também foi criado mais um EndPoint de Listagem de veículos

```
eclipse-workspace - controleveiculos/src/main/java/br/com/controle/veiculos/controller/ListagemController.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer
controleveiculos
src/main/java
br.com.controle.veiculos
ControleveiculosApplication.java
br.com.controle.veiculos.consumer
Fipe.java
RestTemplateApiFipe.java
br.com.controle.veiculos.controller
CarroController.java
FipeController.java
ListagemController.java
UsuarioController.java
br.com.controle.veiculos.events
RecursoCriadoEvent.java
RecursoCriadoListener.java
br.com.controle.veiculos.exception
EntidadeNaoEncontradoException.java
RegraException.java

ListagemController.java
1 package br.com.controle.veiculos.controller;
2
3 import java.util.List;
4
5 @RestController
6 @RequestMapping(path = "/ListagemVeiculos", produces = MediaType.APPLICATION_JSON_VALUE)
7 public class ListagemController {
8
9     @Autowired
10    CarroRepository carroRepository;
11
12    @GetMapping
13    public List<Carro> listCarros(){
14        return carroRepository.findAll();
15    }
16 }

ListagemController.java
1 package br.com.controle.veiculos.controller;
2
3 import java.util.List;
4
5 @RestController
6 @RequestMapping(path = "/ListagemVeiculos", produces = MediaType.APPLICATION_JSON_VALUE)
7 public class ListagemController {
8
9     @Autowired
10    CarroRepository carroRepository;
11
12    @GetMapping
13    public List<Carro> listCarros(){
14        return carroRepository.findAll();
15    }
16 }
```