UPLOADS DE CONHECIMENTO

matera

BOOT CAMP

#let's create the future

Sumário

Informações IMPORTANTES

✔ Convidamos a todos a participar da nossa comunidade do Matera BootCamp. É o ambiente perfeito para você participar e garantir que receberá todos os comunicados.

Participe agora:

✔ O certificado de participação será enviado aos alunos que tiverem 55% ou mais de presença, na semana dos dias **02/10 à 06/10/23** via email.
✔ Faremos um sorteio de uma caneca exclusiva para os participantes que completarem no mínimo 80% de presença. Fique de olho em seu email, o resultado sai no dia **06/10/23!**

✔ Nos conte sua experiência durante o curso, e contribua dando sua opinião e sugestões de melhoria para o projeto Matera BootCamp.
Garantimos que sua resposta será anônima!
Acesse: https://forms.gle/gKw6wQn6jbhDKDWu6

Revisão Sobre Java - Olá, Mundo

Você se lembra da primeira vez que mandou uma mensagem para alguém que gostava muito? Ou talvez a primeira vez que falou em público? É um misto de nervosismo e excitação, não é? Em programação, temos algo semelhante que nos permite dizer "Oi" para esse novo universo, e chamamos de "Olá, Mundo!".

Então, por que "Olá, Mundo!"? Bem, é uma tradição! Pense nisso como um cumprimento universal entre programadores, quase como um aperto de mão secreto. Quando você vê isso, você sabe que alguém está dando seus primeiros passos em uma nova linguagem de programação.

Vamos ver como Java faz isso:

```
public class OlaMundo {
    public static void main(String[] args) {
        System.out.println("Olá, Mundo!");
    }
}
```

Saída:

```
Olá, Mundo!
```

Vamos decifrar isso um pouco:

public class OlaMundo: Estamos definindo uma classe chamada OlaMundo. Pense nisso como se fosse um capítulo em um livro onde queremos contar uma história específica.

public static void main(String[] args): Isso pode parecer um pouco intimidante no início, mas é uma parte essencial. É como a introdução desse capítulo. Em termos simples, é aqui que o Java começa a ler nosso código.

System.out.println("Olá, Mundo!");: Esta é a mensagem real. Estamos pedindo ao Java para imprimir (ou mostrar) a frase "Olá, Mundo!" na tela.

E assim, com apenas essas poucas linhas, você fez seu computador falar com você, dando-lhe uma calorosa saudação. É uma pequena, mas significativa conquista!

E então, com esse "Olá, Mundo!", você começa uma empolgante jornada de descoberta no mundo da programação com Java. Lembre-se de que cada linha que você escreve é um passo na construção de algo incrível. E este é apenas o começo!

Variáveis e seus tipos

Então, você já foi a uma festa de aniversário e viu aquelas etiquetas de nomes que as pessoas colam em suas camisetas? Elas ajudam a identificar quem é quem. As variáveis são como essas etiquetas, mas para os dados. Uma variável dá um "nome" a um pedaço de informação, seja um número, texto ou até mesmo se algo é verdadeiro ou falso.

Vamos dar uma olhada em alguns "tipos" dessas etiquetas:

```java
public class VariaveisSeusTipos {
    public static void main(String[] args) {
        // Um número inteiro, como "Quantos anos você tem?"
        int idade = 25;

        // Um número com casas decimais, como "Qual é a sua altura?"
        double altura = 1.78;

        // Texto ou uma sequência de caracteres, como "Qual é o seu nome?"
        String nome = "Carlos";

        // Verdadeiro ou falso, como "Você gosta de chocolate?"
        boolean gostaDeChocolate = true;

        System.out.println("idade: " + idade);
        System.out.println("altura: " + altura);
        System.out.println("nome: " + nome);
        System.out.println("gostaDeChocolate: " + gostaDeChocolate);
    }
}
```

Saída:

```
idade: 25
altura: 1.78
nome: Carlos
gostaDeChocolate: true
```

Explicando um pouco mais:

int: É um tipo que armazena números inteiros. Como a idade de alguém. Ninguém tem "25,5 anos", certo? São sempre números completos.

double: Já este é para quando precisamos de detalhes. Como a nossa altura. Afinal, você pode muito bem ter 1,78m e não apenas 1 ou 2 metros de altura.

String: Pense nisso como uma corda com contas. Cada conta é uma letra ou caractere, e juntas elas formam palavras ou frases.

boolean: É como um interruptor de luz. Só tem duas posições: ligado (true) ou desligado (false).

Então, por que precisamos de diferentes "tipos" de variáveis? Bem, imagine que você está organizando sua mochila. Você tem um compartimento para o laptop, um bolso menor para o celular e talvez um espaço para garrafas de água. Cada coisa tem seu lugar certo. Da mesma forma, no mundo Java, cada tipo de informação tem seu "lugar" ou tipo de variável adequado.

Variáveis são incrivelmente úteis. Elas são como pequenos baús onde guardamos informações para usar mais tarde. E à medida que você avança no Java, verá que esses "baús" estão em todo lugar, ajudando a tornar seu código inteligente e flexível!

Estrutura de controle e repetições

Você já ficou em frente ao seu guarda-roupa de manhã, tentando decidir o que vestir? "Se estiver frio, vou usar um suéter. Se estiver quente, talvez uma camiseta." No mundo da programação, tomamos decisões semelhantes o tempo todo usando estruturas condicionais.

1. Tomando Decisões com if e else

```
public class TomandoDecisoes {

    public static void main(String[] args) {
        int temperatura = 15;

        if (temperatura < 20) {
            System.out.println("Melhor vestir um suéter!");
        } else {
            System.out.println("Uma camiseta está ótima!");
        }
    }
}
```

Saída:

```
Melhor vestir um suéter!
```

Pense no if como um amigo questionador. Ele sempre pergunta: "Isso é verdade?" No nosso exemplo, ele pergunta: "A temperatura é menor que 20 graus?". Se a resposta for "sim", ele segue o conselho dentro do bloco if. Se não, ele verifica se há um bloco else e segue aquele conselho.

2. Repetindo Ações com Loops

Agora, imagine ter uma playlist de músicas e querer ouvir sua canção favorita três vezes seguidas. Em vez de pressionar o botão de reprodução manualmente a cada vez, você pode configurar para repetir! Em Java, usamos loops para "repetir" ações.

Usando o for:

```
public class UsandoFor {
   public static void main(String[] args) {
       for (int i = 1; i <= 3; i++) {
           System.out.println("Tocando minha canção favorita pela " + i +
                              "ª vez.");
       }
   }
}
```

Saída:

```
Tocando minha canção favorita pela 1ª vez.
Tocando minha canção favorita pela 2ª vez.
Tocando minha canção favorita pela 3ª vez.
```

Aqui, o loop for está nos dizendo: "Ei, vamos tocar essa música 3 vezes!". A variável i é como um contador, começando em 1 e indo até 3, incrementando de um em um.

Outro amigo legal é o while, que repete uma ação enquanto algo for verdadeiro:

```
public class UsandoWhile {
   public static void main(String[] args) {
       int bateriaDoCelular = 10;

       while (bateriaDoCelular > 0) {
           System.out.println("Celular ainda está funcionando... Bateria
                              está com " + bateriaDoCelular + "%.");
           bateriaDoCelular -= 2; // Diminui a bateria em 2% a cada ciclo.
       }
   }
}
```

Saída:

```
Celular ainda está funcionando... Bateria está com 10%.
Celular ainda está funcionando... Bateria está com 8%.
Celular ainda está funcionando... Bateria está com 6%.
Celular ainda está funcionando... Bateria está com 4%.
Celular ainda está funcionando... Bateria está com 2%.
```

Aqui, o loop while nos diz: "Enquanto ainda tivermos bateria no celular, continuamos usando!" E a cada ciclo, a bateria diminui um pouco até acabar.

Decisões e repetições são uma parte essencial da programação. Eles dão ao nosso código a capacidade de pensar, reagir e, claro, ser preguiçoso de vez em quando! Afinal, por que fazer algo manualmente várias vezes quando você pode pedir ao Java para fazer isso por você?

Classes e Objetos

Imagine que você queira construir uma casa. Antes de iniciar a construção, você precisaria de um projeto, certo? Esse projeto determinaria como a casa seria, onde as janelas e portas estariam, a disposição dos quartos e muito mais. No mundo Java, uma Classe é como esse projeto.

1. Classes:

A classe é um esboço ou modelo que descreve o que um objeto será, mas ela não é o objeto em si. Por exemplo, se você estivesse modelando uma classe Carro em Java, poderia parecer com isso:

```
public class Carro {
    // Atributos (também chamados de campos ou variáveis de instância)
    String marca;
    String modelo;
    int ano;
    double velocidade;

    // Métodos
    void acelerar() {
        velocidade += 10;
        System.out.println("Vrummm! Velocidade agora é: " + velocidade + " km/h");
    }

    void frear() {
        velocidade -= 10;
        System.out.println("Reduzindo... Velocidade agora é: " + velocidade + "
                           km/h");
    }
}
```

Entendendo o acima:

- Temos uma classe chamada Carro.
- Essa classe tem atributos, que são como as características do carro: sua marca, modelo, ano e velocidade atual.
- Ela também tem métodos, que são as ações que o carro pode realizar, como acelerar e frear.

2. Objetos:

Um objeto é uma instância da classe. Usando nosso exemplo anterior, a classe Carro é apenas um projeto. Mas e se quisermos ter um carro real? É aí que criamos um objeto!

```
public class TesteCarro {
    public static void main(String[] args) {
        Carro meuCarro = new Carro();
        meuCarro.marca = "Toyota";
        meuCarro.modelo = "Corolla";
        meuCarro.ano = 2022;
        meuCarro.velocidade = 0;

        System.out.println(meuCarro.marca);
        System.out.println(meuCarro.modelo);
        System.out.println(meuCarro.ano);
        System.out.println(meuCarro.velocidade);
        System.out.println("------------------------");
        meuCarro.acelerar();
        meuCarro.frear();
    }
}
```

Saída:

```
Toyota
Corolla
2022
0.0
------------------------
Vrummm! Velocidade agora é: 10.0 km/h
Reduzindo... Velocidade agora é: 0.0 km/h
```

Agora, meuCarro é um objeto da classe Carro. É como se tivéssemos construído uma casa real com base no projeto! E podemos interagir com meuCarro chamando os métodos acelerar e frear.

Em suma, classes são como receitas ou plantas baixas, enquanto objetos são as refeições reais ou as casas construídas a partir desses planos. Uma vez que você entende esse conceito, você realmente começa a entrar na magia da programação orientada a objetos!

Herança

Imagina que você é um designer de carros. Depois de muito trabalho, você finalmente desenhou um modelo básico de carro chamado "CarroBase". Agora, em vez de começar do zero para criar variações desse carro, como um carro esportivo ou um SUV, você simplesmente pega o design do "CarroBase" e faz as modificações necessárias. No mundo da programação Java, este conceito é chamado de Herança.

A herança nos permite criar uma nova classe com base em uma classe já existente. A classe existente é chamada de classe pai (ou superclasse) e a nova classe é a classe filha (ou subclasse).

Exemplificando com Código:

Considerando que temos a classe básica "Carro" do exemplo anterior, digamos que você queira criar um "CarroEsportivo". Em vez de reescrever todo o código do zero, você simplesmente "herda" da classe "Carro":

```
public class CarroEsportivo extends Carro {
    int turbo;

    void ativarTurbo() {
        turbo = 1;
        velocidade += 30;
         System.out.println("Turbo ativado! Velocidade agora é: " + velocidade
+
                            " km/h");
    }
}
```

Veja a palavra-chave extends? Isso indica que CarroEsportivo é uma subclasse de Carro. Além dos métodos básicos de aceleração e frenagem, o CarroEsportivo tem a capacidade adicional de ativar o turbo!

Por que a Herança é Útil?

1. Reutilização de Código: Como vimos, não precisamos reescrever todo o código. Podemos usar o que já temos e simplesmente adicionar ou modificar o que é necessário na subclasse.
2. Estrutura Organizada: Herança nos fornece uma hierarquia clara de classes, facilitando o entendimento das relações entre diferentes classes.
3. Facilidade de Manutenção: Se, no futuro, decidirmos mudar algo na classe básica "Carro", essas mudanças serão automaticamente refletidas em todas as subclasses, como "CarroEsportivo".

Portanto, da próxima vez que você ver diferentes modelos de carros na rua – seja um compacto, um SUV ou um esportivo – pense em como eles compartilham muitos recursos em comum, mas também têm suas características únicas. É assim que a herança funciona no mundo da programação.

Métodos

Imagine que você tem um carro. No mundo real, um carro tem várias funcionalidades e operações que ele pode fazer: pode acelerar, frear, ligar os faróis, ligar o rádio, entre outras coisas.

Agora, pense que cada uma dessas ações é como um "método" no mundo da programação. Quando você gira a chave, está chamando o método "ligar". Quando pisa no acelerador, está chamando o método "acelerar". E assim por diante.

```
public class Carro {

    // ... atributos como velocidade, marca, modelo, etc ...

    // Este é o método para ligar o carro
    void ligar() {
        System.out.println("Carro ligado!");
    }
    // Este é o método para acelerar o carro
    void acelerar() {
        System.out.println("Acelerando...");
    }
    // E aqui temos o método para frear
    void frear() {
        System.out.println("Freando...");
    }
}
```

Cada um desses métodos representa uma ação que o carro pode realizar. E quando você, como programador, quer que o carro faça algo, você "chama" o método correspondente.

Mas por que usar métodos?

Reutilização: Se toda vez que quiséssemos acelerar o carro tivéssemos que explicar todo o processo de injeção de combustível, ignição, etc., seria exaustivo! Com um método, definimos essa ação uma vez e depois a "chamamos" sempre que necessário.

Manutenção: Imagine que descobrimos uma maneira mais eficiente de acelerar. Se temos um método acelerar(), apenas modificamos esse método. Não precisamos procurar em todo o nosso código onde aceleramos o carro.

Legibilidade: Para alguém que está lendo seu código (ou para você mesmo, após algum tempo), é muito mais fácil entender carro.acelerar() do que uma série de comandos e códigos que, juntos, fazem o carro acelerar. Assim, fica claro o que o código está tentando fazer.

Pense nos métodos como os controles de um carro: cada botão, alavanca ou pedal invoca uma ação específica do veículo. Na programação, em vez de pressionar fisicamente um botão, você "chama" um método. E essa é a beleza de usar métodos: eles tornam a operação do nosso "carro de código" suave e eficiente!

Sobrecarga de Métodos (Overloading):
Imagine que você tem um carro que pode acelerar. Mas há diferentes maneiras de acelerar. Você pode querer apenas acelerar sem especificar quanto, ou talvez você queira acelerar a uma certa velocidade específica.

A sobrecarga permite que você tenha múltiplos métodos com o mesmo nome na mesma classe, mas com diferentes parâmetros.

```java
public class Carro {
    double velocidade;

    // Acelerar sem especificar a quantidade
    void acelerar() {
        velocidade += 5; // Acelera em 5km/h
    }

    // Acelerar especificando a quantidade
    void acelerar(double quantidade) {
        velocidade += quantidade;
    }
}
```

Neste exemplo, temos dois métodos acelerar(). Um que acelera o carro em 5km/h toda vez que é chamado e outro que leva um parâmetro e acelera o carro pela quantidade especificada.

Sobrescrita de Métodos (Overriding):
Agora, imagine que você tenha um carro, mas também tem uma categoria especial de carros, digamos, carros elétricos. Carros elétricos podem acelerar de uma maneira um pouco diferente dos carros normais devido ao motor elétrico.

A sobrescrita é um conceito de Programação Orientada a Objetos onde uma subclasse fornece uma implementação específica para um método que já é fornecido por sua superclasse (classe pai).

```java
public class Carro {
    double velocidade;

    void acelerar() {
        velocidade += 5; // Acelera em 5km/h
        System.out.println("Acelerando o carro normal em 5km/h");
    }
}
```

```java
public class CarroEletrico extends Carro {

    // Sobrescrevendo o método acelerar da classe Carro
    @Override
    void acelerar() {
        velocidade += 7; // Acelera em 7km/h devido ao motor elétrico
        System.out.println("Acelerando o carro elétrico em 7km/h");
    }
}
```

Aqui, a classe CarroEletrico é uma subclasse de Carro. Mas porque carros elétricos aceleram de maneira diferente, nós "sobrescrevemos" o método acelerar() na classe CarroEletrico. Quando você chama o método acelerar() em um objeto da classe CarroEletrico, é a versão sobrescrita do método que é executada, não a versão da classe pai Carro.

Em resumo:

- **Sobrecarga** é ter o mesmo método várias vezes na mesma classe, mas com diferentes parâmetros.
- **Sobrescrita** é quando uma subclasse fornece sua própria implementação para um método que já está definido em sua classe pai.

Interfaces

Uma interface é um contrato em Java. Ela não contém a implementação dos métodos que declara, apenas suas assinaturas. Quando uma classe implementa uma interface, ela está essencialmente assinando um contrato que afirma: "Vou fornecer implementações para todos os métodos que essa interface declara".

Isso permite que os objetos sejam tratados de acordo com o comportamento que prometem, em vez de sua estrutura interna ou hierarquia de classes. No mundo dos carros, podemos ter interfaces diferentes que definem diferentes comportamentos.

Por que usar interfaces?

Polimorfismo: Como já discutido, as interfaces permitem que diferentes classes sejam tratadas de maneira uniforme. Um carro elétrico e um carro a gasolina são ambos "reabastecíveis", mesmo que o método para fazê-lo seja diferente.

Separação de responsabilidades: A interface fornece uma maneira de separar o que queremos que um objeto faça (interface) de como ele faz (implementação).

Flexibilidade: Classes diferentes podem implementar a mesma interface de diferentes maneiras. Isso oferece grande flexibilidade no design de software.

Vamos considerar a capacidade de um carro de reabastecer e acelerar:

```
public interface Reabastecivel {
    void reabastecer();
}

public interface Aceleravel {
    void acelerar();
}
```

Agora temos duas interfaces, e qualquer carro que queremos que seja capaz de acelerar ou reabastecer deve implementar essas interfaces.

```
public class CarroGasolina implements Reabastecivel, Aceleravel {
    @Override
    public void reabastecer() {
        System.out.println("Reabastecendo com gasolina no posto.");
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando o carro a gasolina.");
    }
}
```

```
public class CarroEletrico implements Reabastecivel, Aceleravel {
    @Override
    public void reabastecer() {
                System.out.println("Carregando a bateria na estação de
carregamento.");
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando o carro elétrico silenciosamente.");
    }
}
```

A beleza disso é que agora podemos ter um código que trabalhe com qualquer objeto Reabastecivel ou Aceleravel:

```
public static void fazerCarroReabastecer(Reabastecivel carro){
      carro.reabastecer();
}

public static void fazerCarroAcelerar(Aceleravel carro){
      carro.acelerar();
}
```

Graças às interfaces, não precisamos nos preocupar com o tipo específico de carro. Só precisamos saber que ele implementa a interface correta. É como se estivéssemos dizendo: "Não me importo com o tipo exato do seu carro, contanto que você possa acelerar ou reabastecer".

Em resumo, interfaces em Java (e em muitas outras linguagens de programação orientadas a objetos) oferecem uma maneira de definir comportamentos sem especificar como esses comportamentos são implementados. Isso nos dá uma tremenda flexibilidade e é uma ferramenta poderosa para criar sistemas modulares e extensíveis.

Encapsulamento

No mundo da programação orientada a objetos, o encapsulamento é como se você tivesse um carro, mas certos componentes, como o motor, a transmissão ou o sistema elétrico, estivessem protegidos por uma capa ou caixa. Você sabe que esses componentes estão lá e eles fazem o carro funcionar, mas você não interage diretamente com eles toda vez que dirige. Em vez disso, você usa interfaces mais simples, como o volante, os pedais e os botões.

Em Java, o encapsulamento nos permite proteger os "componentes internos" de nossas classes (os atributos) e expor apenas o que é necessário através de métodos (as interfaces mais simples).

Por que o encapsulamento é útil?

- **Controle:** Você pode controlar como os dados internos da sua classe são acessados ou modificados.
- **Flexibilidade:** Você pode alterar a implementação interna sem afetar as classes que usam a sua classe.
- **Proteção:** Você pode evitar que dados sejam alterados de maneiras inesperadas ou indesejadas.

Como aplicar o Encapsulamento?

Usando o nosso exemplo da classe Carro:

```
public class Carro {
    // Atributos privados - a "essência interna" do carro
    private String marca;
    private String modelo;
    private int ano;
    private double velocidade;

     // Métodos públicos - as "interfaces" que nos permitem interagir com o
carro
    public void acelerar() {
        // Lógica para acelerar
    }

    public void frear() {
        // Lógica para frear
    }
```

```
  // Métodos de acesso (Getters e Setters) - nos permitem acessar/alterar os atributos privados
    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }
    // E assim por diante para os outros atributos...
}
```

Neste exemplo:

Atributos são privados: Isso significa que outras classes não podem acessá-los diretamente. É como manter o motor e o sistema elétrico do carro protegidos.

Métodos públicos para ação (como acelerar() e frear()): Esses são os controles que qualquer um pode usar. São as interfaces seguras para interagir com o carro.

Métodos de acesso (Getters e Setters): São como pequenas janelas ou portas que permitem ver ou alterar o estado do carro de uma maneira controlada. Se você quiser saber a marca do carro, pode usar getMarca(), e se quiser mudar a marca, pode usar setMarca(String marca).

Modificadores de Acesso:

Os modificadores de acesso determinam a visibilidade de um método (ou atributo) para outras classes:

- **public:** O método pode ser acessado por qualquer outra classe, independentemente do pacote.
- **protected:** O método pode ser acessado pelas subclasses e por outras classes no mesmo pacote.
- **private:** O método só pode ser acessado dentro da mesma classe. Ele fica oculto para todas as outras classes, incluindo subclasses.
- **(sem modificador):** Também chamado de "default" ou "package-private". O método pode ser acessado apenas por classes no mesmo pacote.

Então, o encapsulamento é uma forma de esconder a complexidade e proteger os detalhes internos de uma classe, expondo apenas o necessário. É um princípio fundamental da programação orientada a objetos e nos ajuda a construir sistemas robustos e flexíveis. Pense nisso como os designs dos carros modernos - muitos detalhes técnicos estão escondidos sob o capô, mas o painel e os controles são intuitivos e fáceis de usar!

Polimorfismo

Polimorfismo é a capacidade de um objeto ser tratado como uma instância de sua própria classe ou de uma de suas classes base. Em termos simples, isso significa que diferentes classes podem ser tratadas como se fossem da mesma classe através da herança.

Pense na ação de dirigir um carro. Seja um carro normal, um carro elétrico ou um carro autônomo, todos têm a ação de dirigir. Mas a maneira como cada um dirige pode ser diferente.

```
public class Carro {
    void dirigir() {
        System.out.println("Dirigindo o carro normalmente.");
    }
}

public class CarroEletrico extends Carro {
    @Override
    void dirigir() {
        System.out.println("Dirigindo o carro elétrico. Aproveitando a eficiência!");
    }
}

public class CarroAutonomo extends Carro {
    @Override
    void dirigir() {
        System.out.println("O carro autônomo está dirigindo sozinho!");
    }
}
```

Agora, graças ao polimorfismo, podemos fazer o seguinte:

```
Carro meuCarro = new CarroEletrico();
meuCarro.dirigir();
// Saída: "Dirigindo o carro elétrico. Aproveitando a eficiência!"
```

Mesmo que meuCarro seja uma referência de Carro, ele está realmente apontando para um CarroEletrico, e o método dirigir() do CarroEletrico é chamado.

Padrão Strategy

O padrão Strategy permite que você mude o comportamento de um objeto em tempo de execução. Ele é usado quando temos várias formas (estratégias) de fazer algo e queremos alternar entre elas dinamicamente.

No nosso contexto de carros, imagine que temos carros com diferentes modos de aceleração.

```
// Interface para a estratégia
interface ModoAcelerar {
    void acelerar();
}

// Estratégias concretas
class AcelerarNormal implements ModoAcelerar {
    @Override
    public void acelerar() {
        System.out.println("Aceleração normal.");
    }
}

class AcelerarEsportivo implements ModoAcelerar {
    @Override
    public void acelerar() {
        System.out.println("Aceleração esportiva! Vai mais rápido.");
    }
}
```

```
public class Carro {
    private ModoAcelerar modoAcelerar;

    public Carro(ModoAcelerar modoAcelerar) {
        this.modoAcelerar = modoAcelerar;
    }

    public void definirModoAcelerar(ModoAcelerar modoAcelerar) {
        this.modoAcelerar = modoAcelerar;
    }

    public void acelerar() {
        modoAcelerar.acelerar();
    }
}
```

Usando este padrão, você pode facilmente mudar o modo de aceleração:

```
public static void main(String[] args) {
    Carro meuCarro = new Carro(new AcelerarNormal());
    meuCarro.acelerar(); // Saída: "Aceleração normal."

    // Mudando o modo de aceleração em tempo de execução
    meuCarro.definirModoAcelerar(new AcelerarEsportivo());
    meuCarro.acelerar(); // Saída: "Aceleração esportiva! Vai mais rápido."
}
```

Deste modo, o polimorfismo (ao permitir que diferentes estratégias implementem a mesma interface) juntamente com o padrão Strategy, nos dá uma maneira flexível e dinâmica de alternar comportamentos.

Spring Framework

O Spring é um framework Java criado para simplificar o desenvolvimento de aplicações. Ele aborda muitas áreas do desenvolvimento Java, desde a construção de aplicações web até a integração de dados. Uma das principais características do Spring é a sua capacidade de gerenciar "beans" (componentes de software) em um contêiner, usando o que é chamado de "Inversão de Controle" (IoC) e injeção de dependências.

Para que serve?

- **Desenvolvimento mais rápido:** O Spring oferece diversos módulos prontos que agilizam a implementação de recursos comuns, como autenticação e autorização, acesso a bancos de dados, mensageria, entre outros.
- **Integração:** Spring se integra facilmente com outras tecnologias Java, como JPA, Hibernate, JMS, e outras.
- **Testabilidade:** O design do Spring é amigável para testes, o que facilita a criação de testes unitários e de integração.
- **Escalabilidade:** Aplicações construídas com Spring são naturalmente escaláveis devido ao seu design modular e baseado em beans.

Spring Boot

Spring Boot é um projeto do ecossistema Spring que simplifica a maneira de criar aplicações stand-alone com mínima configuração. Ele busca eliminar muita da configuração e do boilerplate associados ao uso do Spring, oferecendo uma maneira de criar aplicações "just run".

Benefícios do Spring Boot:

- **Autoconfiguração:** Spring Boot pode detectar bibliotecas no classpath e configurá-las automaticamente. Por exemplo, se o Spring Boot vê um H2 no classpath, ele assume que você está desenvolvendo uma aplicação com banco de dados em memória.
- **Produção pronta:** Spring Boot tem built-in características como monitoramento de saúde e métricas, que torna fácil monitorar e gerenciar aplicações em produção.
- **Opiniões pré-definidas:** Para ajudar os desenvolvedores a serem produtivos, Spring Boot tem uma abordagem opinativa sobre configuração. Mas se você não gosta das opiniões, pode facilmente mudá-las.
- **Empacotamento stand-alone:** As aplicações Spring Boot podem ser empacotadas como JARs e podem ser executadas como aplicações Java normais. Isto é ótimo para microserviços e para ambientes de nuvem.

Para que serve?
Em essência, Spring Boot serve para agilizar e simplificar o desenvolvimento de aplicações com o Spring. Ele automatiza muitas das tarefas tediosas e repetitivas, permitindo que os desenvolvedores se concentrem na lógica de negócios.

Pense no Spring como uma gigantesca caixa de ferramentas para desenvolvedores Java. Quer construir uma API web? Tem um módulo para isso. Precisa acessar um banco de dados? Ele te cobre. E o Spring Boot? Ele é como um assistente inteligente que pega essas ferramentas e as configura para você, para que você possa se concentrar em construir algo incrível sem se preocupar com as minúcias de "como conectar isso a isso". É uma combinação poderosa que muitos desenvolvedores usam e atualmente é o padrão de mercado quando se trata de desenvolvimento web com Java!

Primeiro projeto Spring Boot

Imagine o seguinte cenário: Você quer construir seu próprio carro, mas não deseja começar do zero, soldando o chassi ou montando o motor peça por peça. Ao invés disso, você quer um kit inicial. Isso é o que o Spring Initializr oferece para o desenvolvimento de aplicações. Um começo simplificado!

1. Visite a Oficina (Spring Initializr):

Primeiro, vá até a oficina online de montagem de carros (no nosso caso, o Spring Initializr).

2. Preenchendo a Documentação (Project Metadata):
Antes de começar a montagem, precisamos fornecer alguns detalhes para garantir que seu carro (projeto) seja único:

- **Group:** Imagine isso como a marca do seu carro. No mundo Java, isso seria a empresa ou indivíduo que está construindo a aplicação. Por exemplo, com.minhaempresa.
- **Artifact:** Aqui você dará um nome ao modelo do seu carro. Suponhamos que você chame seu projeto de gerenciadorDeCarros.
- **Packaging:** Escolha Jar. No contexto de carros, pense nisso como a "embalagem" ou "estrutura" do carro. Jar é compacto e eficiente.

3. Escolhendo as Peças (Dependencies):

Você vai querer adicionar funcionalidades ao seu carro. No Spring Initializr, isso se traduz em adicionar Dependencies (Dependências). Para começar, escolha Spring Web porque você quer que seu carro seja conectado e moderno (ou seja, deseja criar uma API REST).

4. Finalize a Configuração e Baixe o Kit:

Clique no botão "Generate" para finalizar sua seleção. Isso irá baixar um arquivo .zip, que é como receber o kit do carro.

5. Montando o Carro (Configurando o Projeto na IDE):

Extraia o arquivo .zip e abra o diretório resultante na sua IDE (IntelliJ, Eclipse, etc.). É como abrir a caixa do seu kit de carro e espalhar todas as peças pelo chão da garagem.

6. Entendendo o Layout do Carro (Estrutura do Projeto):

- **src/main/java:** Aqui é o coração do seu carro, o motor, onde ficará o código.
- **src/main/resources:** Pense nisso como o porta-luvas do carro. Lugar para arquivos auxiliares, como propriedades e configurações.
- **pom.xml:** Esse é o manual de instruções do carro. Ele contém todos os detalhes sobre as peças (dependências) que você escolheu e como elas devem ser montadas. Maven, nosso assistente de montagem, olhará para este arquivo para saber como construir, quais dependências baixar e executar seu projeto.

7. Colocando o Carro para Funcionar (Executando o Projeto):
Dentro do diretório src/main/java, você verá um arquivo chamado GerenciadorDeCarrosApplication.java. Esse é a chave de ignição do seu carro. Execute-o, e o Spring Boot inicializará sua aplicação. Você ouvirá o rugido do motor (verá logs na sua IDE) indicando que tudo está funcionando.

```
@SpringBootApplication
public class GerenciadorDeCarrosApplication {
    public static void main(String[] args) {
        SpringApplication.run(GerenciadorDeCarrosApplication.class, args);
    }
}
```

E pronto! Agora, você tem um projeto Spring Boot rodando. Pode não parecer muito no começo, mas é uma base sólida, e você está pronto para adicionar todos os recursos e funcionalidades que desejar a este "carro".

Inversão de Controle (IoC)

Você já montou um modelo de carro de brinquedo? Tradicionalmente, em programação, é como se você pegasse cada peça individualmente, encaixasse uma na outra, colocasse os adesivos e assim montasse todo o carro. Esse processo é cansativo e repetitivo. Imagine fazer isso para cada carro novo que você quiser montar!

A Inversão de Controle é como se, em vez de você montar cada peça do carro de brinquedo, você simplesmente descrevesse o carro dos seus sonhos e entregasse essa descrição para uma fábrica especial (no caso, o Spring). Essa fábrica então monta o carro pra você, exatamente como você descreveu.

Em termos técnicos, em vez de você criar os objetos manualmente usando o **new** e gerenciar seu ciclo de vida, você deixa o Spring cuidar disso. O Spring se encarrega de criar instâncias dos objetos, configurá-los e entregar pra você quando você precisar.

Delegando a responsabilidade para o Spring

Para que o Spring possa assumir a responsabilidade e criar e gerenciar os objetos, você precisa:

Definir Beans: Ou seja, Você precisa dizer ao Spring quais classes ele deve gerenciar. Você faz isso anotando essas classes com anotações específicas, como **@Component**, **@Service**, **@Repository**, e **@Controller** (ou **@RestController**).

Injeção de Dependências (IoC)

Imagine que seu carro precisa de um motor para funcionar. Em vez de você, pessoalmente, construir o motor e instalar no carro (isso daria um trabalhão, certo?), você simplesmente informa ao seu mecânico (o Spring, neste caso) que seu carro precisa de um motor. O mecânico, então, encontra o motor correto e instala para você. Isso é a injeção de dependências.

Em termos de código, em vez de fazer algo como Motor motor = new Motor(); dentro da classe Carro, você "informa" a classe Carro que ela precisa de um Motor, e deixa o Spring "injetar" essa dependência.

Como a Injeção de Dependências funciona?

O Spring mantém um "container" onde ele cria e gerencia todos os objetos (chamados de "beans"). Quando você diz ao Spring que uma determinada classe precisa de uma dependência, o Spring procura por essa dependência no seu container e a "injeta" na sua classe.

Criando pontos de injeção

Existem três principais maneiras de criar pontos de injeção no Spring:

Construtor: Esta é a maneira recomendada pela maioria dos desenvolvedores e pelo próprio Spring, pois garante que o objeto tenha suas dependências necessárias antes de ser utilizado.

```
@Service
public class CarroService {
    private final Motor motor;

    @Autowired
    public CarroService(Motor motor) {
        this.motor = motor;
    }
}
```

No exemplo acima, o Spring vê que CarroService precisa de um Motor e, então, procura um bean do tipo Motor no seu container e o injeta no CarroService através do construtor.

Setter: Uma alternativa ao construtor é o uso de setters. Essa abordagem é menos preferida porque permite que o objeto seja instanciado sem a dependência necessária, mas pode ser útil em certos cenários.

```
@Service
public class CarroService {
    private Motor motor;

    @Autowired
    public void setMotor(Motor motor) {
        this.motor = motor;
    }
}
```

Atributo: Diretamente no atributo da classe. É a maneira mais concisa, mas não é sempre a mais recomendada, porque torna mais difícil testar a classe isoladamente.

```
@Service
public class CarroService {

    @Autowired
    private Motor motor;
}
```

Com versões mais recentes do Spring, a anotação @Autowired é opcional se a classe tiver um único construtor. O Spring vai automaticamente assumir que você quer injetar as dependências através desse construtor.

Principais Anotações do Spring

Vamos conhecer agora de maneira resumida as principais anotações do Spring:

@Component

O que é?
É a anotação mais genérica do Spring para definir que uma classe é um componente gerenciado pelo Spring. Pense nisso como a etiqueta em um carro genérico no showroom.

No contexto da concessionária:
Imagine que existem carros padrão no showroom - não são especializados para venda ou serviços, apenas carros comuns. O @Component é essa marca padrão. Talvez um carro-conceito que esteja apenas em exibição, ou uma peça especial que a loja queira destacar.

@Service

O que é?
É uma especialização do @Component para indicar que a classe é destinada a regras de negócios e operações.

No contexto da concessionária:
Não é apenas sobre vender carros. A concessionária oferece manutenção, limpeza e outros serviços para os carros vendidos. Quando um cliente volta para um reparo, ele vai diretamente ao @Service, ou seja, aos mecânicos especializados da loja.

@Repository

O que é?
Indica que a classe é uma representação do acesso aos dados, tipicamente usado para interagir com bancos de dados.

No contexto da concessionária:
Todo carro vendido tem um registro - um histórico, detalhes do modelo, etc. O @Repository é como o back-office onde todos esses registros são mantidos. Se precisarmos verificar a história de um carro, iríamos a este depósito.

@Controller

O que é?
É uma indicação de que a classe é um controlador web e pode lidar com solicitações HTTP.

No contexto da concessionária:
A equipe de vendas é o primeiro ponto de contato para os clientes que entram na loja. Eles os ouvem, entendem suas necessidades e os direcionam ao carro certo. Da mesma forma, o @Controller ouve as solicitações dos usuários e as direciona para o serviço ou ação correta.

@RestController

O que é?
É uma combinação de @Controller e @ResponseBody que indica que a classe é um controlador web e que as respostas devem ser automaticamente convertidas para JSON ou XML.

No contexto da concessionária:
Além de uma loja física, a concessionária tem um website para exibir seus carros. O @RestController é como a versão online da equipe de vendas, apresentando carros em um formato digital para os visitantes online.

REST

REST (Representational State Transfer) é um estilo arquitetural proposto por Roy Fielding em sua tese de doutorado no ano 2000. Não é uma tecnologia, um framework ou um protocolo; é, em vez disso, um conjunto de princípios e restrições para criar serviços web escaláveis e fáceis de manter.

Princípios básicos do REST:

- **Statelessness:** Cada solicitação do cliente para o servidor deve conter todas as informações necessárias para compreender e processar a solicitação. Em outras palavras, o servidor não deve manter nenhum estado do cliente entre as solicitações.

- **Client-Server:** A arquitetura REST divide a interface do usuário dos componentes de armazenamento de dados, o que permite melhorar a escalabilidade e a portabilidade da interface do usuário em diferentes plataformas.

- **Cache:** As respostas dos serviços web devem ser explicitamente marcadas como cacheáveis ou não. Isso pode melhorar a performance, reduzindo a quantidade de interações desnecessárias entre cliente e servidor.

- **Interface Uniforme:** Para simplificar a interação entre os componentes, REST determina que os serviços web devem seguir uma interface padronizada e uniforme.

- **Sistema em Camadas:** Os componentes podem ser organizados em camadas hierárquicas, cada camada com uma responsabilidade específica.

- **Código sob Demanda:** O servidor pode, opcionalmente, enviar código executável para o cliente, estendendo assim sua funcionalidade. (Esta é a única restrição que é opcional no REST)

Relação com HTTP

O protocolo HTTP (Hypertext Transfer Protocol) é frequentemente usado para implementar a arquitetura REST, embora o REST possa, em teoria, ser implementado usando outros protocolos. A escolha do HTTP se deve à sua ubiquidade, simplicidade e eficiência.

Com o REST, os recursos (por exemplo, objetos de dados, como "usuários" ou "produtos") são identificados por URLs, como "http://meusite.com/usuarios". As operações CRUD (Criar, Ler, Atualizar e Deletar) são realizadas usando os métodos HTTP padrão:

- **GET:** Ler/receber dados.
- **POST:** Criar novos dados.
- **PUT ou PATCH:** Atualizar dados existentes.
- **DELETE:** Remover dados.

Além disso, o protocolo HTTP fornece uma série de códigos de status (como 200 para "OK", 404 para "Não Encontrado" ou 500 para "Erro Interno do Servidor") que podem ser usados para indicar o resultado da operação solicitada.

O REST, usando o protocolo HTTP, fornece uma maneira padronizada e intuitiva de projetar serviços web. A adesão aos princípios do REST garante que os serviços web sejam escaláveis, manuteníveis e compreensíveis. Como resultado, tornou-se a abordagem dominante para desenvolver serviços web na última década.

Modelo de Maturidade de Richardson

O Modelo de Maturidade de Richardson serve para nos ajudar a entender o quão "RESTful" uma API é. Talvez você esteja se perguntando: "O que é RESTful?". Bom, antes de nos aprofundarmos no modelo, é bom entender que RESTful é um conjunto de boas práticas e convenções para se criar APIs na web. Leonard Richardson foi quem delineou este modelo, apresentando-o como uma escada de quatro degraus (do Nível 0 ao Nível 3) que, gradativamente, nos mostra a evolução de uma API para se tornar verdadeiramente RESTful.

Vamos visualizar esse processo através de uma API fictícia de gerenciamento de carros:

Nível 0: Uso do HTTP como transporte

No Nível 0, a ideia é bem básica: o protocolo HTTP é usado só como um meio de transporte. Não importa qual ação você queira realizar (listar carros, adicionar um carro, remover um carro, etc.), você só terá um único ponto de entrada e usará, na maioria das vezes, somente o método POST do HTTP.

Exemplo:
Imagine que você queira obter detalhes de um carro:

URL de acesso: /api
Método: POST
Corpo da Solicitação: { "ação": "obterCarro", "idCarro": 123 }
Essencialmente, estamos dizendo "Oi, API, quero obter o carro com id 123", tudo isso através de uma única chamada POST. Não é muito intuitivo, né?

Nível 1: Recursos

As coisas começam a melhorar no Nível 1. Em vez de termos uma única entrada, começamos a dividir a funcionalidade da API em "recursos". Cada recurso é como um item ou entidade em nosso sistema.

Exemplo:
Se quiser obter informações de um carro, você poderia fazer:

URL: /carros/123
Método: POST
Agora, temos uma URL que aponta diretamente para o carro que queremos, o que é um avanço. Mas ainda estamos usando o POST para tudo.

Nível 2: Verbos HTTP

O Nível 2 começa a tirar proveito real do protocolo HTTP. Aqui, utilizamos os métodos (ou verbos) HTTP para representar a ação que desejamos realizar.

Exemplos:

Para obter um carro: GET /carros/123
Para adicionar um carro: POST /carros com o corpo da solicitação contendo os detalhes do carro.
Atualizar um carro: PUT /carros/123 com os novos detalhes.
Excluir um carro: DELETE /carros/123
Fica bem mais claro, certo? Cada verbo tem um propósito bem definido.

Nível 3: HATEOAS (Hypermedia as the Engine of Application State)

Aqui, entramos em um território mais avançado. Ao obter um recurso, a API também fornece links para ações relacionadas àquele recurso.

Exemplo:
Ao consultar um carro, além de receber detalhes sobre ele, você também recebe informações sobre o que mais pode fazer com esse carro:

```
{
   "id": 123,
   "marca": "Toyota",
   "modelo": "Corolla",
   "status": "Disponível",
   "_links": {
       "alugar": {
           "href": "/carros/123/alugar",
           "metodo": "POST"
       }
   }
}
```

Se o carro já estiver alugado, talvez o link "alugar" não apareça, mas um link para "devolver" apareça. A ideia aqui é que a própria resposta da API te guie sobre as possíveis ações, sem você precisar consultar uma documentação externa a todo momento.

Status HTTP

Os códigos de status HTTP são códigos de três dígitos que um servidor envia em resposta a uma solicitação feita por um cliente. Estes códigos informam ao cliente o resultado da sua solicitação, seja ela bem-sucedida, com erros ou necessitando de ação adicional.

Famílias de códigos de status
Os códigos de status HTTP são categorizados em cinco classes ou famílias, baseadas no primeiro dígito:

1xx: Informacional

Estes códigos indicam que a solicitação foi recebida e o processo está em continuação.

100 Continue: Basicamente diz ao cliente para continuar com a sua solicitação.

2xx: Sucesso

Indica que a solicitação foi bem-sucedida.

- **200 OK:** Esta é a resposta padrão para solicitações bem-sucedidas. Se você fizer uma solicitação GET para um recurso, e tudo correr bem, você receberá um 200 OK junto com o recurso solicitado.

- **201 Created:** Quando você cria um novo recurso (por exemplo, um novo carro em nossa API de carros), o servidor pode responder com este código, indicando que o recurso foi criado com sucesso.

- **204 No Content:** A solicitação foi bem-sucedida, mas não há conteúdo para enviar na resposta.

3xx: Redirecionamento

Estes códigos indicam que o cliente deve tomar medidas adicionais para completar a solicitação.

- **301 Moved Permanently:** O recurso solicitado foi movido permanentemente para uma nova URL.

- **304 Not Modified:** É uma resposta eficiente que diz ao cliente que o recurso não foi modificado desde a última solicitação e pode ser carregado a partir do cache.

4xx: Erros do Cliente

Estes códigos indicam que a solicitação contém erros ou não pode ser processada pelo servidor.

- **400 Bad Request:** A solicitação não foi compreendida ou foi malformada.

- **401 Unauthorized:** O cliente deve se autenticar para obter a resposta solicitada.

- **403 Forbidden:** O servidor entendeu a solicitação, mas se recusa a atendê-la.

- **404 Not Found:** O recurso solicitado não foi encontrado no servidor.

- **429 Too Many Requests:** O usuário enviou muitas solicitações em um determinado período de tempo.

5xx: Erros do Servidor

Esses códigos indicam que o servidor falhou em concluir uma solicitação aparentemente válida.

- **500 Internal Server Error:** Erro genérico quando ocorre uma condição inesperada e o servidor não pode especificar o problema.
- **502 Bad Gateway:** O servidor estava agindo como um gateway e recebeu uma resposta inválida.
- **503 Service Unavailable:** O servidor não está pronto para processar a solicitação.

Importância dos códigos de status

Os códigos de status são fundamentais na comunicação RESTful, pois fornecem uma forma padronizada de comunicar o resultado das operações. Isso permite que os clientes entendam exatamente o que aconteceu com sua solicitação e tomem medidas apropriadas.

Por exemplo, se você estiver programando um cliente e receber um 404 Not Found, saberá que o recurso que tentou acessar não existe e poderá informar isso ao usuário ou tentar outra ação.

Construindo Nossas Primeiras API's

Agora que já sabemos os conceitos fundamentais sobre Spring, API's e etc. chegou a hora de construirmos nossas primeiras APIs.

1. Configuração Inicial:

Certifique-se de que já tenha criado um projeto Spring Boot com a dependência spring-boot-starter-web utilizando o Spring Initializr. Esta dependência é fundamental para criar aplicações web e REST com Spring.

2. Modelo Carro

Antes de criar o controller, vamos criar uma classe modelo simples para o carro:

```
package com.matera.bootcamp.model;

import lombok.Data;

@Data
public class Carro {
   private Long id;
   private String marca;
   private String modelo;
   private int ano;
   private double velocidade;
}
```

3. Criando o Controller

No Spring, um controller é responsável por manipular as solicitações HTTP. Para criar um, siga estes passos:

1. Crie um pacote chamado controller para manter seus controllers organizados.
2. Dentro desse pacote, crie uma classe chamada CarroController.
3. Anote esta classe com @RestController, que indica que é um controller e que os dados retornados por seus métodos serão escritos diretamente no corpo da resposta, geralmente em formato JSON.
4. Agora, vamos adicionar alguns endpoints básicos:

```java
package com.matera.bootcamp.controller;

import com.matera.bootcamp.model.Carro;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

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

@RestController
@RequestMapping("/api/carros")
public class CarroController {

    // Simulando uma base de dados com uma lista
    private List<Carro> carros = new ArrayList<>();

    // Endpoint para listar carros
    @GetMapping
    public List<Carro> listarCarros() {
        return carros;
    }

    // Endpoint para adicionar um carro
    @PostMapping
    public ResponseEntity<Carro> adicionarCarro(@RequestBody Carro carro) {
        carros.add(carro);
        // Retorna o carro adicionado com status 200 OK
        return ResponseEntity.ok(carro);
    }
}
```

4. Explicação sobre o Controller

@RestController: indica que a classe é um controller e que seus métodos retornarão dados para serem escritos diretamente na resposta.

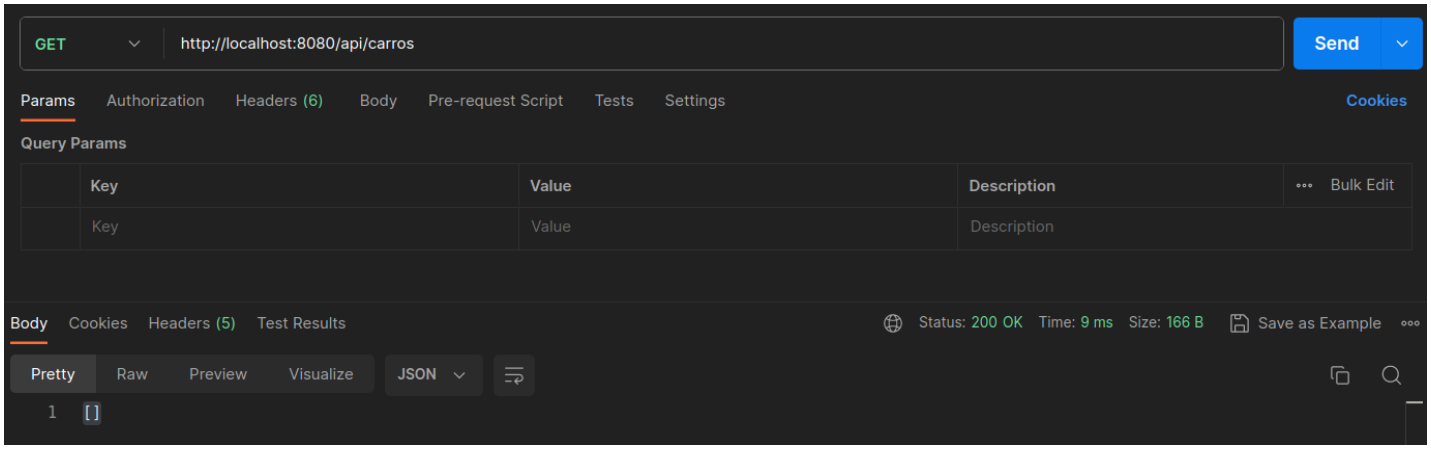
@RequestMapping("/api/carros"): especifica que este controller irá responder a solicitações que começam com /api/carros.

@GetMapping: este é um atalho para @RequestMapping(method = RequestMethod.GET). Ou seja, este método vai responder a solicitações HTTP GET para /api/carros.

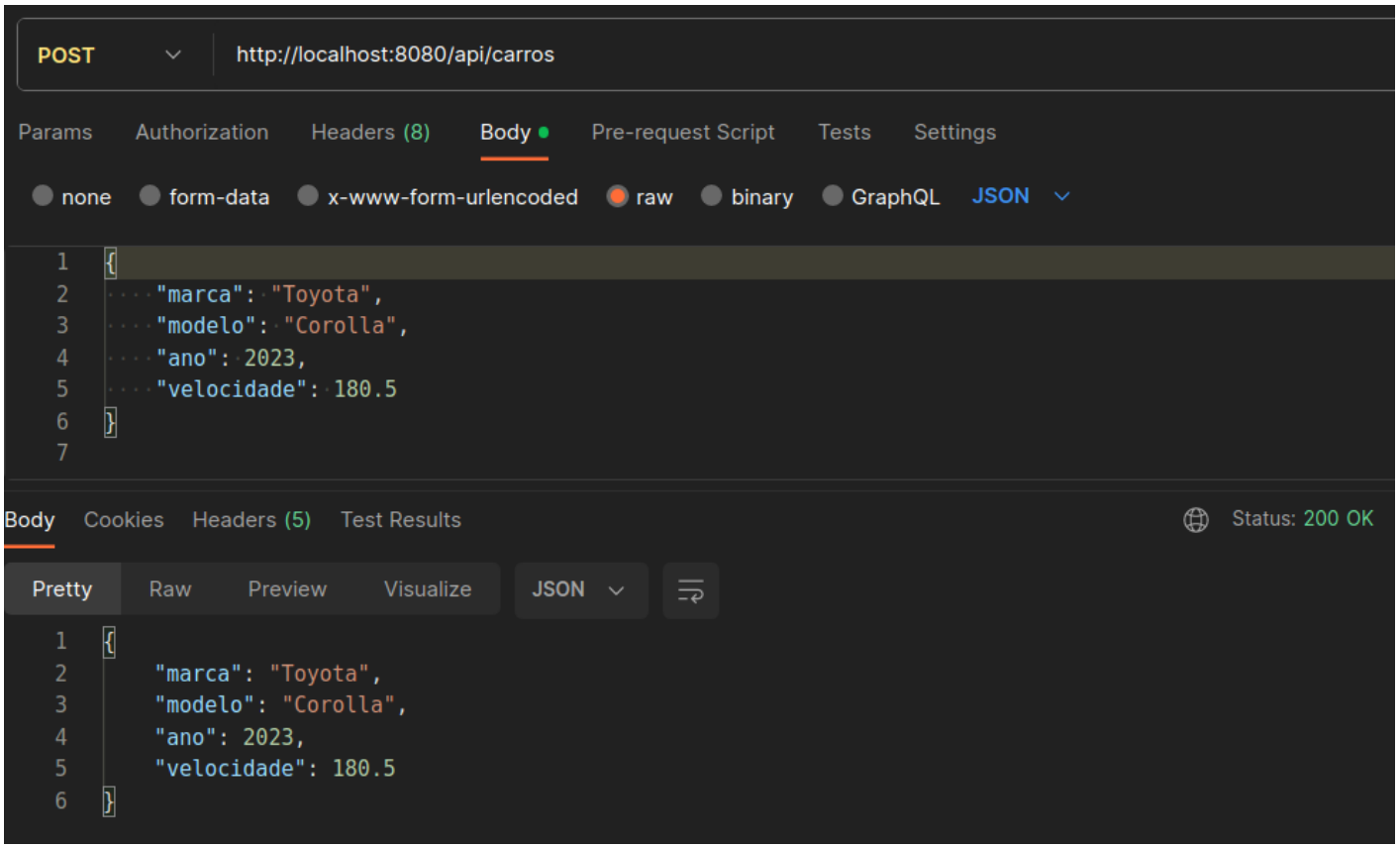
@PostMapping: semelhante ao @GetMapping, mas para solicitações POST.

@RequestBody: este é usado para deserializar o corpo da solicitação HTTP diretamente em um objeto da classe Carro.

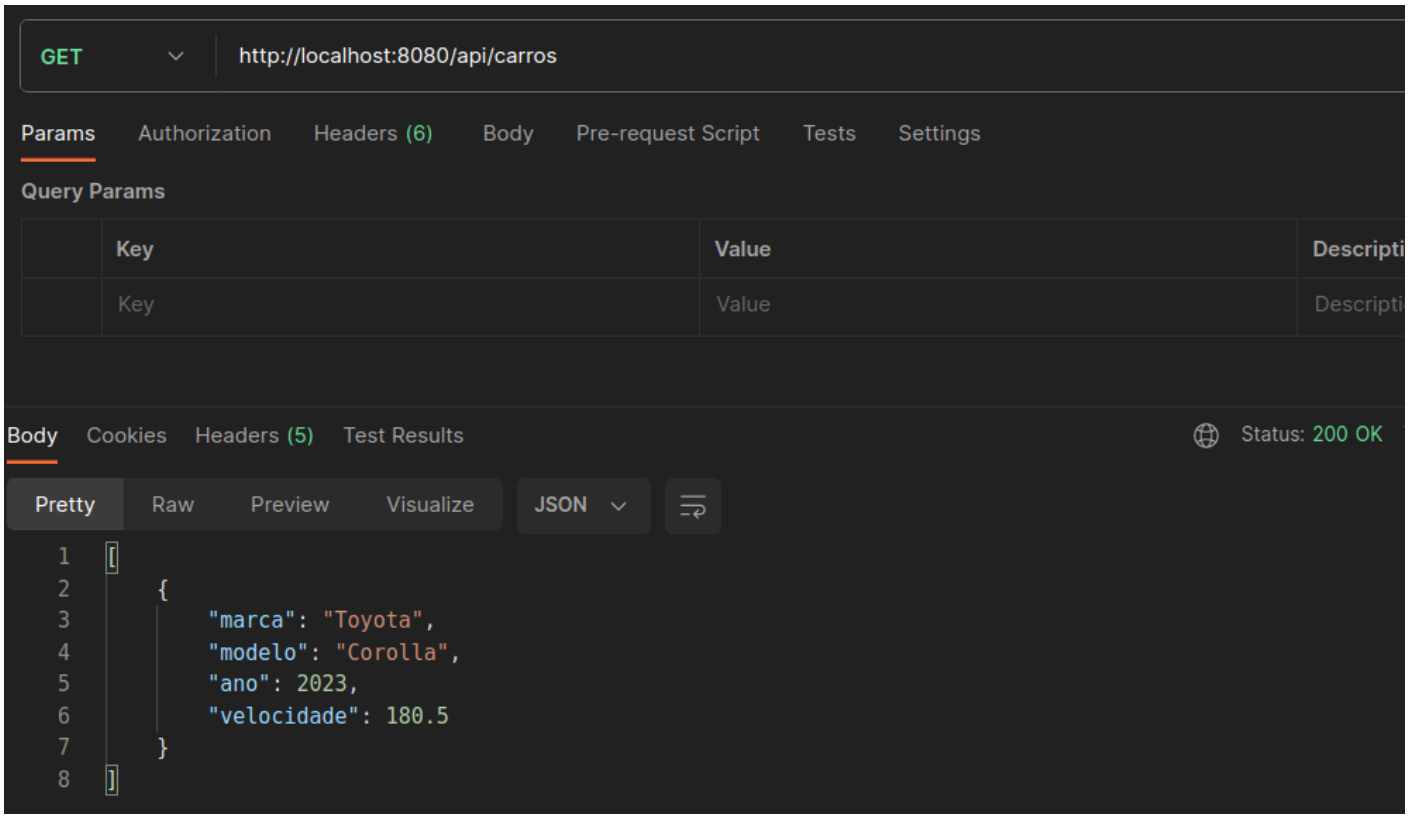
Agora, se você iniciar sua aplicação Spring Boot e fizer uma solicitação GET para http://localhost:8080/api/carros, obterá uma lista vazia.



Se você fizer uma solicitação POST para a mesma URL com um JSON representando um carro, esse carro será adicionado à lista.



Se fizermos uma nova requisição para GET api/carros veremos que o carro foi adicionado:



Agora que criamos o controller básico, vamos adicionar mais funcionalidades, como atualizar, buscar por ID e deletar carros.

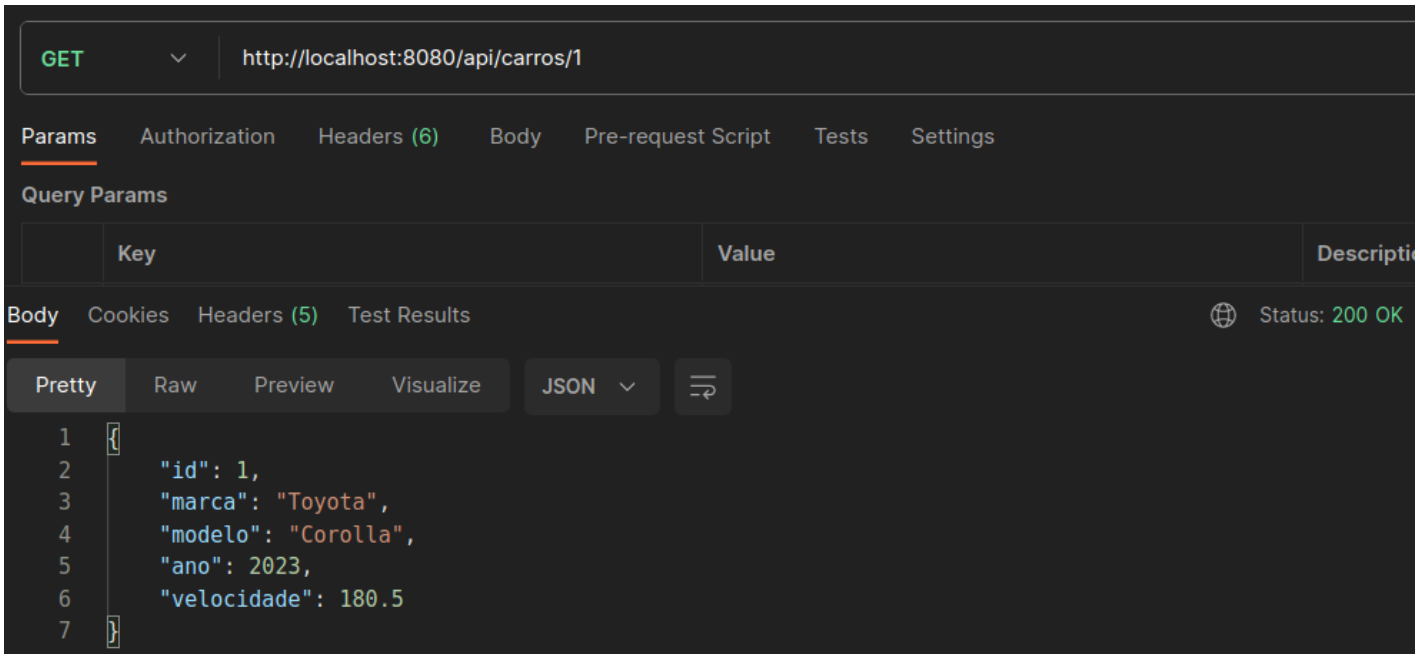
5. Adicionando mais funcionalidades ao Controller

```java
@RestController
@RequestMapping("/api/carros")
public class CarroController {

    private List<Carro> carros = new ArrayList<>();
    private Long currentId = 1L;

    @GetMapping
    public List<Carro> listarCarros() {
        return carros;
    }

    @PostMapping
    public ResponseEntity<Carro> adicionarCarro(@RequestBody Carro carro) {
        carro.setId(currentId++);
        carros.add(carro);
        return ResponseEntity.ok(carro);
    }

    // Buscando carro por ID
    @GetMapping("/{id}")
    public ResponseEntity<Carro> buscarCarroPorId(@PathVariable Long id) {
        Optional<Carro> carroEncontrado = carros.stream()
                        .filter(c -> c.getId().equals(id))
                        .findFirst();

        if (carroEncontrado.isPresent()) {
            return ResponseEntity.ok(carroEncontrado.get());
        } else {
            // Retorna status 404 caso não encontre o carro
            return ResponseEntity.notFound().build();
        }
    }
```

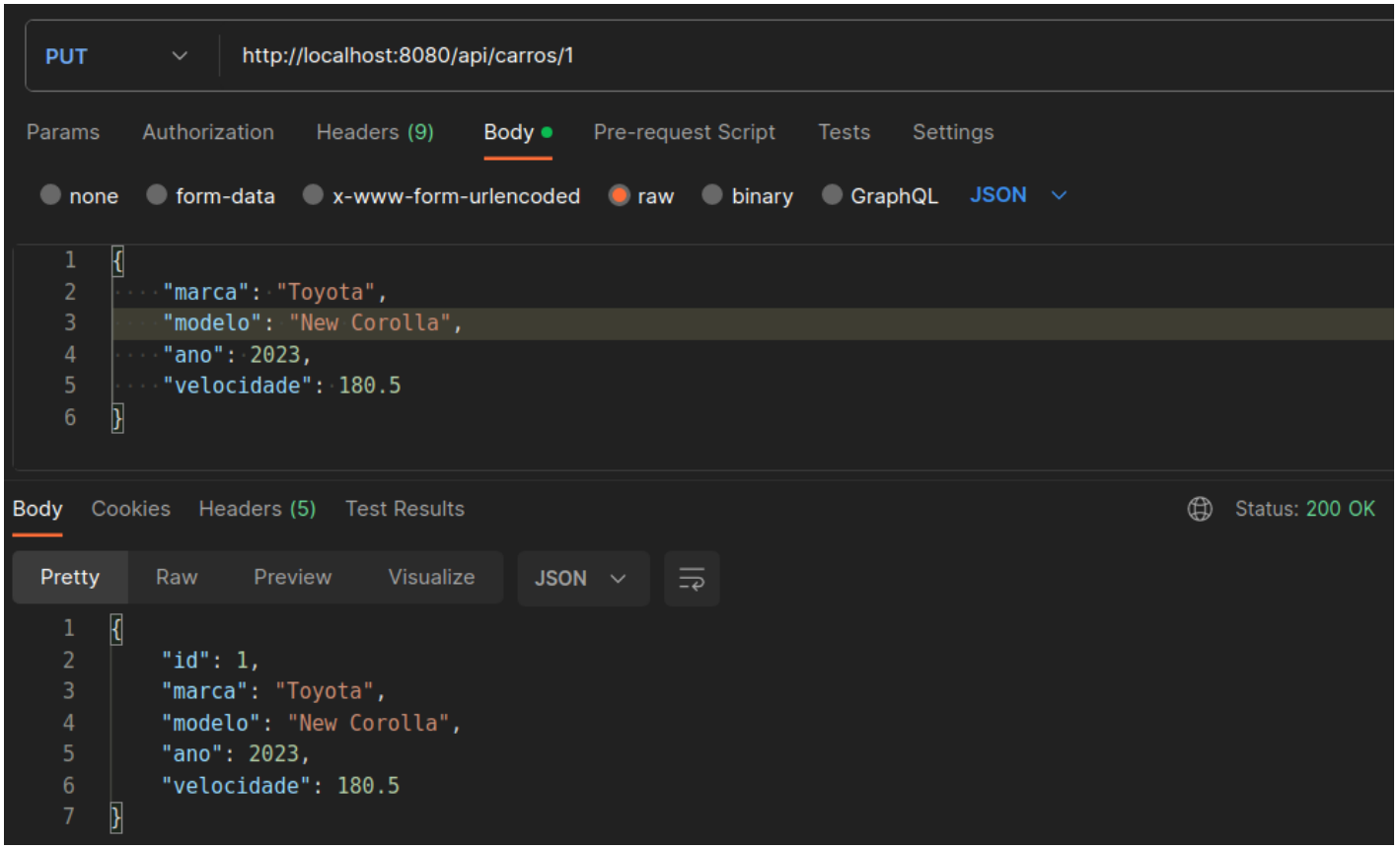
```
// Atualizando carro
    @PutMapping("/{id}")
    public    ResponseEntity<Carro>    atualizarCarro(@PathVariable    Long    id,
@RequestBody Carro carroAtualizado) {
        Optional<Carro> carroExistente = carros.stream()
                                        .filter(c -> c.getId().equals(id))
                                        .findFirst();

        if (carroExistente.isPresent()) {
            carros.remove(carroExistente.get());
            carroAtualizado.setId(id);
            carros.add(carroAtualizado);
            return ResponseEntity.ok(carroAtualizado);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Deletando carro
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deletarCarro(@PathVariable Long id) {
        Optional<Carro> carroParaDeletar = carros.stream()
                                          .filter(c -> c.getId().equals(id))
                                          .findFirst();

        if (carroParaDeletar.isPresent()) {
            carros.remove(carroParaDeletar.get());
            // Retorna status 204 No Content
            return ResponseEntity.noContent().build();
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Explicação:

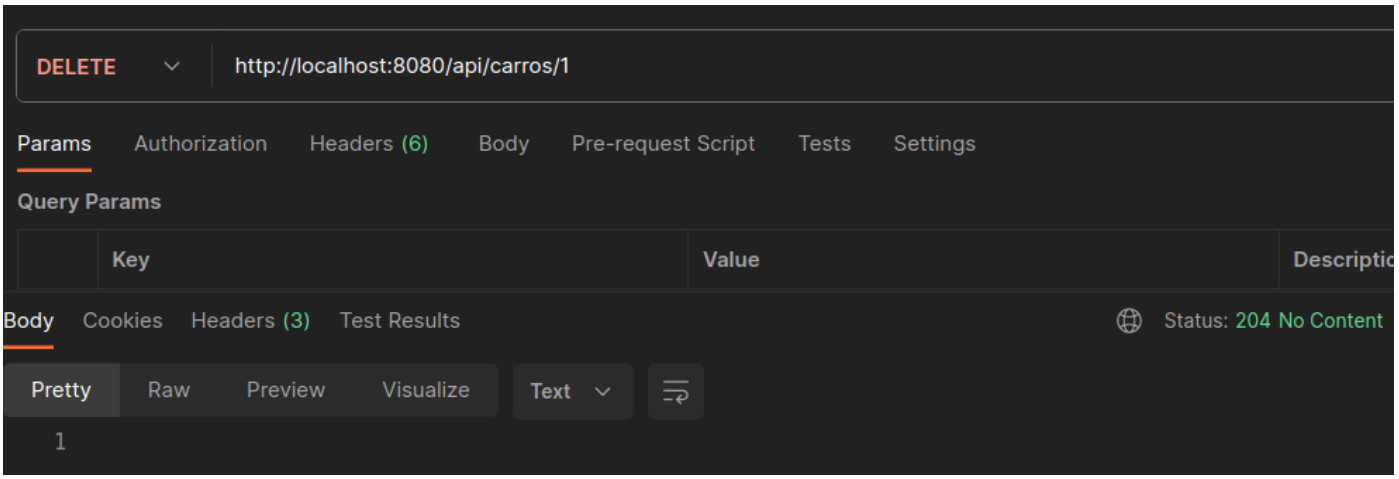
Buscar por ID: Usamos @GetMapping("/{id}") para criar um endpoint que responde a requisições GET a /api/carros/{id}, onde {id} é um ID de carro. O Spring vai pegar esse ID da URL e passar para o método como argumento, graças à anotação @PathVariable.



Atualizar carro: O método PUT é usado para atualizar recursos. Aqui, estamos usando @PutMapping para criar um endpoint que aceita uma representação de um carro no corpo da requisição e atualiza o carro com o ID fornecido na URL.



Deletar carro: O método DELETE é autoexplicativo. Ele deleta o carro com o ID fornecido



Nestes exemplos, estamos simulando um banco de dados com uma lista em memória. Mas, a seguir vamos adicionar um banco de dados. Entretanto, antes disso, vamos entender mais sobre alguns conceitos fundamentais.

JPA (Java Persistence API)

A JPA é uma especificação do Java para persistência de dados, ou seja, para armazenar e recuperar dados de bancos de dados em aplicações Java. Ela permite que você mapeie objetos Java para linhas em tabelas de um banco de dados relacional e vice-versa.

Vantagens da JPA:

Portabilidade: Ao usar JPA, o código relacionado ao banco de dados torna-se independente do sistema de gerenciamento de banco de dados (DBMS).

Redução de Código: A JPA elimina muito do código repetitivo envolvido na manipulação de bancos de dados.

Hibernate

O Hibernate é uma das implementações mais populares da JPA. Ele fornece uma estrutura para mapear objetos Java para registros de banco de dados e vice-versa. O Hibernate também inclui um recurso chamado HQL (Hibernate Query Language), uma linguagem orientada a objetos para fazer consultas ao banco de dados.

Principais características do Hibernate:

Performance: O Hibernate tem várias estratégias otimizadas de fetching de dados.

Flexibilidade: Pode trabalhar com diferentes bancos de dados.

Cache: Suporta cache de primeiro e segundo níveis.

Como tudo isso funciona?
A JPA, junto com o Hibernate, permite a você mapear suas classes Java para tabelas de banco de dados usando simples anotações. Por exemplo:

```
@Entity
public class Carro {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String marca;
   // outros campos e métodos...
}
```

@Entity: Indica que a classe é uma entidade que deve ser mapeada para uma tabela no banco de dados.

@Id: Indica que o campo é uma chave primária.

@GeneratedValue: Especifica como a chave primária será gerada.

Repositórios JPA

O Spring Data JPA fornece um sistema de repositório que permite que você crie interfaces para executar operações CRUD em suas entidades sem precisar escrever implementações específicas. O Spring Data JPA gera a implementação automaticamente!

```
public interface CarroRepository extends JpaRepository<Carro, Long> {
}
```

Neste exemplo, temos um repositório para a entidade Carro. O JpaRepository contém métodos como save(), findAll(), findById(), etc.

Em resumo, a JPA e o Hibernate simplificam a interação do Java com bancos de dados, transformando operações de banco de dados em operações orientadas a objetos. O Spring Data JPA leva isso adiante, eliminando a necessidade de escrever código repetitivo para operações comuns do banco de dados. Tudo isso permite que os desenvolvedores se concentrem na lógica de negócios sem se preocupar muito com detalhes de baixo nível do banco de dados.

Relacionamentos Entre Entidades

O JPA oferece anotações específicas para representar diferentes tipos de relacionamentos entre entidades. Veremos agora os principais:

@OneToOne

O relacionamento @OneToOne indica que uma instância de uma entidade está associada a apenas uma instância de outra entidade e vice-versa. É como dizer: "Para cada item A, há um e apenas um item B correspondente e vice-versa".

Vamos imaginar que temos duas entidades: Carro e RegistroCarro. Cada carro tem um único registro, e cada registro pertence a um único carro.

```
@Entity
public class Carro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String marca;
    private String modelo;

    @OneToOne(mappedBy = "carro", cascade = CascadeType.ALL)
    private RegistroCarro registro;
    // getters, setters...
}
```

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

    @OneToOne
    @JoinColumn(name = "carro_id")
    private Carro carro;
    // getters, setters...
}
```

Neste exemplo:

@OneToOne: Anotação colocada nos campos que representam os lados do relacionamento.

mappedBy: Esta propriedade é usada para definir o lado "não dominante" do relacionamento. No nosso exemplo, Carro é o lado não dominante, pois a tabela RegistroCarro conterá a chave estrangeira.

@JoinColumn: Usado para indicar qual coluna é usada como chave estrangeira. No nosso caso, a tabela RegistroCarro terá uma coluna carro_id que aponta para a tabela Carro.

cascade = CascadeType.ALL: Isso indica que qualquer operação feita no Carro (como salvar, atualizar ou excluir) deve ser refletida no RegistroCarro associado. Por exemplo, se excluirmos um Carro, o RegistroCarro correspondente também será excluído.

Benefícios e Usos:
Integridade dos Dados: O uso de relacionamentos garante a integridade referencial dos dados. Você não terá um RegistroCarro apontando para um Carro que não existe.

Facilidade de Navegação: Uma vez que o relacionamento esteja estabelecido, é fácil navegar entre as entidades. Por exemplo, se você tiver um objeto Carro, pode facilmente obter o RegistroCarro associado e vice-versa.

Eficiência: Como as tabelas são associadas por chaves estrangeiras, as operações de consulta são otimizadas.

Observações:
O relacionamento @OneToOne é útil em situações onde a divisão entre as entidades faz sentido do ponto de vista de domínio ou de design.

É crucial definir corretamente o lado dominante e o lado não dominante do relacionamento para garantir que o mapeamento e as operações de banco de dados sejam feitas corretamente.

@OneToMany

O relacionamento @OneToMany indica que uma instância de uma entidade está associada a múltiplas instâncias de outra entidade. Por outro lado, cada uma dessas múltiplas instâncias se relaciona com apenas uma instância da primeira entidade. Em termos simples, é uma relação "um para muitos".

Imagine que temos uma entidade MarcaCarro e outra entidade Carro. Uma marca pode ter vários carros associados a ela (por exemplo, a marca "Ford" pode ter modelos como "Fiesta", "Focus", "Mustang" etc.), mas cada carro pertence a apenas uma marca.

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

   @OneToMany(mappedBy = "marca", cascade = CascadeType.ALL)
   private List<Carro> carros = new ArrayList<>();
   // getters, setters...
}
```

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

    @ManyToOne
    @JoinColumn(name = "marca_id")
    private MarcaCarro marca;
    // getters, setters...
}
```

Neste exemplo:

@OneToMany: Anotação colocada no campo que representa o lado "um" do relacionamento.

mappedBy: Indica o campo na entidade "muitos" que é o proprietário do relacionamento. No nosso caso, MarcaCarro é o lado "um" e a entidade Carro possui o campo marca que é o proprietário do relacionamento.

@ManyToOne: É a contraparte de @OneToMany e indica o lado "muitos" do relacionamento. Cada carro está associado a uma marca.

@JoinColumn: Indica qual coluna é usada como chave estrangeira. Aqui, a tabela Carro terá uma coluna marca_id que aponta para a tabela MarcaCarro.

cascade = CascadeType.ALL: Assim como no exemplo anterior, isso indica que qualquer operação feita na MarcaCarro (como salvar, atualizar ou excluir) deve ser refletida em todos os Carros associados.

Benefícios e Usos:
Integridade dos Dados: O relacionamento garante a integridade dos dados, pois você não pode ter um Carro sem uma MarcaCarro associada (a menos que o relacionamento seja opcional).

Consulta Eficiente: O relacionamento facilita a recuperação de todos os carros de uma determinada marca com uma simples consulta.

Organização e Hierarquia: Permite agrupar entidades de maneira lógica. No nosso exemplo, todos os carros são agrupados por marcas.

Observações:
Em bancos de dados relacionais, a relação "um para muitos" é geralmente implementada usando uma chave estrangeira na tabela "muitos" apontando para a tabela "um". Assim como em @OneToOne, é importante definir corretamente o lado dominante do relacionamento.

@ManyToMany

O relacionamento @ManyToMany indica que várias instâncias de uma entidade podem estar associadas a várias instâncias de outra entidade. É uma relação "muitos para muitos".

Imagine que temos uma entidade Carro e outra entidade Motorista. Um Carro pode ser dirigido por vários Motoristas em diferentes momentos, e um Motorista pode dirigir vários Carros diferentes.

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

    @ManyToMany
    @JoinTable(
            name = "carro_motorista",
            joinColumns = @JoinColumn(name = "carro_id"),
            inverseJoinColumns = @JoinColumn(name = "motorista_id")
    )
    private List<Motorista> motoristas = new ArrayList<>();
    // getters, setters...
}
```

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

    @ManyToMany(mappedBy = "motoristas")
    private List<Carro> carros = new ArrayList<>();
    // getters, setters...
}
```

Neste exemplo:

@ManyToMany: Anotação usada nos campos que representam ambos os lados do relacionamento.

@JoinTable: Como um relacionamento "muitos para muitos" geralmente requer uma tabela de junção (ou tabela intermediária), usamos @JoinTable para definir essa tabela.

name: Define o nome da tabela de junção.

joinColumns: Define a coluna da tabela de junção que está associada ao ID da entidade atual (neste caso, Carro).

inverseJoinColumns: Define a coluna da tabela de junção que está associada ao ID da outra entidade (neste caso, Motorista).

mappedBy: Indica o campo na outra entidade que é o proprietário do relacionamento. No nosso caso, a entidade Motorista referencia o relacionamento através do campo motoristas na entidade Carro.

Benefícios e Usos:
Relações Complexas: Quando duas entidades têm múltiplas associações entre si, @ManyToMany oferece uma maneira eficaz de mapear essa relação.

Flexibilidade: Permite associar um Carro a vários Motoristas e vice-versa, sem repetir os dados nas entidades principais.

Consulta Eficiente: Você pode facilmente recuperar todos os Motoristas associados a um Carro específico e todos os Carros associados a um Motorista específico.

Observações:
Em bancos de dados relacionais, o relacionamento "muitos para muitos" é implementado usando uma tabela de junção. Esta tabela contém chaves estrangeiras que apontam para as tabelas das duas entidades envolvidas.

É vital definir corretamente qual entidade é o proprietário do relacionamento, pois isso afeta como as operações de atualização são propagadas.

Tipos de carregamento ("fetching strategies")

Os tipos de carregamento, ou "fetching strategies", no JPA (Java Persistence API) determinam quando os dados relacionados de uma entidade serão carregados do banco de dados. Existem principalmente dois tipos de estratégias de carregamento: Lazy (Preguiçoso) e Eager (Ávido). Eles são especialmente importantes em contextos onde as entidades possuem relacionamentos com outras.

Lazy Loading (Carregamento Preguiçoso):

Como funciona:
Com o carregamento preguiçoso, os dados relacionados de uma entidade são carregados sob demanda. Ou seja, eles só são carregados quando você explicitamente acessa esses dados.

Para que serve:
Essa abordagem é útil para melhorar a performance. Se você tem uma entidade Carro e esta tem uma relação com Motorista, ao buscar um Carro, o JPA não buscará imediatamente os Motoristas associados a ele, a menos que você explicitamente acesse essa lista de motoristas.

Suponha que temos uma entidade Carro que tem um relacionamento com Motorista:

```
@Entity
public class Carro {
    // ... outros campos ...

    @OneToMany(fetch = FetchType.LAZY)
    private List<Motorista> motoristas;
}
```

Se buscarmos um Carro do banco de dados, a lista de Motoristas associados a ele não será imediatamente carregada. Apenas quando fizermos algo como carro.getMotoristas(), o JPA irá ao banco de dados e carregará os motoristas.

Eager Loading (Carregamento Esperto):

Como funciona:
Com o carregamento esperto, os dados relacionados de uma entidade são carregados imediatamente, junto com a entidade principal.

Para que serve:
Isso é útil quando sabemos que quase sempre precisaremos dos dados relacionados junto com a entidade principal. Usando o exemplo anterior, se sempre que buscarmos um Carro também quisermos ver seus Motoristas, o carregamento ávido pode ser uma boa escolha.

Novamente, usando a entidade Carro:

```
@Entity
public class Carro {
    // ... outros campos ...

    @OneToMany(fetch = FetchType.EAGER)
    private List<Motorista> motoristas;
}
```

Aqui, ao buscar um Carro, o JPA também carregará todos os Motoristas associados a ele simultaneamente.

Considerações

Performance:
Lazy Loading é geralmente preferido na maioria dos cenários para melhor performance, pois evita carregar dados desnecessários.

Eager Loading pode levar a problemas de performance, especialmente se a entidade principal tiver muitos dados relacionados. Imagine carregar um Carro e acabar trazendo milhares de Motoristas relacionados se você não precisar deles imediatamente.

O problema N+1

O problema N+1 é um desafio comum de performance quando lidamos com sistemas de gerenciamento de banco de dados relacionais e frameworks ORM (Object-Relational Mapping) como o Hibernate/JPA no Java. Esse problema ocorre quando, ao recuperar dados de um banco de dados, o sistema realiza um número excessivo de consultas, causando impactos negativos na performance.

Vamos usar o contexto de carros para entender melhor:
Suponha que você tenha duas entidades: Carro e Motorista, onde um Carro tem um Motorista. Agora, imagine que você queira recuperar uma lista de carros e, para cada carro, exibir as informações do motorista associado.

O que acontece:
Primeira Consulta (1): Você faz uma consulta para recuperar todos os carros.

```
SELECT * FROM carro; -- carroRepository.findAll();
```

Consultas Subsequentes (N): Para cada carro retornado na primeira consulta, você faz uma consulta adicional para recuperar o motorista associado.

```
SELECT * FROM motorista WHERE carro_id = ?;
```

Então, se tivermos 10 carros, teremos 1 consulta inicial para os carros e mais 10 consultas para recuperar os motoristas, totalizando 11 consultas. Daí vem o nome "N+1" - 1 consulta inicial + N consultas para os detalhes.

Por que isso é problemático:
O problema N+1 pode resultar em uma quantidade significativamente grande de consultas ao banco de dados, mesmo para uma quantidade relativamente pequena de registros. Isso pode causar uma série de problemas, incluindo:

Performance degradada: Realizar muitas consultas pequenas é geralmente menos eficiente do que uma consulta bem construída que traz todas as informações necessárias de uma vez.

Aumento no uso de recursos: Isso coloca mais carga tanto na aplicação quanto no banco de dados.

Solução com Spring Data JPA

Usar @EntityGraph com Spring Data JPA:
A anotação @EntityGraph permite especificar quais atributos devem ser carregados de maneira antecipada (Eager). Isso pode ser usado no repositório:

```
@EntityGraph(attributePaths = "motorista")
List<Carro> findAllWithMotorista();
```

Agora, ao chamar findAllWithMotorista(), o JPA carregará os carros e seus motoristas em uma única consulta.

Consultas JPQL personalizadas

Você pode definir uma consulta JPQL personalizada com um JOIN FETCH para carregar os dados relacionados:

```
@Query("SELECT c FROM Carro c JOIN FETCH c.motorista")
List<Carro> findAllWithJoinFetch();
```

Esta consulta trará todos os carros e seus motoristas associados em uma única operação.

Revise os relacionamentos Lazy vs. Eager:

Embora o padrão recomendado em muitos relacionamentos seja usar o carregamento Lazy para evitar carregar dados desnecessários, em alguns cenários, se você sempre precisar dos dados relacionados, pode considerar configurar o relacionamento para ser carregado de forma Eager. No entanto, tenha cuidado com isso, pois pode levar ao carregamento de mais dados do que o necessário em outras operações.

```
@ManyToOne(fetch = FetchType.EAGER)
private Motorista motorista;
```

A combinação de Spring Data JPA com as anotações e técnicas corretas permite otimizar as consultas ao banco de dados, evitando o problema N+1 e melhorando a performance da aplicação. Sempre é bom monitorar e revisar o comportamento das consultas, especialmente em sistemas complexos, para garantir uma operação eficiente.

Lock

Imagine que você tem uma grande garagem que aluga carros. Você tem um sistema que rastreia todos os carros alugados e disponíveis. Se dois clientes tentarem alugar o mesmo carro ao mesmo tempo, isso seria problemático, certo? Especialmente se ambos pensassem que tinham alugado o carro e saíssem dirigindo. Seria um caos!

No mundo dos bancos de dados, o "lock" (travamento) é uma maneira de garantir que, quando um recurso (neste caso, um registro de carro) está sendo usado ou modificado, nenhum outro processo pode acessá-lo ou alterá-lo até que o primeiro processo termine.

Para que serve?

O principal objetivo do "lock" é garantir a integridade e a consistência dos dados. No contexto do nosso exemplo da garagem, se não tivéssemos travas (locks), poderíamos alugar o mesmo carro para dois clientes diferentes ao mesmo tempo!

Como funciona?

Vamos aprofundar um pouco o exemplo:

Lock Otimista (Optimistic Locking)

É como se, ao alugar um carro, o sistema anotasse a hora atual e a comparasse quando o carro fosse realmente alugado. Se outra pessoa tentasse alugar o mesmo carro enquanto a primeira transação ainda não estivesse concluída e a hora registrada fosse diferente, o sistema saberia e impediria o segundo aluguel.

No mundo JPA, isso é frequentemente implementado usando uma coluna de versão (@Version). Cada vez que um registro é atualizado, a coluna de versão é incrementada. Se outra transação tentar atualizar o mesmo registro e a versão for diferente da que foi lida inicialmente, uma exceção é lançada.

Lock Pessimista (Pessimistic Locking)

No caso pessimista, quando um cliente decide alugar um carro, o sistema reserva imediatamente esse carro, impedindo que qualquer outro cliente o alugue até que a primeira transação seja concluída ou a reserva expire.

No JPA, podemos usar o método lock ou a cláusula FOR UPDATE em SQL para implementar isso.

Vantagens

- **Consistência dos Dados:** As travas garantem que os dados sejam consistentes e que as transações sejam concluídas sem interferências indesejadas.

- **Evita Condições de Corrida:** Especialmente em sistemas altamente concorrentes, o locking pode prevenir problemas onde múltiplas operações tentam modificar o mesmo recurso simultaneamente.

Desvantagens

- **Desempenho:** O uso excessivo de travamento, especialmente o pessimista, pode reduzir o desempenho, pois as transações têm que esperar que os locks sejam liberados.

- **Deadlocks:** Se dois processos estiverem esperando um ao outro para liberar um recurso, pode ocorrer um deadlock. É uma situação onde ambos os processos ficam esperando indefinidamente.

- **Complexidade:** Gerenciar locks, especialmente em sistemas grandes, pode tornar o código e a lógica de negócios mais complexos.

Assim como você não gostaria que dois clientes alugassem o mesmo carro ao mesmo tempo, os locks em bancos de dados são essenciais para garantir a integridade e a consistência dos dados. No entanto, é fundamental equilibrar a necessidade de locking com o desempenho e a complexidade do sistema.

H2 - Bando de Dados Em Memória

O H2 é um banco de dados relacional leve escrito em Java. Uma das principais vantagens do H2 é que ele pode ser incorporado em aplicações Java, o que significa que você não precisa instalar e configurar um servidor de banco de dados separado; ele roda junto com sua aplicação. Isso torna o H2 uma excelente opção para **testes** e **prototipagem**.

Características do H2

- **Modos de Operação:** O H2 pode funcionar tanto em modo incorporado (dentro da sua aplicação) quanto em modo servidor (como um servidor de BD tradicional).
- **Interface Web de Console:** O H2 vem com uma interface de console web simples que permite executar consultas e gerenciar o banco de dados.
- **Rápido e Leve:** Por ser um banco de dados em memória, o H2 é muito rápido, tornando-o ideal para testes unitários.
- **Compatibilidade SQL:** Ele suporta a maioria das funcionalidades SQL padrão.

Configurando H2 no Spring Boot

Dependências

Adicione as dependências necessárias ao seu arquivo pom.xml (se estiver usando Maven):

```
<dependencies>
    <!-- outras dependencias… -->

    <!-- H2 Database -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

</dependencies>
```

Configuração

No arquivo application.properties ou application.yml, adicione as seguintes configurações:

```
# Habilita o console web do H2
spring.h2.console.enabled=true

# Configuração da fonte de dados
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# JPA e Hibernate config
spring.jpa.hibernate.ddl-auto=update
```

Aqui, o banco de dados H2 é criado na memória (jdbc:h2:mem:testdb). O Hibernate tentará criar ou atualizar as tabelas conforme necessário.

Console Web

Após iniciar sua aplicação Spring Boot, você pode acessar o console web do H2 em: http://localhost:8080/h2-console. Use as credenciais fornecidas nas configurações (sa e senha em branco, neste exemplo) para acessar o banco de dados.

Entidades e Repositórios

Com o H2 e o Spring Data JPA configurados, você pode criar suas entidades (classes de domínio) e repositorios (interfaces para consulta). O Spring Data JPA e o Hibernate cuidarão da criação das tabelas e das operações CRUD.

O H2 é uma excelente ferramenta para desenvolvimento e testes, mas não é recomendado para ambientes de produção devido à sua natureza em memória (os dados são perdidos após reiniciar a aplicação). Para ambientes de produção, é aconselhável usar bancos de dados mais robustos como PostgreSQL, MySQL, Oracle, entre outros.Adicionando Banco de Dados as Nossas APIs

1. Entidade Carro

Uma entidade é, na prática, uma representação de uma tabela no banco de dados. No Hibernate/JPA, usamos classes Java com anotações específicas para representar essas tabelas.

Na classe Carro:

- **@Entity:** Esta anotação diz ao JPA que a classe Carro é uma entidade que deve ser mapeada para uma tabela de banco de dados.
- **@Id:** Esta anotação indica que o campo id é a chave primária da tabela.

- **@GeneratedValue:** Esta anotação diz ao JPA para gerar automaticamente um valor para o campo id sempre que um novo registro for inserido.

```
package com.matera.bootcamp.model;

import lombok.Data;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Data
@Entity
public class Carro {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String marca;
   private String modelo;
   private int ano;
   private double velocidade;
   private String placa;
}
```

2. Repositório

Em JPA, um repositório é uma camada que facilita o acesso ao banco de dados. Pense nisso como um conjunto de operações comuns que você deseja realizar no banco de dados, como salvar, encontrar ou deletar registros.

Criando o CarroRepository

JpaRepository: Ao estender esta interface, você obtém muitas operações básicas de CRUD (criar, ler, atualizar, deletar) sem precisar escrevê-las manualmente.

```
package com.matera.bootcamp.repository;

import com.matera.bootcamp.model.Carro;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CarroRepository extends JpaRepository<Carro, Long> {
}
```

Anteriormente, estávamos usando uma lista em memória para armazenar carros. Mas agora, queremos persistir esses dados em um banco de dados. Para isso, usaremos nosso CarroRepository.

Service

Vamos criar a classe CarroService. A camada de serviço é o coração da lógica de negócios e, frequentemente, lidamos com tratamentos de exceções, validações, transformações de dados e chamadas para outras camadas ou serviços aqui.

Primeiro, a estrutura básica:

```
package com.matera.bootcamp.service;

import com.matera.bootcamp.repository.CarroRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class CarroService {

    @Autowired
    private CarroRepository carroRepository;

    // Métodos serão inseridos aqui
}
```

1. Listar Carros

Este é um método simples que busca todos os carros no repositório e os retorna:

```
public List<Carro> listarCarros() {
    return carroRepository.findAll();
}
```

2. Adicionar Carro

Quando um novo carro é adicionado, ele deve ser salvo no repositório:

```
    public Carro adicionarCarro(Carro carro) throws Exception {
        // 1. Verificando se a placa do carro já existe
        Carro carroExistente = carroRepository.findByPlaca(carro.getPlaca());
        if (carroExistente != null) {
            throw new Exception("Um carro com essa placa já existe.");
        }

        // 2. Validando campos obrigatórios
        if (carro.getModelo() == null || carro.getModelo().trim().isEmpty()) {
            throw new Exception("O modelo do carro é obrigatório.");
        }

        if (carro.getMarca() == null || carro.getMarca().trim().isEmpty()) {
            throw new Exception("A marca do carro é obrigatória.");
        }

        // Aqui poderíamos adicionar mais validações conforme necessário
```

```
        return carroRepository.save(carro);
    }
```

Ajustando o CarroRepository:

```
@Repository
public interface CarroRepository extends JpaRepository<Carro, Long> {
    Carro findByPlaca(String placa);
}
```

Explicação:

Estamos fazendo uma busca no banco de dados pela placa do carro que se deseja adicionar. Se encontrarmos um carro com a mesma placa, lançamos uma exceção.

Em seguida, verificamos se os campos modelo e marca estão preenchidos. Caso contrário, lançamos exceções.

Lembre-se: é crucial fornecer mensagens de erro claras e informativas. Elas ajudam muito, especialmente quando você está depurando ou quando um cliente está tentando entender por que algo não está funcionando como esperado.

Além disso, em um cenário real, em vez de lançar a exceção genérica *Exception*, você provavelmente usaria (ou criaria) exceções específicas, como ValidacaoException, RecursoDuplicadoException, entre outras. Isso torna o código mais limpo e o tratamento de exceções mais granular.

3. Buscar Carro por ID

Retorna um carro baseado no ID:

```
public Optional<Carro> buscarCarroPorId(Long id) {
    return carroRepository.findById(id);
}
```

4. Atualizar Carro

Antes de atualizar um carro, queremos verificar se ele já existe. Se não existir, podemos lançar uma exceção:

```
public Carro atualizarCarro(Long id, Carro carroAtualizado) throws Exception {
    if (!carroRepository.existsById(id)) {
        throw new Exception("Carro não encontrado.");
    }
    // Garantindo que o ID do carroAtualizado é o mesmo que queremos atualizar
    carroAtualizado.setId(id);
    return carroRepository.save(carroAtualizado);
}
```

5. Deletar Carro

Novamente, antes de excluir, verificamos a existência:

```
public void deletarCarro(Long id) throws Exception {
    if (!carroRepository.existsById(id)) {
        throw new Exception("Carro não encontrado.");
    }
    carroRepository.deleteById(id);
}
```

Entendendo o que fizemos:
Uso do @Service: Esta anotação informa ao Spring que a classe deve ser tratada como um componente de serviço. Isso também significa que o Spring gerenciará o ciclo de vida desta classe (como a criação de uma instância).

Injeção do CarroRepository com @Autowired: Estamos delegando ao Spring a responsabilidade de fornecer uma instância do repositório. Isso simplifica a gestão de dependências.

Tratamento de exceções: Em métodos como "atualizar" e "deletar", é essencial verificar se o recurso (neste caso, o carro) realmente existe antes de realizar a operação. Se não existir, lançamos uma exceção. Em um ambiente real, você usaria exceções personalizadas para tratar esses casos com mais clareza.

Uso do Optional: O Spring Data JPA retorna Optional em métodos como findById. O Optional é uma caixa que pode ou não conter um valor e é uma forma de lidar com possíveis valores nulos sem ter que fazer verificações nulas explícitas. Se o valor dentro do Optional existir, ele pode ser obtido; caso contrário, pode-se lidar com a ausência de valor de maneira adequada.

Ajustando Nosso Controller

```
package com.matera.bootcamp.controller;

import com.matera.bootcamp.service.CarroService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import lombok.extern.slf4j.Slf4j;

@Slf4j
@RestController
@RequestMapping("/api/carros")
public class CarroController {

    @Autowired
    private CarroService carroService;

    // adicione os métodos a seguir, aqui
}
```

1. API Listar Carros

```
@GetMapping
public ResponseEntity<List<Carro>> listarCarros() {
    return ResponseEntity.ok(carroService.listarCarros());
}
```

2. API Adicionar Carro

```
@PostMapping
public ResponseEntity<Carro> adicionarCarro(@RequestBody Carro carro) {
    try {
        Carro carroAdicionado = carroService.adicionarCarro(carro);
        return ResponseEntity.ok(carroAdicionado);
    } catch (Exception e) {
        return ResponseEntity.badRequest().body(null);
    }
}
```

3. API Buscar Carro Por Id

```
@GetMapping("/{id}")
public ResponseEntity<Carro> buscarCarroPorId(@PathVariable Long id) {
    Optional<Carro> carroOptional = carroService.buscarCarroPorId(id);
    if (carroOptional.isPresent()) {
        Carro carro = carroOptional.get();
        return ResponseEntity.ok(carro);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

4. API Atualizar Carro

```
@PutMapping("/{id}")
public ResponseEntity<Carro> atualizarCarro(@PathVariable Long id,
@RequestBody Carro carroAtualizado) {
    try {
        Carro carroAtual = carroService.atualizarCarro(id, carroAtualizado);
        if (carroAtual != null) {
            return ResponseEntity.ok(carroAtual);
        } else {
            return ResponseEntity.notFound().build();
        }
    } catch (Exception e) {
        return ResponseEntity.badRequest().body(null);
    }
}
```

5. API Apagar Carro

```
@DeleteMapping("/{id}")
public ResponseEntity<Void> deletarCarro(@PathVariable Long id) {
    try {
        carroService.deletarCarro(id);
        return ResponseEntity.noContent().build();
    } catch (Exception e) {
        log.error("Erro ao deletar carro de id {}", id, e);
        return ResponseEntity.notFound().build();
    }
}
```

Explicação:
@Autowired: Essa anotação injeta a instância de CarroService no controller, permitindo que você chame seus métodos.

adicionarCarro: Aqui, quando tentamos adicionar um carro, delegamos essa tarefa para o método adicionarCarro do serviço. Se tudo funcionar corretamente, retornamos o carro adicionado. Se ocorrer algum erro (por exemplo, carro com a mesma placa ou campos obrigatórios ausentes), retornamos um status 400 Bad Request.

buscarCarroPorId: A busca agora é delegada para o serviço, e o controller só precisa verificar se o carro foi encontrado ou não.

atualizarCarro: Novamente, delegamos para o serviço a lógica de atualização. Se houver algum problema, como não encontrar o carro pelo ID fornecido, retornamos 404 Not Found. Se ocorrer um erro durante a atualização (por exemplo, validações), retornamos 400 Bad Request.

deletarCarro: Delegamos a operação de exclusão para o serviço e verificamos se a operação foi bem-sucedida.

A separação de responsabilidades e a delegação da lógica de negócios para o serviço tornam o código mais organizado e fácil de manter. O controller agora lida principalmente com a recepção de requisições HTTP, delegando as operações específicas para o serviço.