Desenvolvimento de APIs REST

06 - Relacionamento entre entidades (continuação)

- Relacionamentos Um para Um
- Relacionamentos Um para Muitos / Muitos para Um
- Relacionamentos Muitos para Muitos
- Operações em Cascata

serratec
Parque Tecnológico Região Serrana

Revisão

- H2 - Banco em Memória e Disco
- Relacionamento Embedded
- Enum e Exceções
- Herança

Relacionamento Um para Um

O relacionamento um para um, também conhecido como one-to-one, é utilizado para dividir uma tabela em duas, para deixar as informações de forma mais organizada.

Utilizando o projeto da **aula anterior** onde utilizamos o h2 como banco de dados.
inserir a classe **Proprietario** abaixo:

A linha abaixo é opcional para classe **Proprietario** geralmente mapeamos somente o lado onde vai ter a chave estrangeira do relacionamento.

```
@OneToOne(mappedBy = "proprietario")
private Veiculo veiculo;
```

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

    private String nome;


    //... gets e sets
    @Override
    public int hashCode() {
        return Objects.hash(id);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Proprietario other = (Proprietario) obj;
        return Objects.equals(id, other.id);
    }
}
```

- Inserir **Getter** e **Setter**
- **Equals e HashCode** para o atributo **id**

Relacionamento Um para Um

```java
@Entity
@Table(name="veiculo")
public class Veiculo {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @NotBlank(message="Preencha a placa")
   @Size(max=7)
   @Column(nullable = false, length=7)
   private String placa;

   @NotBlank(message="Preencha a marca")
   @Size(max=30)
   @Column(nullable = false, length=30)
   private String marca;

   @NotBlank(message="Preencha o modelo")
   @Size(max=40)
   @Column(nullable = false, length=40)
   private String modelo;

   @Embedded @Valid
   private Caracteristica caracteristica;

   @OneToOne
   @JoinColumn(name="id_proprietario")
   private Proprietario proprietario;
```

Adicionar o atributo proprietário na classe **Veiculo**

Podemos mapear os dois lados do relacionamento, mas geralmente mapeamentos no lado que contém a chave estrangeira do relacionamento.

A propriedade **JoinColumn** é onde definimos qual será a chave estrangeira na tabela.

Relacionamento Um para Um

Vamos executar a aplicação, podemos verificar o resultado no console do STS e também no **h2**.

```
2022-09-04 13:10:59.808  INFO 1410504 --- [  restartedMain] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing Pers
Hibernate: alter table veiculo add column id_proprietario bigint
Hibernate: alter table veiculo add constraint FK9odaviqr8n5wtjkbuior6dil4 foreign key (id_proprietario) references proprietario
```

Verificando as alterações no **h2**
http://localhost:8080/h2-console/

Relacionamento Um para Um

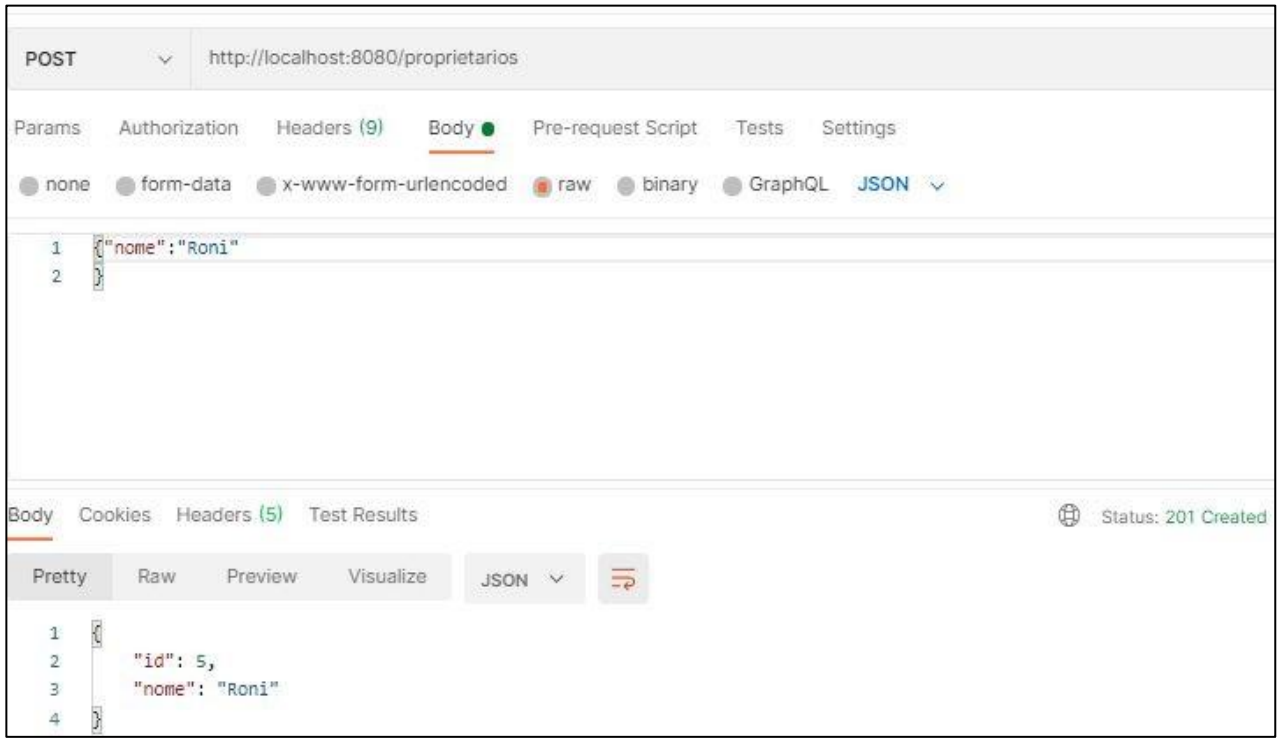
Vamos criar a interface **ProprietarioRepository** e a classe **ProprietarioController**

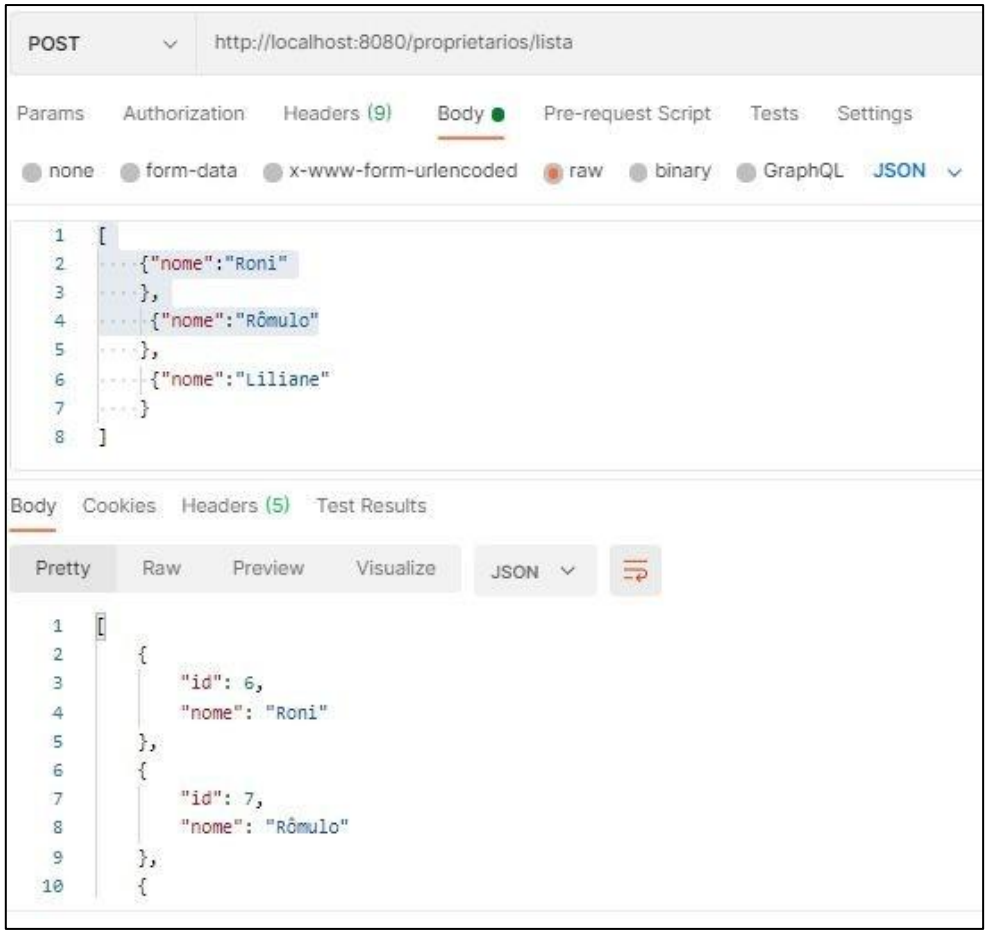
```
@Repository
public interface ProprietarioRepository extends JpaRepository<Proprietario, Long>{

}
```

```
@RestController
public class ProprietarioController {
    @Autowired
    private ProprietarioRepository proprietarioRepository;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Proprietario inserir(@RequestBody Proprietario proprietario){
        return proprietarioRepository.save(proprietario);
    }

    @PostMapping("/lista")
    @ResponseStatus(HttpStatus.CREATED)
    public List<Proprietario> inserirVarios(@RequestBody List<Proprietario> proprietarios) {
        return proprietarioRepository.saveAll(proprietarios);
    }

    @GetMapping
    public ResponseEntity<List<Proprietario>> listar(){
        List<Proprietario> proprietarios = proprietarioRepository.findAll();
        return ResponseEntity.ok(proprietarios);
    }

}
```

Relacionamento Um para Um

Testando no Postman as inclusões

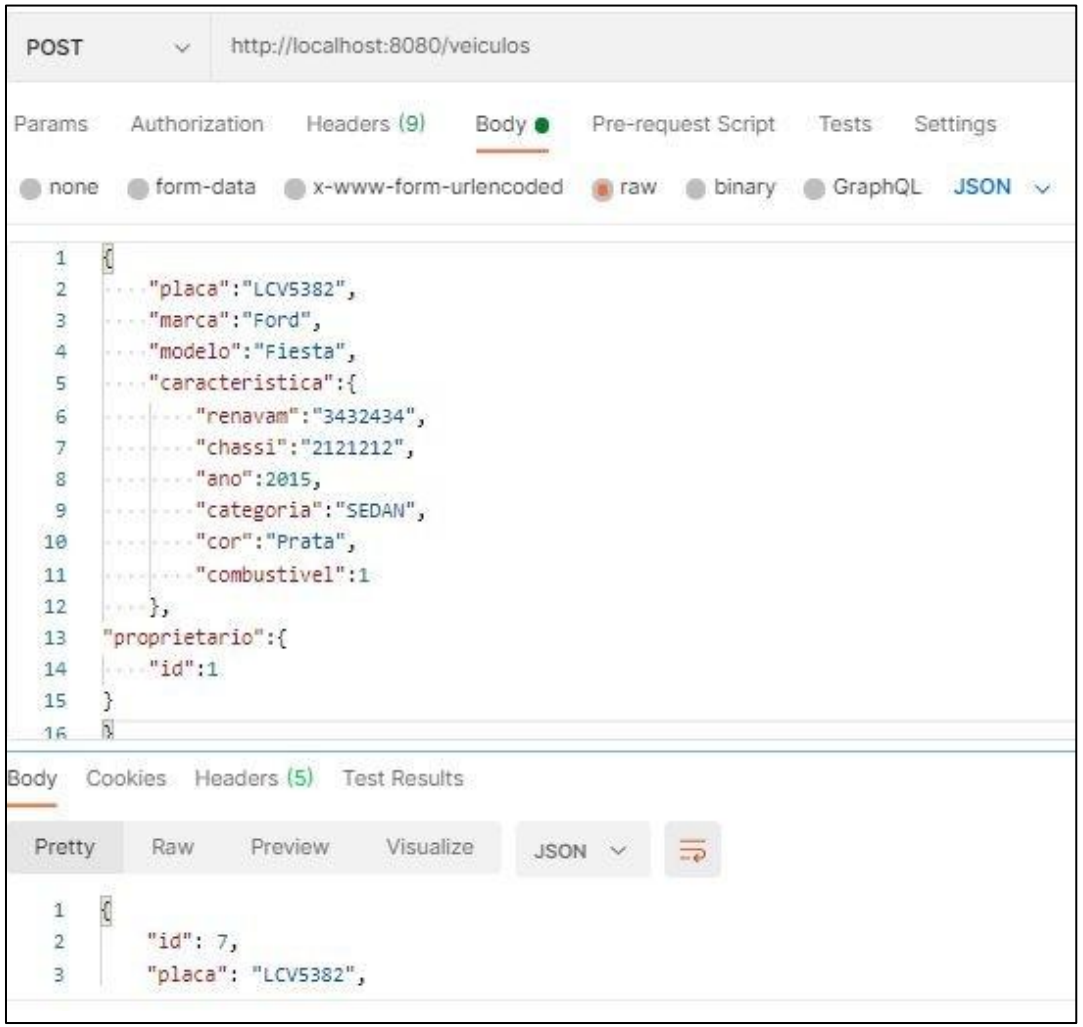




serratec

Relacionamento Um para Um

Testando no Postman o cadastro do proprietário do veículo. Não é necessário preencher todos os campos somente o id do veículo.



serratec

Fetch.Lazy e Fetch.Eager

fetch.LAZY (buscar preguiçoso)

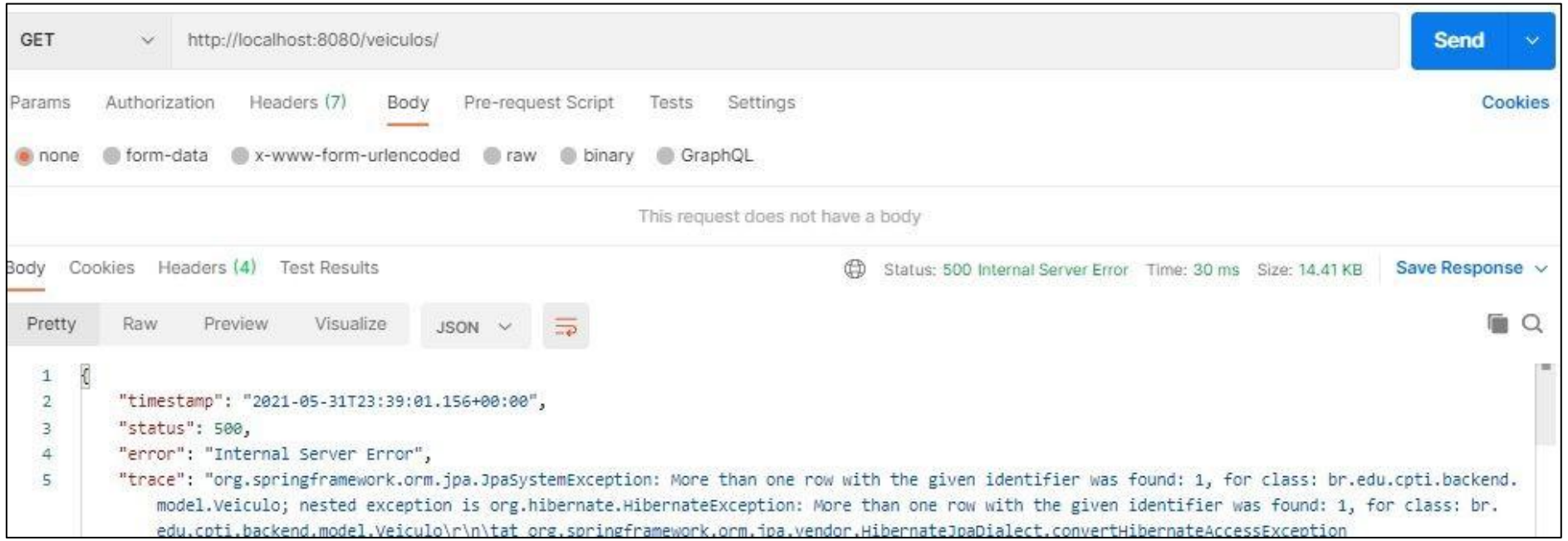
Os mapeamentos do Hibernate são Lazy por padrão. Quando acessamos o objeto ele não traz instantaneamente todas as dependências. Somente quando precisamos do atributo dependente é que a pesquisa é realizada, economizando memória.

fetch.EAGER (buscar ansioso)

Retorna tudo que está dentro do objeto, se existir um relacionamento 1 para N, será carregado todas as referências dele.
Obs: Esta propriedade tem problemas com performance, não sendo quase utilizado.

Fetch.Lazy e Fetch.Eager

Ao listar todos veículos temos o erro abaixo:



Inserir na classe Veiculo:

```
@OneToOne(fetch = FetchType.EAGER)
   @JoinColumn(name="id_proprietario")
   private Proprietario proprietario;
```

Um para Muitos / Muitos para Um

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

    @Column(name="data_entrada")
    private LocalDate dataEntrada;

    @Column(name="data_saida")
    private LocalDate dataSaida;

    @Column
    private String obs;

    @ManyToOne
    @JoinColumn(name="id_veiculo")
    private Veiculo veiculo;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

Vamos inserir a classe **Manutencao**

Várias manutenções para um determinado veículo.
@JoinColumn – Estamos dizendo qual é a coluna da tabela que é a chave estrangeira.

Inserir **Getter** e **Setter**
Equals e HashCode para o atributo **id**

Um para Muitos / Muitos para Um

Vamos inserir o atributo **manutencoes** em destaque na classe **Veiculo**

Um veículo pode ter uma ou várias manutenções

mappedBy – Estamos dizendo qual é o nome do atributo da classe **Manutencao** que contém o relacionamento

```
@Entity
@Table(name="veiculo")
public class Veiculo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message="Preencha a placa")
    @Size(max=7)
    @Column(nullable = false, length=7)
    private String placa;

    @NotBlank(message="Preencha a marca")
    @Size(max=30)
    @Column(nullable = false, length=30)
    private String marca;

    @NotBlank(message="Preencha o modelo")
    @Size(max=40)
    @Column(nullable = false, length=40)
    private String modelo;

    @Embedded @Valid
    private Caracteristica caracteristica;

    @OneToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="id_proprietario")
    private Proprietario proprietario;

    @OneToMany(mappedBy = "veiculo")
    private List<Manutencao> manutencoes;
```

Um para Muitos / Muitos para Um

Executando a aplicação verificamos no console que o hibernate criou a tabela **manutencao** no banco de dados

```
Hibernate: create table manutencao (id bigint generated by default as identity, data_entrada date, data_saida date, obs varchar(255), id_veiculo bigint, primary key (id))
Hibernate: alter table manutencao add constraint FKdfe27xk4hnghxvnrfvnsh6a0d foreign key (id_veiculo) references veiculo
```

Verificando no **h2** a tabela foi criada com a chave estrangeira da tabela **veiculo**

Um para Muitos / Muitos para Um

```java
@RestController
@RequestMapping("/manutencoes")
public class ManutencaoController {
    @Autowired
    private ManutencaoRepository manutencaoRepository;
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Manutencao inserir(@RequestBody Manutencao manutencao) {
        return manutencaoRepository.save(manutencao);
    }
    @GetMapping
    public ResponseEntity<List<Manutencao>> listar(){
        List<Manutencao> manutencoes = manutencaoRepository.findAll();
        return ResponseEntity.ok(manutencoes);
    }
    @GetMapping("{id}")
    public ResponseEntity<Manutencao> buscar(@PathVariable Long id) {
        Optional<Manutencao> manutencao = manutencaoRepository.findById(id);
        if (manutencao.isPresent()){
            return ResponseEntity.ok(manutencao.get());
        }
        return ResponseEntity.notFound().build();
    }

}
```

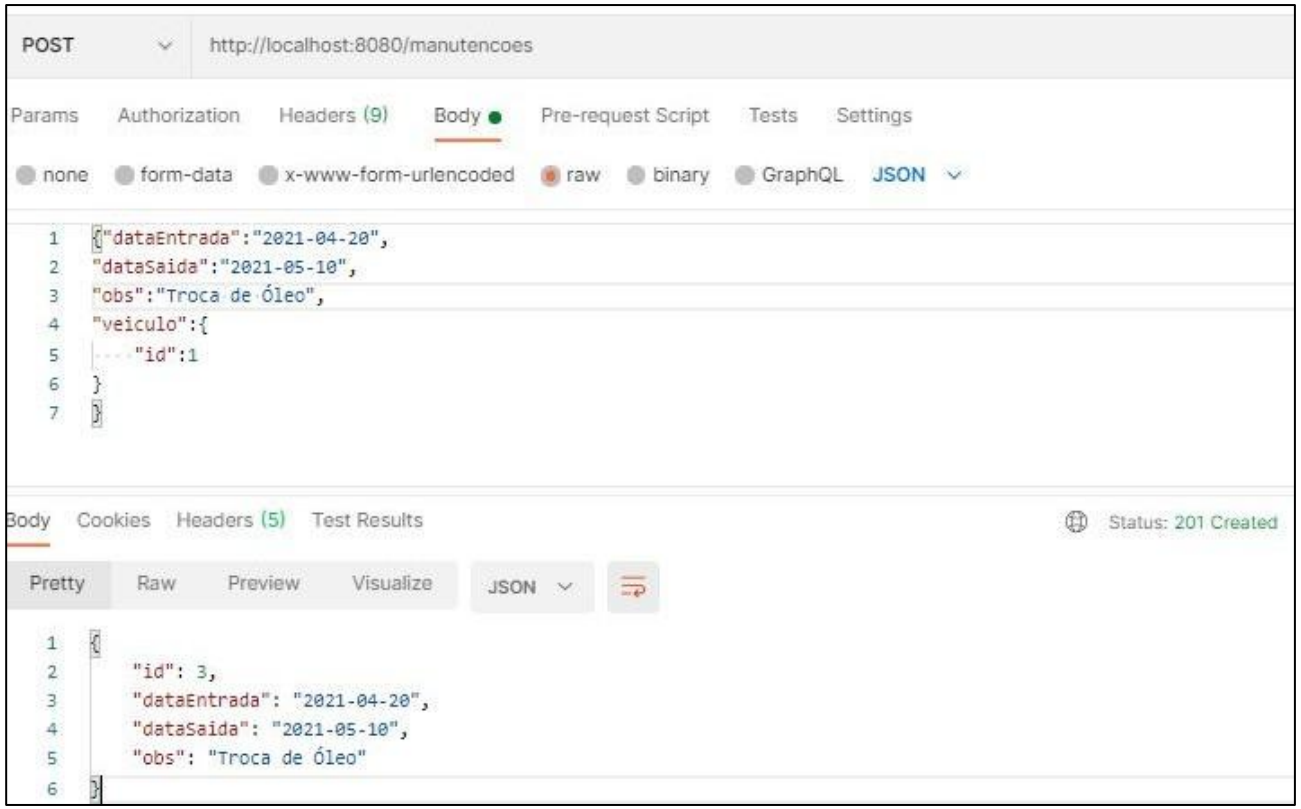
Inserir o **controller** e o **repository** para **manutencao**

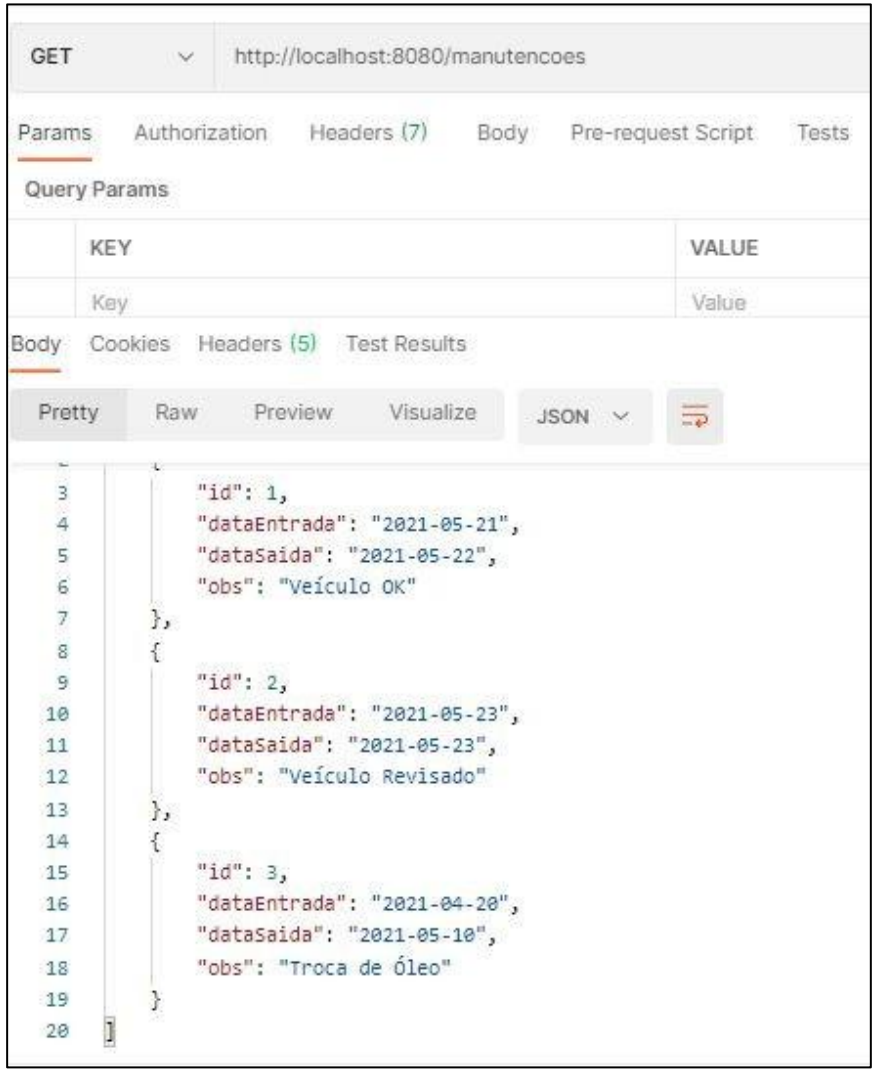
```java
@Repository
public interface ManutencaoRepository extends
            JpaRepository<Manutencao,Long>{

}
```

serratec

Um para Muitos / Muitos para Um

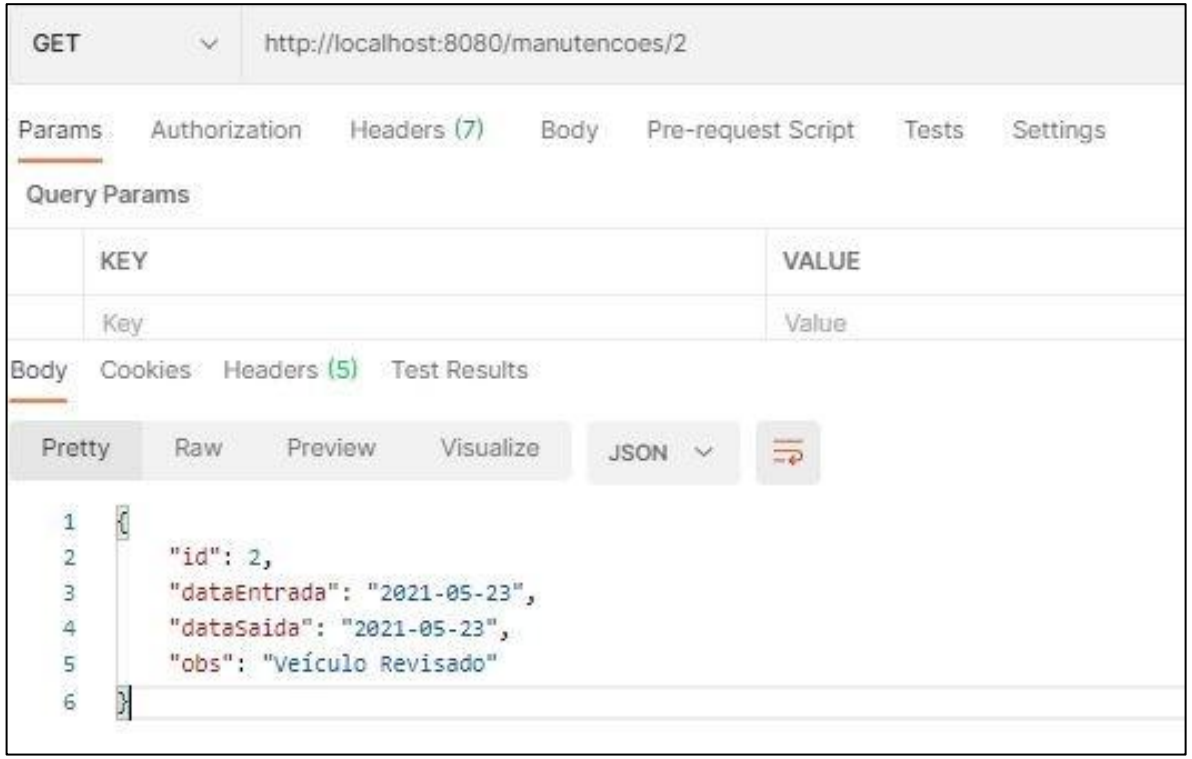
Testando no Postman

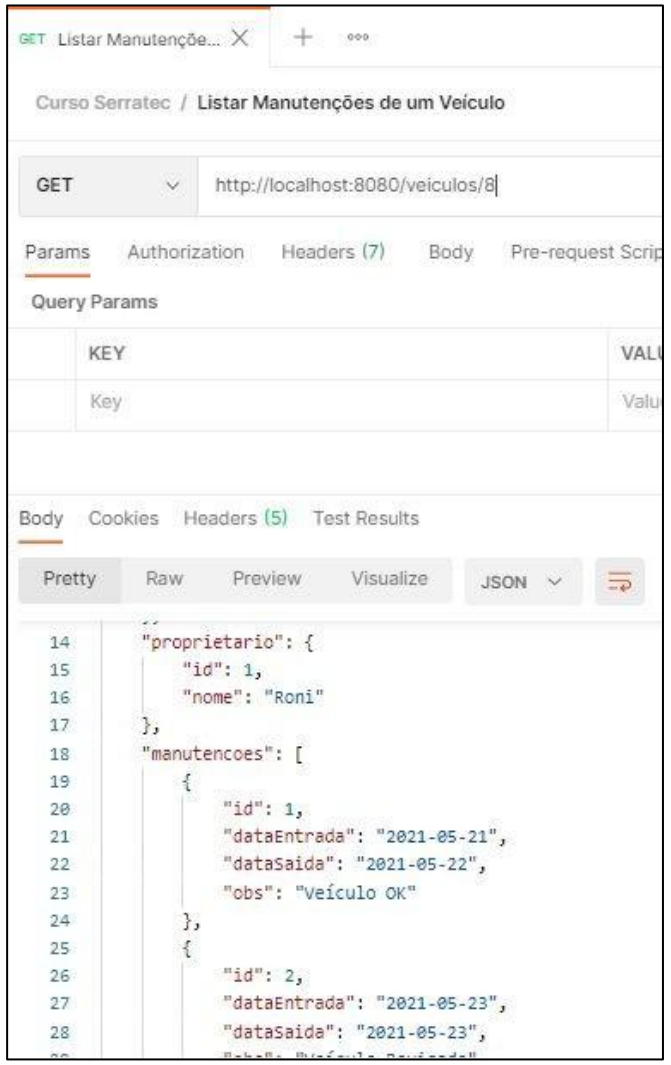




Um para Muitos / Muitos para Um

Testando no Postman





serratec

Muitos para Muitos

Vamos criar a classe **Servico** para utilizar neste exemplo

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

   @Column
   private String descricao;

   @Column
   private BigDecimal valor;

   public Long getId() {
       return id;
   }

   public void setId(Long id) {
       this.id = id;
   }
```

Inserir **Getter** e **Setter**
Equals e HashCode para o atributo **id**

Muitos para Muitos

Vamos inserir o atributo **serviços** na classe **Manutencao**

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

    @Column(name="data_entrada")
    private LocalDate dataEntrada;

    @Column(name="data_saida")
    private LocalDate dataSaida;

    @Column
    private String obs;

    @ManyToOne
    @JoinColumn(name="id_veiculo")
    private Veiculo veiculo;

    @ManyToMany
    @JoinTable(name="manutencao_servico",
    joinColumns = @JoinColumn(name="id_manutencao"),
    inverseJoinColumns = @JoinColumn(name="id_servico"))
    private List<Servico> servicos;
```

Um veículo pode ter mais de um tipo de serviço para fazer no dia da manutenção como por exemplo, troca de óleo, revisão e outros.
Um serviço pode estar em diversas manutenções de um determinado veículo.

Uma manutenção de um veículo pode ter diversos serviços.

Quando temos um relacionamento muitos para muitos é criada uma tabela intermediária para este modelo.

@JoinTable - serve para configurar a tabela intermediária. Precisamos ter duas colunas uma para referência ao tipo de serviço (id_servico) e outra para o identificador da manutenção (id_manutencao).

joinColumns - vamos configurar as colunas
@JoinColumn(name = "id_manutencao") - coluna referente a tabela **manutencao**

@inverseJoinColumns - responsável por mapear colunas do lado inverso do relacionamento.

@JoinColumn(name = "id_servico") - coluna referente a tabela **servico**

Muitos para Muitos

Vamos executar a aplicação e verificar no **h2**.

Cascade

```
@ManyToOne(cascade = CascadeType.ALL)
@ManyToMany(cascade = CascadeType.ALL)
@OneToOne(cascade = CascadeType.ALL)
```

- ALL - propaga todas as operações do “pai” para os “filhos”. Ex: efetuar uma operação e uma manutenção, todos os serviços associados sofreram a mesma operação (persistência, exclusão, etc)
- PERSIST - propaga as operações de persistência (insert)
- MERGE - operações de atualização (insert para “filhos” novos e update nos que foram alterados)
- REMOVE - ao remover o “pai”, os “filhos” também serão removidos
- DETACH - remove a entidades do “contexto” de persistência
- LOCK - “re-adiciona” a entidade ao “contexto” de persistência
- REFRESH - atualiza a instância com os dados do banco de dados
- REPLICATE - apenas quando existe mais de um banco de dados (replicação) e se deseja manter os dados sincronizados
- SAVE_UPDATE - propaga operações de inserção e update

Mais informações https://www.baeldung.com/jpa-cascade-types

Exercícios

Exercícios

Vamos acessar o projeto em que foi utilizado o banco de dados no **Postgres** para fazer este exercício.
Inserir o script abaixo no banco.

```
CREATE TABLE pedido(id_pedido serial PRIMARY KEY, data_pedido date, hora_pedido time, data_entrega date, id_cliente bigint,
FOREIGN KEY (id_cliente) REFERENCES cliente(id_cliente));

INSERT INTO PEDIDO (data_pedido, hora_pedido, data_entrega, id_cliente) VALUES ('2021-10-20', '20:00', '2021-10-28',1);
INSERT INTO PEDIDO (data_pedido, hora_pedido, data_entrega, id_cliente) VALUES ('2021-09-21', '21:00', '2021-09-26',2);
INSERT INTO PEDIDO (data_pedido, hora_pedido, data_entrega, id_cliente) VALUES ('2021-10-19', '14:00', '2021-10-21',1);
INSERT INTO PEDIDO (data_pedido, hora_pedido, data_entrega, id_cliente) VALUES ('2021-10-16', '12:00', '2021-10-18',2);
```

Exercício One to Many / Many To One

```java
@Entity
public class Pedido {
   @Id

@GeneratedValue(strategy=Generation
Type.IDENTITY)
   @Column(name="id_pedido")
   private Long id;

   @Column(name="data_pedido")
   private LocalDate dataPedido;

   @Column(name="hora_pedido")
   private LocalTime horaPedido;

   @Column(name="data_entrega")
   private LocalDate dataEntrega;




   public Long getId() {
       return id;
   }
```

Vamos inserir a classe **Pedido**

Fazemos o mapeamento em que temos vários pedidos de um determinado cliente.

@JoinColumn – Estamos dizendo qual é a coluna da tabela que é a chave estrangeira.

Inserir **Getter** e **Setter**

Equals e HashCode para o atributo **id**

Exercício One to Many / Many To One

Vamos inserir a interface **PedidoRepository**

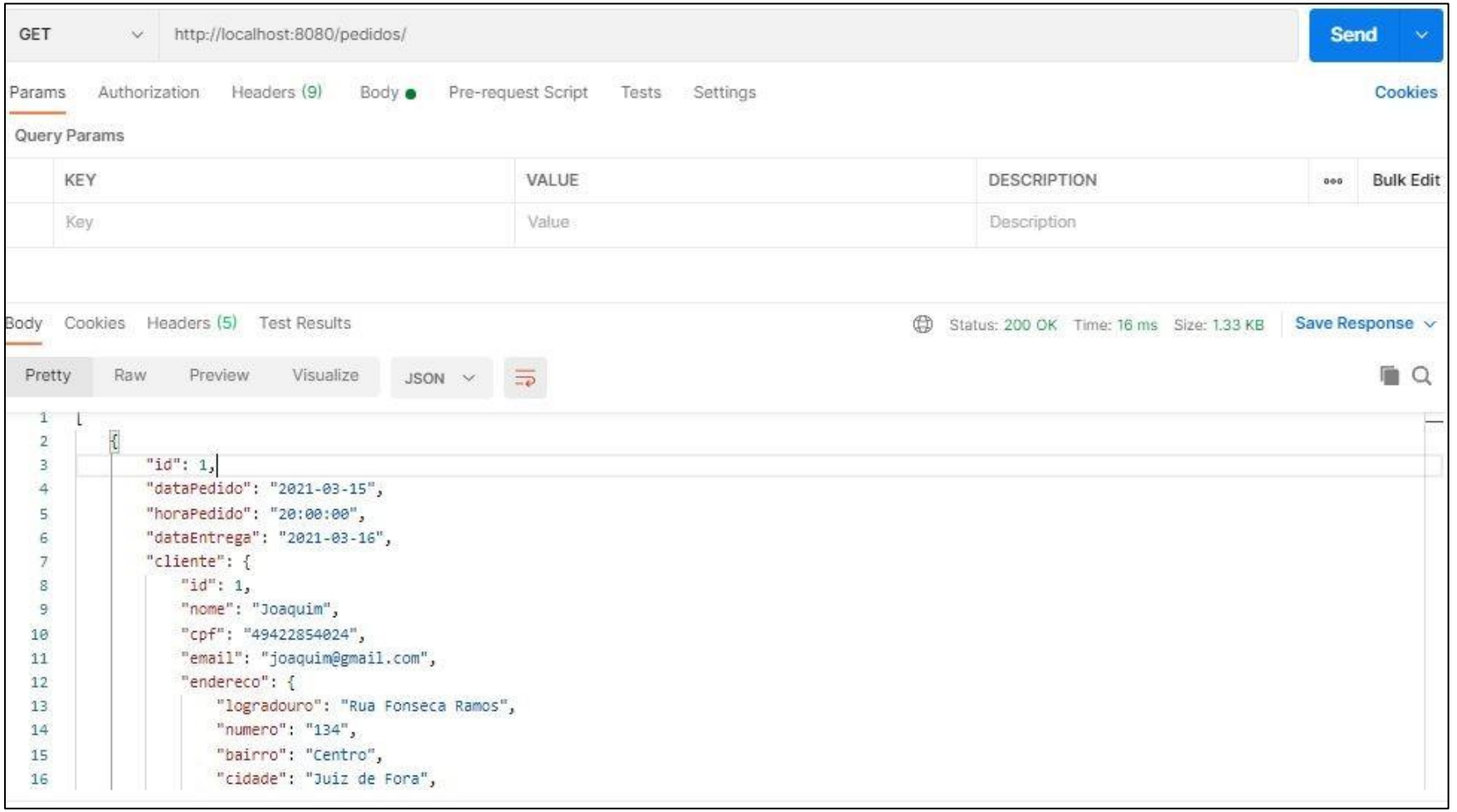
```
@Repository
public interface PedidoRepository extends
     JpaRepository<Pedido,Long>   {

}
```

Vamos inserir a classe **PedidoController** para listar todos pedidos e por id

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {
   @Autowired
   private PedidoRepository pedidoRepository;
   @GetMapping
   public List<Pedido> listar(){
       return pedidoRepository.findAll();
   }
   @GetMapping("/{id}")
   public ResponseEntity<Pedido> buscar(@PathVariable Long id){
       Optional<Pedido> pedido = pedidoRepository.findById(id);
       if (pedido.isPresent()) {
           return ResponseEntity.ok(pedido.get());
       }
       return ResponseEntity.notFound().build();
   }
}
```

Exercício One to Many / Many To One

Vamos testar no Postman



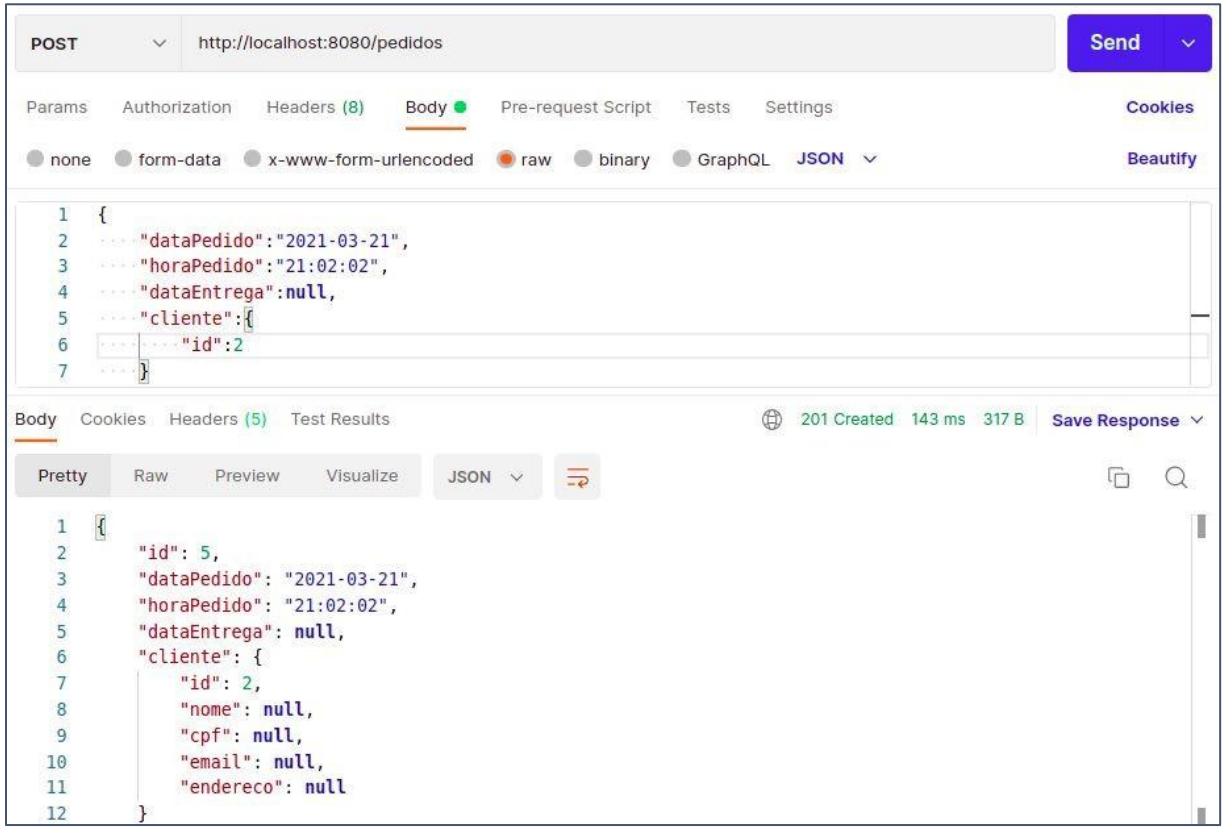
serratec

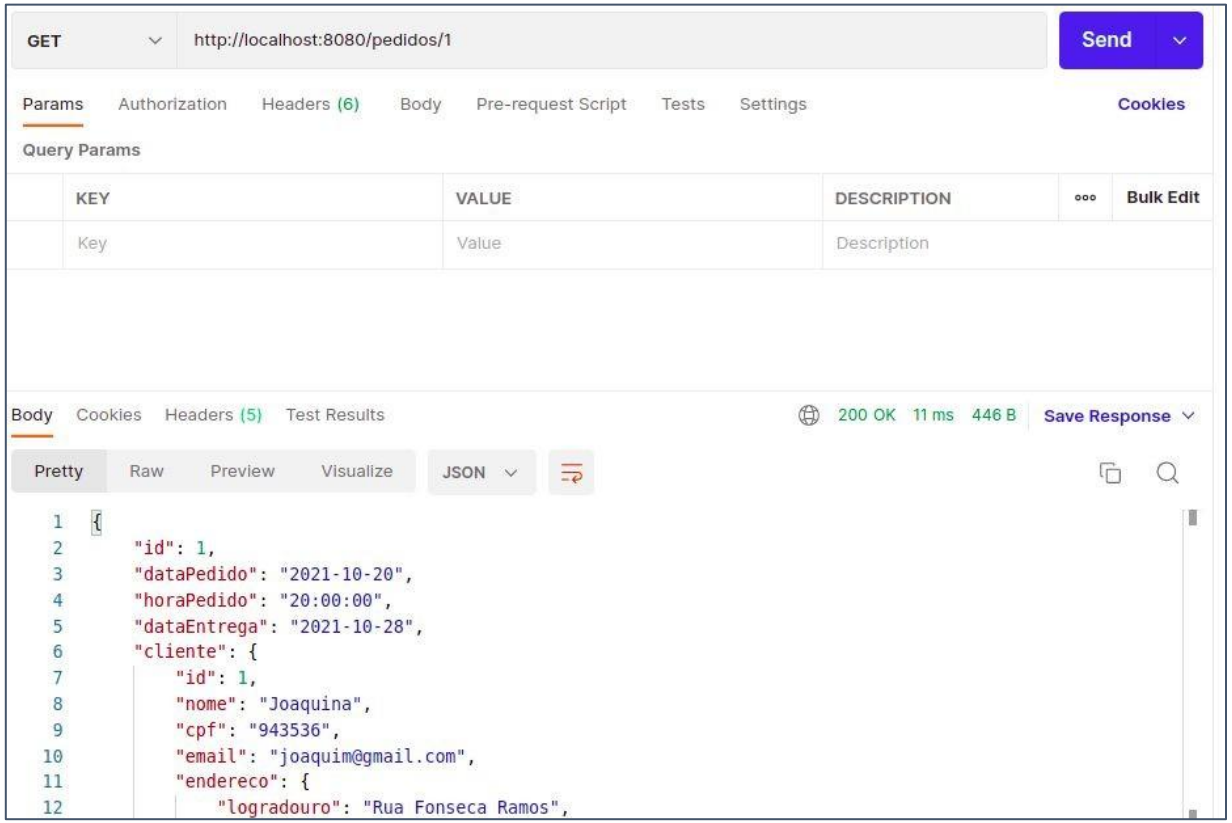
Exercício One to Many / Many To One

Adicionar o método para inserir o pedido

Testar no Postman

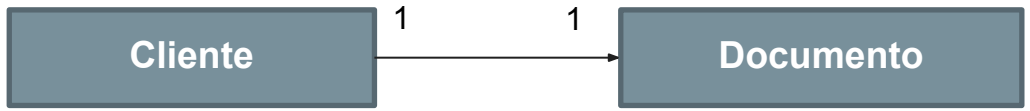
```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Pedido inserir(@Valid @RequestBody Pedido pedido) {
    return pedidoRepository.save(pedido);
}
```





Exercício One to One

O relacionamento um para um, também conhecido como one-to-one, é utilizado para dividir uma tabela em duas, para deixar as informações de forma mais organizada.



Um cliente possui somente um **documento**.

Um documento pertence a um determinado cliente.

Adicionar o scprit abaixo no banco de dados do **Postgres**

```
CREATE SEQUENCE documento_id_seq;
CREATE TABLE documento (
  id_documento bigint PRIMARY KEY DEFAULT nextval('documento_id_seq'),
  identidade varchar(15),
  orgao_expeditor varchar(20),
  cnh varchar(20),
  titulo_eleitor varchar(15),
  id_cliente bigint unique,
  FOREIGN KEY (id_cliente) REFERENCES cliente(id_cliente)
);

INSERT INTO documento (identidade, orgao_expeditor, cnh, titulo_eleitor, id_cliente)
VALUES ('098442','IFP','234343242','234243',1);
```

Exercício One to One

Criar a classe **Documento**

```java
@Entity
public class Documento {
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   @Column(name="id_documento")
   private Long id;

   @Column
   private String identidade;

   @Column(name="orgao_expeditor")
   private String orgaoExpeditor;

   @Column
   private String cnh;

   @Column(name="titulo_eleitor")
   private String tituloEleitor;

   @OneToOne
   @JoinColumn(name="id_cliente")
   private Cliente cliente;
```

Podemos mapear nos dois lados do relacionamento mas geralmente mapeamentos no lado que contém a chave estrangeira do relacionamento.

A propriedade **JoinColumn** é onde definimos qual será a chave estrangeira na tabela

Inserir **Getter** e **Setter**

Equals e HashCode para o atributo **id**