Java Spring Professional

Coleção Java Spring

Nelio Alves

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Alves, Nelio
Java spring professional [livro eletrônico] / Nelio Alves. -- Belo Horizonte, MG : Neme Digital, 2025. -- (Coleção java spring)
PDF

ISBN 978-65-999289-6-3

1. Java (Linguagem de programação) 2. Linguagem de programação para computadores - Estudo e ensino I. Título. II. Série.

25-263290 CDD-005.133

Índices para catálogo sistemático:

1. Java : Linguagem de programação : Computadores : Processamento de dados 005.133

Eliane de Freitas Leite - Bibliotecária - CRB 8/8415

Introdução

Visão geral do Java Spring Professional

Bem-vindo(a) à jornada pelo universo do Java Spring Professional, um percurso meticulosamente planejado para transformar desenvolvedores aspirantes em verdadeiros mestres do desenvolvimento com Spring Boot. Este livro é mais do que apenas um manual; é um convite para você se aprofundar nas práticas e ferramentas que definem a excelência em desenvolvimento Java moderno.

Por que Spring Boot?

A escolha de focar em Spring Boot não é por acaso. Este framework robusto, baseado na plataforma Java, é reconhecido por sua capacidade de simplificar o desenvolvimento de aplicações robustas e altamente escaláveis. Com Spring Boot, você pode esperar uma redução drástica na quantidade de código necessário para configurar e lançar novos aplicativos, permitindo que você se concentre no que realmente importa: a lógica de negócios e a inovação.

Estrutura do Curso

Este livro está organizado em seis capítulos detalhados, cada um abordando aspectos fundamentais para dominar o Spring Boot. Você começará entendendo os **Componentes e Injeção de Dependência**, essenciais para qualquer aplicação Spring. Este capítulo inicial não só fundamenta seu conhecimento nas capacidades básicas do Spring, mas também o prepara para explorar configurações mais complexas e personalizadas.

À medida que avançamos, você mergulhará no mundo dos **Modelos de Domínio e ORM**, descobrindo como mapear objetos para um banco de dados de forma eficiente, passando por todas as complexidades de JPA e Hibernate. Essas habilidades são cruciais para trabalhar com dados de maneira profissional e eficaz.

No terceiro capítulo, o foco será em construir e estruturar uma **API REST**. Você aprenderá sobre padrões de design de API, CRUD, e como lidar com exceções e validações, garantindo que sua aplicação não só funcione bem, mas também seja fácil de manter e expandir.

Os capítulos subsequentes cobrirão tópicos avançados como **JPA, Consultas SQL e JPQL**, onde você aprimorará sua habilidade em otimizar consultas e entenderá melhor a performance de aplicações. O capítulo sobre **Login e Controle de Acesso** demonstra a implementação de autenticação e segurança, aspectos cruciais para qualquer aplicativo moderno.

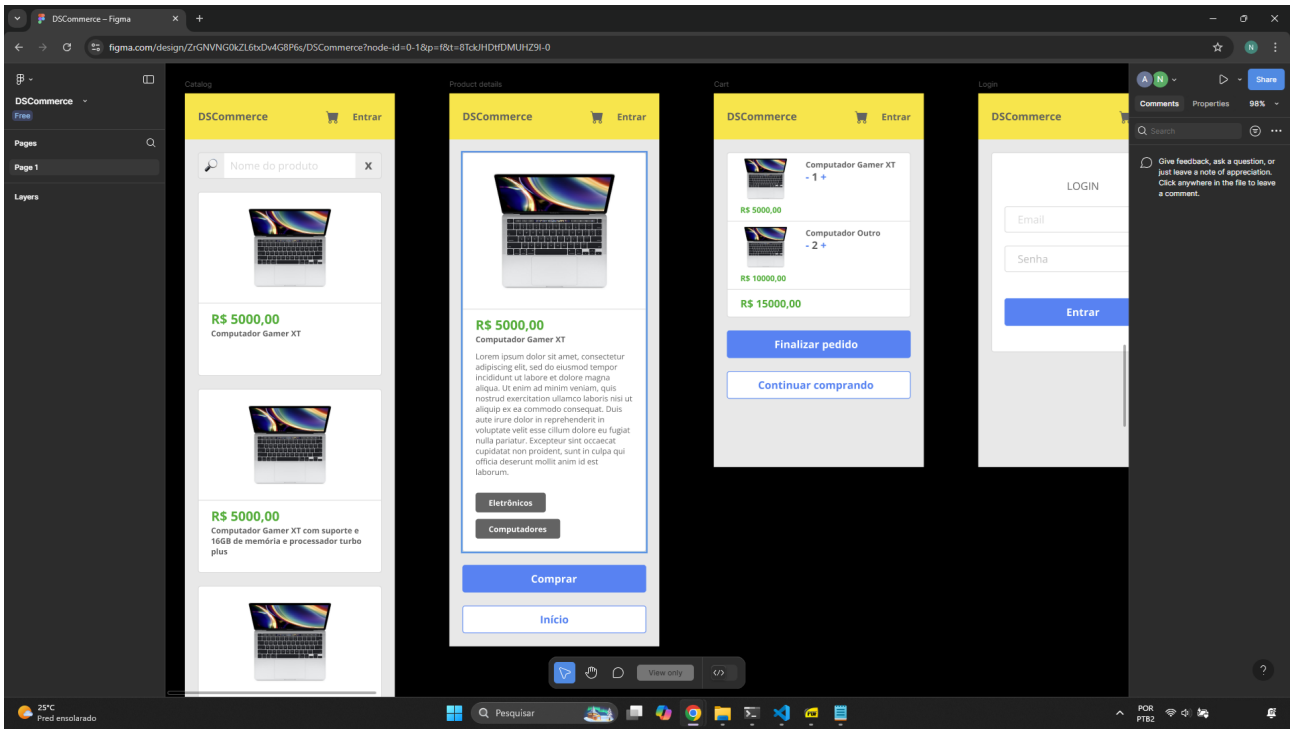
Por fim, o livro conclui com práticas essenciais de **Homologação e Implantação com**

CI/CD, preparando você para o mundo real das aplicações empresariais, onde a entrega contínua e a integração contínua definem o ritmo do desenvolvimento.

O sistema DSCommerce

Vamos construir passo a passo um sistema de comércio, o qual chamamos de DSCommerce. Durante o processo, vamos consultar o documento de requisitos, bem como o design Figma do front end, valendo lembrar que este livro é sobre a construção do back end deste sistema, então na verdade o design Figma nos servirá para ajudar a compreender melhor o que deve ser implementado.

O documento de requisitos (que inclusive contém o link para o design Figma) pode ser acessado no material de apoio anexo.



Design Figma do sistema DSCommerce. Será utilizado durante o desenvolvimento para nos ajudar a compreender melhor o que deve ser implementado.

Durante o processo de construção do back end do DSCommerce, vamos implementar as seguintes funcionalidades:

- **Consulta de catálogo**. Esta funcionalidade permitirá listar produtos, inclusive filtrando os mesmos. Esta funcionalidade servirá para o usuário encontrar produtos para colocar em seu carrinho de compras.
- **Login**, por meio da qual o usuário informa suas credenciais e obtem um token de acesso às funcionalidades protegidas do sistema, conforme seus perfis de acesso.
- **Manter produtos**, ou seja, as quatro operações básicas de criar, recuperar, atualizar e deletar produtos, também conhecida como CRUD (Create, Retrieve, Update, Delete). Esta funcionalidade servirá para a área administrativa do sistema.
- **Registrar pedido**. Esta funcionalidade é responsável por registrar um pedido do cliente no sistema, a partir das informações do carrinho de compras.

Pré-requisitos

Este não é um conteúdo para quem ainda não sabe nada de programação. Para que você consiga acompanhar este conteúdo com entendimento, é preciso ter alguns conhecimentos básicos listados a seguir:

- **Fundamentos de programação (independente da linguagem)**
  - Conhecimento básico de lógica e orientação a objetos em qualquer linguagem
  - Conhecimento básico de banco de dados relacional e SQL
  - Conhecimento básico de Git
- **Conhecimento básico de Java**
  - Sintaxe básica de classes, atributos e funções
  - Construtores, modificadores public/private, funções get/set
  - Pacotes Java

Seu Caminho para a Maestria

Com cada capítulo, você enfrentará desafios projetados para testar e consolidar seu conhecimento, garantindo que ao final do livro, você não apenas compreenda teoricamente o Spring Boot, mas também seja capaz de aplicar efetivamente esses conhecimentos em cenários do mundo real.

Estamos entusiasmados para guiá-lo nesta jornada de aprendizado. Prepare-se para mergulhar de cabeça e emergir como um profissional qualificado e confiante, pronto para enfrentar os desafios de desenvolvimento mais exigentes com Spring Boot. Vamos começar!

Preparação de ambiente

Antes de mergulharmos a fundo do desenvolvimento com Spring Boot, é crucial que seu ambiente de trabalho esteja devidamente configurado. Esta seção guiará você por um processo passo a passo para preparar seu computador, garantindo que você tenha todas as ferramentas necessárias para começar a desenvolver aplicações em Java com Spring. Vamos abordar a instalação do JDK, do Spring Boot, e de uma IDE de sua escolha.

Instalação do Java JDK

O Java Development Kit (JDK) é o coração do desenvolvimento Java, e o Spring Boot é construído sobre ele. Para começar, você precisa instalar a versão mais recente do JDK. Aqui estão os passos gerais:

1. **Visite o site que oferece uma distribuição do JDK**. Você pode acessar o site oficial da Oracle, ou pode acessar um site que distribui *builds* gratuitas do JDK, tal como o site da Azul Zulu Builds of OpenJDK (https://www.azul.com/downloads).
2. **Selecione a última versão LTS do JDK**, lembrando que as versões LTS são as versões de longo período de suporte, tais como a 11, 17, 21 e assim por diante.
3. **Baixe e instale** o JDK para o seu sistema operacional, seguindo as instruções fornecidas no site.

Certifique-se de configurar a variável de ambiente `JAVA_HOME` para apontar para o diretório onde o JDK foi instalado e atualize o caminho do sistema para incluir o diretório `bin` do JDK.

Escolha da IDE

Para desenvolver com Java e Spring, você pode utilizar a IDE que preferir. Aqui estão algumas opções populares:

- **IntelliJ IDEA**: Conhecida por sua eficiência e integração com o ecossistema Spring. A versão Community é gratuita, enquanto a versão Ultimate oferece recursos adicionais voltados para o desenvolvimento web e empresarial.
- **Spring Tool Suite (STS)**: Uma versão do Eclipse customizada para desenvolvimento Spring. Inclui ferramentas específicas para o Spring e é totalmente gratuita.
- **Visual Studio Code (VS Code)**: Leve e versátil, com excelente suporte para Java através de extensões, como o Java Extension Pack, que inclui tudo necessário para começar a programar em Java.

Conteúdo

Capítulo 1

Componentes e injeção de dependência

1.1 Material de apoio do capítulo

Bem-vindos ao capítulo "Componentes e injeção de dependência".

Este curso conta com um material de apoio anexo, onde você terá acesso a alguns conteúdos suplementares.

Para este capítulo, vamos utilizar o seguinte arquivo contido no material de apoio anexo:

- 01 Componentes e injeção de dependência (slides).pdf

1.2 Sistema e componentes

Na arquitetura de software, um sistema eficaz é aquele que é bem estruturado e organizado em componentes bem definidos. Cada componente é uma parte do sistema que tem uma responsabilidade específica, e juntos, eles colaboram para formar um todo coeso e funcional. Nesta seção, exploraremos a importância de construir componentes coesos e desacoplados e como esses princípios contribuem para um sistema robusto e adaptável.

Componentes Coesos

A coesão refere-se à medida em que as responsabilidades de um único módulo ou componente são unicamente definidas e estritamente alinhadas. Componentes coesos possuem várias características distintas:

- **Responsabilidade Única**: Cada componente deve ter uma única responsabilidade ou função dentro do sistema. Isso simplifica o entendimento do sistema, facilita a manutenção e minimiza os impactos de mudanças, pois as alterações em um componente coeso raramente afetam outros componentes.
- **Clareza**: A clareza na definição de responsabilidades ajuda outros desenvolvedores a entender rapidamente o papel de cada componente no sistema, sem a necessidade de decifrar uma teia complexa de dependências e funcionalidades misturadas.

Desacoplamento de Componentes

Desacoplamento é o princípio de reduzir as dependências diretas entre diferentes componentes de um sistema. A adoção dessa prática traz diversos benefícios:

- **Independência**: Componentes desacoplados podem ser desenvolvidos, testados, e modificados de forma independente uns dos outros, o que aumenta a eficiência do desenvolvimento e reduz o risco de erros propagados.
- **Flexibilidade**: Menos dependências facilitam a reconfiguração e a adaptação do sistema para atender a novas necessidades ou integrar novas tecnologias.
- **Facilidade de substituição**: Substituir um componente se torna uma tarefa trivial quando ele é desacoplado, pois suas interfaces com o restante do sistema são bem definidas e limitadas.

Objetivos dos Princípios de Coesão e Desacoplamento

A aplicação dos princípios de coesão e desacoplamento não é meramente técnica, ela serve a objetivos estratégicos essenciais para o sucesso a longo prazo de um projeto de software:

- **Flexibilidade**: Sistemas projetados com alta coesão e baixo acoplamento são mais flexíveis, permitindo que a equipe de desenvolvimento responda mais rapidamente às mudanças no ambiente de negócios ou nas exigências do usuário.
- **Manutenção e Substituição Facilitada**: Facilita a manutenção e eventual substituição de componentes sem a necessidade de revisões profundas ou correções em cascata.
- **Reaproveitamento**: Componentes bem definidos e independentes são mais fáceis de serem reutilizados em diferentes partes do sistema ou até em novos projetos.

Em suma, ao projetar seu sistema em termos de componentes coesos e desacoplados, você está não apenas construindo um software mais robusto, mas também criando uma base que suportará o crescimento e a evolução contínua de suas aplicações. A utilização desses princípios no desenvolvimento com Spring Boot é crucial para aproveitar ao máximo as capacidades do framework e para garantir a longevidade e a escalabilidade dos sistemas que você desenvolve.

1.3 Exemplo didático

A implementação de programas pode variar significativamente dependendo de como os conceitos de design de software são aplicados. Nesta seção, exploraremos um exemplo prático que ilustra a diferença entre implementar uma solução sem o uso de componentes e uma solução que os utiliza, destacando as vantagens da abordagem baseada em componentes.

Enunciado do Problema

O problema proposto é desenvolver um programa que leia os dados de um funcionário (nome e salário bruto) e depois calcule e mostre o salário líquido, considerando descontos de impostos e previdência. As regras para os cálculos são as seguintes: 1) Imposto é 20% 2) Previdência é 10%

Exemplo:

```
Nome: Maria
Salário bruto: 4000.00
Salário líquido = 2800.00
```

Solução sem Componentes

A primeira abordagem é uma solução direta sem a utilização de componentes. O código a seguir realiza todas as operações dentro do método principal, incluindo a entrada de dados, o cálculo do salário líquido e a exibição dos resultados:

```
import java.util.Locale;
import java.util.Scanner;

public class Program {
    Locale.setDefault(Locale.US);

    Scanner sc = new Scanner(System.in);
    System.out.print("Nome: ");
    String name = sc.nextLine();
    System.out.print("Salario bruto: ");
    double grossSalary = sc.nextDouble();

    double netSalary = grossSalary * 0.7; // Desconto direto de 30% para
   impostos e previdência
    System.out.printf("Salario liquido = %.2f%n", netSalary);

    sc.close();
}
```

Solução com Componentes

A segunda abordagem utiliza uma arquitetura baseada em componentes, dividindo o problema em partes menores e mais gerenciáveis. A estrutura do código inclui classes separadas para o funcionário (`Employee`), serviços de cálculo de salário (`SalaryService`), imposto (`TaxService`) e previdência (`PensionService`):

```
// Arquivo principal que inicia o programa
package app;

import java.util.Locale;
import java.util.Scanner;

import entities.Employee;
import services.SalaryService;

public class Program {

    public static void main(String[] args) {
        Locale.setDefault(Locale.US);
        Scanner sc = new Scanner(System.in);
```

```
        System.out.print("Nome: ");
        String name = sc.nextLine();
        System.out.print("Salario bruto: ");
        double grossSalary = sc.nextDouble();

        Employee employee = new Employee(name, grossSalary);

        SalaryService salaryService = new SalaryService();

        double netSalary = salaryService.netSalary(employee);

        System.out.printf("Salario liquido = %.2f%n", netSalary);

        sc.close();
    }

}
```

```
package entities;

public class Employee {

    private String name;
    private Double grossSalary;

    public Employee() {
    }

    public Employee(String name, Double grossSalary) {
        this.name = name;
        this.grossSalary = grossSalary;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getGrossSalary() {
        return grossSalary;
    }

    public void setGrossSalary(Double grossSalary) {
        this.grossSalary = grossSalary;
    }
}
```

```
package services;

public class PensionService {
```

```
    public double discount(double amount) {
        return amount * 0.1;
    }
}
```

```
package services;

public class TaxService {

    public double tax(double amount) {
        return amount * 0.2;
    }
}
```

```
package services;

import entities.Employee;

public class SalaryService {

    // FORMA ERRADA
    private TaxService taxService = new TaxService();
    private PensionService pensionService = new PensionService();

    public double netSalary(Employee employee) {
        return employee.getGrossSalary() - taxService.tax(employee.
   getGrossSalary())
                - pensionService.discount(employee.getGrossSalary());
    }
}
```

Discussão

A abordagem baseada em componentes, embora mais extensa, traz uma série de benefícios:

- **Modularidade**: Cada parte do programa tem uma responsabilidade clara e definida.
- **Manutenabilidade**: É mais fácil de modificar ou corrigir bugs em um sistema modularizado, já que as mudanças em um componente geralmente não afetam outros.
- **Reaproveitamento**: Componentes como `TaxService` ou `PensionService` podem ser reutilizados em outras partes do sistema ou em outros projetos.

Considerações Finais

Por enquanto, as dependências nos componentes foram instanciadas diretamente dentro dos serviços, o que não é uma prática ideal. Nas próximas seções, introduziremos conceitos de

inversão de controle e **injeção de dependência** para melhorar ainda mais a estrutura do código, aumentando sua flexibilidade e a capacidade de manutenção.

1.4 Inversão de controle e injeção de dependência

A inversão de controle (IoC) e a injeção de dependência são princípios fundamentais no design de software moderno, especialmente no desenvolvimento de aplicações robustas e facilmente testáveis. Esses conceitos são cruciais para entender como frameworks como o Spring Boot gerenciam as dependências entre componentes de uma forma elegante e eficiente.

Inversão de Controle

Analogia do carro: Pense em um carro, onde o motor depende da bateria para funcionar. Embora essa dependência exista, a base de encaixe da bateria não está localizada dentro do motor. Isso ocorre porque, se fosse necessário trocar a bateria, não seria preciso desmontar o motor inteiro. Este design facilita a manutenção e substituição da bateria.

Generalizando: Essa analogia pode ser aplicada ao desenvolvimento de software. Se um componente A depende de um componente B, idealmente, A não deve controlar como essa dependência é gerenciada ou instanciada. Se A fosse responsável por criar B diretamente, qualquer mudança em B poderia exigir uma mudança correspondente em A, o que viola o princípio de baixo acoplamento e alta coesão. A ideia é "inverter o controle", transferindo a responsabilidade de gerenciar a dependência para fora do componente A, geralmente para um framework ou container.

Injeção de Dependência

Implementando a inversão de controle: Uma vez que a inversão de controle é estabelecida como princípio, surge a necessidade de "injetar" as dependências necessárias nos componentes. A injeção de dependência pode ser realizada de várias maneiras:

- **Construtor**: A dependência é fornecida através do construtor do componente, garantindo que o componente não possa ser criado sem suas dependências. Este método é amplamente utilizado por sua simplicidade e pela garantia de que as dependências são imutáveis após a criação do objeto.
- **Método set**: A dependência é injetada através de um método setter. Isso permite a alteração das dependências de um componente após sua criação. Embora flexível, este método pode levar a estados inconsistentes se o componente for usado antes de todas as suas dependências serem definidas.
- **Container de injeção de dependência**: Frameworks modernos, como o Spring, usam containers que automaticamente gerenciam as dependências. Esses containers cuidam da criação e fornecimento de todas as dependências necessárias, baseando-se em configurações definidas pelo desenvolvedor, como anotações no código ou configurações XML.

A inversão de controle e a injeção de dependência são essenciais para a construção de aplicações que são fáceis de testar, manter e evoluir. Eles permitem que os desenvolvedores se concentrem na lógica de negócios, enquanto o gerenciamento de dependências é tratado de forma transparente pelo framework. No contexto do Spring Boot, estas práticas não apenas facilitam a

manutenção e a expansão de aplicações, mas também promovem um design limpo e modular. Ao longo deste livro, você verá como esses princípios são aplicados para criar aplicações eficientes e elegantes.

1.5 Trocando a cependência sem abrir o componente pai

Um dos princípios mais poderosos em design de software é o Princípio Aberto-Fechado (Open-Closed Principle - OCP) de SOLID, que afirma que um software deve ser aberto para extensão, mas fechado para modificação. Isso significa que deveríamos poder estender um comportamento de um módulo sem alterar o código existente. A injeção de dependência, um dos pilares da inversão de controle, é uma técnica que nos ajuda a alcançar esse princípio de forma eficaz.

Exemplificação com SalaryService

Considere o seguinte cenário: inicialmente, temos um serviço genérico de cálculo de impostos (`TaxService`), mas as regras fiscais mudam ou precisamos adaptar o serviço para atender especificidades fiscais do Brasil. Para isso, criamos uma nova classe `BrazilTaxService` que estende `TaxService` e sobrescreve o método de cálculo de impostos para aplicar uma alíquota de 30%.

```
package services;

public class BrazilTaxService extends TaxService {

    @Override
    public double tax(double amount) {
        return amount * 0.3;  // Alíquota de imposto específica para o Brasil
    }
}
```

Implementação Flexível com Injeção de Dependência

A classe `SalaryService` é desenhada para ser flexível e aderir ao princípio OCP. Ela aceita qualquer implementação de `TaxService`, permitindo que o cálculo do imposto seja definido externamente. A classe não precisa ser alterada se decidirmos mudar a forma como o imposto é calculado; apenas passamos uma diferente implementação de `TaxService` para o construtor.

```
package services;

import entities.Employee;

public class SalaryService {

    private TaxService taxService;
    private PensionService pensionService;

    public SalaryService(TaxService taxService, PensionService pensionService) {
        this.taxService = taxService;
        this.pensionService = pensionService;
    }
```

```
    public double netSalary(Employee employee) {
        return employee.getGrossSalary() - taxService.tax(employee.
    getGrossSalary())
                - pensionService.discount(employee.getGrossSalary());
    }
}
```

Demonstração na Aplicação

Na aplicação principal, a troca do serviço de impostos é feita simplesmente passando uma instância de `BrazilTaxService` ao invés de uma instância de `TaxService` para o `SalaryService`. Isso ilustra como a inversão de controle e a injeção de dependência permitem a substituição de componentes sem necessidade de alterar os componentes que os utilizam.

```
package app;

import java.util.Locale;
import java.util.Scanner;

import entities.Employee;
import services.BrazilTaxService;
import services.PensionService;
import services.SalaryService;
import services.TaxService;

public class Program {

    public static void main(String[] args) {

        Locale.setDefault(Locale.US);

        Scanner sc = new Scanner(System.in);
        System.out.print("Nome: ");
        String name = sc.nextLine();
        System.out.print("Salario bruto: ");
        double grossSalary = sc.nextDouble();

        Employee employee = new Employee(name, grossSalary);

        TaxService taxService = new BrazilTaxService();
        PensionService pensionService = new PensionService();
        SalaryService salaryService = new SalaryService(taxService,
    pensionService);

        double netSalary = salaryService.netSalary(employee);

        System.out.printf("Salario liquido = %.2f%n", netSalary);

        sc.close();
    }

}
```

Essa flexibilidade demonstra o poder da inversão de controle combinada com a injeção de dependência. `SalaryService` se mantém inalterado, mesmo quando a implementação concreta de `TaxService` muda, mantendo o sistema aderente ao princípio OCP e facilitando a manutenção e evolução do software.

1.6 Frameworks

No mundo do desenvolvimento de software, os frameworks desempenham um papel crucial ao oferecer uma estrutura ou uma "armação" robusta que simplifica e acelera a criação de aplicações complexas. Essas ferramentas não só fornecem uma fundação sobre a qual os desenvolvedores podem construir, mas também gerenciam aspectos repetitivos e complexos do desenvolvimento, permitindo que os desenvolvedores se concentrem na lógica de negócios específica de seus projetos.

O que é um Framework?

Literalmente traduzido como "estrutura" ou "estrutura de trabalho", um framework em programação é um conjunto de ferramentas que oferece uma infraestrutura para desenvolver sistemas de maneira produtiva. Essa infraestrutura geralmente inclui:

- **Injeção de Dependência**: Facilita o gerenciamento de dependências entre os componentes, promovendo um acoplamento mais fraco e uma maior modularidade.
- **Gerenciamento de Transações**: No back end, os frameworks podem gerenciar automaticamente o início e o fim de transações, garantindo consistência e integridade dos dados.
- **Ciclo de Vida e Escopo de Componentes**: Controla como e quando os componentes são criados, destruídos e reutilizados, otimizando o uso de recursos e a performance da aplicação.
- **Configurações**: Permite definir e modificar configurações de maneira centralizada, sem alterar o código base.
- **Integrações**: Facilita a integração com outras aplicações, serviços ou APIs, expandindo as funcionalidades da aplicação sem grandes complicações.
- **Outras funcionalidades**: Pode incluir segurança, mapeamento de dados, rotinas de teste, entre outros.

Trabalhando com Injeção de Dependência em Frameworks

Para aproveitar plenamente as capacidades de um framework, especialmente no que tange à injeção de dependência, dois passos fundamentais devem ser seguidos:

1. **Registrar os Componentes**: Definir quais classes ou componentes estão disponíveis para uso pelo framework. Isso geralmente envolve anotar classes ou registrá-las em algum tipo de arquivo de configuração.
2. **Indicar as Dependências**: Especificar quais componentes dependem de quais outros componentes. Isso pode ser feito através de anotações no código, injeção via construtor, métodos setters ou arquivos de configuração.

Uma vez realizados estes dois passos, o framework assumirá a responsabilidade de:

- **Instanciar Componentes**: Cria instâncias dos componentes conforme necessário, gerenciando sua criação de forma eficiente.
- **Resolver Dependências**: Automagicamente conecta os componentes com as dependências requeridas, garantindo que todas as necessidades sejam satisfeitas antes de um componente ser utilizado.
- **Reaproveitar Componentes**: Mantém instâncias de componentes que podem ser reutilizados, reduzindo a necessidade de criação constante de novos objetos e, assim, melhorando a performance.
- **Gerenciar Escopo e Ciclo de Vida**: Controla a duração e a visibilidade dos componentes, decidindo se são de longa duração, de sessão, de requisição ou de aplicação, além de gerenciar quando são criados e destruídos.

Frameworks são fundamentais no desenvolvimento moderno por encapsularem práticas de codificação complexas e repetitivas, permitindo aos desenvolvedores focar na criação de funcionalidades únicas para suas aplicações. Ao abstrair a complexidade de tarefas como gerenciamento de dependências e ciclo de vida de componentes, eles não só aumentam a produtividade, mas também promovem um código mais limpo, testável e manutenível.

1.7 Criando projeto Spring Boot

Agora que já aprendemos como resolver nosso problema exemplo usando um projeto Java com componentes, e também aprendemos o que é um framework, nesta e nas próximas seções vamos refazer a solução do nosso projeto, porém agora utilizando o framework Spring Boot.

Spring Boot é uma extensão do framework Spring que simplifica o processo de configuração e desenvolvimento de aplicações em Java. Uma das maneiras mais rápidas e eficientes de iniciar um novo projeto Spring Boot é através do Spring Initializr, uma ferramenta online que permite gerar a estrutura básica de um projeto com as configurações desejadas. Vamos detalhar como usar essa ferramenta para criar seu primeiro projeto Spring Boot.

Utilizando o Spring Initializr

Para começar, siga estes passos simples para configurar e gerar seu projeto:

1. **Acesse o Spring Initializr**: Vá para start.spring.io.
2. **Configurações do Projeto**:
    - **Project**: Escolha 'Maven Project'. Maven é um poderoso gerenciador de projetos e dependências para Java que facilita a construção e o gerenciamento de qualquer projeto Java.
    - **Language**: Selecione 'Java'. Escolha a última versão LTS do Java disponível para garantir suporte de longo prazo.
    - **Spring Boot**: Escolha a última versão estável do Spring Boot para aproveitar as mais recentes melhorias e funcionalidades.
3. **Metadados do Projeto**:
    - **Group**: Digite 'com.devsuperior'. Isso define o identificador do grupo do seu projeto, que é comummente utilizado para organizar projetos em grandes corporações ou repositórios.

- **Artifact**: Digite 'aula'. Este é o nome do seu projeto e do artefato que será construído.
- **Name**: O nome será preenchido automaticamente com base no artefato. Você pode deixá-lo como está ou personalizá-lo.
- **Description**: Forneça uma breve descrição do projeto.
- **Package name**: Será automaticamente preenchido com base no grupo e no artefato, mas você pode modificar se necessário.
- **Packaging**: Deixe como 'Jar', que é o padrão para projetos Spring Boot.
- **Java**: Selecione a versão do Java que você escolheu anteriormente.

4. **Dependências**:
   - Adicione a dependência 'Spring Web'. Esta dependência permite que o Spring Boot sirva conteúdo web e RESTful APIs.
5. **Gere o Projeto**: Clique em 'Generate' para baixar um arquivo zip contendo o projeto.

Estrutura do projeto criado

Quando você importa o projeto, encontrará a seguinte estrutura de arquivos e diretórios:

- `src/`: Contém duas subpastas, `main/` e `test/`. A pasta `main/` contém o código da aplicação, enquanto `test/` é usada para os testes.
  - `java/`: Dentro de `main/`, contém os arquivos de código fonte Java.
    - `com/devsuperior/aula/`: Contém uma classe principal que inicia a aplicação.
  - `resources/`: Esta pasta contém recursos estáticos como propriedades de configuração (`application.properties` ou `application.yml`).
- `pom.xml`: Arquivo de configuração do Maven que contém as dependências do projeto e outras configurações.

Importando o Projeto no STS e IntelliJ

Após criar seu projeto Spring Boot utilizando o Spring Initializr, o próximo passo é importá-lo em uma IDE que você escolher para continuar o desenvolvimento. Spring Tool Suite (STS) e IntelliJ IDEA são duas das IDEs mais populares para desenvolvimento Java com Spring Boot devido ao seu robusto suporte para projetos Spring. Aqui estão os passos detalhados para importar seu projeto em cada uma dessas IDEs.

Spring Tool Suite (STS)

Spring Tool Suite é uma IDE baseada em Eclipse otimizada para desenvolvimento Spring. Siga estes passos para importar seu projeto Spring Boot:

1. **Abrir o STS**: Inicie o Spring Tool Suite.
2. **Importar Projeto**:
   - Vá até o menu `File` e selecione `Import...`.
   - Na janela de importação, expanda `Maven` e selecione `Existing Maven Projects`.
   - Clique em `Next`.
3. **Selecionar o Diretório do Projeto**:

- Clique em `Browse` e navegue até o diretório onde você extraiu o projeto Spring Boot.
- Selecione o diretório e clique em `Finish`. O STS detectará o arquivo `pom.xml` e configurará o projeto automaticamente.

4. **Verificar o Projeto**:
    - Após a importação, o projeto aparecerá no `Package Explorer`.
    - Expanda o projeto para verificar se a estrutura de diretórios e arquivos está corretamente importada.

IntelliJ IDEA

IntelliJ IDEA é conhecida por sua poderosa funcionalidade e suporte integrado para projetos Spring Boot. Siga estes passos para importar seu projeto:

1. **Abrir o IntelliJ IDEA**:
    - Inicie o IntelliJ IDEA.
2. **Importar Projeto**:
    - Na tela inicial, clique em `Open`.
    - Alternativamente, se você já tem o IntelliJ aberto, vá até o menu `File` e selecione `Open`.
3. **Selecionar o Projeto**:
    - Navegue até o diretório onde o projeto Spring Boot foi extraído.
    - Selecione o diretório do projeto ou o arquivo `pom.xml` e clique em `OK`.
4. **Configurar Projeto**:
    - O IntelliJ irá abrir uma janela perguntando se você deseja abrir o diretório como um projeto, anexar à janela atual ou abrir em uma nova janela. Escolha `Open as Project`.
    - O IntelliJ automaticamente configura o projeto baseado no `pom.xml`, identificando-o como um projeto Maven.
5. **Verificar e Configurar JDK**:
    - Após abrir o projeto, vá para `File` > `Project Structure` > `Project`.
    - Verifique se o projeto está configurado com a JDK correta (a mesma versão que você selecionou no Spring Initializr).
    - Se a JDK não estiver configurada, você pode adicionar uma JDK através desta janela.

1.8 Importando o projeto no STS e Intellij

O processo de importação de um projeto Spring Boot nas IDEs mais populares, como Spring Tools for Eclipse e IntelliJ IDEA, é uma etapa fundamental para começar o desenvolvimento de uma aplicação. Vamos abordar os passos detalhados para realizar esta tarefa em ambas as plataformas.

Spring Tools for Eclipse (STS)

Spring Tools for Eclipse, também conhecido como STS, é uma extensão do Eclipse otimizada para o desenvolvimento de aplicações Spring. Siga estes passos para importar seu projeto Spring Boot:

1. **Abrir STS**:

- Inicie o Spring Tools Suite.

2. **Importar Projeto**:
   - No menu, vá em `File` > `Import...`.
   - Na janela de importação, escolha `Existing Maven Projects` sob o diretório `Maven`.
   - Clique em `Next`.
3. **Selecionar o Diretório do Projeto**:
   - Clique em `Browse` e encontre o diretório onde seu projeto Spring Boot foi descompactado.
   - Selecione o diretório e certifique-se de que o `pom.xml` está selecionado.
   - Clique em `Finish`. O STS agora deverá resolver as dependências do Maven e configurar o projeto automaticamente.
4. **Verificar o Projeto**:
   - O projeto agora aparecerá na `Project Explorer` ou `Package Explorer`.
   - Expanda o projeto para ver a estrutura de arquivos, incluindo pacotes e recursos.

IntelliJ IDEA

IntelliJ IDEA é uma IDE desenvolvida pela JetBrains que suporta uma integração extensa com o ecossistema Spring, incluindo Spring Boot. Siga estes passos para importar um projeto Spring Boot:

1. **Abrir IntelliJ IDEA**:
   - Inicie o IntelliJ IDEA. Se for a primeira inicialização, você pode importar o projeto diretamente na tela de boas-vindas. Caso contrário, você pode selecionar `File` > `Open` no menu principal.
2. **Selecionar o Projeto**:
   - Navegue até o diretório onde o projeto Spring Boot foi descompactado.
   - Você pode selecionar o diretório do projeto ou diretamente o arquivo `pom.xml`. Clique em `OK`.
3. **Configurar o Projeto**:
   - IntelliJ reconhecerá automaticamente que se trata de um projeto Maven. Se for solicitado, configure a JDK para o projeto, selecionando a versão do Java que você usou no Spring Initializr.
   - O IntelliJ irá configurar o projeto e resolver as dependências.
4. **Verificar o Projeto**:
   - Uma vez que o projeto esteja aberto, explore a estrutura de arquivos no painel à esquerda.
   - Abra algumas classes ou recursos para garantir que tudo foi importado e está sendo exibido corretamente.

1.9 Implementando o projeto Spring Boot

Ao construir aplicações com Spring Boot, a organização e a estruturação correta do código são essenciais para aproveitar ao máximo os recursos do framework. Nesta seção, vamos implementar a solução para calcular o salário líquido de um funcionário, utilizando componentes gerenciados pelo Spring. Este exemplo prático demonstrará como registrar e injetar componentes usando as anotações `@Component`, `@Service`, e `@Autowired`.

Estrutura do Projeto

Primeiro, vamos organizar o nosso projeto em pacotes e classes, utilizando as convenções e práticas recomendadas pelo Spring Boot.

Classe `Employee`

Esta classe representa a entidade `Employee`. Ela armazena informações como nome e salário bruto do funcionário.

```
package com.devsuperior.aula.entities;

public class Employee {

    private String name;
    private Double grossSalary;

    public Employee() {
    }

    public Employee(String name, Double grossSalary) {
        this.name = name;
        this.grossSalary = grossSalary;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getGrossSalary() {
        return grossSalary;
    }

    public void setGrossSalary(Double grossSalary) {
        this.grossSalary = grossSalary;
    }
}
```

Serviço `TaxService`

Este serviço é responsável pelo cálculo do imposto. A anotação `@Service` indica que o Spring deve gerenciar esta classe como um componente de serviço.

```
package com.devsuperior.aula.service;

import org.springframework.stereotype.Service;

@Service
public class TaxService {
```

```
    public double tax(double amount) {
        return amount * 0.2; // Imposto de 20%
    }
}
```

Serviço PensionService

Semelhante ao `TaxService`, este serviço calcula a contribuição da previdência.

```
package com.devsuperior.aula.service;

import org.springframework.stereotype.Service;

@Service
public class PensionService {

    public double discount(double amount) {
        return amount * 0.1; // Desconto de 10% para a previdência
    }
}
```

Serviço SalaryService

Este é o serviço central que utiliza `TaxService` e `PensionService` para calcular o salário líquido. A injeção de dependências é feita via anotação `@Autowired`, permitindo que o Spring Boot cuide da criação e gerenciamento desses componentes.

```
package com.devsuperior.aula.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.devsuperior.aula.entities.Employee;

@Service
public class SalaryService {

    @Autowired
    private TaxService taxService;

    @Autowired
    private PensionService pensionService;

    public double netSalary(Employee employee) {
        return employee.getGrossSalary() - taxService.tax(employee.
   getGrossSalary())
                - pensionService.discount(employee.getGrossSalary());
    }
}
```

Classe de Aplicação `AulaApplication`

Finalmente, a classe de aplicação onde o Spring Boot é inicializado e onde é realizado o teste do cálculo do salário líquido.

```
package com.devsuperior.aula;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.devsuperior.aula.entities.Employee;
import com.devsuperior.aula.service.SalaryService;

@SpringBootApplication
public class AulaApplication implements CommandLineRunner {

    @Autowired
    private SalaryService salaryService;

    public static void main(String[] args) {
        SpringApplication.run(AulaApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        Employee employee = new Employee("Maria", 4000.0);
        System.out.println("Salario líquido = " + salaryService.netSalary(
   employee));
    }
}
```

Execução do projeto

A execução do projeto dentro da IDE é bastante simples. Geralmente você vai contar com uma opção menu similar a `run` a partir do clique com o botão direito sobre o nome do projeto.

Particularmente na IDE STS, por exemplo, há também uma aba "Boot Dashboard" na interface gráfica da IDE, onde pode-se clicar com o botão direito sobre o nome do projeto e escolher `(Re)start` para executar o projeto. Esta opção é recomendada, pois se o projeto já estiver em execução, a execução será automaticamente interrompida e reiniciada.

Uma vez executado o projeto, você pode acompanhar o log de execução na aba `Console` da sua IDE. Nesta aba o projeto vai imprimindo informações sobre a execução, e você deverá visualizar lá o valor `2800.0`, que é o resultado do processamento do salário líquido do funcionário.

Registro e Injeção de Componentes

Como você pode observar, cada serviço é registrado no contexto do Spring utilizando `@Service`. Isso permite que o Spring gerencie esses componentes, incluindo sua criação, injeção e destruição conforme necessário. A injeção de dependências através do `@Autowired` facilita a manutenção

do código, pois desacopla a criação de objetos do uso dos mesmos, seguindo os princípios de inversão de controle e injeção de dependência discutidos anteriormente.

Vale ressaltar que poderíamos ter utilizado a *annotation* `@Component` ao invés da `@Service`, pois ambas tem o mesmo efeito, com a diferença que `@Service` apenas possui um nome mais sugestivo para o tipo de classe que estamos registrando como componentes, que são serviços do sistema.

Vale ressaltar também que a injeção dos componentes poderia ter sido feita por meio de **construtores** ao invés do `@Autowired`. O framework Spring aceita ambas as formas. Um exemplo de injeção de dependência com construtor é mostrado a seguir na implementação alternativa da classe `SalaryService`.

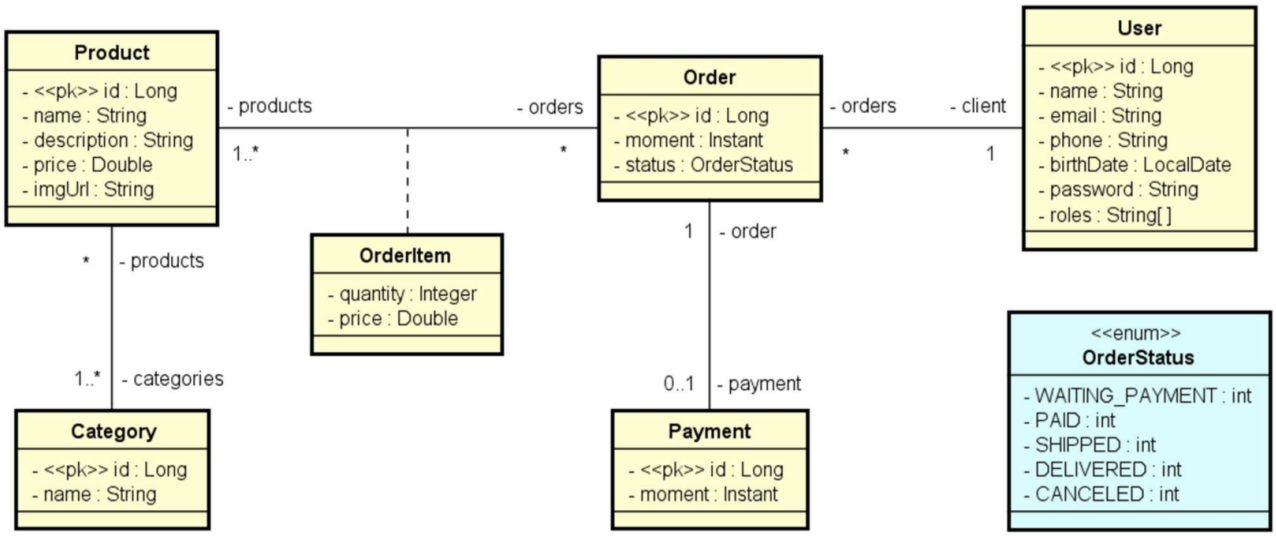
```
package com.devsuperior.aula.service;

import org.springframework.stereotype.Service;

import com.devsuperior.aula.entities.Employee;

@Service
public class SalaryService {

    private TaxService taxService;

    private PensionService pensionService;

    public SalaryService(TaxService taxService, PensionService pensionService) {
        this.taxService = taxService;
        this.pensionService = pensionService;
    }

    public double netSalary(Employee employee) {
        return employee.getGrossSalary() - taxService.tax(employee.
   getGrossSalary())
                - pensionService.discount(employee.getGrossSalary());
    }
}
```

Capítulo 2

Modelo de Domínio e ORM

2.1 Apresentação do capítulo

Bem-vindos ao capítulo sobre modelagem de domínio, uma etapa crucial no desenvolvimento de aplicações robustas com Java e Spring. Neste capítulo, vamos explorar como implementar o modelo de domínio do projeto DSCommerce, utilizando algumas das melhores ferramentas disponíveis no ecossistema Spring para otimizar e simplificar o trabalho com dados em aplicações Java.



Projeto do modelo de domínio do sistema DSCommerce.

Nossa jornada pela modelagem de domínio será conduzida principalmente pelo uso do **Java Persistence API (JPA)** e **Spring Data JPA**. O JPA é uma poderosa especificação padrão para mapeamento objeto-relacional, que facilita a integração de aplicações Java com bancos de dados relacionais. Com ele, podemos definir como objetos em nossa aplicação correspondem às tabelas em um banco de dados, manipulando esses dados de forma mais intuitiva e orientada a objetos.

Por sua vez, o Spring Data JPA é um subframework do ecossistema Spring que simplifica ainda mais a implementação do JPA, reduzindo a quantidade de código boilerplate necessário e automatizando muitas das tarefas repetitivas associadas ao gerenciamento de dados. Com

Spring Data JPA, você pode focar mais na lógica do negócio enquanto o framework cuida da complexidade do acesso a dados.

Durante este capítulo, adotaremos a abordagem **code first** para o mapeamento objeto-relacional (ORM). Isso significa que começaremos escrevendo o código das nossas classes de domínio em Java, equipadas com annotations do JPA que definem como essas classes e seus campos se relacionam com as tabelas e colunas de um banco de dados. A partir dessas definições, a estrutura do banco de dados será gerada automaticamente, permitindo uma integração fluida e direta entre o código da aplicação e o banco de dados.

Utilizar essa abordagem nos permite modelar o domínio de nossa aplicação de maneira clara e coesa, refletindo as necessidades reais do negócio diretamente no código. As annotations do JPA servirão como pontes, traduzindo nosso modelo de objetos para o modelo relacional de maneira eficiente e controlada.

Este capítulo é uma oportunidade fantástica para aprofundar seus conhecimentos e habilidades no trabalho com banco de dados em aplicações Java. Ao dominar a modelagem de domínio com JPA e Spring Data JPA, você estará equipado para construir aplicações mais robustas, escaláveis e fáceis de manter. Prepare-se para transformar seu modo de interagir com dados em suas aplicações, tornando o processo mais intuitivo, estruturado e produtivo. Vamos embarcar juntos nesta jornada de aprendizado e descoberta, e transformar os desafios em oportunidades para crescer como desenvolvedores!

2.2 Material de apoio do capítulo

Para este capítulo, vamos utilizar os seguintes arquivos contidos no material de apoio anexo:

- 02 Modelo de domínio e ORM (slides).pdf
- 02 DESAFIO Modelo de domínio e ORM.pdf
- Documento de requisitos DSCommerce.pdf
- Projeto DSCommerce ao final do capítulo 2

2.3 Antes de prosseguir com o capítulo

Antes de prosseguirmos com o conteúdo deste capítulo, é importante compreender como uma aplicação Java com Spring se comunica com o banco de dados relacional.

A interação de uma aplicação Java com um banco de dados é uma parte crucial do desenvolvimento de sistemas. Esta interação pode ser complexa e multifacetada, envolvendo várias camadas de abstração para facilitar o desenvolvimento e melhorar a manutenibilidade do código. Vamos explorar como essas camadas interagem, começando pelo JDBC, passando pelo JPA, e finalmente chegando ao Spring Data JPA.

JDBC (Java Database Connectivity)

JDBC é uma API padrão em Java projetada para conectar-se a bancos de dados. É uma das bibliotecas mais antigas e mais utilizadas para acessar bases de dados relacionais. Com JDBC, desenvolvedores podem executar operações SQL diretamente a partir do código Java, como consultas, atualizações e inserções de dados. A API JDBC fornece um meio para estabelecer uma conexão com o banco, executar comandos SQL e gerenciar respostas. Essa biblioteca é

fundamental para aplicações que necessitam de interação direta com o banco de dados, sem o uso de abstrações adicionais.

JPA (Java Persistence API)

O JPA é uma especificação da linguagem Java que fornece um framework para mapeamento relacional de objetos, conhecido como ORM (Object-Relational Mapping). O JPA permite que desenvolvedores mapeiem classes Java a tabelas de banco de dados de forma a gerenciar as operações de banco de dados em um alto nível de abstração. Utilizando anotações ou arquivos XML, os desenvolvedores podem definir como as entidades Java correspondem às tabelas do banco, como as operações de CRUD são executadas, e como as relações entre entidades são gerenciadas. O JPA transforma as operações em entidades Java em comandos SQL adequados, abstraindo a complexidade do JDBC.

Spring Data JPA

Spring Data JPA é um subprojeto do ecossistema Spring que faz uso da especificação JPA para simplificar ainda mais o desenvolvimento de aplicações que acessam dados. Ele fornece uma camada de abstração sobre o JPA, oferecendo uma maneira mais simples e poderosa de integrar a persistência de dados na sua aplicação. Com Spring Data JPA, é possível criar repositórios de dados com operações pré-definidas e customizáveis sem necessidade de implementar o código repetitivo usualmente associado com operações de banco de dados. O framework também suporta a geração automática de consultas baseadas em nomes de métodos, critérios de busca e muito mais.

Conteúdos Complementares Recomendados no YouTube

Para aprofundar seu entendimento e habilidade em trabalhar com banco de dados em Java, recomendamos os seguintes vídeos:

1. **Revisão de Álgebra Relacional e SQL**:
   - **Objetivo**: Relembrar as operações básicas com SQL.
   - https://www.youtube.com/watch?v=GHpE5xOxXXI
2. **Super revisão de OO e SQL com Java e JDBC**:
   - **Objetivo**: Compreender na prática como é consultar os dados de um banco de dados somente com Java e JDBC, sem utilizar uma ferramenta ORM.
   - https://www.youtube.com/watch?v=xC_yKw3MYX4
3. **Nivelamento ORM - JPA e Hibernate**:
   - **Objetivo**: Ter uma introdução teórica e prática sobre ORM com JPA, antes de ir direto para o Spring com o Spring Data JPA.
   - https://www.youtube.com/watch?v=CAP1IPgeJkw

Esses recursos são essenciais para entender as bases tecnológicas sobre as quais construímos sistemas modernos e eficientes. Eles fornecem um fundamento sólido que será extremamente útil ao avançarmos no desenvolvimento de aplicações com Spring Data JPA.

2.4 Documento de requisitos do sistema DSCommerce

Introdução ao Documento de Requisitos

Ao iniciar o desenvolvimento de qualquer sistema, é fundamental ter uma compreensão clara dos requisitos. O documento de requisitos do sistema DSCommerce, o qual pode ser acessado no material anexo, serve como um mapa para desenvolvedores e stakeholders, delineando funcionalidades, regras de negócio, e a interface do usuário. Este documento é essencial para garantir que todos os envolvidos no projeto compartilhem uma visão comum do que precisa ser construído.

Estrutura do Documento

O documento de requisitos do DSCommerce está organizado em várias seções principais, cada uma abordando diferentes aspectos do sistema:

1. **Premissas**: Esta seção estabelece as bases do sistema, explicando o contexto e os objetivos por trás do desenvolvimento. Ela especifica que o DSCommerce deve ser simples, porém abrangente, permitindo a aplicação de conhecimentos fundamentais em um contexto prático.
2. **Visão Geral do Sistema**: Fornece uma descrição concisa do que o sistema deve fazer, detalhando as funcionalidades principais, como manutenção de cadastro de usuários, produtos, categorias, e a funcionalidade do carrinho de compras. Ela descreve como os usuários interagem com o sistema, diferenciando os acessos e permissões entre clientes e administradores.
3. **Protótipos de Tela**: Embora não detalhado textualmente aqui, esta seção do documento original inclui links para protótipos visuais que ajudam a ilustrar como as interfaces do usuário devem aparecer e funcionar. Protótipos são cruciais para alinhar as expectativas visuais e funcionais entre desenvolvedores e stakeholders.
4. **Modelo Conceitual**: Descreve a estrutura de dados do sistema, incluindo entidades importantes como produtos, pedidos, e usuários, e suas relações. Este modelo ajuda os desenvolvedores a entender como organizar e relacionar os dados no banco de dados.
5. **Casos de Uso (Visão Geral e Detalhamento)**: Detalha cada operação que os usuários podem realizar no sistema, dividido em visões gerais e cenários de uso detalhados. Cada caso de uso descreve as ações dos usuários, as respostas do sistema, e as condições sob as quais essas ações ocorrem. Esta seção é vital para compreender como o sistema deve responder às interações dos usuários. ### A Importância do Documento de Requisitos

Compreender este documento é essencial para qualquer pessoa envolvida no desenvolvimento do DSCommerce, seja implementando diretamente o código ou realizando testes e manutenções. Ele serve como uma diretriz clara para o que precisa ser desenvolvido e como cada parte do sistema interage com as outras.

Ao modelar o domínio usando Java e Spring, utilizaremos JPA e Spring Data JPA para facilitar o mapeamento objeto-relacional. Com essas ferramentas, transformaremos as especificações dos modelos de dados e casos de uso em estruturas de código que o Spring pode gerenciar, usando uma abordagem code first que nos permite criar o banco de dados relacional diretamente a partir das classes Java.

À medida que você se aprofundar neste capítulo e começar a implementar o sistema DSCommerce, mantenha este documento de requisitos sempre acessível. Ele será seu guia através do processo de desenvolvimento, garantindo que todos os aspectos do sistema sejam abordados e implementados conforme esperado. Encorajamos você a mergulhar nos detalhes e a utilizar este documento como uma ferramenta para construir um sistema robusto e eficiente. Vamos transformar esses requisitos em realidade com precisão e criatividade!

2.5 Baixando o projeto pronto

Neste momento, caso você tenha alguma experiência com o Spring, e caso você prefira baixar o projeto pronto ao final deste capítulo no material anexo, ao invés de construir o projeto do zero seguindo nosso passo a passo, você pode fazer isso.

Basta acessar no material anexo o projeto DSCommerce ao final do capítulo 2, e importar na sua IDE favorita. Desta forma, você pode apenas passar pelo conteúdo com mais rapidez para entender como nós construímos o projeto.

Entretanto, faço a advertência de que, caso você ainda esteja aprendendo a programar com Spring, é importante que você construa o projeto do zero seguindo nosso passo a passo, pois com isso você vai construir e sedimentar seu conhecimento. Lembre-se: não pule etapas.

2.6 Criando o projeto DSCommerce

Nesta seção, vamos criar o projeto DSCommerce, que será um projeto Spring Boot. Vamos utilizar o Spring Initializr, configurá-lo com as dependências necessárias e, em seguida, importá-lo e executá-lo na IDE IntelliJ IDEA. Ao final, testaremos a aplicação para garantir que tudo está funcionando conforme o esperado.

Passo 1: Criar o Projeto no Spring Initializr

1. **Acesse o Spring Initializr**:
   - Vá para start.spring.io.
2. **Configure seu Projeto**:
   - **Project Type**: Escolha `Maven Project`.
   - **Language**: Selecione `Java`.
   - **Spring Boot**: Escolha a última versão estável disponível.
   - **Project Metadata**:
     - **Group**: `com.devsuperior`
     - **Artifact**: `dscommerce`
     - **Name**: Pode deixar como `dscommerce` ou alterar conforme sua preferência.
     - **Description**: Descreva brevemente o projeto, por exemplo, "Demo project for Spring Boot".
     - **Package name**: Será automaticamente preenchido como `com.devsuperior.dscommerce`.
3. **Adicione as Dependências**:
   - Clique em `Add Dependencies` e procure pelas seguintes:
     - `Spring Web`: Para construir serviços web, incluindo RESTful applications.

- `Spring Data JPA`: Para integração com bancos de dados usando Java Persistence API.
- `H2 Database`: Um banco de dados em memória, ideal para desenvolvimento e testes.

4. **Gerar o Projeto**:
   - Após adicionar as dependências, clique em `Generate` para baixar o projeto como um arquivo ZIP.
5. **Descompacte o Arquivo**:
   - Salve e descompacte o arquivo em uma pasta de sua escolha no seu computador.

Passo 2: Importar o Projeto no IntelliJ IDEA

1. **Abra o IntelliJ IDEA**:
   - Se for a primeira inicialização, selecione `Open or Import`. Caso já tenha o IntelliJ aberto, vá para `File > Open`.
2. **Encontre e Selecione o Projeto**:
   - Navegue até a pasta onde você descompactou o projeto e selecione o diretório `dscommerce`.
3. **Abrir como Projeto**:
   - Confirme que deseja abrir o diretório como um projeto. O IntelliJ começará a indexar e configurar o projeto baseado no `pom.xml`.
4. **Verifique as Configurações do Projeto**:
   - Certifique-se de que a versão da JDK está configurada corretamente para Java 21 em `File > Project Structure > Project`.

Passo 3: Executar o Projeto

1. **Encontrar a Classe Principal**:
   - Navegue até `src/main/java/com/devsuperior/dscommerce/DscommerceApplication.java`.
2. **Executar a Aplicação**:
   - Clique com o botão direito sobre o arquivo e selecione `Run 'DscommerceApplication'`. O IntelliJ irá compilar e executar o projeto.

Passo 4: Testar a Aplicação

1. **Acessar a Aplicação**:
   - Abra um navegador e acesse http://localhost:8080. Como ainda não definimos nenhum endpoint específico, você provavelmente verá uma página de erro padrão do Spring ou uma página em branco.
2. **Verificar Logs**:
   - No console do IntelliJ, verifique os logs para garantir que não há erros e que o servidor iniciou corretamente.

2.7 Entidade User, banco H2

Nesta seção, vamos criar uma classe de entidade chamada `User` dentro de um subpacote `entities` no nosso projeto Spring Boot. Esta entidade será mapeada para uma tabela no banco de dados usando a Java Persistence API (JPA).

Estrutura do Código da Classe User

```
package com.devsuperior.dscommerce.entities;

import jakarta.persistence.*;

import java.time.LocalDate;
import java.util.Objects;

@Entity
@Table(name = "tb_user")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @Column(unique = true)
    private String email;
    private String phone;
    private LocalDate birthDate;
    private String password;

    public User() {
    }

    public User(Long id, String name, String email, String phone, LocalDate
   birthDate, String password) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.phone = phone;
        this.birthDate = birthDate;
        this.password = password;
    }

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }
```

```
    public void setEmail(String email) {
        this.email = email;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public LocalDate getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(LocalDate birthDate) {
        this.birthDate = birthDate;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        User user = (User) o;

        return Objects.equals(id, user.id);
    }

    @Override
    public int hashCode() {
        return id != null ? id.hashCode() : 0;
    }
}
```

Anotações e Campos

- **@Entity**: Declara que a classe é uma entidade JPA, que será mapeada para uma tabela no banco de dados.
- **@Table(name = "tb_user")**: Especifica o nome da tabela no banco de dados.
- **@Id**: Indica que o campo é a chave primária da tabela.
- **@GeneratedValue(strategy = GenerationType.IDENTITY)**: Configura a geração automática dos valores da chave primária, delegando ao banco de dados a responsabilidade de gerar o identificador único.
- **@Column(unique = true)**: Define que a coluna `email` deve ser única no banco de

dados.
- **LocalDate birthDate**: Utiliza a classe `LocalDate` para armazenar datas sem informação de tempo.

Configurando o Banco de Dados H2

Para integrar o banco de dados H2 ao projeto, é necessário configurar alguns parâmetros nos arquivos `.properties` do Spring Boot.

Arquivo `application.properties`:

```
spring.profiles.active=test
spring.jpa.open-in-view=false
```

- **spring.profiles.active=test**: Ativa o perfil `test`, que vai buscar as configurações específicas do arquivo `application-test.properties`.
- **spring.jpa.open-in-view=false**: Desabilita a estratégia Open Session in View para evitar problemas de performance relacionados à persistência.

Arquivo `application-test.properties`:

```
# Dados de conexão com o banco H2
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=

# Configuração do cliente web do banco H2
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# Configuração para mostrar o SQL no console
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

- **spring.datasource.url**: Define a URL de conexão para o banco de dados em memória H2.
- **spring.h2.console.enabled=true**: Habilita o console web do H2, permitindo acessar o banco de dados através de um navegador.
- **spring.h2.console.path**: Define o caminho para acessar o console do H2.
- **spring.jpa.show-sql=true**: Ativa a exibição das consultas SQL no console, útil para debugging.
- **spring.jpa.properties.hibernate.format_sql=true**: Formata o SQL exibido no console para facilitar a leitura.

Executando a Aplicação e Acessando o H2 Console

Para executar a aplicação:

1. **Abra o IntelliJ IDEA** e localize a classe principal da aplicação (`DscommerceApplication`).
2. **Execute a aplicação** clicando com o botão direito do mouse sobre o arquivo e selecionando `Run 'DscommerceApplication'`.

Para acessar o H2 Console:

1. **Abra um navegador** e digite o URL `http://localhost:8080/h2-console`.
2. **Configure o JDBC URL** como `jdbc:h2:mem:testdb`, deixe o usuário como `sa` e a senha em branco.
3. **Conecte-se** para gerenciar o banco de dados através do console web.

Assim, nesta seção, criamos nossa primeira entidade do sistema, a classe `User`, e configuramos o mapeamento objeto-relacional para gerar automaticamente a base de dados no banco H2. Este processo demonstra a eficiência do Spring Boot e do Spring Data JPA na gestão e abstração do acesso a dados, facilitando a manipulação e o gerenciamento do banco de dados de forma automatizada e simplificada.

2.8 Order, Enum, relacionamento muitos-para-um

Nesta seção, vamos explorar a criação de um tipo enum `OrderStatus` e a entidade `Order` em um projeto Spring Boot. Também abordaremos o relacionamento desta entidade com a entidade `User`, configurando um relacionamento de mão dupla entre `User` e `Order`.

Criação do Enum OrderStatus

Enums são tipos de dados que consistem em um conjunto fixo de constantes. No Java, a declaração `enum` é usada para definir valores fixos e imutáveis.

```
package com.devsuperior.dscommerce.entities;

public enum OrderStatus {
    WAITING_PAYMENT, PAID, SHIPPED, DELIVERED, CANCELED;
}
```

- **Enum `OrderStatus`**: Define os possíveis estados de um pedido. Cada constante do enum representa um status diferente que um pedido pode ter ao longo do seu ciclo de vida.

Criação da Classe Order

A classe `Order` representa um pedido dentro do sistema e é mapeada para uma tabela no banco de dados usando JPA.

```
package com.devsuperior.dscommerce.entities;

import java.time.Instant;
import jakarta.persistence.*;

@Entity
@Table(name = "tb_order")
public class Order {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(columnDefinition = "TIMESTAMP WITHOUT TIME ZONE")
    private Instant moment;
    private OrderStatus status;

    @ManyToOne
    @JoinColumn(name = "client_id")
    private User client;

    public Order() {
    }

    public Order(Long id, Instant moment, OrderStatus status, User client) {
        this.id = id;
        this.moment = moment;
        this.status = status;
        this.client = client;
    }

    // Getters and Setters are omitted for brevity
}
```

Explicação do Código:

- **@Entity** e **@Table(name = "tb_order")**: Anotações que indicam que esta classe é uma entidade JPA e será mapeada para a tabela `tb_order` no banco de dados.
- **@Id** e **@GeneratedValue(strategy = GenerationType.IDENTITY)**: Indicam que o campo `id` é a chave primária da tabela e que será gerado automaticamente pelo banco de dados.
- **@Column(columnDefinition = "TIMESTAMP WITHOUT TIME ZONE")**: Especifica que o campo `moment` deve ser armazenado como um timestamp sem informações de fuso horário.
- **@ManyToOne** e **@JoinColumn(name = "client_id")**: Estabelecem um relacionamento muitos-para-um entre `Order` e `User`. O `JoinColumn` indica que a coluna `client_id` na tabela `tb_order` é a chave estrangeira que referencia a tabela `tb_user`.

Modificações na Classe User para Relacionamento de Mão Dupla

Para completar o relacionamento de mão dupla entre `User` e `Order`, modificamos a classe `User` para incluir uma lista de pedidos associados ao usuário.

```
package com.devsuperior.dscommerce.entities;

import jakarta.persistence.*;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
```

```
@Entity
@Table(name = "tb_user")
public class User {

    // Código existente omitido para brevidade

    @OneToMany(mappedBy = "client")
    private List<Order> orders = new ArrayList<>();

    // Código existente omitido para brevidade

    public List<Order> getOrders() {
        return orders;
    }
}
```

- **@OneToMany(mappedBy = "client")**: Define um relacionamento de um-para-muitos entre `User` e `Order`. O atributo `mappedBy` indica que a classe `Order` possui o campo `client` que mantém o relacionamento.

Execução e Verificação no H2 Console

Após implementar as classes `User` e `Order` com o relacionamento especificado, execute o projeto novamente:

1. **Execute o Projeto**: Use o IntelliJ IDEA ou outra IDE para executar a aplicação Spring Boot.
2. **Acesse o H2 Console**: Abra um navegador e digite o URL `http://localhost:8080/h2-console`.
3. **Conecte-se ao Banco**: Use as credenciais e a URL JDBC configuradas para acessar o banco de dados H2.

Verifique se a tabela `tb_order` foi criada corretamente e se possui a chave estrangeira `client_id`, conforme definido no mapeamento objeto-relacional.

2.9 Payment, relacionamento um-para-um

Nesta seção, exploraremos a criação da entidade `Payment` e seu mapeamento um-para-um com a entidade `Order` usando a Java Persistence API (JPA). Também detalharemos as modificações necessárias na classe `Order` para implementar um mapeamento de mão dupla, e explicaremos como essas configurações refletem no banco de dados.

A Classe Payment

A classe `Payment` representa os detalhes de pagamento associados a um pedido. Vamos analisar o código e entender cada parte do seu mapeamento com JPA.

```
package com.devsuperior.dscommerce.entities;

import jakarta.persistence.*;
```

```java
import java.time.Instant;

@Entity
@Table(name = "tb_payment")
public class Payment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(columnDefinition = "TIMESTAMP WITHOUT TIME ZONE")
    private Instant moment;

    @OneToOne
    @MapsId
    private Order order;

    public Payment() {
    }

    public Payment(Long id, Instant moment, Order order) {
        this.id = id;
        this.moment = moment;
        this.order = order;
    }

    // Getters and setters omitted for brevity
}
```

Explicação do Código:

- **@Entity** e **@Table(name = "tb_payment")**: Define a classe `Payment` como uma entidade e mapeia para a tabela `tb_payment`.
- **@Id** e **@GeneratedValue(strategy = GenerationType.IDENTITY)**: A chave primária `id` é gerada automaticamente pelo banco de dados.
- **@Column(columnDefinition = "TIMESTAMP WITHOUT TIME ZONE")**: Especifica como o campo `moment` deve ser armazenado no banco de dados.
- **@OneToOne** e **@MapsId**: Estas anotações estabelecem um relacionamento um-para-um com a entidade `Order`. `@MapsId` indica que o `id` de `Payment` será mapeado para o mesmo `id` de `Order`, compartilhando assim a chave primária entre as duas entidades.

Modificações na Classe Order

Para completar o mapeamento de mão dupla entre `Order` e `Payment`, a classe `Order` deve ser modificada para incluir uma referência ao `Payment` associado.

```java
@Entity
@Table(name = "tb_order")
public class Order {

    // Codigo existente
```

```
    @OneToOne(mappedBy = "order", cascade = CascadeType.ALL)
    private Payment payment;

    // Codigo existente

    public Payment getPayment() {
        return payment;
    }

    public void setPayment(Payment payment) {
        this.payment = payment;
    }

    // Codigo existente
}
```

Explicação do Mapeamento:

- **@OneToOne(mappedBy = "order", cascade = CascadeType.ALL)**: Configura o relacionamento de mão dupla, onde `mappedBy` aponta para o campo `order` na entidade `Payment`. A opção `cascade = CascadeType.ALL` significa que as operações de persistência, atualização ou exclusão em `Order` serão propagadas para `Payment`.

Execução e Verificação no H2 Console

Após implementar as modificações:

1. **Execute a Aplicação**: Rode o projeto Spring Boot novamente usando sua IDE preferida para que as configurações de mapeamento sejam aplicadas ao banco de dados.
2. **Acesse o H2 Console**: Verifique se as tabelas `tb_order` e `tb_payment` foram criadas corretamente. A URL de acesso usual é `http://localhost:8080/h2-console`.

Impacto no Modelo Relacional

Com este mapeamento, a tabela `tb_payment` no banco de dados terá seu `id` como uma chave primária que também é uma chave estrangeira referenciando `id` na tabela `tb_order`. Isso significa que cada pagamento está diretamente associado a um pedido, compartilhando o mesmo identificador, o que facilita a gestão de dados relacionados entre pedidos e seus pagamentos.

2.10 Muitos-para-muitos, column unique e text

Nesta seção, vamos explorar a implementação das entidades `Product` e `Category` no projeto. Essas duas classes têm um relacionamento muitos-para-muitos. Vamos detalhar cada aspecto do código, focando especialmente no relacionamento bilateral muitos para muitos, uso da annotation `@Column` com argumentos específicos, e a criação das tabelas correspondentes no banco de dados.

Classe Category

A classe `Category` representa uma categoria de produtos. Vejamos seu código e as configurações relevantes:

```
package com.devsuperior.dscommerce.entities;

import jakarta.persistence.*;

import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

@Entity
@Table(name = "tb_category")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @ManyToMany(mappedBy = "categories")
    private Set<Product> products = new HashSet<>();

    public Category() {
    }

    public Category(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Set<Product> getProducts() {
        return products;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
```

```
        Category category = (Category) o;

        return Objects.equals(id, category.id);
    }

    @Override
    public int hashCode() {
        return id != null ? id.hashCode() : 0;
    }
}
```

Explicação do Código:

- **@Entity** e **@Table(name = "tb_category")**: Define a classe como uma entidade JPA e mapeia para a tabela `tb_category`.
- **@Id** e **@GeneratedValue(strategy = GenerationType.IDENTITY)**: Especifica que o campo `id` é a chave primária e é gerado automaticamente pelo banco de dados.
- **@ManyToMany(mappedBy = "categories")**: Estabelece um relacionamento muitos-para-muitos com a entidade `Product`. O `mappedBy` indica que a entidade `Product` é o proprietário do relacionamento.

Classe Product

A classe `Product` representa produtos que podem pertencer a múltiplas categorias.

```
package com.devsuperior.dscommerce.entities;

import jakarta.persistence.*;

import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

@Entity
@Table(name = "tb_product")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @Column(columnDefinition = "TEXT")
    private String description;
    private Double price;
    private String imgUrl;

    @ManyToMany
    @JoinTable(name = "tb_product_category",
            joinColumns = @JoinColumn(name = "product_id"),
            inverseJoinColumns = @JoinColumn(name = "category_id"))
    private Set<Category> categories = new HashSet<>();
```

```
    public Product() {
    }

    public Product(Long id, String name, String description, Double price,
    String imgUrl) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.price = price;
        this.imgUrl = imgUrl;
    }

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public String getImgUrl() {
        return imgUrl;
    }

    public void setImgUrl(String imgUrl) {
        this.imgUrl = imgUrl;
    }

    public Set<Category> getCategories() {
        return categories;
    }

    @Override
```

```
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Product product = (Product) o;

        return Objects.equals(id, product.id);
    }

    @Override
    public int hashCode() {
        return id != null ? id.hashCode() : 0;
    }
}
```

Explicação do Código:

- **@Column(columnDefinition = "TEXT")**: Define que o campo `description` deve ser armazenado como texto, o que é útil para descrições longas que excedem o limite padrão de caracteres.
- **@ManyToMany** e **@JoinTable**: Configura o relacionamento muitos-para-muitos, especificando a tabela de junção `tb_product_category` e os campos de chave estrangeira para ambas as entidades.

Execução e considerações finais

Após implementar as classes, execute o projeto novamente e acesse o H2 Console para verificar a criação das tabelas `tb_product`, `tb_category`, e `tb_product_category`. O relacionamento muitos-para-muitos deve ser refletido corretamente na tabela de associação `tb_product_category`, com chaves estrangeiras apontando para as tabelas de produtos e categorias.

As configurações feitas nas classes `Product` e `Category` demonstram como implementar e configurar um relacionamento muitos-para-muitos em JPA, usando anotações específicas para definir detalhes importantes como a unicidade e o tipo de dados das colunas. Isso garante que o modelo de dados seja robusto e adequadamente representado no banco de dados.

2.11 Muitos-para-muitos com classe de associação

Nesta seção, exploraremos a implementação de um relacionamento muitos para muitos entre as entidades `Product` e `Order` através de uma classe de associação `OrderItem`. Esse tipo de relacionamento é necessário quando o relacionamento em si precisa armazenar dados adicionais, além das chaves estrangeiras. Também discutiremos a implementação da chave composta usando a classe `OrderItemPK`.

Classe OrderItemPK

A classe `OrderItemPK` representa a chave composta da entidade `OrderItem`. No JPA, chaves compostas são implementadas usando a anotação `@Embeddable`, que indica que a classe pode ser embutida em outra entidade.

```java
package com.devsuperior.dscommerce.entities;

import jakarta.persistence.Embeddable;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;

import java.util.Objects;

@Embeddable
public class OrderItemPK {

    @ManyToOne
    @JoinColumn(name = "order_id")
    private Order order;

    @ManyToOne
    @JoinColumn(name = "product_id")
    private Product product;

    public OrderItemPK() {
    }

    public Order getOrder() {
        return order;
    }

    public void setOrder(Order order) {
        this.order = order;
    }

    public Product getProduct() {
        return product;
    }

    public void setProduct(Product product) {
        this.product = product;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        OrderItemPK that = (OrderItemPK) o;

        if (!Objects.equals(order, that.order)) return false;
        return Objects.equals(product, that.product);
    }

    @Override
    public int hashCode() {
        int result = order != null ? order.hashCode() : 0;
        result = 31 * result + (product != null ? product.hashCode() : 0);
        return result;
    }
}
```

Explicação do Código:

- **@Embeddable**: Marca a classe como incorporável em outras entidades.
- **@ManyToOne**: Define um relacionamento muitos-para-um, significando que muitos itens de pedido podem estar associados a um único pedido ou produto.
- **@JoinColumn**: Especifica o nome da coluna na tabela que estabelece o vínculo.

Classe OrderItem

`OrderItem` usa a chave composta `OrderItemPK` para estabelecer relações com as entidades `Product` e `Order` e armazenar informações adicionais como `quantity` e `price`.

```
package com.devsuperior.dscommerce.entities;

import jakarta.persistence.*;

import java.util.Objects;

@Entity
@Table(name = "tb_order_item")
public class OrderItem {

    @EmbeddedId
    private OrderItemPK id = new OrderItemPK();

    private Integer quantity;
    private Double price;

    public OrderItem() {
    }

    public OrderItem(Order order, Product product, Integer quantity, Double
   price) {
        id.setOrder(order);
        id.setProduct(product);
        this.quantity = quantity;
        this.price = price;
    }

    public Order getOrder() {
        return id.getOrder();
    }

    public void setOrder(Order order) {
        id.setOrder(order);
    }

    public Product getProduct() {
        return id.getProduct();
    }

    public void setProduct(Product product) {
        id.setProduct(product);
    }

    public Integer getQuantity() {
```

```
        return quantity;
    }

    public void setQuantity(Integer quantity) {
        this.quantity = quantity;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        OrderItem orderItem = (OrderItem) o;

        return Objects.equals(id, orderItem.id);
    }

    @Override
    public int hashCode() {
        return id != null ? id.hashCode() : 0;
    }
}
```

Explicação do Código:

- **@Entity** e **@Table(name = "tb_order_item")**: Indica que esta é uma entidade e mapeia para uma tabela específica.
- **@EmbeddedId**: Utiliza `OrderItemPK` como chave primária da entidade, aproveitando as configurações de chave composta.

Alterações nas Classes Product e Order

Para suportar o acesso bidirecional entre `Product` e `Order`, as classes são modificadas para incluir um conjunto de `OrderItem`.

Note que, além do método `getItems` para retornar uma lista de `OrderItem`, foram implementados também métodos `getOrders` e `getProducts` nas classes `Product` e `Order` respectivamente, para que o acesso aos pedidos e produtos fique transparente ao programador, a partir das classes `Product` e `Order` respectivamente.

Classe Product

```
@Entity
@Table(name = "tb_product")
```

```
public class Product {

    // codigo existente

    @OneToMany(mappedBy = "id.product")
    private Set<OrderItem> items = new HashSet<>();

    // codigo existente

    public Set<OrderItem> getItems() {
        return items;
    }

    public List<Order> getOrders() {
        return items.stream().map(x -> x.getOrder()).toList();
    }

    // codigo existente
}
```

Classe Order

```
@Entity
@Table(name = "tb_order")
public class Order {

    // codigo existente

    @OneToMany(mappedBy = "id.order")
    private Set<OrderItem> items = new HashSet<>();

    // codigo existente

    public Set<OrderItem> getItems() {
        return items;
    }

    public List<Product> getProducts() {
        return items.stream().map(x -> x.getProduct()).toList();
    }

    // codigo existente
}
```

Verificação no H2 Console e considerações finais

Após implementar as alterações, execute o projeto novamente e acesse o H2 Console para verificar se as tabelas `tb_order_item`, `tb_product`, e `tb_order` foram corretamente criadas, incluindo a tabela de associação `tb_order_item` com as chaves estrangeiras `order_id` e `product_id`.

Implementamos um relacionamento muitos para muitos com atributos adicionais usando a classe de associação `OrderItem` e uma chave composta `OrderItemPK`. As modificações nas classes `Product`

e `Order` permitem uma interação fácil e direta, facilitando o acesso aos pedidos e produtos a partir de cada classe, respectivamente. Essa abordagem não apenas atende aos requisitos de modelagem de dados complexos mas também alinha com as boas práticas de design de banco de dados e desenvolvimento de aplicativos.

2.12 Seeding da base de dados

Agora que nossa base de dados está criada, precisamos popular essa base de dados com alguns dados de teste, para que tenhamos dados para testar as lógicas de negócio que vamos implementar em seguida. O processo de popular a base de dados com dados iniciais é chamado de *seeding*.

Como implementar o seeding da base de dados no Spring Boot

Para implementar o seeding da base de dados em um projeto Spring Boot, o primeiro passo é criar um arquivo de nome `import.sql` dentro da pasta `src/main/resources`.

Uma vez criado este arquivo, podemos colocar nele comandos INSERT para inserir os dados. Por exemplo, comecemos inserindo apenas algumas categorias:

```
INSERT INTO tb_category(name) VALUES ('Livros');
INSERT INTO tb_category(name) VALUES ('Eletrônicos');
INSERT INTO tb_category(name) VALUES ('Computadores');
```

Depois de incluir os comandos INSERT de algumas categorias do arquivo `import.sql`, ao executar o projeto e verificar a tabela `tb_category`, podemos observar que as categorias foram inseridas na base de dados.

Uma vez validado que nosso arquivo `import.sql` está sendo executado, podemos prosseguir incluindo outros comandos INSERT para fazer o *seeding* nas demais tabelas da nossa base de dados.

Cuidados ao escrever o script de seeding

- **Chaves estrangeiras**: como estamos adotando aqui chaves primárias auto-incrementais que são geradas pelo próprio banco de dados, precisamos observar a sequência de contagem iniciando pelo valor 1. Por exemplo, se inserimos três categorias 'Livros', 'Eletrônicos' e 'Computadores', nesta ordem, os valores das chaves primárias destas três categorias serão 1, 2 e 3 respectivamente. Assim, quando precisarmos, por exemplo, preencher o valor de uma chave estrangeira que se refere à categoria 'Eletrônicos', o valor dessa chave estrangeira deverá ser 2.
- **Dependência entre tabelas e ordem de inserção**: você deve observar a ordem coerente de inserção dos dados, pois você não pode inserir um valor de chave estrangeira para um registro que ainda não foi inserido no banco de dados. Por exemplo, eu não posso preencher uma chave estrangeira referente a uma categoria de chave 2, se ainda não há no banco de dados uma categoria com código 2.

- **Nomes de tabelas e campos**: os nomes das tabelas e campos do seu script SQL deve ser exatamente equivalente aos nomes definidos nas suas classes Java. Se algum nome estiver errado no script, vai ocorrer um erro na execução do *seeding*.

Script completo para seeding do DSCommerce

No link anexo está disponibilizado um script completo para *seeding* da base de dados do DSCommerce.

https://gist.github.com/acenelio/664c3508edd4d418d566ed86179fdf8b

Capítulo 3

API REST, camadas, CRUD, exceções, validações

3.1 Visão geral do capítulo

Bem-vindos ao capítulo "API REST, camadas, CRUD, exceções, validações". Agora que aprendemos a implementar o modelo de domínio, precisamos aprender como implementar as funcionalidades de um sistema, que inclui organizar a aplicação em camadas, realizar operações de manipulação dos dados, tratar exceções e validações de dados e, por fim, disponibilizar uma API Rest para que essas operações fiquem disponíveis na web para outras aplicações consumirem.

Neste capítulo vamos começar implementando as funcionalidades de CRUD, que basicamente são as famosas "telas de cadastro" de um sistema, as quais disponibilizam operações para criar, recuperar, atualizar e deletar registros. A sigla CRUD vem do inglês: Create, Retrieve, Update e Delete.

Entretanto, como estamos tratando aqui apenas do backend do sistema, nós não vamos implementar as "telas de cadastro" propriamente ditas, pois as "telas" gráficas correspondem ao frontend do sistema, que não pertence ao escopo deste material. Mesmo assim, no backend nós implementamos as funcionalidades de CRUD no lado do backend, ou seja, a parte do sistema responsável por realizar operações de manipulação dos dados.

Também vamos aprender como organizar a aplicação no padrão arquitetural "camadas", para que possamos ter uma aplicação organizada, com os componentes com responsabilidades bem definidas, e de fácil manutenção.

Vamos aprender também como tratar exceções e validações de dados, retornando os códigos de erro apropriados ao protocolo HTTP. Vamos aprender também boas práticas de implementação de uma API para disponibilizar as funcionalidades do backend, e também vamos passar pelos principais conceitos relacionados ao desenvolvimento web de uma API.

Este capítulo é muito importante e foi criado com muito carinho para que você possa ter a melhor experiência de aprendizado neste assuntos que são essenciais para um desenvolvedor backend. Vamos começar!

3.2 Material de apoio do capítulo

Para este capítulo, vamos utilizar os seguintes arquivos contidos no material de apoio anexo:

- 03 API REST, camadas, CRUD, exceções, validações (slides).pdf
- 03 DESAFIO CRUD de clientes.pdf
- Projeto DSCommerce ao final do capítulo 3

3.3 O que é uma API REST?

Nesta seção, vamos explorar o conceito de API REST, um componente essencial no desenvolvimento de aplicações modernas que interagem através da web. Vamos começar entendendo o que é uma API e progredir até as especificidades das APIs REST.

API: Application Programming Interface

Uma API, ou Interface de Programação de Aplicações, é fundamentalmente um conjunto de definições e protocolos para a construção e integração de software de aplicação. É um contrato estabelecido entre um provedor de funcionalidades e o consumidor dessas funcionalidades, o que permite que diferentes programas se comuniquem entre si sem necessidade de conhecer detalhes da implementação interna.

- **Funcionalidades Expostas**: Uma API define funcionalidades que estão disponíveis para serem utilizadas por outros sistemas e módulos, sem expor os detalhes internos de como essas funcionalidades são implementadas.
- **Contrato**: Funciona como um contrato que especifica as entradas e saídas que os consumidores podem esperar, garantindo uma interação padronizada e previsível.

API Web

Uma API Web estende o conceito de APIs para o ambiente web. Neste contexto, as funcionalidades são acessadas por meio de endpoints web. Estes são pontos de contato onde as APIs podem ser acessadas e incluem:

- **Host e Porta**: O endereço do servidor onde a API está hospedada e a porta através da qual as comunicações ocorrem.
- **Rota**: O caminho específico no servidor onde a API ou um recurso específico pode ser acessado.
- **Parâmetros e Corpo (Payload)**: Dados enviados para a API para refinar uma solicitação ou enviar informações para processamento.
- **Cabeçalhos**: Informações adicionais enviadas junto com a solicitação ou resposta que podem incluir tokens de autenticação, tipos de conteúdo esperados, entre outros.

API REST: Princípios do Padrão REST

Uma API REST é uma forma de API Web que segue os princípios do Representational State Transfer (REST), um estilo arquitetural definido por Roy Fielding em sua dissertação doctoral em 2000. REST não é uma norma ou protocolo, mas um conjunto de restrições arquitetônicas que, quando seguidas, facilitam a criação de projetos de software web que são escaláveis, flexíveis, e eficientes. Vamos detalhar cada um dos princípios fundamentais do REST:

1. Cliente/Servidor com HTTP

O modelo REST é baseado na separação entre cliente e servidor, uma separação de responsabilidades que permite que ambos evoluam de forma independente. O cliente não precisa saber os detalhes de armazenamento de dados do servidor, enquanto o servidor não precisa se preocupar com a interface e o estado do usuário, permitindo que a interface do usuário seja melhorada sem afetar a lógica do servidor.

- **HTTP**: REST utiliza protocolo HTTP para as comunicações, empregando seus métodos padrão (GET, POST, PUT, DELETE, etc.) para realizar operações CRUD (Create, Read, Update, Delete) em recursos representados geralmente em formatos como JSON ou XML.

2. Comunicação Stateless

Cada requisição de cliente para o servidor deve conter toda a informação necessária para entender e completar a requisição. O servidor não deve armazenar nada sobre o estado mais recente do cliente entre as requisições. Isso permite que o sistema seja mais fácil de escalar, pois não depende do contexto armazenado no servidor para processar as requisições.

3. Cache

A capacidade de cache é uma característica integral do REST. As respostas devem, implícita ou explicitamente, definir a si mesmas como cacheáveis ou não, permitindo que os clientes reutilizem respostas armazenadas para melhorar a eficiência e a escalabilidade. Isso reduz a carga no servidor e melhora a latência percebida pelo usuário final.

4. Interface Uniforme

Um dos principais princípios do REST é a uniformidade da interface entre componentes, que simplifica e desacopla a arquitetura, o que permite que cada parte evolua independentemente. As seguintes condições devem ser satisfeitas para alcançar uma interface uniforme:

- **Identificação de recursos**: Os recursos são identificados em requisições usando URIs de forma padronizada.
- **Manipulação de recursos através de representações**: Os recursos são manipulados através de suas representações (como JSON ou XML), e as operações são realizadas no servidor.
- **Mensagens autoexplicativas**: As mensagens devem ser suficientemente autoexplicativas para descrever como processá-las.
- **HATEOAS (Hypermedia as the Engine of Application State)**: Os clientes interagem com a aplicação inteiramente através de hyperlinks fornecidos dinamicamente pelo servidor.

5. Sistema em Camadas

O sistema em camadas restringe a interação entre componentes, permitindo que eles operem dentro de camadas hierárquicas. Um cliente não pode geralmente ver além da camada com a qual está interagindo, permitindo que sistemas intermediários melhorem a escalabilidade por meio de balanceamento de carga ou caches compartilhados.

6. Código sob Demanda (Opcional)

Este é o único princípio opcional do REST, permitindo que servidores transfiram executáveis ou scripts temporários para os clientes quando necessário para estender a funcionalidade do cliente. Isso oferece uma forma de estender e personalizar a lógica do cliente sem a necessidade de um novo deployment ou atualização.

Para mais detalhes, consulte:

https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api

Diferença entre Backend e Frontend em um Sistema Web

- **Frontend**: Refere-se à parte da aplicação web que é executada no navegador do cliente. Inclui tudo o que o usuário interage diretamente, como páginas da web e interfaces de usuário.
- **Backend**: É o lado do servidor da aplicação, onde ocorrem processamento de dados, autenticação de usuários, operações de banco de dados e lógica de negócios.

É importante destacar que a API não é exatamente o mesmo que o backend. Enquanto o backend refere-se a toda a parte do servidor responsável pela lógica de negócios, gerenciamento de dados e autenticação, a API é a parte do backend que é exposta para que outras aplicações ou o próprio frontend possam interagir com esses serviços de backend.

3.4 Recursos, URL, Parâmetros de Consulta e de Rota

Na construção de APIs REST e aplicações web em geral, a organização e acesso à funcionalidade ou informações são concebidos através de **recursos**. Nesta seção, exploraremos como esses recursos são acessados usando URLs, e como os parâmetros de consulta e de rota são utilizados para refinar e especificar esses acessos.

Recursos

Num sistema web, um **recurso** é uma entidade ou uma coleção de entidades que são consideradas parte essencial da funcionalidade do sistema. Por exemplo, em uma aplicação de e-commerce, produtos, clientes, pedidos, e categorias são todos recursos. Cada recurso é identificado e acessado através de uma URL única.

URL - Universal Resource Locator

A **URL** (Localizador Uniforme de Recursos) é o endereço utilizado para acessar um recurso na web. Ela define a localização de um recurso na rede, que pode ser acessado utilizando protocolos como HTTP ou HTTPS. A URL especifica o protocolo usado, o host (ou endereço IP), a porta (opcional), e o caminho (path) até o recurso.

Estrutura de uma URL para Acessar Recursos

Os recursos em uma aplicação REST são acessados principalmente por URLs que são estruturadas para refletir a organização dos dados de forma lógica e hierárquica. Vejamos alguns exemplos típicos de como as URLs podem ser estruturadas para acessar recursos:

- **GET** `host:port/products`: Esta URL acessa a lista de todos os produtos disponíveis. Usando o método HTTP GET, essa operação é destinada a apenas recuperar dados sem modificar nada no servidor.
- **GET** `host:port/products?page=3`: Acessa a lista de produtos, mas com um parâmetro de consulta `page` que especifica que queremos os produtos listados na página 3. Este é um exemplo de como parâmetros de consulta podem ser usados para modificar a resposta do servidor sem alterar o recurso em si.
- **GET** `host:port/products/1`: Acessa um produto específico identificado pelo ID, que neste caso é 1. Este é um exemplo de parâmetro de rota, onde o identificador do recurso faz parte do caminho da URL.
- **GET** `host:port/products/1/categories`: Recupera as categorias associadas ao produto com ID 1, demonstrando como os caminhos da URL podem ser usados para acessar recursos relacionados ou sub-recursos.

Parâmetros de URL e Parâmetros de Rota

Parâmetros de Rota: - São utilizados para identificar um recurso específico ou um grupo de recursos. Eles são parte do caminho (path) da URL. No exemplo `host:port/products/1`, o número `1` é um parâmetro de rota que especifica qual produto deve ser retornado.

Parâmetros de Consulta: - São opções adicionais que podem ser usadas para ordenar, filtrar, ou determinar o formato da resposta retornada do servidor. Eles não fazem parte do caminho da URL, mas são anexados ao final dela com um prefixo `?`. Por exemplo, em `host:port/products ?page=3`, o parâmetro de consulta `page` com valor `3` instrui o servidor a retornar apenas a terceira página de produtos.

3.5 Padrões de URL, Verbos HTTP e Códigos de Resposta

Esta seção discute os padrões de URL, verbos HTTP e códigos de resposta que formam a base das comunicações na web, especialmente em APIs RESTful. Compreender esses componentes é crucial para desenvolver serviços web eficientes e aderentes às melhores práticas.

Padrões de URL

Em uma API REST, a ação desejada sobre um recurso deve ser expressa principalmente pelo verbo HTTP e não pela URL. A URL deve identificar o recurso, enquanto o verbo HTTP indica a ação a ser realizada nesse recurso.

Exemplos Incorretos e Corretos:

Incorreto: - **GET**: `host:port/insertProduct` - **GET**: `host:port/listProduct`

Esses exemplos são incorretos porque utilizam o método GET para ações que devem ser expressas por outros verbos HTTP, além de descrever ações na própria URL.

Correto: - **POST**: `host:port/products` - **GET**: `host:port/products`

Nesses exemplos corretos, o verbo POST é usado para criar um novo produto, e GET é usado para listar produtos. As URLs descrevem os recursos, e os verbos descrevem as ações.

Verbos (Métodos) HTTP Mais Utilizados

Os verbos HTTP definem o tipo de operação que se deseja realizar em um recurso. Os mais comuns são:

- **GET**: Usado para obter um ou mais recursos. GET deve ser seguro e, idealmente, idempotente, o que significa que não altera nenhum estado no servidor.
- **POST**: Utilizado para criar um novo recurso. POST não é idempotente, o que significa que enviar várias solicitações POST idênticas podem resultar em múltiplas criações.
- **PUT**: Empregado para atualizar um recurso existente ou criar um recurso se ele não existir, de maneira idempotente.
- **DELETE**: Usado para remover um recurso. DELETE é idempotente pois deletar o mesmo recurso repetidamente tem o mesmo efeito que deletá-lo uma única vez.

Operação Idempotente: Uma operação é idempotente se não causar efeitos adicionais se for executada mais de uma vez com o mesmo input.

Para mais detalhes, consulte:

https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods

Códigos de Resposta HTTP

Os códigos de resposta HTTP indicam se uma operação na web foi realizada com sucesso, encontrou um erro, ou algo intermediário. Eles são agrupados da seguinte maneira:

- **Respostas de Informação (100-199)**: Indicam que a solicitação foi recebida e entendida e está sendo processada.
- **Respostas de Sucesso (200-299)**: Confirmam que a solicitação foi recebida, entendida e aceita com sucesso.
- **Redirecionamentos (300-399)**: Informam que ações adicionais precisam ser tomadas para completar a solicitação.
- **Erros do Cliente (400-499)**: Indicam que houve um erro na solicitação e que a solicitação não pode ser processada.
- **Erros do Servidor (500-599)**: Indicam que o servidor falhou ao tentar processar uma solicitação válida.

Para uma descrição completa dos códigos de resposta, visite:

https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status

3.6 Padrão Camadas

O padrão de camadas é uma abordagem de arquitetura de software que ajuda a organizar as diferentes partes de um sistema de forma clara e estruturada. Este padrão é crucial para a criação de aplicações robustas e facilmente mantidas, permitindo que cada seção do sistema tenha responsabilidades bem definidas.

Organização em Camadas

A organização em camadas divide os componentes do sistema em grupos horizontais, cada um com uma responsabilidade específica. Cada camada só pode interagir com a camada direta-

mente abaixo dela, garantindo um acoplamento baixo e uma alta coesão dentro das camadas. Essa separação clara facilita tanto a manutenção quanto a escalabilidade do sistema.

Camadas e Suas Responsabilidades

1. **Controlador (ou Camada de Apresentação)**:
   - **Responsabilidade**: Responder às interações do usuário. Em uma API REST, essas interações são representadas pelas requisições HTTP. A camada de controlador captura essas requisições e as encaminha para a camada de serviço apropriada, tratando também de devolver as respostas ao cliente.
2. **Service (ou Camada de Negócio)**:
   - **Responsabilidade**: Executar as operações de negócio. Esta camada é o coração da lógica de negócios da aplicação, onde métodos significativos são implementados. Por exemplo, um método `registrarPedido` na camada de serviço poderia realizar várias tarefas como verificar estoque, salvar o pedido no banco de dados, atualizar o estoque e enviar um e-mail de confirmação.
   - **Transações**: É nesta camada que geralmente se definem os limites transacionais das operações com o banco de dados, garantindo a integridade e a consistência dos dados.
   - **Uso de DTOs**: Ao devolver dados para o controlador, a camada de serviço utiliza Objetos de Transferência de Dados (DTOs), que são versões simplificadas das entidades, desvinculadas do ORM (Object-Relational Mapping). Isso evita problemas relacionados ao ciclo de vida da sessão ORM e reduz o overhead de dados desnecessários sendo enviados ou recebidos.
3. **Repository (ou Camada de Persistência)**:
   - **Responsabilidade**: Gerenciar as operações de banco de dados em nível mais granular. Esta camada lida diretamente com o acesso ao banco de dados, executando operações como busca, inserção, atualização e exclusão de registros. O Repository deve focar em operações "atômicas", deixando a lógica de negócios para a camada de serviço.

Interações entre as Camadas

- **Controlador → Serviço**: O controlador recebe requisições do usuário (ou de outro sistema) e delega a execução das operações à camada de serviço. Após o processamento, recebe de volta os DTOs para responder à requisição inicial.
- **Serviço → Repository**: A camada de serviço solicita ao repository a execução de operações de banco de dados específicas, como a recuperação ou gravação de dados. O serviço compila os resultados das várias chamadas ao repository para formar a resposta de negócios completa.
- **Repository**: A camada de repository interage diretamente com o banco de dados, sem nenhuma lógica de negócios, apenas traduzindo as solicitações em comandos SQL ou utilizando um framework ORM para mapeamento objeto-relacional.

A utilização do padrão de camadas na arquitetura de uma aplicação não só facilita a organização e manutenção do código, como também promove uma melhor separação de preocupações. Isso é especialmente importante em sistemas complexos, onde a clareza na definição de responsabilidades é fundamental para o sucesso do desenvolvimento e manutenção do software. A discussão sobre DTOs será expandida em seções subsequentes, onde exploraremos mais profundamente

seu papel e implementação em APIs REST.

3.7 Aviso: DSCommerce preparado

Atenção: neste momento preciso que você esteja com seu projeto DSCommerce preparado na sua IDE conforme foi finalizado no capítulo anterior, ou seja, o projeto precisa estar com o modelo de domínio implementado, o *seeding* do banco de dados funcionando, e todas tabelas criadas e populadas com os dados, conforme você pode conferir rodando o projeto e acessando o H2 Console no seu navegador.

Vamos precisar do projeto correto desta forma para prosseguir com as aulas práticas que veremos nas próximas seções.

3.8 Primeiro Teste da API

Nesta seção, iniciaremos a implementação prática da nossa API, construindo um componente essencial em qualquer aplicação baseada em arquitetura REST: o controlador. O controlador é responsável por responder a requisições web, atuando como a camada de interface entre o usuário (ou um sistema externo) e os serviços de aplicação. Vamos detalhar a criação de um controlador simples para nossa aplicação.

Criação do Controlador de Produtos

Vamos começar definindo uma classe `ProductController` dentro do pacote `com.devsuperior.dscommerce.controllers`. Este controlador terá a responsabilidade inicial de responder a requisições destinadas aos produtos da nossa aplicação.

```
package com.devsuperior.dscommerce.controllers;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/products")
public class ProductController {

    @GetMapping
    public String teste() {
        return "Ola mundo!";
    }
}
```

Anotações e Seus Significados

- **@RestController**: Esta anotação marca a classe como um controlador onde cada método retorna um domínio de objeto em vez de uma vista. Isso indica que a classe está pronta para processar requisições web.

- **@RequestMapping(value = "/products")**: Define que todas as requisições que chegam para a rota `/products` serão tratadas por este controlador.
- **@GetMapping**: Especifica que o método `teste()` deve responder a requisições GET. Quando não se especifica um caminho dentro da anotação `@GetMapping`, assume-se que o método responde pela URL definida no `@RequestMapping` da classe.

Testando o Controlador

Após implementar o controlador, o próximo passo é testar se ele está funcionando como esperado. Para isso, basta acessar a URL `http://localhost:8080/products` através de um navegador web. Se tudo estiver configurado corretamente, a página deverá exibir a mensagem "Ola mundo!", que é o retorno do método `teste()` no nosso controlador.

Introdução ao Postman

O Postman é uma ferramenta poderosa usada para testar APIs. Ele permite que desenvolvedores façam requisições a endereços de API de forma controlada, podendo manipular facilmente os cabeçalhos, o corpo da requisição e ver as respostas HTTP. Usar o Postman é uma forma eficiente de testar, desenvolver e documentar APIs.

Criando uma Collection no Postman

Para organizar seus testes de API no Postman, você pode criar uma "collection" ou coleção:

1. Abra o Postman.
2. No painel à esquerda, clique em "New" e selecione "Collection".
3. Dê um nome para sua coleção, por exemplo, "Testes API DSCommerce".

Adicionando Requisições à Collection

Dentro da coleção que você criou, você pode começar a adicionar requisições específicas:

1. Com a coleção selecionada, clique em "Add request".
2. Nomeie sua requisição, como "Teste GET Products".
3. Na aba de configuração da requisição, selecione o método GET e insira a URL `http://localhost:8080/products`.
4. Clique em "Save".

Testando a API no Postman

Depois de salvar sua requisição:

1. Selecione a requisição criada dentro da coleção.
2. Clique em "Send" para executar a requisição.
3. O Postman exibirá a resposta da API no painel abaixo, onde você deve ver "Ola mundo!" como resposta.

Este primeiro teste valida a configuração básica do nosso controlador e introduz o uso de ferramentas essenciais como o Postman para testar e desenvolver APIs.

3.9 Primeiro Teste com Repository

Nesta seção, vamos explorar como realizar um teste inicial com o componente Repository em uma aplicação Spring Boot, utilizando o Spring Data JPA. O objetivo é verificar a funcionalidade de busca de um produto no banco de dados e retornar seu nome através de um endpoint da API.

Spring Data JPA e JpaRepository

Spring Data JPA é uma parte do ecossistema Spring Data que visa simplificar a implementação de repositórios de acesso a dados. Ele permite a fácil integração entre a aplicação Spring e a API de Persistência Java (JPA), automatizando a implementação de repositórios comuns.

JpaRepository é uma interface do Spring Data que fornece métodos CRUD para manipulação de entidades de forma simplificada. Estender esta interface em um repositório de Spring permite o acesso a um rico conjunto de métodos, como `save`, `delete`, `findById`, entre outros, além de permitir a definição de consultas customizadas usando apenas assinaturas de métodos.

Implementação do ProductRepository

Vamos criar um repositório para a entidade `Product`. Este repositório estenderá `JpaRepository`, permitindo que utilizemos os métodos fornecidos para interagir com o banco de dados.

```
package com.devsuperior.dscommerce.repositories;

import com.devsuperior.dscommerce.entities.Product;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

Neste código, `ProductRepository` é uma interface que não requer implementação manual dos métodos. O Spring Data JPA se encarrega de prover a implementação em tempo de execução.

Modificações na Classe ProductController

Agora, vamos modificar a classe `ProductController` para incluir uma operação de busca usando o `ProductRepository`. Esse teste é provisório e será aprimorado posteriormente.

```
package com.devsuperior.dscommerce.controllers;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.devsuperior.dscommerce.entities.Product;
import com.devsuperior.dscommerce.repositories.ProductRepository;
```

```
@RestController
@RequestMapping(value = "/products")
public class ProductController {

    @Autowired
    private ProductRepository repository;

    @GetMapping
    public String teste() {
        Optional<Product> result = repository.findById(1L);
        Product product = result.get();
        return product.getName();
    }
}
```

Explicação do Código

- **@Autowired**: Anotação que solicita ao Spring a injeção de dependência do `ProductRepository`.
- **repository.findById(1L)**: Busca no banco de dados o produto com o ID 1. Esta operação retorna um `Optional<Product>`, que pode ou não conter um produto.
- **result.get()**: Obtém o produto do `Optional`. Em uma aplicação real, seria prudente verificar se o `Optional` realmente contém um valor para evitar exceções.

Testando a Aplicação

Para testar essa funcionalidade, acesse a URL `http://localhost:8080/products` em um navegador ou através de uma ferramenta como Postman. Se o banco de dados estiver corretamente configurado com os dados iniciais (seed), e o produto com ID 1 for "The Lord of the Rings", o endpoint retornará este nome.

Este primeiro teste com o `ProductRepository` demonstra a capacidade de integrar a camada de persistência com a camada web de uma aplicação Spring Boot. Reforça a importância da separação de responsabilidades e introduz as práticas recomendadas de acesso a dados em aplicações modernas. Em etapas futuras, vamos refinar essa implementação para incluir tratamentos de erros adequados e melhorar a estrutura do código para suportar uma manutenção fácil e uma expansão segura.

3.10 Criando DTO e Estruturando Camadas

Neste segmento do livro, vamos estruturar melhor nosso projeto em camadas, introduzindo a camada de serviço e utilizando o padrão DTO (Data Transfer Object) para otimizar a representação e transferência de dados.

Responsabilidades das Camadas

Relembrando as responsabilidades fundamentais de cada camada em nossa arquitetura:

- **Controlador**: Responsável por responder às interações do usuário, neste contexto de uma API REST, essas interações são as requisições HTTP.
- **Service**: Realiza operações de negócio. Por exemplo, um método na camada de serviço como `registrarPedido` pode realizar várias operações como verificar estoque, salvar pedido, baixar estoque, e enviar email.
- **Repository**: Executa operações individuais de acesso ao banco de dados, funcionando como a ponte entre a camada de negócios e o banco de dados.

Padrão DTO (Data Transfer Object)

O padrão DTO é utilizado para transferir dados entre sub-sistemas de uma aplicação, uma forma eficaz de passar dados com múltiplos atributos em uma única chamada de método, reduzindo o número de chamadas necessárias. Vamos explorar os detalhes e vantagens deste padrão:

- **Definição**: DTO é um objeto simples usado para transferir dados entre processos, não contendo qualquer lógica de negócio que manipule esses dados além da simples transferência.
- **Características**:
  - Simples e serializável.
  - Não gerenciado por frameworks ORM, evitando o overhead de monitoramento de estado.
  - Pode incluir outros DTOs aninhados, mas nunca entidades diretamente.

Vantagens do Uso de DTOs

- **Projeção de Dados**:
  - **Segurança**: Permite omitir dados sensíveis que não devem ser expostos.
  - **Economia de Tráfego**: Reduz a quantidade de dados enviados pela rede.
  - **Flexibilidade**: Facilita a customização da representação de dados dependendo do contexto de uso.
- **Separação de Responsabilidades**:
  - **Camadas Service e Repository**: Gerenciam transações e monitoramento ORM.
  - **Controlador**: Simplifica o tráfego de dados, utilizando DTOs que não estão atrelados ao ORM.

Implementação do ProductDTO

Vamos definir um DTO para a entidade `Product`, que servirá para transferir informações de produtos de maneira eficiente e segura.

```
package com.devsuperior.dscommerce.dto;

import com.devsuperior.dscommerce.entities.Product;

public class ProductDTO {

    private Long id;
```

```
    private String name;
    private String description;
    private Double price;
    private String imgUrl;

    public ProductDTO() {
    }

    public ProductDTO(Long id, String name, String description, Double price,
   String imgUrl) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.price = price;
        this.imgUrl = imgUrl;
    }

    public ProductDTO(Product entity) {
        id = entity.getId();
        name = entity.getName();
        description = entity.getDescription();
        price = entity.getPrice();
        imgUrl = entity.getImgUrl();
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public Double getPrice() {
        return price;
    }

    public String getImgUrl() {
        return imgUrl;
    }
}
```

Implementação da Classe ProductService

A classe `ProductService` é uma parte crucial da nossa arquitetura em camadas, responsável pela lógica de negócios relacionada aos produtos. Utiliza o repositório para acessar dados e manipula esses dados de acordo com as regras de negócio antes de transferi-los para outras partes da aplicação, como a camada de controladores. Vamos explorar em detalhes a implementação dessa classe.

```java
package com.devsuperior.dscommerce.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.devsuperior.dscommerce.dto.ProductDTO;
import com.devsuperior.dscommerce.entities.Product;
import com.devsuperior.dscommerce.repositories.ProductRepository;

@Service
public class ProductService {

    @Autowired
    private ProductRepository repository;

    @Transactional(readOnly = true)
    public ProductDTO findById(Long id) {
        Product product = repository.findById(id).get();
        return new ProductDTO(product);
    }
}
```

Anotações e Seus Papéis

- **@Service**: Esta anotação marca a classe como um componente de serviço do Spring, o que a categoriza como uma candidata para a detecção automática de beans durante a configuração baseada em anotações e a auto-configuração. Ela é destinada a classes que implementam a lógica de negócio.
- **@Autowired**: Usada para injeção de dependência automática pelo Spring. Aqui, ela é colocada no `ProductRepository` para que o Spring forneça a instância necessária do repositório sem a necessidade de inicialização manual.

Uso de @Transactional

- **@Transactional(readOnly = true)**: Esta anotação é usada para definir as características da transação.
  - **readOnly = true** significa que a transação é somente leitura. Esta configuração é importante pois informa ao gerenciador de transações e ao banco de dados que essa operação não irá realizar nenhuma alteração no estado dos dados. Isso pode ajudar a otimizar o desempenho das operações, especialmente em operações que envolvem muitos dados ou são chamadas frequentemente.
  - A presença de `@Transactional` também garante que se ocorrerem exceções durante a operação, a transação será revertida (rollback), mantendo a consistência dos dados.

Método findById

- O método `findById(Long id)` é responsável por recuperar um produto específico pelo seu ID. O uso do método `findById` do repositório retorna um `Optional<Product>`, o que é uma prática recomendada para lidar com a possibilidade de que o ID fornecido não corresponda a nenhum produto.

- A chamada `get()` no `Optional` é usada para extrair o produto, se presente. No entanto, em uma aplicação real, você deve tratar a possibilidade de `Optional` estar vazio para evitar `NoSuchElementException`, por exemplo, usando `orElseThrow()` com uma exceção customizada.
- O produto recuperado é então convertido em um `ProductDTO` usando o construtor correspondente que aceita um objeto `Product`. Isso desacopla o modelo de dados do banco de dados do modelo usado nas respostas da API, permitindo que alterações internas no modelo de entidade não afetem os clientes da API.

Nova Implementação do ProductController

O `ProductController` agora utiliza o `ProductService`, respeitando a divisão em camadas e delegando as responsabilidades de negócio para a camada de serviço.

```
package com.devsuperior.dscommerce.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.devsuperior.dscommerce.dto.ProductDTO;
import com.devsuperior.dscommerce.services.ProductService;

@RestController
@RequestMapping(value = "/products")
public class ProductController {

    @Autowired
    private ProductService service;

    @GetMapping(value = "/{id}")
    public ProductDTO findById(@PathVariable Long id) {
        return service.findById(id);
    }
}
```

3.11 Dica da Biblioteca ModelMapper para DTO

Na construção de aplicações modernas, especialmente aquelas que seguem uma arquitetura baseada em API, frequentemente nos deparamos com a necessidade de transferir dados entre as camadas de modelo de domínio e a camada de apresentação ou rede. No caso dos objetos DTO que estamos utilizando, a biblioteca ModelMapper oferece uma solução eficiente e flexível para a automação desse processo de mapeamento.

O que é ModelMapper?

ModelMapper é uma biblioteca Java que automatiza o processo de mapeamento de objetos de um tipo para outro. É amplamente utilizada em aplicações Java para mapear objetos de

domínio (entidades) para objetos DTO e vice-versa. O objetivo principal da ModelMapper é simplificar o código que mapeia entre tipos de objeto, reduzindo a quantidade de código manual que os desenvolvedores precisam escrever e manter.

Para aqueles interessados em explorar mais sobre como usar a biblioteca ModelMapper, incluindo exemplos práticos, configurações avançadas, e melhores práticas, recomendo a leitura do seguinte artigo disponível em Baeldung, uma respeitada fonte de aprendizado para desenvolvedores Java e Spring:

https://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application

3.12 CRUD

O termo CRUD é fundamental na construção de aplicações web e sistemas de gerenciamento de informações. Vamos explorar o que significa CRUD, como ele se aplica tanto no desenvolvimento de front-end quanto de back-end, e preparar o terreno para uma exploração detalhada dessas operações no contexto de Java e Spring.

O que é CRUD?

CRUD é a sigla para Create, Retrieve, Update, Delete. Essas quatro operações representam as ações básicas que são realizadas em dados em aplicações de banco de dados. A capacidade de completar operações CRUD é essencial para permitir que os usuários interajam com qualquer sistema de dados de maneira eficaz. Vamos detalhar cada uma dessas operações:

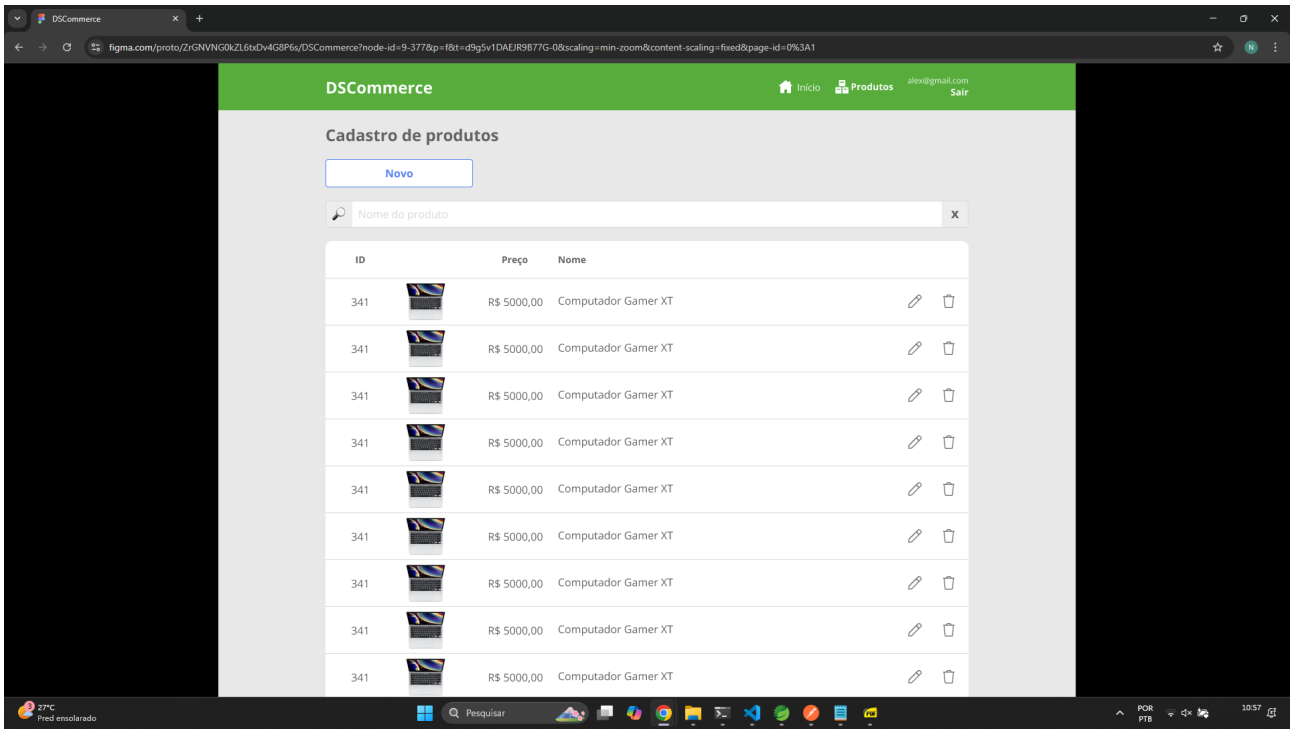
- **Create**: A operação de criar novos registros no banco de dados.
- **Retrieve**: A operação de buscar ou recuperar dados já existentes. Esta operação pode ser subdividida em buscar todos os registros ou buscar um registro específico por um identificador.
- **Update**: A operação de modificar registros existentes.
- **Delete**: A operação de remover registros do banco de dados.

CRUD no Frontend

No contexto do frontend, um CRUD geralmente se refere a uma interface de usuário que permite aos usuários realizar todas essas quatro operações. Uma "tela de cadastro", por exemplo, típicamente permite que os usuários vejam uma listagem dos registros existentes, que podem ser filtrados por critérios específicos, além de oferecer opções para adicionar novos registros, editar registros existentes ou excluir registros. Essas interfaces são essenciais para a gestão eficaz dos dados dentro de sistemas empresariais, e-commerce, blogs, e outros sistemas que necessitam de interação contínua com o banco de dados. No contexto do nosso sistema DSCommerce, o exemplo que vamos explorar é o que seria nossa tela de cadastro de produtos, conforme mostrado na imagem.

CRUD no Backend

No backend, as operações de CRUD correspondem às funcionalidades implementadas para manipular dados no sistema. No contexto de APIs, estas são algumas das operações que os desenvolvedores implementam para permitir que o frontend interaja com o banco de dados:



Tela de cadastro de produtos, no sistema DSCommerce

- **CREATE**: Salvar um novo registro no banco de dados.
- **READ**: Recuperar todos os registros existentes de uma tabela, geralmente implementado com suporte a paginação para eficiência em grandes conjuntos de dados.
- **READ**: Recuperar um registro específico dado um identificador, como um ID de usuário ou de produto.
- **UPDATE**: Atualizar um registro existente, identificado por um ID.
- **DELETE**: Deletar um registro existente, também identificado por um ID.

Implementação de um CRUD com Java e Spring

A partir deste ponto, iniciaremos uma série de seções dedicadas a demonstrar como implementar um CRUD completo no backend usando Java e Spring. Cada seção abordará em detalhes a implementação de uma parte específica do CRUD, explorando as melhores práticas, padrões de design e as capacidades do Spring Framework para facilitar e otimizar essas operações.

3.13 Busca Paginada de Produtos

Nesta seção, vamos explorar como implementar a busca paginada de produtos em uma aplicação que utiliza Spring Boot, focando na camada de serviço e controlador para gerenciar a interação com um banco de dados de produtos.

Implementação do Método findAll na Classe ProductService

A busca paginada é essencial em sistemas que lidam com uma grande quantidade de dados. Ela permite aos usuários visualizar informações em pequenas partes, o que melhora a performance e a usabilidade. Vejamos a implementação do método `findAll` em `ProductService`:

```java
package com.devsuperior.dscommerce.services;

import com.devsuperior.dscommerce.dto.ProductDTO;
import com.devsuperior.dscommerce.entities.Product;
import com.devsuperior.dscommerce.repositories.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class ProductService {

    @Autowired
    private ProductRepository repository;

    @Transactional(readOnly = true)
    public ProductDTO findById(Long id) {
        Product product = repository.findById(id).get();
        return new ProductDTO(product);
    }

    @Transactional(readOnly = true)
    public Page<ProductDTO> findAll(Pageable pageable) {
        Page<Product> result = repository.findAll(pageable);
        return result.map(x -> new ProductDTO(x));
    }
}
```

Explicação do Código

- **@Transactional(readOnly = true)**: Assegura que o método é apenas para leitura, otimizando o acesso ao banco de dados e garantindo que nenhuma alteração será feita durante a execução deste método.
- **findAll(Pageable pageable)**: Este método faz uso do Spring Data JPA para buscar todos os produtos. O objeto `Pageable` é um parâmetro do Spring que contém informações sobre a paginação e ordenação. Esse método retorna uma `Page<ProductDTO>`, que é uma coleção paginada de `ProductDTO`.

Método findAll na Classe ProductController

O `ProductController` lida com as requisições HTTP e interage com a `ProductService` para buscar dados. Vamos analisar a implementação deste método:

```java
package com.devsuperior.dscommerce.controllers;

import com.devsuperior.dscommerce.dto.ProductDTO;
import com.devsuperior.dscommerce.services.ProductService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
```

```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/products")
public class ProductController {

    @Autowired
    private ProductService service;

    @GetMapping(value = "/{id}")
    public ProductDTO findById(@PathVariable Long id) {
        return service.findById(id);
    }

    @GetMapping
    public Page<ProductDTO> findAll(Pageable pageable) {
        return service.findAll(pageable);
    }
}
```

Explicação do Código

- **@GetMapping**: Define que o método `findAll` responderá a requisições GET para a URL base `/products`. Quando acompanhado de parâmetros `Pageable`, o Spring automaticamente configura a paginação baseada nos parâmetros recebidos na requisição, como `page`, `size`, e `sort`.

Testando a Busca Paginada

Ao chamar a URL `http://localhost:8080/products` com um cliente HTTP como o Postman, você receberá um JSON paginado que lista os produtos. Esse JSON incluirá metadados sobre a paginação e os próprios produtos.

```
{
    "content": [
        {
            "id": 1,
            "name": "The Lord of the Rings",
            "description": "Lorem ipsum ...",
            "price": 90.5,
            "imgUrl": "https://raw.githubusercontent.com/devsuperior/dscatalog-
   resources/master/backend/img/1-big.jpg"
        },
        ...
    ],
    "pageable": {
        "sort": {
            "empty": false,
            "sorted": true,
            "unsorted": false
```

```
        },
        "offset": 0,
        "pageSize": 12,
        "pageNumber": 0,
        "unpaged": false,
        "paged": true
    },
    "last": false,
    "totalElements": 25,
    "totalPages": 3,
    "size": 12,
    "number": 0,
    "sort": {
        "empty": false,
        "sorted": true,
        "unsorted": false
    },
    "first": true,
    "numberOfElements": 12,
    "empty": false
}
```

Parâmetros de Paginação

É possível utilizar parâmetros de paginação na busca paginada. Os principais parâmetros são:

- **size**: Define o número de registros por página.
- **page**: Especifica o número da página que você deseja recuperar.
- **sort**: Permite especificar por qual campo os resultados devem ser ordenados e a direção (ascendente ou descendente).

Por exemplo, a URL `http://localhost:8080/products?size=12&page=0&sort=name,desc` solicitará ao servidor que retorne a primeira página de produtos, com 12 produtos por página, ordenados pelo nome em ordem descendente.

3.14 Inserindo Novo Produto com POST

Nesta seção, vamos abordar como adicionar um novo produto ao banco de dados usando um método POST. Este é um componente crucial para permitir a interação do usuário com o sistema de gerenciamento de produtos através de uma API REST.

Método insert na Classe ProductService

O método `insert` na classe `ProductService` é responsável por criar um novo produto no banco de dados. Vamos examinar cada passo deste processo:

```
@Service
public class ProductService {

    @Autowired
    private ProductRepository repository;
```

```
    @Transactional
    public ProductDTO insert(ProductDTO dto) {
        Product entity = new Product();
        entity.setName(dto.getName());
        entity.setDescription(dto.getDescription());
        entity.setPrice(dto.getPrice());
        entity.setImgUrl(dto.getImgUrl());

        entity = repository.save(entity);

        return new ProductDTO(entity);
    }
}
```

Explicação do Código

- **@Transactional**: Esta anotação inicia uma transação de banco de dados. Isso significa que se algo der errado durante a execução deste método, todas as alterações feitas no banco de dados serão automaticamente revertidas.
- **Construção da Entidade**: Um novo objeto `Product` é criado e suas propriedades são configuradas com base nos valores fornecidos no `ProductDTO`. Isso desacopla o objeto de transferência de dados (DTO) do objeto de entidade, o que é uma prática recomendada em aplicações de camadas múltiplas.
- **Salvar a Entidade**: O objeto `entity` é salvo no banco de dados usando o método `save` do repositório. Este método retorna a entidade persistida, agora com um `id` gerado pelo banco de dados.
- **Retorno como DTO**: Por fim, a entidade salva é convertida de volta para um `ProductDTO` antes de ser retornada ao cliente. Isso garante que a resposta contenha a representação mais atualizada do produto, incluindo seu identificador único.

Método insert na Classe ProductController

Na camada do controlador, a lógica é simplificada para apenas tratar da recepção e resposta das requisições HTTP:

```
@RestController
@RequestMapping(value = "/products")
public class ProductController {

    @Autowired
    private ProductService service;

    @PostMapping
    public ProductDTO insert(@RequestBody ProductDTO dto) {
        return service.insert(dto);
    }
}
```

Explicação do Código

- **@PostMapping**: Indica que este método responde a requisições POST na URL especificada em `@RequestMapping`.
- **@RequestBody ProductDTO dto**: Este parâmetro indica que o método espera receber dados no corpo da requisição, e que esses dados devem ser convertidos automaticamente para um objeto `ProductDTO` pelo Spring.

Testando a Requisição no Postman

Para testar a criação de um novo produto:

1. **Abra o Postman** e crie uma nova requisição.
2. **Selecione o método POST** e insira a URL `http://localhost:8080/products`.
3. Na aba **Body**, selecione **raw** e escolha **JSON** como formato.
4. Insira os dados do novo produto conforme o exemplo abaixo:

```
{
    "name": "Meu produto",
    "description": "Descrição do produto",
    "imgUrl": "https://teste.com/produto.jpg",
    "price": 50
}
```

5. **Envie a requisição** e observe a resposta. Se tudo estiver configurado corretamente, você deverá receber o `ProductDTO` do produto recém-criado, incluindo seu ID gerado pelo banco de dados.

3.15 Customizando Resposta com ResponseEntity

Na construção de APIs REST com Spring Boot, o objeto `ResponseEntity` é uma ferramenta poderosa que permite customizar as respostas HTTP. Essa customização pode incluir o status da resposta, os headers e o corpo da mensagem. Vamos explorar como o `ResponseEntity` é utilizado na classe `ProductController` para aprimorar a comunicação entre o cliente e o servidor.

Classe ProductController Atualizada

A classe `ProductController` foi atualizada para usar `ResponseEntity` em cada um de seus métodos. Isso oferece um controle mais granular sobre a resposta HTTP enviada ao cliente, como veremos a seguir:

```
package com.devsuperior.dscommerce.controllers;

import java.net.URI;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.ResponseEntity;
```

```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
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 org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import com.devsuperior.dscommerce.dto.ProductDTO;
import com.devsuperior.dscommerce.services.ProductService;

@RestController
@RequestMapping(value = "/products")
public class ProductController {

    @Autowired
    private ProductService service;

    @GetMapping(value = "/{id}")
    public ResponseEntity<ProductDTO> findById(@PathVariable Long id) {
        ProductDTO dto = service.findById(id);
        return ResponseEntity.ok(dto);  // 200 OK with body
    }

    @GetMapping
    public ResponseEntity<Page<ProductDTO>> findAll(Pageable pageable) {
        Page<ProductDTO> dto = service.findAll(pageable);
        return ResponseEntity.ok(dto);  // 200 OK with paged body
    }

    @PostMapping
    public ResponseEntity<ProductDTO> insert(@RequestBody ProductDTO dto) {
        dto = service.insert(dto);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(dto.getId()).toUri();
        return ResponseEntity.created(uri).body(dto);  // 201 Created with
    location header
    }
}
```

Explicação do Código

- **@GetMapping(value = "/{id}")**:
  - Este método usa `ResponseEntity.ok(dto)` para retornar uma resposta HTTP 200 (OK), com o DTO do produto como corpo da resposta.
- **@GetMapping**:
  - Similar ao método anterior, mas aplicado à busca paginada. A resposta também é 200 (OK), mas inclui uma página de `ProductDTO` como corpo.
- **@PostMapping**:
  - O método `insert` é mais complexo e demonstra o poder do `ResponseEntity` para manipular respostas HTTP de forma eficaz:
    - **Criação do URI**: Após a inserção do produto, um `URI` é construído usando `ServletUriComponentsBuilder`. Este URI representa o endereço do novo recurso

criado. A função `fromCurrentRequest()` pega a URI da requisição corrente, `path ("/{id}")` adiciona o ID do produto ao caminho, e `buildAndExpand(dto.getId())` substitui `{id}` pelo ID real do produto.

* **ResponseEntity.created(uri)**: Retorna uma resposta com o status 201 (Created), que é ideal para operações de criação. O método `created` também configura o header 'Location' para o URI do novo recurso, conforme construído anteriormente. O corpo da resposta contém o `ProductDTO` do produto inserido.

Testando no Postman

Para testar a inserção de um novo produto:

1. **Configure o Postman**:
   - Método: POST
   - URL: `http://localhost:8080/products`
   - Na aba Body, selecione raw e JSON.
   - Corpo da requisição:

```
{
    "name": "Meu produto",
    "description": "Descrição do produto",
    "imgUrl": "https://teste.com/produto.jpg",
    "price": 50
}
```

2. **Envie a requisição**:
   - Ao enviar, o Postman deve mostrar uma resposta com status 201 (Created). O header 'Location' mostrará o URI do novo produto criado, e o corpo da resposta incluirá os detalhes do produto.

O uso de `ResponseEntity` no Spring Boot oferece um controle detalhado sobre as respostas HTTP, permitindo aos desenvolvedores configurar precisamente como os endpoints devem responder às requisições. Isso não apenas melhora a semântica das respostas da API, mas também enriquece a experiência do desenvolvedor e do usuário final ao interagir com a aplicação.

3.16 Atualizando Produto com PUT

Nesta seção, exploramos como atualizar um produto existente no banco de dados usando o método HTTP PUT. Esse processo envolve modificações tanto na camada de serviço quanto no controlador para garantir que as atualizações sejam aplicadas corretamente e de forma eficiente.

Atualizações no Código de ProductService

A classe `ProductService` é responsável por intermediar as operações entre o controlador e o repositório. Vamos detalhar as mudanças feitas para permitir a atualização de produtos.

```
@Service
public class ProductService {
```

```
    ...

    @Transactional
    public ProductDTO insert(ProductDTO dto) {
        Product entity = new Product();
        copyDtoToEntity(dto, entity);
        entity = repository.save(entity);
        return new ProductDTO(entity);
    }

    @Transactional
    public ProductDTO update(Long id, ProductDTO dto) {
        Product entity = repository.getReferenceById(id);
        copyDtoToEntity(dto, entity);
        entity = repository.save(entity);
        return new ProductDTO(entity);
    }

    private void copyDtoToEntity(ProductDTO dto, Product entity) {
        entity.setName(dto.getName());
        entity.setDescription(dto.getDescription());
        entity.setPrice(dto.getPrice());
        entity.setImgUrl(dto.getImgUrl());
    }
}
```

Explicação do Código

- **getReferenceById(id)**: Este método do JPA é usado para buscar uma referência à entidade `Product` com o ID fornecido sem necessariamente carregar todos os seus dados imediatamente. Isso é útil para operações de atualização onde você não precisa trabalhar com todos os dados da entidade, mas apenas modificar alguns atributos.
- **copyDtoToEntity(dto, entity)**: Este método auxiliar é criado para evitar a duplicação de código. Ele copia os dados de um `ProductDTO` para a entidade `Product`. Isso assegura que os campos da entidade sejam atualizados conforme os dados recebidos do DTO.

Método update na Classe ProductController

O controlador é atualizado para manipular requisições PUT, permitindo a atualização de produtos existentes.

```
@RestController
@RequestMapping(value = "/products")
public class ProductController {

    @PutMapping(value = "/{id}")
    public ResponseEntity<ProductDTO> update(@PathVariable Long id, @RequestBody
    ProductDTO dto) {
        dto = service.update(id, dto);
        return ResponseEntity.ok(dto);
    }
```

```
}
```

Explicação do Código

- **@PutMapping(value = "/{id}")**: Define que este método responderá a requisições PUT enviadas para `/products/{id}`, onde `{id}` é o ID do produto a ser atualizado.
- **ResponseEntity**: O método retorna um `ResponseEntity` contendo o DTO atualizado. O uso de `ResponseEntity.ok(dto)` encapsula o DTO atualizado em uma resposta HTTP 200 (OK), indicando que a atualização foi bem-sucedida.

Testando a Requisição no Postman

Para testar a atualização de um produto:

1. **Configure o Postman**:
   - Método: PUT
   - URL: `http://localhost:8080/products/{id}`, substitua `{id}` pelo ID do produto que deseja atualizar.
   - Na aba Body, selecione raw e JSON.
2. **Corpo da Requisição**:
   - Insira os dados atualizados para o produto. Por exemplo:

```
{
    "name": "Produto Atualizado",
    "description": "Nova descrição do produto",
    "price": 75.5,
    "imgUrl": "https://novaurl.com/produto.jpg"
}
```

3. **Envie a Requisição**:
   - O Postman deve mostrar uma resposta 200 OK com os detalhes do produto atualizado.

A implementação da funcionalidade de atualização é essencial para permitir que os usuários modifiquem dados existentes de maneira controlada e segura. O uso do método `getReferenceById` para otimizar o acesso ao banco de dados e a separação clara entre a lógica de transferência de dados e a lógica de negócios ajudam a manter o código organizado e eficiente.

3.17 Deletando Produto com DELETE

Nesta seção, vamos implementar a operação de deleção de um produto no banco de dados utilizando o método HTTP DELETE. Essa explicação será básica, pois mais adiante aprenderemos como tratar possíveis erros, como tentar excluir um produto inexistente ou um produto que esteja associado a um pedido.

Método delete na Classe ProductService

A responsabilidade da deleção de um produto no banco de dados recai sobre a camada de serviço, onde utilizamos o `ProductRepository` para executar essa operação.

```
@Service
public class ProductService {

    ...

    @Transactional
    public void delete(Long id) {
        repository.deleteById(id);
    }
}
```

Explicação do Código

- **@Transactional**: Garante que a operação de exclusão será executada dentro de uma transação. Se algum erro ocorrer durante a execução do método, a transação será revertida.
- **deleteById(id)**: Esse método do `JpaRepository` busca a entidade pelo ID fornecido e a exclui do banco de dados. Se o ID fornecido não existir, uma exceção `EmptyResultDataAccessException` será lançada. Essa exceção ainda não está sendo tratada, mas aprenderemos como lidar com ela em seções futuras.
- **Retorno `void`**: Como a deleção não retorna um objeto específico, o método foi definido com retorno `void`, apenas executando a remoção.

Método delete na Classe ProductController

No controlador, implementamos o endpoint que recebe a requisição DELETE e repassa a operação para o serviço.

```
@RestController
@RequestMapping(value = "/products")
public class ProductController {

    ...

    @DeleteMapping(value = "/{id}")
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        service.delete(id);
        return ResponseEntity.noContent().build();
    }
}
```

Explicação do Código

- **@DeleteMapping(value = "/{id}")**: Define que esse método responderá a requisições HTTP DELETE enviadas para a URL `/products/{id}`, onde `{id}` é o identificador do produto a ser excluído.
- **@PathVariable Long id**: Captura o ID do produto a ser deletado da URL.
- **service.delete(id)**: Chama o método `delete` do `ProductService`, que executa a operação de exclusão.
- **ResponseEntity**: O tipo de retorno `Void` indica que o corpo da resposta não conterá nenhum conteúdo, pois não há necessidade de retornar dados ao cliente após a exclusão.
- **ResponseEntity.noContent().build()**:
  - Retorna uma resposta HTTP com status **204 (No Content)**, que é o código de status recomendado para operações de deleção bem-sucedidas.
  - Esse status indica que a requisição foi processada corretamente, mas não há conteúdo na resposta.

Testando a Deleção no Postman

Para testar a operação de deleção de um produto no Postman:

1. **Configure a requisição**:
   - Método: **DELETE**
   - URL: `http://localhost:8080/products/{id}`
   - Substitua `{id}` pelo identificador do produto que deseja excluir.
2. **Envie a requisição**:
   - Se o produto existir e puder ser excluído, o Postman retornará o código **204 No Content** e um corpo vazio.
   - Se o ID informado não existir, um erro será gerado (mais adiante veremos como tratar esse erro).

3.18 Criando Exceções de Serviço Customizadas

Ao desenvolver APIs RESTful, é crucial implementar um sistema de tratamento de exceções eficaz. Sem um tratamento adequado, os erros não são comunicados de forma clara ao consumidor da API, resultando frequentemente em respostas de erro genéricas como 500 (Internal Server Error), que não fornecem informações úteis sobre o que realmente deu errado.

Códigos de Erro Comuns

Antes de implementarmos nossa própria classe de exceção, é útil entender alguns códigos de status HTTP comumente usados para comunicar diferentes tipos de erros:

- **400 Bad Request**: Indica que o servidor não pode ou não vai processar a requisição devido a algo que foi percebido como um erro do cliente (e.g., sintaxe da requisição malformada).
- **401 Unauthorized**: Falha na autenticação, quando é necessário que o usuário se autentique para obter a resposta.

- **403 Forbidden**: O servidor entendeu a requisição, mas se recusa a autorizá-la.
- **404 Not Found**: O recurso solicitado não foi encontrado mas poderá estar disponível no futuro.
- **409 Conflict**: Indica um conflito na requisição, como tentar criar um recurso duplicado.
- **415 Unsupported Media Type**: O tipo de mídia dos dados requisitados não é suportado pelo servidor.
- **422 Unprocessable Entity**: A requisição está bem formada, mas foi impossível segui-la devido a erros semânticos.

Implementação da Classe de Exceção Customizada

Para melhorar a maneira como nossa API lida com erros, vamos criar uma exceção customizada que será usada especificamente para situações onde um recurso não é encontrado:

```
package com.devsuperior.dscommerce.services.exceptions;

public class ResourceNotFoundException extends RuntimeException {

    public ResourceNotFoundException(String msg) {
        super(msg);
    }
}
```

Esta classe estende `RuntimeException`, permitindo que ela seja lançada sem exigir tratamento obrigatório (catch ou declaração throws).

Atualização do Método findById em ProductService

Com nossa exceção customizada pronta, podemos agora modificar o método `findById` em `ProductService` para usar essa nova exceção, melhorando a clareza da resposta em caso de falhas:

```
@Service
public class ProductService {

    ...

    @Transactional(readOnly = true)
    public ProductDTO findById(Long id) {
        Product product = repository.findById(id).orElseThrow(
            () -> new ResourceNotFoundException("Recurso não encontrado"));
        return new ProductDTO(product);
    }
}
```

Explicação do Código

- **orElseThrow()**: Este método é parte da classe `Optional` e é usado aqui para lançar uma `ResourceNotFoundException` caso o método `findById` não encontre o produto com o

ID fornecido. A mensagem “Recurso não encontrado” ajuda a identificar claramente o problema.

Testando a Implementação

Se o endpoint for testado neste momento com um ID que não existe, a API ainda retornará uma resposta de erro 500. No entanto, se você verificar o stack trace de erros no console da aplicação Spring Boot, poderá observar que a exceção `ResourceNotFoundException` está sendo lançada corretamente.

Esta seção preparou o terreno para um sistema robusto de tratamento de exceções. Implementamos uma exceção customizada para lidar com situações específicas de recursos não encontrados. No próximo passo, vamos aprender como capturar essa exceção dentro do controlador ou com um manipulador global de exceções para retornar respostas apropriadas e informativas aos usuários da API.

3.19 Tratando Exceção com Resposta Customizada

Quando se trata de APIs REST, um bom tratamento de exceções é crucial para a comunicação clara e eficiente dos erros ao consumidor da API. Nesta seção, vamos abordar como tratar exceções com respostas customizadas no Spring Boot, usando um `ControllerAdvice` para manipular exceções de forma centralizada.

Representação dos Dados de Erro

Primeiramente, vamos criar uma classe DTO chamada `CustomError` para padronizar a forma como os erros são apresentados ao consumidor da API. Esta classe contém informações como o timestamp do erro, o status HTTP, a mensagem de erro e o caminho da requisição que causou o erro.

```
package com.devsuperior.dscommerce.dto;

import java.time.Instant;

public class CustomError {

    private Instant timestamp;
    private Integer status;
    private String error;
    private String path;

    public CustomError(Instant timestamp, Integer status, String error, String
   path) {
        this.timestamp = timestamp;
        this.status = status;
        this.error = error;
        this.path = path;
    }

    // Getters
    public Instant getTimestamp() {
        return timestamp;
```

```
    }

    public Integer getStatus() {
        return status;
    }

    public String getError() {
        return error;
    }

    public String getPath() {
        return path;
    }
}
```

O que é um Controller Advice?

Um `ControllerAdvice` é uma anotação do Spring que permite definir um manipulador global de exceções para controladores em toda a aplicação. Isso ajuda a evitar a duplicação de código de tratamento de exceções e centraliza o controle de erros em um único local.

Implementação da Classe ControllerExceptionHandler

A classe `ControllerExceptionHandler` demonstra como podemos usar o `ControllerAdvice` para tratar exceções específicas de forma elegante e eficiente.

```
package com.devsuperior.dscommerce.controllers.handlers;

import com.devsuperior.dscommerce.dto.CustomError;
import com.devsuperior.dscommerce.services.exceptions.ResourceNotFoundException;
import jakarta.servlet.http.HttpServletRequest;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import java.time.Instant;

@ControllerAdvice
public class ControllerExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<CustomError> resourceNotFound(
   ResourceNotFoundException e, HttpServletRequest request) {
        HttpStatus status = HttpStatus.NOT_FOUND;
        CustomError err = new CustomError(Instant.now(), status.value(), e.
   getMessage(), request.getRequestURI());
        return ResponseEntity.status(status).body(err);
    }
}
```

Explicação do Código

- **@ControllerAdvice**: Indica que esta classe é um conselheiro de controladores, capaz de interceptar exceções lançadas por métodos anotados com `@RequestMapping` e similares.
- **@ExceptionHandler(ResourceNotFoundException.class)**: Especifica que o método `resourceNotFound` deve ser invocado para tratar exceções do tipo `ResourceNotFoundException` .
- **ResponseEntity**: Constrói a resposta para a exceção com o status HTTP apropriado e o corpo contendo um objeto `CustomError`.
- **CustomError Creation**: Um novo objeto `CustomError` é criado com o timestamp atual, o status HTTP, a mensagem de erro e o caminho da URI solicitada.

Testando a Implementação no Postman

Após implementar o `ControllerAdvice` e o método de tratamento para `ResourceNotFoundException` , se tentarmos acessar um produto por um ID inexistente, a API agora retornará uma resposta com o código de status 404 (Not Found) e um corpo de resposta contendo detalhes do erro.

1. **Configure a requisição no Postman**:
   - Método: GET
   - URL: `http://localhost:8080/products/{id}` (substitua `{id}` por um valor inexistente)
2. **Envie a requisição**:
   - A resposta deve incluir:
     - Status: 404 Not Found
     - Body:

```
{
    "timestamp": "2023-09-29T12:34:56.789Z",
    "status": 404,
    "error": "Recurso não encontrado",
    "path": "/products/{id}"
}
```

3.20 Implementando Outras Exceções

Nesta seção, abordaremos a implementação de tratamentos para exceções adicionais relacionadas ao banco de dados, expandindo o sistema de tratamento de erros de nossa API para garantir respostas adequadas e informativas aos consumidores da API.

Classe DatabaseException

A `DatabaseException` é utilizada para capturar e tratar exceções que ocorrem devido a operações de banco de dados que falham por razões de integridade de dados ou restrições do banco.

```
package com.devsuperior.dscommerce.services.exceptions;
```

```
public class DatabaseException extends RuntimeException {

    public DatabaseException(String msg) {
        super(msg);
    }
}
```

Essa exceção é derivada de `RuntimeException`, permitindo que seja lançada sem exigências de tratamento explícito (try-catch).

Atualização da Classe ControllerExceptionHandler

Incluímos um manipulador adicional em `ControllerExceptionHandler` para tratar `DatabaseException` , configurando a resposta para retornar um status HTTP 400 (Bad Request) quando essa exceção é capturada.

```
package com.devsuperior.dscommerce.controllers.handlers;

import com.devsuperior.dscommerce.dto.CustomError;
import com.devsuperior.dscommerce.services.exceptions.DatabaseException;
import com.devsuperior.dscommerce.services.exceptions.ResourceNotFoundException;
import jakarta.servlet.http.HttpServletRequest;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

import java.time.Instant;

@ControllerAdvice
public class ControllerExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<CustomError> resourceNotFound(
   ResourceNotFoundException e, HttpServletRequest request) {
        HttpStatus status = HttpStatus.NOT_FOUND;
        CustomError err = new CustomError(Instant.now(), status.value(), e.
   getMessage(), request.getRequestURI());
        return ResponseEntity.status(status).body(err);
    }

    @ExceptionHandler(DatabaseException.class)
    public ResponseEntity<CustomError> database(DatabaseException e,
   HttpServletRequest request) {
        HttpStatus status = HttpStatus.BAD_REQUEST;
        CustomError err = new CustomError(Instant.now(), status.value(), e.
   getMessage(), request.getRequestURI());
        return ResponseEntity.status(status).body(err);
    }
}
```

Atualização dos Métodos em ProductService

Ampliamos a funcionalidade de `ProductService` para lidar com as exceções customizadas durante as operações de CRUD.

```
@Service
public class ProductService {

    ...

    @Transactional(readOnly = true)
    public ProductDTO findById(Long id) {
        Product product = repository.findById(id).orElseThrow(
                () -> new ResourceNotFoundException("Recurso não encontrado"));
        return new ProductDTO(product);
    }

    @Transactional
    public ProductDTO update(Long id, ProductDTO dto) {
        try {
            Product entity = repository.getReferenceById(id);
            copyDtoToEntity(dto, entity);
            entity = repository.save(entity);
            return new ProductDTO(entity);
        }
        catch (EntityNotFoundException e) {
            throw new ResourceNotFoundException("Recurso não encontrado");
        }
    }

    @Transactional(propagation = Propagation.SUPPORTS)
    public void delete(Long id) {
        if (!repository.existsById(id)) {
            throw new ResourceNotFoundException("Recurso não encontrado");
        }
        try {
            repository.deleteById(id);
        } catch (DataIntegrityViolationException e) {
            throw new DatabaseException("Falha de integridade referencial");
        }
    }

    ...
}
```

Explicações Detalhadas

- **`@Transactional(propagation = Propagation.SUPPORTS)`**: Esta configuração é usada no método `delete` para indicar que o método não necessariamente precisa ser executado dentro de uma transação, mas pode participar de uma se já existir.
- **Tratamento de Erros**:
  - `ResourceNotFoundException`: Lançada quando um produto não é encontrado, seja na busca por ID ou antes de uma exclusão.

– `DatabaseException`: Lançada em caso de violação de integridade, como tentar deletar um produto que está vinculado a pedidos.

Testando Cenários de Exceções

Instruímos o leitor a testar os seguintes cenários para verificar o comportamento da API:

1. **Buscar produto por id inexistente**: Deverá retornar 404 Not Found.
2. **Atualizar produto com id inexistente**: Também resultará em 404 Not Found.
3. **Deletar produto com id inexistente**: Retorna 404 Not Found.
4. **Deletar produto que faz parte de um pedido**: Retorna 400 Bad Request com uma mensagem explicativa sobre falha de integridade referencial.

3.21 Validação com Bean Validation

Introdução ao Bean Validation

Bean Validation é uma especificação Java que fornece um mecanismo padrão para validar dados em modelos Java, conhecidos como JavaBeans. Ela define uma série de anotações e uma API para configurar e executar validações de maneira declarativa, evitando a necessidade de implementar manualmente códigos de validação espalhados pelo aplicativo. Com Bean Validation, as regras de validação são definidas diretamente nos atributos das classes de modelo usando anotações, o que facilita a manutenção e garante uma maior consistência no tratamento de dados em toda a aplicação.

Adicionando a Dependência do Bean Validation

Para usar o Bean Validation em um projeto Spring Boot, é necessário incluir a dependência `spring-boot-starter-validation` no arquivo `pom.xml`. Isso pode ser feito adicionando o seguinte bloco de dependência:

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

Ao salvar o arquivo `pom.xml`, a IDE (Spring Tools ou IntelliJ, por exemplo) reconhece a nova dependência e automaticamente realiza o download das bibliotecas necessárias.

Anotações de Validação na Classe ProductDTO

A validação no nível do DTO é crucial para garantir que os dados recebidos pela API estejam de acordo com as regras de negócio antes de qualquer processamento ou persistência. Abaixo, detalhamos as anotações usadas na classe `ProductDTO` para implementar as validações especificadas:

```
package com.devsuperior.dscommerce.dto;
```

```
import com.devsuperior.dscommerce.entities.Product;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Positive;
import jakarta.validation.constraints.Size;

public class ProductDTO {

    private Long id;

    @Size(min = 3, max = 80, message = "Nome precisar ter de 3 a 80 caracteres")
    @NotBlank(message = "Campo requerido")
    private String name;

    @Size(min = 10, message = "Descrição precisa ter no mínimo 10 caracteres")
    @NotBlank(message = "Campo requerido")
    private String description;

    @NotNull(message = "Campo requerido")
    @Positive(message = "O preço deve ser positivo")
    private Double price;

    private String imgUrl;

    ...
```

Explicação das Anotações

- **@NotBlank**: Garante que o campo não seja nulo e que o comprimento da string seja maior que zero, excluindo espaços em branco.
- **@Size**: Define os limites mínimo e máximo para o comprimento de uma string. Utilizado aqui para garantir que o nome e a descrição tenham os tamanhos adequados conforme o documento de requisitos.
- **@Positive**: Assegura que o valor do campo seja um número positivo, adequado para representar valores como preço.

Validação nos Métodos do Controlador

Para que as validações sejam efetivamente aplicadas, é necessário incluir a anotação `@Valid` nos parâmetros dos métodos nos controladores que recebem o `ProductDTO`:

```
@RestController
@RequestMapping(value = "/products")
public class ProductController {

    ...

    @PostMapping
    public ResponseEntity<ProductDTO> insert(@Valid @RequestBody ProductDTO dto)
    {
        dto = service.insert(dto);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(dto.getId()).toUri();
```

```
        return ResponseEntity.created(uri).body(dto);
    }

    @PutMapping(value = "/{id}")
    public ResponseEntity<ProductDTO> update(@PathVariable Long id, @Valid
   @RequestBody ProductDTO dto) {
        dto = service.update(id, dto);
        return ResponseEntity.ok(dto);
    }

    ...
}
```

Testando a Validação

Se o projeto for testado agora, com as validações configuradas, qualquer violação das regras de validação resultará em uma resposta com erro, mas as mensagens específicas dos erros de validação ainda não serão apresentadas de forma clara. O tratamento e a customização das mensagens de erro serão abordados na próxima seção, permitindo uma melhor comunicação dos problemas aos consumidores da API.

3.22 Customizando a Resposta da Validação

Introdução

Para aprimorar o tratamento de erros em nossa API, introduziremos uma customização mais detalhada das mensagens de erro de validação. Isso proporcionará feedback mais claro e direcionado aos consumidores da API, indicando exatamente o que está errado com os dados enviados.

Definição da Classe FieldMessage

A classe `FieldMessage` será usada para representar detalhadamente os erros de validação por campo, especificando qual campo gerou o erro e qual a mensagem de erro associada.

```
package com.devsuperior.dscommerce.dto;

public class FieldMessage {

    private String fieldName;
    private String message;

    public FieldMessage(String fieldName, String message) {
        this.fieldName = fieldName;
        this.message = message;
    }

    public String getFieldName() {
        return fieldName;
    }
```

```
    public String getMessage() {
        return message;
    }
}
```

Classe ValidationError

A `ValidationError` é uma extensão da classe `CustomError`, que além das informações básicas de erro, inclui uma lista de `FieldMessage`, permitindo representar múltiplos erros de validação que podem ocorrer durante o processamento de uma única requisição.

```
package com.devsuperior.dscommerce.dto;

import java.time.Instant;
import java.util.ArrayList;
import java.util.List;

public class ValidationError extends CustomError {

    private List<FieldMessage> errors = new ArrayList<>();

    public ValidationError(Instant timestamp, Integer status, String error,
    String path) {
        super(timestamp, status, error, path);
    }

    public List<FieldMessage> getErrors() {
        return errors;
    }

    public void addError(String fieldName, String message) {
        errors.add(new FieldMessage(fieldName, message));
    }
}
```

Atualização da Classe ControllerExceptionHandler

Na classe `ControllerExceptionHandler`, implementamos um tratamento especializado para a exceção `MethodArgumentNotValidException`, que é disparada quando ocorrem falhas de validação definidas pelas anotações do Bean Validation no DTO. Vamos detalhar como essa exceção é tratada para transformar erros técnicos em mensagens compreensíveis para os usuários da API.

```
package com.devsuperior.dscommerce.controllers.handlers;

import com.devsuperior.dscommerce.dto.CustomError;
import com.devsuperior.dscommerce.dto.ValidationError;
import com.devsuperior.dscommerce.services.exceptions.DatabaseException;
import com.devsuperior.dscommerce.services.exceptions.ResourceNotFoundException;
import jakarta.servlet.http.HttpServletRequest;
import org.springframework.http.HttpStatus;
```

```java
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

import java.time.Instant;

@ControllerAdvice
public class ControllerExceptionHandler {

    ...

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<CustomError> methodArgumentNotValidation(
   MethodArgumentNotValidException e, HttpServletRequest request) {
        HttpStatus status = HttpStatus.UNPROCESSABLE_ENTITY;
        ValidationError err = new ValidationError(Instant.now(), status.value(),
    "Dados inválidos", request.getRequestURI());

        for (FieldError f : e.getBindingResult().getFieldErrors()) {
            err.addError(f.getField(), f.getDefaultMessage());
        }

        return ResponseEntity.status(status).body(err);
    }
}
```

Explicação do Código

- **@ExceptionHandler(MethodArgumentNotValidException.class)**: Esta anotação define que o método `methodArgumentNotValidation` deve ser chamado quando uma exceção do tipo `MethodArgumentNotValidException` ocorrer. Essa exceção é comumente lançada pelo Spring quando a validação de um objeto falha no nível do argumento de método anotado com `@Valid`.
- **HttpStatus.UNPROCESSABLE_ENTITY (422)**: Utilizamos este status HTTP para indicar que a requisição foi bem formada mas não pôde ser seguida devido a erros semânticos, ou seja, dados inválidos fornecidos pelo cliente.
- **Percorrendo os Erros de Campo**: Usamos `e.getBindingResult().getFieldErrors()` para obter uma lista de `FieldError`, onde cada `FieldError` representa um erro específico de validação que ocorreu. Estes são adicionados à resposta de erro através do método `addError`.

Testando as Validações

Encorajamos o leitor a testar os vários cenários possíveis que podem gerar erros de validação, utilizando a classe `ProductDTO` que definimos anteriormente. Isso inclui:

- Enviar um `name` com menos de 3 caracteres ou mais de 80.
- Enviar uma `description` com menos de 10 caracteres.
- Definir um `price` negativo.

Cada um desses testes deve resultar em uma resposta com código 422 (Unprocessable Entity), contendo detalhes específicos sobre cada erro de validação encontrado. Essa abordagem melhora significativamente a usabilidade e a robustez da API, fornecendo aos desenvolvedores informações precisas para corrigir os dados enviados.

Capítulo 4

JPA, consultas SQL e JPQL

4.1 Visão geral do capítulo

Bem-vindos ao capítulo "JPA, consultas SQL e JPQL". Uma habilidade essencial para a atividade profissional de um desenvolvedor backend é a realização de consultas ao banco de dados relacional.

A forma mais clássica de se realizar consultas a um banco de dados relacional é por meio da linguagem SQL, que fornece uma sintaxe simples e elegante para recuperar informações do banco de dados. Este é um fundamento importante que os estudantes de programação aprendem geralmente já no início de sua jornada.

Além disso, as linguagens modernas geralmente oferecem recursos adicionais para facilitar a consulta aos bancos de dados e instanciar objetos com esses dados. Esses recursos incluem linguagens específicas para consultas, como é o caso de Java com sua JPA - Java Persistence API, que oferece a JPQL, que é uma linguagem de consulta parecida com a SQL, porém com uma sintaxe mais próxima do modelo de objetos.

Neste capítulo vamos passar alguns dos fundamentos mais importantes da JPA, e também vamos explorar vários exercícios e recursos para que você possa aprender em profundidade como fazer consultas ao banco de dados usando Java e Spring. Vamos resolver exercícios tanto em SQL como em JPQL, para você possa adquirir uma base de conhecimento em ambas tecnologias. Vamos começar!

4.2 Material de apoio do capítulo

Para este capítulo, vamos utilizar os seguintes arquivos contidos no material de apoio anexo:

- 04 JPA, consultas SQL e JPQL (slides).pdf
- 04 DESAFIO Consulta vendas.pdf
- Código fonte dos exercícios SQL e JPQL
- Projeto DSCommerce ao final do capítulo 4

4.3 Sessão JPA e Estados de Entidades

Gerenciamento de Entidades com JPA

Java Persistence API (JPA) é uma especificação padrão que gerencia o relacionamento entre objetos Java e dados de um banco de dados relacional. A JPA facilita o gerenciamento de entidades em uma aplicação Java, abstraindo muitos dos detalhes complexos associados ao acesso direto ao banco de dados.

Sessão JPA

Uma sessão JPA é o contexto durante o qual a JPA gerencia as entidades enquanto interage com o banco de dados. Esse gerenciamento ocorre através de uma unidade de persistência chamada `EntityManager`, que é responsável por gerenciar o ciclo de vida das entidades, realizar operações de banco de dados e manter o contexto de persistência.

- **EntityManager**: Este objeto central na JPA encapsula uma conexão com o banco de dados e mantém o estado das entidades que estão sendo gerenciadas. As operações como persistir, remover, ou buscar entidades são feitas através do `EntityManager`.

Uso da JPA e Spring Data JPA

Embora a JPA possa ser usada diretamente, frameworks como o Spring simplificam ainda mais o gerenciamento de entidades com o Spring Data JPA, que abstrai muitas das operações comuns de `EntityManager` por meio de interfaces de repositório.

Exemplo de Uso Direto da JPA:

```
EntityManagerFactory emf = ...
EntityManager em = emf.createEntityManager();

Product prod = new ...
em.getTransaction().begin();
em.persist(prod);
em.getTransaction().commit();
```

Neste exemplo, um `EntityManager` é criado e usado para gerenciar uma transação onde um novo produto é persistido no banco de dados.

Exemplo com Spring Data JPA:

```
@Autowired
private ProductRepository repository;

@Transactional
public void meuMetodo() {
    Product prod = new ...
    repository.save(prod);
}
```

Aqui, o `ProductRepository` abstrai a complexidade do gerenciamento direto do `EntityManager`, utilizando anotações como `@Transactional` para controlar as transações.

Ciclo de Vida das Entidades JPA

Entidades em JPA passam por vários estados durante seu ciclo de vida, e o `EntityManager` é responsável por gerenciar esses estados:

- **Transient**: A entidade está apenas na memória e ainda não está associada a uma sessão de persistência. Qualquer objeto recém-criado que ainda não foi persistido está neste estado.
- **Managed**: A entidade está associada a uma sessão JPA e qualquer alteração feita a ela será sincronizada com o banco de dados na conclusão da transação. As entidades entram nesse estado após serem persistidas ou recuperadas da base de dados.
- **Detached**: A entidade foi persistida ou recuperada em uma sessão, mas agora a sessão foi fechada. Ela não está mais sob o gerenciamento do `EntityManager` atual.
- **Removed**: A entidade está marcada para remoção. Uma vez que a transação é confirmada, a entidade será removida do banco de dados.

Cada estado é crucial para entender como as operações sobre as entidades afetarão a base de dados e como as alterações no estado dos objetos são sincronizadas entre a aplicação e o banco de dados.

Entender o gerenciamento de entidades e o ciclo de vida das entidades em JPA é fundamental para desenvolver aplicações robustas e eficientes que interagem com bancos de dados. O uso do Spring Data JPA pode simplificar significativamente esse gerenciamento, permitindo que os desenvolvedores se concentrem mais na lógica do negócio enquanto o framework cuida da persistência dos dados.

4.4 Salvando entidade associada para um PARTE 1

Nesta seção vamos iniciar um estudo de caso no qual precisamos salvar, usando uma única requisição, um pessoa e seu respectivo departamento.

Nosso objetivo aqui é salvar, usando uma única requisição, um pessoa e seu respectivo departamento, presumindo que o departamento já esteja previamente cadastrado.

Do ponto de vista da requisição web, podemos defini-la de duas formas: nossa requisição pode passar os dados via objeto Json ou com um objeto Department aninhado, ou com o id do departamento:

Objeto Json com departamento aninhado

POST http://localhost:8080/people

```
{
    "name": "Nova Pessoa",
    "salary": 8000.0,
    "department": {
        "id": 1
```

```
    }
}
```

Objeto Json apenas com id do departamento

POST http://localhost:8080/people

```
{
    "name": "Nova Pessoa",
    "salary": 8000.0,
    "departmentId": 1
}
```

Para isto, vamos utilizar um projeto Spring Boot iniciado, o qual está disponível no link a seguir. Caso queira seguir o passo a passo desta seção, por favor abra o projeto no seu computador:

https://github.com/devsuperior/aula-salvar-para-um

Observe que no projeto temos duas entidades `Person` e `Department`, onde uma pessoa está associada com um departamento.

Classe Department

```
package com.devsuperior.aula.entities;

import jakarta.persistence.*;

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

@Entity
@Table(name = "tb_department")
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @OneToMany(mappedBy = "department")
    private List<Person> people = new ArrayList<>();

    public Department() {
    }

    public Department(Long id, String name) {
        this.id = id;
        this.name = name;
    }

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

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Person> getPeople() {
        return people;
    }
}
```

Classe Person

```
package com.devsuperior.aula.entities;

import jakarta.persistence.*;

@Entity
@Table(name = "tb_person")
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private Double salary;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    public Person() {
    }

    public Person(Long id, String name, Double salary, Department department) {
        this.id = id;
        this.name = name;
        this.salary = salary;
        this.department = department;
    }

    public Long getId() {
        return id;
    }

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

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getSalary() {
        return salary;
    }

    public void setSalary(Double salary) {
        this.salary = salary;
    }

    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {
        this.department = department;
    }
}
```

Implementando DTO para departamento aninhado

Para permitir o envio de um Json com o departamento aninhado, precisamos criar os DTOs correspondentes:

Classe DepartmentDTO

```
package com.devsuperior.aula.dto;

import com.devsuperior.aula.entities.Department;

public class DepartmentDTO {

    private Long id;
    private String name;

    public DepartmentDTO(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public DepartmentDTO(Department entity) {
        id = entity.getId();
        name = entity.getName();
    }

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

```
    public String getName() {
        return name;
    }
}
```

Classe PersonDepartmentDTO

```
package com.devsuperior.aula.dto;

import com.devsuperior.aula.entities.Person;

public class PersonDepartmentDTO {

    private Long id;
    private String name;
    private Double salary;

    private DepartmentDTO department;

    public PersonDepartmentDTO(Long id, String name, Double salary,
   DepartmentDTO department) {
        this.id = id;
        this.name = name;
        this.salary = salary;
        this.department = department;
    }

    public PersonDepartmentDTO(Person entity) {
        id = entity.getId();
        name = entity.getName();
        salary = entity.getSalary();
        department = new DepartmentDTO(entity.getDepartment());
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public Double getSalary() {
        return salary;
    }

    public DepartmentDTO getDepartment() {
        return department;
    }
}
```

Implementando DTO para pessoa com id do departamento

Para permitir o envio de um Json com os dados da pessoa e o id do departamento apenas, precisamos criar um DTO correspondente.

Classe PersonDTO

```
package com.devsuperior.aula.dto;

import com.devsuperior.aula.entities.Person;

public class PersonDTO {

    private Long id;
    private String name;
    private Double salary;
    private Long departmentId;

    public PersonDTO(Long id, String name, Double salary, Long departmentId) {
        this.id = id;
        this.name = name;
        this.salary = salary;
        this.departmentId = departmentId;
    }

    public PersonDTO(Person entity) {
        id = entity.getId();
        name = entity.getName();
        salary = entity.getSalary();
        departmentId = entity.getDepartment().getId();
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public Double getSalary() {
        return salary;
    }

    public Long getDepartmentId() {
        return departmentId;
    }
}
```

Nas próximas seções vamos prosseguir com nosso estudo de caso, implementando os códigos restantes para atingir nosso objetivo.

4.5 Salvando entidade associada para um PARTE 2

Dando prosseguimento ao nosso estudo de caso, agora vamos mostrar a implementação da operação responsável por salvar uma pessoa e seu respectivo departamento, considerando o caso em que informamos na requisição um objeto Json com os dados da pessoa e um objeto aninhado para o departamento.

Classe PersonService

Vamos começar com a implementação da operação `insert` na classe `PersonService`. Caso seja necessário, revise a implementação da classe `PersonDepartmentDTO` para compreender sua estrutura.

```
package com.devsuperior.aula.services;

import com.devsuperior.aula.dto.PersonDTO;
import com.devsuperior.aula.dto.PersonDepartmentDTO;
import com.devsuperior.aula.entities.Department;
import com.devsuperior.aula.entities.Person;
import com.devsuperior.aula.repositories.DepartmentRepository;
import com.devsuperior.aula.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class PersonService {

    @Autowired
    private PersonRepository repository;

    @Autowired
    private DepartmentRepository departmentRepository;

    public PersonDepartmentDTO insert(PersonDepartmentDTO dto) {

        Person entity = new Person();
        entity.setName(dto.getName());
        entity.setSalary(dto.getSalary());

        Department dept = new Department();
        dept.setId(dto.getDepartment().getId());

        entity.setDepartment(dept);

        entity = repository.save(entity);

        return new PersonDepartmentDTO(entity);
    }
}
```

Aqui é importante notar que primeiro precisamos preparar os objetos `Person` e `Department`, devidamente associados, antes de chamar o método `save` do objeto `repository`. Repare que nós instanciamos um objeto `Department` usando a cláusula `new`, e depois definimos o id deste

departamento como sendo o id do objeto aninhado:

```
Department dept = new Department();
dept.setId(dto.getDepartment().getId());
```

Classe PersonController

Para disponibilizar um endpoint para inserir uma pessoa e seu respectivo departamento, vamos implementar uma classe controladora `PersonController`, de forma similar à que já aprendemos anteriormente.

```
package com.devsuperior.aula.controllers;

import com.devsuperior.aula.dto.PersonDTO;
import com.devsuperior.aula.dto.PersonDepartmentDTO;
import com.devsuperior.aula.services.PersonService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
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 org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.net.URI;

@RestController
@RequestMapping(value = "/people")
public class PersonController {

    @Autowired
    private PersonService service;

    @PostMapping
    public ResponseEntity<PersonDepartmentDTO> insert(@RequestBody
   PersonDepartmentDTO dto) {
        dto = service.insert(dto);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(dto.getId()).toUri();
        return ResponseEntity.created(uri).body(dto);
    }
}
```

Testando a requisição

Para testar esta requisição, devemos preparar no Postman uma requisição POST no caminho `http://localhost:8080/people`, com o seguinte corpo Json:

```
{
    "name": "Nova Pessoa",
```

```
    "salary": 8000.0,
    "department": {
        "id": 1
    }
}
```

4.6 Salvando entidade associada para um PARTE 3

Para finalizar nosso estudo de caso, agora vamos mostrar a implementação da operação responsável por salvar uma pessoa e seu respectivo departamento, considerando o caso em que informamos na requisição um objeto Json com os dados da pessoa, mais o id do departamento.

Classe PersonService

Aqui temos o código do novo método na classe `PersonService`. Caso seja necessário, revise a implementação da classe `PersonDTO` para compreender sua estrutura.

```
@Service
public class PersonService {

    ...

    public PersonDTO insert(PersonDTO dto) {

        Person entity = new Person();
        entity.setName(dto.getName());
        entity.setSalary(dto.getSalary());

        Department dept = new Department();
        dept.setId(dto.getDepartmentId());

        entity.setDepartment(dept);

        entity = repository.save(entity);

        return new PersonDTO(entity);
    }
}
```

Novamente, aqui é importante notar que primeiro precisamos preparar os objetos `Person` e `Department`, devidamente associados, antes de chamar o método `save` do objeto `repository`. Repare que nós instanciamos um objeto `Department` usando a cláusula `new`, e depois definimos o id deste departamento como sendo o id do departamento, representado pelo campo `departmentId`:

```
Department dept = new Department();
dept.setId(dto.getDepartmentId());
```

Classe PersonController

O método que implementa o endpoint é praticamente o mesmo anterior, com a diferença que agora o parâmetro é do tipo `PersonDTO`.

```
package com.devsuperior.aula.controllers;

import com.devsuperior.aula.dto.PersonDTO;
import com.devsuperior.aula.dto.PersonDepartmentDTO;
import com.devsuperior.aula.services.PersonService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
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 org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.net.URI;

@RestController
@RequestMapping(value = "/people")
public class PersonController {

    @Autowired
    private PersonService service;

    @PostMapping
    public ResponseEntity<PersonDTO> insert(@RequestBody PersonDTO dto) {
        dto = service.insert(dto);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(dto.getId()).toUri();
        return ResponseEntity.created(uri).body(dto);
    }
}
```

Testando a requisição

Para testar esta requisição, devemos preparar no Postman uma requisição POST no caminho `http://localhost:8080/people`, com o seguinte corpo Json:

```
{
    "name": "Nova Pessoa",
    "salary": 8000.0,
    "departmentId": 1
}
```

4.7 Salvando entidades associadas para muitos

Outra operação muito comum que o desenvolvedor backend precisa dominar, é quando se deseja salvar objetos associados por meio de um relacionamento "para muitos".

Para compreender este tópico, vamos usar um estudo de caso de um sistema de produtos e categorias. O objetivo aqui é salvar, em uma única requisição, um produto e suas respectivas categorias, presumindo que as categorias já estejam previamente cadastradas.

No corpo da requisição web, vamos passar os dados do produto, juntamente com uma lista de objetos com o id de cada categoria:

POST http://localhost:8080/products

```
{
    "name": "Produto novo",
    "price": 1000.0,
    "categories": [
        {
            "id": 2
        },
        {
            "id": 3
        }
    ]
}
```

Vamos utilizar um projeto Spring Boot iniciado, o qual está disponível no link a seguir. Caso queira seguir o passo a passo desta seção, por favor abra o projeto no seu computador:

https://github.com/devsuperior/aula-salvar-para-muitos

Observe que no projeto temos duas entidades `Product` e `Category`, onde um produto está associado a várias categorias.

Classe Product

```
package com.devsuperior.aula.entities;

import jakarta.persistence.*;

import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "tb_product")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private Double price;
```

```
    @ManyToMany
    @JoinTable(name = "tb_product_category",
            joinColumns = @JoinColumn(name = "product_id"),
            inverseJoinColumns = @JoinColumn(name = "category_id"))
    private Set<Category> categories = new HashSet<>();

    public Product() {
    }

    public Product(Long id, String name, Double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public Set<Category> getCategories() {
        return categories;
    }
}
```

Classe Category

```
package com.devsuperior.aula.entities;

import jakarta.persistence.*;

import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "tb_category")
public class Category {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @ManyToMany(mappedBy = "categories")
    private Set<Product> products = new HashSet<>();

    public Category() {
    }

    public Category(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Set<Product> getProducts() {
        return products;
    }
}
```

Classes DTO

Para representar os dados que serão trafegados na requisição, vamos criar um tipo `CategoryDTO` e `ProductDTO`, de modo que cada objeto tipo `ProductDTO` esteja associado com vários objetos tipo `CategoryDTO`. Vamos já criar também o construtor de `ProductDTO` para instanciar um objeto a partir de uma entidade tipo `Product`.

Classe CategoryDTO

```
package com.devsuperior.aula.dto;

import com.devsuperior.aula.entities.Category;

public class CategoryDTO {

    private Long id;
```

```
    private String name;

    public CategoryDTO() {
    }

    public CategoryDTO(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public CategoryDTO(Category entity) {
        id = entity.getId();
        name = entity.getName();
    }

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Classe ProductDTO

```
package com.devsuperior.aula.dto;

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

import com.devsuperior.aula.entities.Category;
import com.devsuperior.aula.entities.Product;

public class ProductDTO {

    private Long id;
    private String name;
    private Double price;

    private List<CategoryDTO> categories = new ArrayList<>();

    public ProductDTO() {
    }

    public ProductDTO(Long id, String name, Double price) {
        this.id = id;
        this.name = name;
```

```
        this.price = price;
    }

    public ProductDTO(Product entity) {
        id = entity.getId();
        name = entity.getName();
        price = entity.getPrice();
        for (Category cat : entity.getCategories()) {
            categories.add(new CategoryDTO(cat));
        }
    }

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public List<CategoryDTO> getCategories() {
        return categories;
    }
}
```

Classe ProductService

Agora que já temos a representação DTO de um produto associado com suas categorias, vamos implementar a classe `ProductService`, que será responsável por preparar os objetos das entidades `Product` e `Category`, devidamente associados, para que depois sejam salvos no banco de dados.

```
package com.devsuperior.aula.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.devsuperior.aula.dto.CategoryDTO;
import com.devsuperior.aula.dto.ProductDTO;
```

```
import com.devsuperior.aula.entities.Category;
import com.devsuperior.aula.entities.Product;
import com.devsuperior.aula.repositories.ProductRepository;

@Service
public class ProductService {

    @Autowired
    private ProductRepository repository;

    public ProductDTO insert(ProductDTO dto) {

        Product entity = new Product();

        entity.setName(dto.getName());
        entity.setPrice(dto.getPrice());

        for (CategoryDTO catDto : dto.getCategories()) {
            Category cat = new Category();
            cat.setId(catDto.getId());
            entity.getCategories().add(cat);
        }

        entity = repository.save(entity);

        return new ProductDTO(entity);
    }
}
```

Classe ProductController

Para disponibilizar um endpoint em nossa API, vamos implementar a classe `ProductController`.

```
package com.devsuperior.aula.controllers;

import java.net.URI;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
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 org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import com.devsuperior.aula.dto.ProductDTO;
import com.devsuperior.aula.services.ProductService;

@RestController
@RequestMapping(value = "/products")
public class ProductController {

    @Autowired
    private ProductService service;
```

```
    @PostMapping
    public ResponseEntity<ProductDTO> insert(@RequestBody ProductDTO dto) {
        dto = service.insert(dto);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(dto.getId()).toUri();
        return ResponseEntity.created(uri).body(dto);
    }
}
```

Testando a requisição

Para testar esta requisição, devemos preparar no Postman uma requisição POST no caminho `http://localhost:8080/products`, com o seguinte corpo Json:

```
{
    "name": "Produto novo",
    "price": 1000.0,
    "categories": [
        {
            "id": 2
        },
        {
            "id": 3
        }
    ]
}
```

4.8 Evitando Degradação de Performance

No desenvolvimento de aplicações que utilizam Java Persistence API (JPA), a performance é um aspecto crítico que necessita atenção especial. Um dos principais desafios ao usar JPA é evitar a degradação de performance, que pode ser causada por múltiplas idas e vindas ao banco de dados devido ao carregamento de entidades associadas.

O carregamento lazy (preguiçoso) é uma estratégia de fetch que carrega as entidades associadas sob demanda. Por exemplo, se temos uma entidade `Order` com uma coleção de `Product`, o carregamento lazy não irá buscar os produtos até que o código acesse explicitamente a lista de produtos. Embora isso possa ser útil para economizar recursos, frequentemente leva a um problema de se realizar várias consultas ao banco de dados, degradando significativamente a performance.

Projeto Exemplo

Para demonstrar as práticas de otimização de performance na prática, disponibilizamos um projeto exemplo que será explorado nas próximas seções. Este projeto ilustra diversos cenários de uso da JPA e apresenta soluções para os problemas comuns de performance relacionados ao carregamento lazy. O projeto está disponível no GitHub e pode ser baixado a partir do seguinte link:

https://github.com/devsuperior/aula-lazy

4.9 Carregamento EAGER e LAZY

Introdução ao Carregamento de Dados na JPA

Java Persistence API (JPA) oferece flexibilidade no gerenciamento de como e quando as entidades associadas são carregadas do banco de dados através dos conceitos de carregamento EAGER (ansioso) e LAZY (preguiçoso). A escolha entre esses dois modos impacta diretamente a performance e o comportamento da aplicação.

Carregamento Padrão de Entidades Associadas

Em JPA, o tipo de carregamento de uma entidade associada pode afetar significativamente como as consultas ao banco de dados são realizadas:

- **Relacionamento para-um (OneToOne, ManyToOne)**: O padrão é EAGER. Isso significa que a entidade associada é carregada imediatamente quando a entidade principal é carregada, independentemente de a entidade associada ser utilizada ou não.
- **Relacionamento para-muitos (OneToMany, ManyToMany)**: O padrão é LAZY. Aqui, as entidades associadas não são carregadas quando a entidade principal é carregada. Elas só são buscadas do banco de dados quando são explicitamente acessadas pela primeira vez no código.

Comportamento do Carregamento Lazy

Quando um relacionamento está configurado para carregamento lazy, o acesso às entidades associadas enquanto a sessão JPA estiver ativa pode desencadear consultas adicionais ao banco de dados. Isso ocorre porque a JPA precisa buscar as informações da entidade associada que não foram inicialmente carregadas. Este comportamento, apesar de útil para economizar recursos, pode levar ao problema das consultas N+1 se não for bem gerenciado.

Alterando o Comportamento de Carregamento

Embora os comportamentos padrão de carregamento sejam geralmente adequados, há situações em que podem ser necessárias alterações para melhorar a performance ou adequar às necessidades específicas da aplicação. Aqui estão algumas formas de modificar o comportamento de carregamento:

- **Atributo Fetch no Relacionamento da Entidade**: Pode-se especificar explicitamente o tipo de carregamento (EAGER ou LAZY) no mapeamento da entidade usando o atributo `fetch`. Por exemplo:

```
@ManyToOne(fetch = FetchType.LAZY)
private Department department;
```

No entanto, alterar o padrão pode gerar comportamentos inesperados, como carregamentos excessivos ou inadequados, dependendo do contexto da operação.

- **Cláusula JOIN FETCH**: Utilizada em consultas JPQL para especificar que entidades associadas devem ser carregadas juntamente com a entidade principal para evitar consultas subsequentes:

```
SELECT p FROM Product p JOIN FETCH p.category
```

  Esta abordagem é eficaz para evitar o problema de N+1 em casos específicos onde sabemos que vamos precisar das entidades associadas imediatamente após a carga da entidade principal.

- **Consultas Customizadas**: Criar consultas específicas que buscam exatamente o que é necessário pode ser a melhor solução, pois permite um controle total sobre o que é carregado e quando. Utilizar o JPQL ou o Criteria API para definir essas consultas garante que apenas os dados necessários sejam recuperados, otimizando a performance.

A compreensão profunda do carregamento EAGER e LAZY e de como manipular esses comportamentos é crucial para desenvolver aplicações eficientes com JPA. A escolha entre EAGER e LAZY deve ser feita com base nas necessidades específicas de cada caso de uso.

4.10 Analisando o Carregamento Lazy dos Funcionários

O carregamento lazy (preguiçoso) é uma técnica usada pela Java Persistence API (JPA) para otimizar o acesso ao banco de dados, carregando dados sob demanda. Isso significa que os dados associados a uma entidade não são carregados imediatamente com a entidade principal, mas sim quando são explicitamente acessados pela primeira vez no código. Essa abordagem pode reduzir o uso desnecessário de recursos, mas também pode levar a múltiplas consultas ao banco de dados se não for gerenciada adequadamente.

Comportamento das Consultas no Método findEmployeesByDepartment

No projeto "aula-lazy" que fornecemos anteriormente, na classe `DepartmentRepository` o método `findEmployeesByDepartment` ilustra o comportamento do carregamento lazy ao buscar empregados associados a um departamento específico. Vamos detalhar o processo passo a passo para entender como a JPA gerencia este cenário:

```
@Service
public class DepartmentService {

    @Autowired
    private DepartmentRepository repository;

    @Transactional(readOnly = true)
    public List<EmployeeMinDTO> findEmployeesByDepartment(Long id) {
        Optional<Department> result = repository.findById(id);
        List<Employee> list = result.get().getEmployees();
        return list.stream().map(x -> new EmployeeMinDTO(x)).toList();
    }
}
```

Passo 1: Busca do Departamento

Inicialmente, o método `findById` do `DepartmentRepository` é chamado para buscar o departamento com o ID fornecido:

- **Consulta ao Banco de Dados**: Uma consulta SQL é executada para buscar o departamento especificado pelo ID. Neste ponto, apenas os dados do departamento são recuperados, não incluindo os empregados associados, assumindo que o relacionamento entre `Department` e `Employee` está configurado como lazy.

Passo 2: Acesso aos Empregados Associados

- **Acesso aos Empregados**: A linha `List<Employee> list = result.get().getEmployees();` acessa a lista de empregados do departamento. Como este relacionamento é lazy, a JPA não carregou esses empregados na consulta inicial.
- **Segunda Consulta ao Banco de Dados**: Ao acessar a lista de empregados pela primeira vez, a JPA detecta que esses dados ainda não foram carregados e executa uma segunda consulta SQL para buscar os empregados associados ao departamento recuperado. Esta consulta é disparada automaticamente pelo framework devido ao acesso ao atributo lazy.

Passo 3: Transformação em DTOs

- **Conversão para DTO**: Os empregados recuperados são então mapeados para `EmployeeMinDTO`, uma classe de transferência de dados que pode conter menos atributos que a entidade original, otimizando o tráfego de dados para o cliente.

Considerações de Performance

- **Dupla Consulta**: Este método ilustra uma situação típica onde o carregamento lazy pode resultar em múltiplas consultas ao banco de dados, conhecido como o problema N+1. Cada acesso a uma lista de entidades lazy dentro de um loop ou operação similar pode resultar em uma nova consulta ao banco.
- **Otimização**: Para otimizar esse comportamento, poderíamos considerar o uso de técnicas como `JOIN FETCH` em uma consulta JPQL customizada ou `Entity Graphs` para carregar os empregados simultaneamente com o departamento, se o cenário de uso exigir frequentemente ambos os dados.

4.11 Alterando o Atributo Fetch dos Relacionamentos

Nesta seção vamos explorar um exemplo em que altermos o comportamento padrão de carregamento de objetos associados. Para isto, vamos analisar novamente nosso projeto "aula-lazy", em específico a classe `Department`, que possui um relacionamento um-para-muitos com a entidade `Employee`. Originalmente, este relacionamento não especifica o modo de fetch, portanto, ele segue o comportamento padrão, que é LAZY.

```
package com.devsuperior.aulalazy.entities;
```

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

import jakarta.persistence.*;

@Entity
@Table(name = "tb_department")
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long id;
    public String name;

    @OneToMany(mappedBy = "department")
    public List<Employee> employees = new ArrayList<>();

    public Department() {
    }

    public Department(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Employee> getEmployees() {
        return employees;
    }
}
```

Efeitos de `FetchType.EAGER` em `@OneToMany`

Ao alterar o parâmetro `fetch` para `FetchType.EAGER` na anotação `@OneToMany`, modificamos significativamente como os dados são carregados do banco de dados. Aqui está como a anotação modificada apareceria:

```java
@OneToMany(mappedBy = "department", fetch = FetchType.EAGER)
public List<Employee> employees = new ArrayList<>();
```

Implicações do Carregamento Eager

1. **Carregamento Imediato**: Com `FetchType.EAGER`, a JPA carrega todas as entidades associadas (`Employee` neste caso) imediatamente junto com a entidade `Department`, independentemente de se elas serão usadas ou não.
2. **Consumo de Recursos**: Esse carregamento imediato pode resultar em um uso significativo de recursos de memória e processamento, especialmente se o número de `Employee` associados for grande ou se as consultas forem frequentes e não necessitarem dos dados dos empregados.
3. **Impacto no Desempenho**: Em sistemas com muitos dados ou em operações que envolvem muitos departamentos, o carregamento eager pode levar a uma degradação perceptível no desempenho devido ao grande número de dados carregados em cada consulta. Além disso, pode ocorrer o problema de N+1 mesmo com eager fetching se o mapeamento não for feito cuidadosamente.
4. **Comportamento da Aplicação**: Alterar para eager fetching pode mudar o comportamento da aplicação de maneiras não previstas, especialmente em sistemas complexos onde as interações entre entidades são intrincadas e diversas partes do sistema dependem dessas interações de maneiras específicas.

A decisão entre usar `FetchType.LAZY` ou `FetchType.EAGER` deve ser baseada numa compreensão detalhada das necessidades específicas de acesso a dados da aplicação. Em muitos casos, é preferível manter o carregamento lazy padrão e utilizar técnicas como `JOIN FETCH` em consultas específicas onde a carga imediata de entidades associadas é necessária. Isso oferece um equilíbrio entre desempenho e disponibilidade de dados, permitindo otimizações mais granulares e contextualizadas conforme a necessidade.

4.12 Otimizando Consultas com a Cláusula JOIN FETCH

No desenvolvimento de aplicações que utilizam JPA (Java Persistence API), frequentemente nos deparamos com a necessidade de otimizar consultas para evitar o problema das "N+1 consultas" (falaremos em detalhes sobre este problema mais adiante neste capítulo). Uma técnica eficaz para isso é o uso da cláusula `JOIN FETCH`, que permite carregar dados associados de maneira eficiente, eliminando múltiplas consultas ao banco de dados que seriam necessárias em alguns casos para buscar as entidades associadas.

Considerando o relacionamento entre as entidades `Employee` e `Department` de nosso sistema "aula-lazy", pode-se observar que cada funcionário está associado a um departamento:

```
@Entity
@Table(name = "tb_employee")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long id;
    public String name;
    public String email;
```

```
    @ManyToOne
    @JoinColumn(name = "department_id")
    public Department department;

    ...
}
```

Para buscar funcionários junto com seus respectivos departamentos, podemos definir um método no repositório `EmployeeRepository` que utiliza a cláusula `JOIN FETCH`. Isso garante que o JPA carregue os funcionários e seus departamentos em uma única consulta, reduzindo o overhead de múltiplas idas e vindas ao banco de dados.

```
package com.devsuperior.aulalazy.repositories;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import com.devsuperior.aulalazy.entities.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    @Query("SELECT obj FROM Employee obj JOIN FETCH obj.department")
    List<Employee> findEmployeesWithDepartments();
}
```

Funcionamento da Cláusula JOIN FETCH

O que é JOIN FETCH?

A cláusula `JOIN FETCH` no JPQL (Java Persistence Query Language) instrui o JPA a fazer um join entre as entidades `Employee` e `Department` e a carregar todos os dados relacionados de uma vez. Isto é particularmente útil quando sabemos que vamos precisar acessar os dados associados imediatamente após a recuperação da entidade principal.

Benefícios

- **Redução de Latência**: Menos consultas ao banco, pois dados associados são carregados na mesma consulta da entidade principal.
- **Prevenção do Problema N+1**: Evita múltiplas consultas que normalmente ocorrem quando acessamos entidades relacionadas configuradas com fetch tipo lazy.

Vale ressaltar que, embora `JOIN FETCH` seja muito útil, ele tem limitações, especialmente quando se trata de consultas paginadas, onde ele não se comporta corretamente, necessitando outras abordagens para recuperar dados associados.

4.13 Entendendo Transactional e Open-In-View

Annotation @Transactional do Spring

A anotação `@Transactional` do Spring Framework é uma ferramenta poderosa para gerenciar transações com o banco de dados de maneira declarativa. Ela pode ser aplicada em métodos ou classes inteiras para definir o comportamento de transação de operações de banco de dados.

Funcionalidades da @Transactional

- **Gerenciamento de Transações**: A anotação `@Transactional` automaticamente cria e gerencia transações de banco de dados, garantindo que as operações realizadas dentro do método anotado sejam completadas com sucesso ou, em caso de falha, revertidas (rollback). Isso simplifica significativamente o desenvolvimento, pois o desenvolvedor não precisa manualmente controlar a abertura, commit ou rollback de transações.
- **Resolução de Dependências Lazy**: Ao utilizar esta anotação, qualquer acesso a dados que necessite de carregamento lazy será resolvido dentro do escopo da transação. Isso significa que, enquanto a transação está ativa, todas as pendências de carregamento lazy com o banco de dados serão atendidas sem problemas, evitando exceções como `LazyInitializationException`, que ocorre quando se tenta acessar dados lazy fora do contexto de uma transação ativa.

Exemplo de Uso

```
@Transactional
public void updateProduct(Product product) {
    repository.save(product);
}
```

Neste exemplo, o método `updateProduct` é envolto em uma transação. Se ocorrer alguma exceção durante o `save`, todas as modificações feitas no banco de dados serão revertidas.

Propriedade Open-In-View

A propriedade `open-in-view` está relacionada ao padrão Open Session in View, que é comumente utilizado em aplicações Hibernate/JPA. Esta propriedade controla se a sessão do Hibernate (ou sessão JPA) fica aberta durante toda a requisição HTTP, incluindo a renderização da view (camada de apresentação).

Configuração e Implicações

- **spring.jpa.open-in-view=false**: Configurar esta propriedade como `false` significa que a sessão JPA será fechada antes de a execução retornar à camada de apresentação (controller). Isso ajuda a evitar problemas de performance e uso de recursos associados à manutenção de sessões abertas por mais tempo que o necessário. No entanto, também significa que qualquer acesso a dados que requer carregamento lazy não será possível após a conclusão do método de serviço, a menos que a transação ainda esteja ativa.

Problemas Associados ao Open-In-View

Quando `open-in-view` está habilitado (o padrão em muitas configurações do Spring Boot é `true`), a sessão JPA permanece aberta durante toda a requisição, o que pode levar a:

- **Problemas de Performance**: Manter a sessão aberta pode resultar em uma maior utilização de recursos, pois a sessão pode ficar retida por mais tempo que o necessário.
- **Riscos de Segurança**: A extensão da sessão aumenta o risco de ataques, como aqueles que tentam manipular a sessão persistente.

4.14 Consultas com Query Methods

Introdução aos Query Methods do Spring

Os Query Methods são uma das funcionalidades mais poderosas e flexíveis do Spring Data JPA. Eles permitem aos desenvolvedores definir consultas ao banco de dados de maneira simples e declarativa, usando convenções de nomenclatura em interfaces de repositório. Essa abordagem reduz significativamente a complexidade do código e aumenta a legibilidade ao abstrair a construção de consultas SQL ou JPQL.

Funcionamento dos Query Methods

O Spring Data JPA interpreta o nome dos métodos em interfaces de repositório para gerar as consultas correspondentes automaticamente. O nome do método descreve a consulta que você quer executar em termos de condições de pesquisa. O framework analisa o nome do método e cria uma consulta que busca os resultados conforme especificado.

Exemplo Básico

Consideremos uma entidade `Product` com campos `name` e `category`. Podemos definir um repositório com métodos que busquem produtos baseados em diferentes critérios usando apenas a assinatura do método:

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    List<Product> findByName(String name);

    List<Product> findByCategoryName(String categoryName);
}
```

Neste exemplo, o Spring Data JPA geraria consultas para buscar produtos por `name` ou pela `categoryName` automaticamente, sem a necessidade de escrever qualquer SQL ou JPQL.

Vantagens dos Query Methods

- **Simplicidade**: Define consultas complexas com simples assinaturas de métodos, sem necessidade de escrever código de consulta explícito.

- **Manutenção**: Facilita a manutenção do código, pois as alterações nas entidades ou nos requisitos de consulta podem frequentemente ser geridas apenas alterando os nomes dos métodos.
- **Segurança**: Reduz o risco de injeção de SQL, pois as consultas são construídas automaticamente pelo framework usando parâmetros de consulta seguros.

Limitações e Considerações

Apesar de sua utilidade, os Query Methods têm limitações:

- **Complexidade**: Para consultas extremamente complexas, os nomes dos métodos podem se tornar muito longos e menos intuitivos.
- **Customização**: Em alguns casos, pode ser necessário maior controle sobre a consulta do que o que pode ser alcançado através de um Query Method. Para esses casos, o Spring Data JPA permite o uso de anotações `@Query` para definir consultas JPQL ou SQL customizadas diretamente.

4.15 Introdução sobre JPQL

O que é JPQL?

JPQL, ou Java Persistence Query Language, é uma linguagem de consulta orientada a objetos utilizada em aplicações Java para interagir com bancos de dados no contexto da Java Persistence API (JPA). JPQL é fortemente inspirada em SQL, mas opera em termos de objetos, classes e propriedades, ao invés de tabelas e colunas, fazendo a ponte entre o paradigma orientado a objetos do Java e o modelo relacional de bancos de dados.

Propósito da JPQL

A principal vantagem da JPQL é sua capacidade de abstrair a complexidade das operações de banco de dados, permitindo que desenvolvedores escrevam consultas de maneira mais natural e alinhada com o modelo de domínio da aplicação. Com JPQL, é possível realizar operações de seleção, inserção, atualização e exclusão utilizando a sintaxe familiar do SQL, mas de uma maneira que respeite a orientação a objetos do código Java.

Características da JPQL

- **Independência de Plataforma**: Uma das maiores vantagens da JPQL é sua independência de banco de dados específico. Isso significa que o mesmo código JPQL pode ser executado em diferentes bancos de dados sem modificação, desde que o JPA esteja corretamente configurado.
- **Integração com JPA**: JPQL é parte integrante da especificação JPA e trabalha em conjunto com o `EntityManager`, que gerencia o ciclo de vida das entidades JPA. Isso proporciona uma integração profunda e eficiente entre as consultas e as operações de persistência.
- **Orientação a Objetos**: Ao contrário do SQL tradicional, JPQL permite realizar consultas considerando a estrutura das classes e dos objetos. Por exemplo, é possível realizar uma consulta sobre objetos de uma classe com herança, tratando-as através de suas especificidades de classe.

Exemplo de Uso de JPQL

Consideremos a seguinte consulta JPQL que busca todos os produtos de uma determinada categoria:

```
String jpql = "SELECT p FROM Product p WHERE p.category.name = :categoryName";
Query query = entityManager.createQuery(jpql);
query.setParameter("categoryName", "Electronics");
List<Product> products = query.getResultList();
```

Neste exemplo, `Product` e `Category` são classes de entidade, e a consulta é escrita em termos dessas classes e de suas propriedades, como `p.category.name`.

Benefícios do Uso de JPQL

- **Sintaxe de Alto Nível**: JPQL fornece uma sintaxe de alto nível para a realização de consultas, o que facilita a escrita, leitura e manutenção do código.
- **Segurança**: Consultas JPQL são menos suscetíveis a injeções SQL, já que os parâmetros são tipados e validados pela JPA.
- **Performance**: Embora JPQL possa não ser tão otimizada quanto SQL nativo para operações específicas de banco de dados, a diferença de performance é geralmente compensada pela robustez e pela portabilidade entre diferentes SGBDs.

4.16 Polêmica: Vale a Pena Especializar na JPQL?

Enquanto a JPQL oferece diversas vantagens ao se integrar de forma fluente com o ecossistema Java e JPA, há um debate contínuo sobre se vale a pena se especializar em JPQL, especialmente quando SQL tradicional ainda é uma opção viável e poderosa. Nesta seção, exploraremos as vantagens e desvantagens da JPQL para ajudar a elucidar esta questão.

Vantagens da JPQL

1. **Simplicidade em Alguns Casos**: JPQL pode simplificar a escrita de consultas em cenários onde a lógica de negócios se alinha estreitamente com o modelo de domínio, permitindo consultas que são mais legíveis e diretas comparadas ao SQL tradicional.
2. **Integração com Spring Data JPA**: JPQL se beneficia plenamente das capacidades do Spring Data JPA, como suporte automático para paginação e a habilidade de retornar diretamente objetos dentro do contexto de persistência do Spring. Isso facilita o desenvolvimento, reduzindo a quantidade de código boilerplate necessário para implementar funcionalidades comuns de aplicativos de dados.
3. **Entidades Gerenciadas**: Os objetos resultantes de uma consulta JPQL são automaticamente entidades gerenciadas pela JPA. Isso significa que qualquer alteração nessas entidades pode ser automaticamente persistida no banco de dados ao final da transação, simplificando a manipulação de dados.

Desvantagens da JPQL

1. **Complexidade em Consultas Avançadas**: Para consultas particularmente complexas, a JPQL pode ser mais desafiadora para escrever e validar. A natureza orientada a objetos da JPQL pode tornar difícil expressar algumas consultas que são trivialmente implementadas em SQL puro, especialmente aquelas que requerem operações avançadas de junção ou subconsultas complexas.
2. **Ausência de Operações de União**: JPQL não suporta operações de união (`UNION`), o que pode ser um grande limitador para certos tipos de consultas que necessitam combinar resultados de múltiplas tabelas de formas não suportadas por `JOIN`s.
3. **Curva de Aprendizado**: Aprender JPQL pode exigir um investimento significativo em tempo e recursos, especialmente para desenvolvedores que já estão confortáveis com SQL. Além disso, especializar-se em JPQL pode não ser tão benéfico se o ambiente de trabalho demandar flexibilidade entre diferentes tecnologias de banco de dados ou se a aplicação fizer uso intensivo de funcionalidades SQL que estão fora do escopo da JPA.

Discussão: Especializar-se em JPQL Vale a Pena?

A decisão de se especializar em JPQL deve considerar tanto o contexto do projeto quanto as preferências e a experiência da equipe de desenvolvimento. Se o projeto se beneficia amplamente das características de integração e gerenciamento de entidades da JPA, e as consultas não tendem a ser excessivamente complexas, a especialização em JPQL pode aumentar a produtividade e a manutenibilidade do código.

Por outro lado, em ambientes onde a flexibilidade e a performance são críticas, ou onde são necessárias consultas avançadas que vão além das capacidades da JPQL, pode ser mais prático manter uma forte competência em SQL tradicional, utilizando a JPQL de forma complementar, mas não exclusiva.

4.17 Preparando para os estudos de caso de consultas

Nesta seção, vamos apresentar orientações para você se preparar para os exercícios sobre SQL e JPQL que vamos apresentar nas seções subsequentes.

Revisão sobre SQL

Em primeiro lugar, recomendamos algumas referências para você fazer um repasse dos fundamentos de SQL. A primeira referência é nosso vídeo “Revisão de Álgebra Relacional e SQL”, conforme link:

https://www.youtube.com/watch?v=GHpE5xOxXXI

A segunda referência é o Tutorial de SQL do W3Schools, conforme link:

https://www.w3schools.com/sql/default.asp

Por fim, recomendamos que você faça uma conta na plataforma Beecrowd (antigamente chamada de URI), onde você poderá fazer vários exercícios de SQL. Nós fizemos uma classificação prévia dos exercícios, agrupando por assunto, para facilitar seu acesso:

- **Grupo 1: projeção, restrição**: exercícios 2602, 2603, 2604, 2607, 2608, 2615, 2624
- **Grupo 2: JOIN**: exercícios 2605, 2606, 2611, 2613, 2614, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2742
- **Grupo 3: GROUP BY, subconsultas**: exercícios 2609, 2993, 2994, 2995, 2996
- **Grupo 4: Expressões na projeção**: exercícios 2610, 2625, 2738, 2739, 2741, 2743, 2744, 2745, 2746, 3001
- **Grupo 5: Diferença, União**: exercícios 2616, 2737, 2740, 2990
- **Grupo 6: Difíceis**: exercícios 2988, 2989, 2991, 2992, 2997, 2998, 2999

Preparação do Postgresql

Os exercícios sobre SQL e JPQL que vamos apresentar nas próximas seções serão para o banco de dados relacional Postgresql, que inclusive é o banco de dados padrão para os exercícios da plataforma Beecrowd. Sendo assim, você precisará instalar o Postgresql, bem como uma ferramenta para acessar e fazer consultas ao banco de dados. Sugerimos duas opções:

- **Opção 1 - instalação direta**
  - Instale o Postgresql (versão 12 ou 13 ou 14)
  - Instale o pgAdmin ou o DBeaver
- **Opção 2 - Instalação via Docker Compose**
  - Se você tem experiência com Docker, pode instalar o Postgresql e o pgAdmin usando Docker Compose. Um exemplo de script pode ser obtido no link:

https://gist.github.com/acenelio/5e40b27cfc40151e36beec1e27c4ff71

Estudos de caso SQL e JPQL

Os exercícios sobre SQL e JPQL que faremos nas próximas seções serão estudos de caso de alguns projetos que selecionamos da plataforma Beecrowd, quais sejam:

- URI 2602 - busca simples
- URI 2611 - join simples
- URI 2621 - like e between
- URI 2609 - group by
- URI 2737 - união
- URI 2990 - diferença / left join

Nós já preparamos os projetos Spring para cada exercício, com o modelo de domínio já implementado, correspondente à base de dados de cada exercício. Você pode baixar os projetos no link do Github a seguir:

https://github.com/devsuperior/bdsconsultas

4.18 URI 2602 - Elaborando a Consulta

Enunciado do Exercício

Sua empresa está fazendo um levantamento de quantos clientes estão cadastrados nos estados, porém, faltou levantar os dados do estado do Rio Grande do Sul. Então você deve Exibir o nome de todos os clientes cujo estado seja 'RS'.

Estrutura da Base de Dados

Para abordar essa tarefa, trabalharemos com a tabela `customers`, que inclui informações detalhadas sobre cada cliente. Veja o script de criação da tabela:

```
CREATE TABLE customers (
  id NUMERIC PRIMARY KEY,
  name CHARACTER VARYING (255),
  street CHARACTER VARYING (255),
  city CHARACTER VARYING (255),
  state CHAR (2),
  credit_limit NUMERIC
);
```

A tabela contém várias colunas que descrevem o cliente, mas estamos particularmente interessados na coluna `state` para este exercício.

Dados Inseridos

Os dados dos clientes são inseridos da seguinte forma:

```
INSERT INTO customers (id, name, street, city, state, credit_limit)
VALUES
  (1, 'Pedro Augusto da Rocha', 'Rua Pedro Carlos Hoffman', 'Porto Alegre', 'RS'
   , 700.00),
  (2, 'Antonio Carlos Mamel', 'Av. Pinheiros', 'Belo Horizonte', 'MG', 3500.50),
  (3, 'Luiza Augusta Mhor', 'Rua Salto Grande', 'Niteroi', 'RJ', 4000.00),
  (4, 'Jane Ester', 'Av 7 de setembro', 'Erechim', 'RS', 800.00),
  (5, 'Marcos Antônio dos Santos', 'Av Farrapos', 'Porto Alegre', 'RS', 4250.25)
   ;
```

Elaboração da Consulta SQL

A consulta para responder ao enunciado é direta e utiliza a cláusula `WHERE` para filtrar os clientes pelo estado do Rio Grande do Sul (RS):

```
SELECT name
FROM customers
WHERE state = 'RS'
```

Explicação da Consulta

- **`SELECT name`**: Esta parte da consulta especifica que queremos retornar apenas o nome dos clientes.
- **`FROM customers`**: Especifica que a consulta será realizada sobre a tabela `customers`.
- **`WHERE state = 'RS'`**: Filtra os registros para incluir somente aqueles onde a coluna `state` é igual a 'RS', que é o critério dado pelo exercício.

4.19 Baixando os projetos iniciados dos estudos de caso

Esta seção é para avisar que neste momento você deve estar com os projetos Spring Boot dos estudos de casos iniciados. Caso não tenha baixado os projetos ainda, eles estão no repositório Github a seguir:

https://github.com/devsuperior/bdsconsultas

Você precisa também estar com o banco de dados Postgresql instalado no seu computador.

4.20 URI 2602 Spring Boot SQL

Agora vamos dar continuidade ao exercício URI 2602, elaborando a consulta SQL dentro de um projeto Spring Boot. Para começar, abra o projeto "uri2602" fornecido e prepare o ambiente para execução.

Configuração do Banco de Dados

Antes de prosseguir, é necessário criar uma base de dados chamada "uri2602" no seu sistema de gerenciamento de banco de dados PostgreSQL.

Configurações do Projeto Spring Boot

O arquivo de properties do projeto Spring Boot contém configurações essenciais para a conexão com o banco de dados:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/uri2602
spring.datasource.username=postgres
spring.datasource.password=1234567

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.hibernate.ddl-auto=none

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Essas configurações especificam como o Spring Boot deve conectar-se ao PostgreSQL e como o Hibernate deve se comportar em relação ao esquema do banco de dados e à formatação das consultas SQL.

Classe Entidade Customer

O projeto já inclui uma classe `Customer` que mapeia para a tabela `customers` no banco de dados:

```
package com.devsuperior.uri2602.entities;

import jakarta.persistence.*;

@Entity
```

```
@Table(name = "customers")
public class Customer {

    @Id
    private Long id;
    private String name;
    private String street;
    private String city;
    private String state;
    private Double creditLimit;

    ...
}
```

Projection para Resultados da Consulta

Para representar os dados retornados pela consulta, foi criada uma projection chamada `CustomerMinProjection`:

```
package com.devsuperior.uri2602.projections;

public interface CustomerMinProjection {
    String getName();
}
```

DTO para Resultados de Consulta

Implementamos também um DTO para resultados de consultas.

```
package com.devsuperior.uri2602.dto;

import com.devsuperior.uri2602.projections.CustomerMinProjection;

public class CustomerMinDTO {

    private String name;

    public CustomerMinDTO() {
    }

    public CustomerMinDTO(String name) {
        this.name = name;
    }

    public CustomerMinDTO(CustomerMinProjection projection) {
        name = projection.getName();
    }

    String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "CustomerMinDTO [name=" + name + "]";
    }
}
```

Implementação do Repositório

O repositório `CustomerRepository` inclui uma consulta SQL nativa que busca nomes de clientes no estado especificado:

```
package com.devsuperior.uri2602.repositories;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import com.devsuperior.uri2602.projections.CustomerMinProjection;

public interface CustomerRepository extends JpaRepository<Customer, Long> {

    @Query(nativeQuery = true, value = "SELECT name "
            + "FROM customers "
            + "WHERE UPPER(state) = UPPER(:state)")
    List<CustomerMinProjection> search1(String state);
}
```

Teste da Consulta

Finalmente, a classe principal `Uri2602Application` executa a consulta ao iniciar a aplicação, imprimindo os resultados:

```
@SpringBootApplication
public class Uri2602Application implements CommandLineRunner {

    @Autowired
    private CustomerRepository repository;

    public static void main(String[] args) {
        SpringApplication.run(Uri2602Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
```

```
        List<CustomerMinProjection> list = repository.search1("RS");
        List<CustomerMinDTO> result1 = list.stream().map(x -> new CustomerMinDTO
    (x)).collect(Collectors.toList());

        System.out.println("\n*** RESULTADO SQL RAIZ:");
        for (CustomerMinDTO obj : result1) {
            System.out.println(obj);
        }
        System.out.println("\n\n");
    }
}
```

4.21 URI 2602 Spring Boot JPQL

Prosseguindo com o exercício URI 2602, agora vamos abordar a elaboração da consulta utilizando JPQL no projeto Spring Boot, proporcionando uma alternativa ao uso de SQL nativo anteriormente discutido.

Implementação da Consulta JPQL

Dentro do `CustomerRepository`, adicionamos um método que utiliza JPQL para realizar a mesma operação de busca de clientes por estado. O código deste método é detalhado abaixo:

```
@Query("SELECT new com.devsuperior.uri2602.dto.CustomerMinDTO(obj.name) "
        + "FROM Customer obj "
        + "WHERE UPPER(obj.state) = UPPER(:state)")
List<CustomerMinDTO> search2(String state);
```

Explicação do Código

- **`@Query` Annotation**: Esta anotação define a consulta JPQL que será executada quando o método `search2` for chamado. O uso de JPQL permite integrar mais naturalmente com o modelo de objetos da aplicação, diferentemente do SQL nativo que trabalha diretamente com as tabelas e colunas do banco de dados.
- **Construtor de DTO**: `new com.devsuperior.uri2602.dto.CustomerMinDTO(obj.name)` cria uma nova instância de `CustomerMinDTO` diretamente na consulta. Isso é uma característica poderosa do JPQL, permitindo que a transformação de entidades em DTOs seja feita diretamente na camada de persistência.
- **Filtro de Estado**: A cláusula `WHERE` filtra os clientes pelo estado, usando a função `UPPER` para garantir que a comparação seja insensível a maiúsculas e minúsculas. A variável `:state` é um parâmetro que será passado ao método `search2` em tempo de execução.

Testando a Consulta JPQL

Para verificar o funcionamento da consulta JPQL, acrescentamos código na classe principal `Uri2602Application` para executar e imprimir os resultados da nova consulta. O código adicional é explicado a seguir:

```
@SpringBootApplication
public class Uri2602Application implements CommandLineRunner {

    ...

    @Override
    public void run(String... args) throws Exception {

        ...

        List<CustomerMinDTO> result2 = repository.search2("RS");

        System.out.println("\n*** RESULTADO JPQL:");
        for (CustomerMinDTO obj : result2) {
            System.out.println(obj);
        }
    }
}
```

Explicação do Código de Teste

- **Execução da Consulta JPQL**: `List<CustomerMinDTO> result2 = repository.search2("RS");` executa a consulta JPQL definida no repositório, passando "RS" como parâmetro para o estado.
- **Impressão dos Resultados**: O laço `for` itera sobre cada `CustomerMinDTO` retornado pela consulta e imprime seu conteúdo. Este passo é crucial para verificar visualmente se a consulta está retornando os dados esperados.

4.22 URI 2611 Elaborando a Consulta

Enunciado do Exercício

Uma Vídeo locadora contratou seus serviços para catalogar os filmes dela. Eles precisam que você selecione o código e o nome dos filmes cuja descrição do gênero seja 'Action'.

Estrutura da Base de Dados

Para abordar este exercício, trabalhamos com duas tabelas principais: `genres` e `movies`. Abaixo estão os scripts SQL para criar essas tabelas e inserir os dados iniciais:

Criação de Tabelas

```
CREATE TABLE genres (
  id numeric PRIMARY KEY,
  description varchar(50)
);

CREATE TABLE movies (
  id numeric PRIMARY KEY,
  name varchar(50),
```

```
  id_genres numeric REFERENCES genres (id)
);
```

Inserção de Dados

```
INSERT INTO genres (id, description)
VALUES
  (1, 'Animation'),
  (2, 'Horror'),
  (3, 'Action'),
  (4, 'Drama'),
  (5, 'Comedy');

INSERT INTO movies (id, name, id_genres)
VALUES
  (1, 'Batman', 3),
  (2, 'The Battle of the Dark River', 3),
  (3, 'White Duck', 1),
  (4, 'Breaking Barriers', 4),
  (5, 'The Two Hours', 2);
```

Elaboração da Consulta SQL

Para atender ao pedido da locadora, precisamos de uma consulta que selecione o código e o nome dos filmes cujo gênero é 'Action'. Aqui está a solução para o exercício:

```
SELECT movies.id, movies.name
FROM movies
INNER JOIN genres ON movies.id_genres = genres.id
WHERE genres.description = 'Action'
```

Explicação da Consulta

- **Seleção de Colunas**: `SELECT movies.id, movies.name` indica que queremos retornar o identificador (`id`) e o nome (`name`) dos filmes.
- **Fonte de Dados**: `FROM movies` especifica que a tabela principal para a consulta é `movies`.
- **Junção com Genres**: `INNER JOIN genres ON movies.id_genres = genres.id` cria um vínculo entre as tabelas `movies` e `genres` com base nos identificadores de gênero. Isso permite acessar informações de ambas as tabelas na mesma consulta.
- **Filtro de Gênero**: `WHERE genres.description = 'Action'` restringe os resultados aos filmes cuja descrição do gênero é 'Action'. Este filtro é crucial para satisfazer o requisito de listar apenas filmes de ação.

4.23 URI 2611 Spring Boot SQL e JPQL

Avançando no exercício URI 2611, vamos implementar as consultas SQL e JPQL no projeto Spring Boot. Para isso, abra o projeto "uri2611" fornecido e prepare-se para integrar o código com uma nova base de dados chamada "uri2611" criada no PostgreSQL.

Modelo de Domínio

O projeto Spring Boot já está equipado com o modelo de domínio correspondente ao modelo relacional do banco de dados:

```
@Entity
@Table(name = "movies")
public class Movie {

    @Id
    private Long id;
    private String name;

    @ManyToOne
    @JoinColumn(name = "id_genres")
    private Genre genre;

    ...
```

```
@Entity
@Table(name = "genres")
public class Genre {

    @Id
    private Long id;
    private String description;

    @OneToMany(mappedBy = "genre")
    private List<Movie> movies = new ArrayList<>();
}
```

Projections

Implementamos uma projeção para representar o resultado da consulta SQL de forma simplificada:

```
package com.devsuperior.uri2611.projections;

public interface MovieMinProjection {

    Long getId();
    String getName();
}
```

DTO

Implementamos também um DTO para resultado de consultas:

```
package com.devsuperior.uri2611.dto;

import com.devsuperior.uri2611.projections.MovieMinProjection;

public class MovieMinDTO {

    private Long id;
    private String name;

    public MovieMinDTO() {
    }

    public MovieMinDTO(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public MovieMinDTO(MovieMinProjection projection) {
        id = projection.getId();
        name = projection.getName();
    }

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "MovieMinDTO [id=" + id + ", name=" + name + "]";
    }
}
```

Repositório com Consultas SQL e JPQL

O `MovieRepository` inclui métodos para realizar as consultas tanto em SQL nativo quanto em JPQL.

- `search1`: Método que utiliza SQL nativo para buscar filmes pelo gênero, realizando um `INNER JOIN` entre as tabelas `movies` e `genres`.

- `search2`: Método que emprega JPQL para obter o mesmo resultado, porém utilizando a abordagem orientada a objetos do JPA.

```
package com.devsuperior.uri2611.repositories;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import com.devsuperior.uri2611.dto.MovieMinDTO;
import com.devsuperior.uri2611.entities.Movie;
import com.devsuperior.uri2611.projections.MovieMinProjection;

public interface MovieRepository extends JpaRepository<Movie, Long> {

    @Query(nativeQuery = true, value = "SELECT movies.id, movies.name "
            + "FROM movies "
            + "INNER JOIN genres ON movies.id_genres = genres.id "
            + "WHERE genres.description = :genreName")
    List<MovieMinProjection> search1(String genreName);

    @Query("SELECT new com.devsuperior.uri2611.dto.MovieMinDTO(obj.id, obj.name)
    "
            + "FROM Movie obj "
            + "WHERE obj.genre.description = :genreName")
    List<MovieMinDTO> search2(String genreName);
}
```

Testando as Consultas

A classe principal `Uri2611Application` é configurada para testar ambas as consultas ao iniciar a aplicação.

- Executamos ambas as consultas para o gênero "Action".
- Os resultados são mapeados para `MovieMinDTO` e impressos para verificar a consistência entre as abordagens SQL e JPQL.

```
@SpringBootApplication
public class Uri2611Application implements CommandLineRunner {

    @Autowired
    private MovieRepository repository;

    public static void main(String[] args) {
        SpringApplication.run(Uri2611Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {

        List<MovieMinProjection> list = repository.search1("Action");
        List<MovieMinDTO> result1 = list.stream().map(x -> new MovieMinDTO(x)).
```

```
    collect(Collectors.toList());

        System.out.println("\n*** RESULTADO SQL RAIZ:");
        for (MovieMinDTO obj : result1) {
            System.out.println(obj);
        }
        System.out.println("\n\n");

        List<MovieMinDTO> result2 = repository.search2("Action");

        System.out.println("\n*** RESULTADO JPQL:");
        for (MovieMinDTO obj : result2) {
            System.out.println(obj);
        }
    }
}
```

4.24 URI 2621 Elaborando a consulta

Enunciado do Exercício

Na hora de entregar o relatório de quantos produtos a empresa tem em estoque, uma parte do relatório ficou corrompida, por isso o responsável do estoque lhe pediu uma ajuda, ele quer que você exiba os seguintes dados para ele.

Exiba o nome dos produtos cujas quantidades estejam entre 10 e 20 e cujo nome do fornecedor inicie com a letra 'P'.

Estrutura da Base de Dados

O seguinte script SQL define a estrutura das tabelas `providers` e `products`:

```
CREATE TABLE providers (
  id numeric PRIMARY KEY,
  name varchar(255),
  street varchar(255),
  city varchar(255),
  state char(2)
);

CREATE TABLE products (
  id numeric PRIMARY KEY,
  name varchar(255),
  amount numeric,
  price numeric,
  id_providers numeric REFERENCES providers (id)
);

INSERT INTO providers (id, name, street, city, state)
VALUES
  (1, 'Ajax SA', 'Rua Presidente Castelo Branco', 'Porto Alegre', 'RS'),
  (2, 'Sansul SA', 'Av Brasil', 'Rio de Janeiro', 'RJ'),
  (3, 'Pr Sheppard Chairs', 'Rua do Moinho', 'Santa Maria', 'RS'),
```

```
  (4, 'Elon Electro', 'Rua Apolo', 'São Paulo', 'SP'),
  (5, 'Mike Electro', 'Rua Pedro da Cunha', 'Curitiba', 'PR');

INSERT INTO products (id, name, amount, price, id_providers)
VALUES
  (1, 'Blue Chair', 30, 300.00, 5),
  (2, 'Red Chair', 50, 2150.00, 2),
  (3, 'Disney Wardrobe', 400, 829.50, 4),
  (4, 'Executive Chair', 17, 9.90, 3),
  (5, 'Solar Panel', 30, 3000.25, 4);
```

Saída Esperada

Com base nos dados fornecidos, a saída esperada da consulta SQL seria:

name
Executive Chair

Solução do Exercício

A consulta SQL para resolver o exercício é:

```
SELECT products.name
FROM products
INNER JOIN providers ON products.id_providers = providers.id
WHERE products.amount BETWEEN 10 AND 20
AND providers.name LIKE 'P%'
```

Explicação Detalhada da Consulta:

- **`SELECT products.name`:** Esta parte da consulta especifica que o resultado deve incluir o nome dos produtos.
- **`FROM products`:** Indica que estamos buscando dados da tabela `products`.
- **`INNER JOIN providers ON products.id_providers = providers.id`:** Esta instrução une a tabela `products` com a tabela `providers` baseada na relação entre `id_providers` de `products` e `id` de `providers`. Isso é necessário para acessar o nome do fornecedor.
- **`WHERE products.amount BETWEEN 10 AND 20`:** Filtra os produtos que têm quantidades entre 10 e 20.
- **`AND providers.name LIKE 'P%'`:** Adicionalmente, filtra os produtos cujo nome do fornecedor começa com a letra 'P'. O operador `%` é um coringa que representa qualquer sequência de caracteres.

4.25 URI 2621 Spring Boot SQL e JPQL

Vamos prosseguir com a implementação das consultas SQL e JPQL no projeto Spring Boot. Abra o projeto "uri2621" fornecido anteriormente e crie uma base de dados chamada "uri2621" no PostgreSQL do seu computador para integrar o modelo de dados necessário.

Modelo de Domínio

O projeto já contém o modelo de domínio correspondente ao esquema relacional, que é crucial para a persistência dos dados:

Entidade Product:

```
@Entity
@Table(name = "products")
public class Product {

    @Id
    private Long id;
    private String name;
    private Integer amount;
    private Double price;

    @ManyToOne
    @JoinColumn(name = "id_providers")
    private Provider provider;

    // Getters and Setters...
}
```

Entidade Provider:

```
@Entity
@Table(name = "providers")
public class Provider {

    @Id
    private Long id;
    private String name;
    private String street;
    private String city;
    private String state;

    @OneToMany(mappedBy = "provider")
    private List<Product> products = new ArrayList<>();

    // Getters and Setters...
}
```

Projections

A projeção `ProductMinProjection` é criada para simplificar o resultado das consultas, focando apenas no nome do produto:

```
package com.devsuperior.uri2621.projections;

public interface ProductMinProjection {
```

```
    String getName();
}
```

DTO

A classe `ProductMinDTO` é responsável por encapsular os dados do resultado em um objeto de transferência de dados:

```
package com.devsuperior.uri2621.dto;

import com.devsuperior.uri2621.projections.ProductMinProjection;

public class ProductMinDTO {

    private String name;

    public ProductMinDTO() {
    }

    public ProductMinDTO(String name) {
        this.name = name;
    }

    public ProductMinDTO(ProductMinProjection projection) {
        name = projection.getName();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "ProductMinDTO [name=" + name + "]";
    }
}
```

Repositório com Consultas SQL e JPQL

O `ProductRepository` inclui métodos para realizar consultas em SQL nativo e em JPQL:

```
package com.devsuperior.uri2621.repositories;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
```

```
import org.springframework.data.jpa.repository.Query;

import com.devsuperior.uri2621.dto.ProductMinDTO;
import com.devsuperior.uri2621.entities.Product;
import com.devsuperior.uri2621.projections.ProductMinProjection;

public interface ProductRepository extends JpaRepository<Product, Long> {

    @Query(nativeQuery = true, value = "SELECT products.name "
            + "FROM products "
            + "INNER JOIN providers ON providers.id = products.id_providers "
            + "WHERE products.amount BETWEEN :min AND :max "
            + "AND providers.name LIKE CONCAT(:beginName, '%')")
    List<ProductMinProjection> search1(Integer min, Integer max, String
   beginName);

    @Query("SELECT new com.devsuperior.uri2621.dto.ProductMinDTO(obj.name) "
            + "FROM Product obj "
            + "WHERE obj.amount BETWEEN :min AND :max "
            + "AND obj.provider.name LIKE CONCAT(:beginName, '%')")
    List<ProductMinDTO> search2(Integer min, Integer max, String beginName);
}
```

Método `search1`: - **Tipo de Consulta:** SQL Nativo. Este método executa uma consulta SQL diretamente no banco de dados usando a anotação `@Query` com o atributo `nativeQuery = true`. - **Consulta Realizada:** A consulta faz um `INNER JOIN` entre as tabelas `products` e `providers` baseado na relação entre `products.id_providers` e `providers.id`. - **Filtro de Dados:** Filtra produtos cuja quantidade está entre os valores especificados por `:min` e `:max` e cujo nome do fornecedor começa com uma letra ou mais especificada por `:beginName`. O `CONCAT(:beginName, '%')` constrói uma string que é usada no operador `LIKE` para filtrar os nomes dos fornecedores. - **Projeção de Resultados:** Retorna uma lista de `ProductMinProjection`, que são projeções focadas apenas no nome do produto.

Método `search2`: - **Tipo de Consulta:** JPQL. Este método usa JPQL para formular a consulta, permitindo trabalhar de forma mais alinhada ao modelo de domínio do projeto. - **Consulta Realizada:** A consulta acessa a entidade `Product` e faz referência ao seu relacionamento com `Provider` diretamente pelo objeto, `obj.provider`. - **Filtro de Dados:** Similar ao método `search1`, porém utilizando a sintaxe JPQL que se alinha melhor com o gerenciamento de entidades do JPA. Filtros são aplicados diretamente sobre os atributos das entidades. - **Criação de DTO:** Utiliza o construtor da classe `ProductMinDTO` para criar novos objetos DTO diretamente na consulta. Este método é típico de JPQL e permite a construção eficiente de objetos de transferência de dados sem necessidade de conversões adicionais no código Java.

Teste das Consultas

O código a seguir, na classe principal `Uri2621Application`, testa as consultas SQL e JPQL implementadas no repositório:

```
@SpringBootApplication
public class Uri2621Application implements CommandLineRunner {

    @Autowired
```

```
    private ProductRepository repository;

    public static void main(String[] args) {
        SpringApplication.run(Uri2621Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {

        List<ProductMinProjection> list = repository.search1(10, 20, "P");
        List<ProductMinDTO> result1 = list.stream().map(x -> new ProductMinDTO(x
   )).collect(Collectors.toList());

        System.out.println("\n*** RESULTADO SQL RAIZ:");
        for (ProductMinDTO obj : result1) {
            System.out.println(obj);
        }

        System.out.println("\n*** RESULTADO JPQL:");
        List<ProductMinDTO> result2 = repository.search2(10, 20, "P");
        for (ProductMinDTO obj : result2) {
            System.out.println(obj);
        }
    }
}
```

4.26 Dica - pratique um pouco se você estiver precisando

Esta seção é para avisar que, caso você sinta que esteja precisando treinar um pouco de SQL básico, este é um ótimo momento para isso, pois nós acabamos de fazer três exercícios mais básicos do Beecrowd, então neste momento você está com uma boa preparação para praticar com os exercícios mais fáceis, antes de prosseguirmos com os exercícios mais difíceis.

Caso queira praticar, recomendamos fazer os exercícios dos grupos 1 e 2 que selecionamos:

- **Grupo 1: projeção, restrição**: exercícios 2602, 2603, 2604, 2607, 2608, 2615, 2624
- **Grupo 2: JOIN**: exercícios 2605, 2606, 2611, 2613, 2614, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2742

4.27 URI 2609 Elaborando a Consulta

Enunciado do Exercício

Como de costume o setor de vendas está fazendo uma análise de quantos produtos temos em estoque, e você poderá ajudar eles.

Então seu trabalho será exibir o nome e a quantidade de produtos de cada uma categoria.

Estrutura da Base de Dados

A base de dados para este exercício está estruturada da seguinte maneira:

```sql
CREATE TABLE categories (
  id numeric PRIMARY KEY,
  name varchar
);

CREATE TABLE products (
  id numeric PRIMARY KEY,
  name varchar(50),
  amount numeric,
  price numeric(7,2),
  id_categories numeric REFERENCES categories (id)
);

INSERT INTO categories (id, name)
VALUES
  (1,   'wood'),
  (2,   'luxury'),
  (3,   'vintage'),
  (4,   'modern'),
  (5,   'super luxury');

INSERT INTO products (id, name, amount, price, id_categories)
VALUES
  (1,   'Two-doors wardrobe', 100,  800,    1),
  (2,   'Dining table', 1000,   560,    3),
  (3,   'Towel holder', 10000,  25.50,  4),
  (4,   'Computer desk',    350,    320.50, 2),
  (5,   'Chair',    3000,   210.64, 4),
  (6,   'Single bed',   750,    460,    1);
```

Saída Esperada

Com base nos dados inseridos na base de dados, a saída esperada para a consulta, em formato de tabela, é a seguinte:

name	sum
luxury	350
modern	13000
wood	850
vintage	1000

Solução do Exercício

A consulta SQL que resolve este exercício é:

```sql
SELECT categories.name, SUM(products.amount)
FROM categories
INNER JOIN products ON products.id_categories = categories.id
GROUP BY categories.name
```

Explicação Detalhada da Consulta

- `SELECT categories.name, SUM(products.amount)`:
  - Esta linha seleciona duas colunas: `name` da tabela `categories` e a soma (`SUM()`) dos valores da coluna `amount` da tabela `products`. `SUM(products.amount)` calcula o total de produtos para cada categoria.
- `FROM categories`:
  - Especifica que a tabela principal para a consulta é `categories`.
- `INNER JOIN products ON products.id_categories = categories.id`:
  - Este comando realiza uma junção interna (`INNER JOIN`) entre as tabelas `categories` e `products`. A junção é feita onde o `id` das categorias corresponde ao `id_categories` dos produtos.
- `GROUP BY categories.name`:
  - Agrupa os resultados pelo nome da categoria. Isso é necessário porque a função `SUM()` é uma função de agregação que combina várias linhas de entrada em grupos de resumo, neste caso, baseados nos nomes das categorias.

4.28 URI 2609 Spring Boot SQL e JPQL

Agora vamos dar continuidade ao exercício URI 2609, elaborando as consultas SQL e JPQL no projeto Spring Boot. Abra o projeto "uri2609" fornecido e crie uma base de dados chamada "uri2609" no banco de dados PostgreSQL do seu computador.

Modelo de Domínio

O projeto Spring Boot contém o modelo de domínio que corresponde ao modelo relacional do banco de dados:

```
@Entity
@Table(name = "categories")
public class Category {

    @Id
    private Long id;
    private String name;

    @OneToMany(mappedBy = "category")
    private List<Product> products = new ArrayList<>();

    ...
```

```
@Entity
@Table(name = "products")
public class Product {

    @Id
    private Long id;
    private String name;
    private Integer amount;
```

```
    private BigDecimal price;

    @ManyToOne
    @JoinColumn(name = "id_categories")
    private Category category;

    ...
```

Projections

Foi necessário implementar uma projection para representar o resultado da consulta SQL:

```
package com.devsuperior.uri2609.projections;

public interface CategorySumProjection {

    String getName();
    Long getSum();
}
```

DTO

Foi implementado também um DTO para representar os dados resultantes:

```
package com.devsuperior.uri2609.dto;

import com.devsuperior.uri2609.projections.CategorySumProjection;

public class CategorySumDTO {

    private String name;
    private Long sum;

    public CategorySumDTO() {
    }

    public CategorySumDTO(String name, Long sum) {
        this.name = name;
        this.sum = sum;
    }

    public CategorySumDTO(CategorySumProjection projection) {
        name = projection.getName();
        sum = projection.getSum();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public Long getSum() {
        return sum;
    }

    public void setSum(Long sum) {
        this.sum = sum;
    }

    @Override
    public String toString() {
        return "CategorySumDTO [name=" + name + ", sum=" + sum + "]";
    }
}
```

Consultas SQL e JPQL

O repositório foi implementado com as consultas SQL e JPQL. Vamos detalhar cada parte do código:

```
package com.devsuperior.uri2609.repositories;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import com.devsuperior.uri2609.dto.CategorySumDTO;
import com.devsuperior.uri2609.entities.Category;
import com.devsuperior.uri2609.projections.CategorySumProjection;

public interface CategoryRepository extends JpaRepository<Category, Long> {

    @Query(nativeQuery = true, value = "SELECT categories.name, SUM(products.
    amount) "
            + "FROM categories "
            + "INNER JOIN products ON products.id_categories = categories.id "
            + "GROUP BY categories.name")
    List<CategorySumProjection> search1();

    @Query("SELECT new com.devsuperior.uri2609.dto.CategorySumDTO(obj.category.
    name, SUM(obj.amount)) "
            + "FROM Product obj "
            + "GROUP BY obj.category.name")
    List<CategorySumDTO> search2();
}
```

- `search1()`: Este método utiliza uma consulta SQL nativa para somar a quantidade de produtos por categoria. O resultado é agrupado pelo nome da categoria.
- `search2()`: Este método utiliza JPQL para realizar a mesma tarefa, criando instâncias de

`CategorySumDTO` diretamente na consulta, o que pode melhorar a performance ao evitar o carregamento completo dos objetos de domínio.

Testando a Solução

O código abaixo foi implementado na classe principal do projeto Spring Boot para testar as consultas:

```
package com.devsuperior.uri2609;

import java.util.List;
import java.util.stream.Collectors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.devsuperior.uri2609.dto.CategorySumDTO;
import com.devsuperior.uri2609.projections.CategorySumProjection;
import com.devsuperior.uri2609.repositories.CategoryRepository;

@SpringBootApplication
public class Uri2609Application implements CommandLineRunner {

    @Autowired
    private CategoryRepository repository;

    public static void main(String[] args) {
        SpringApplication.run(Uri2609Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {

        List<CategorySumProjection> list = repository.search1();
        List<CategorySumDTO> result1 = list.stream().map(x -> new CategorySumDTO
   (x)).collect(Collectors.toList());

        System.out.println("\n*** RESULTADO SQL RAIZ:");
        for (CategorySumDTO obj : result1) {
            System.out.println(obj);
        }
        System.out.println("\n\n");

        List<CategorySumDTO> result2 = repository.search2();

        System.out.println("\n*** RESULTADO JPQL:");
        for (CategorySumDTO obj : result2) {
            System.out.println(obj);
        }
    }
}
```

4.29 URI 2737 Elaborando a consulta

Enunciado do Exercício

O diretor da Mangojata Advogados ordenou que lhe fosse entregue um relatório sobre seus advogados atuais.

O diretor quer que você mostre para ele o nome do advogado que têm mais clientes, o nome do advogado que tem menos clientes e a média de clientes entre todos os advogados.

OBS: Antes de apresentar a média mostre um campo chamado Average a fim de deixar o relatório mais apresentável. A média deverá ser apresentada em inteiros.

Estrutura da Base de Dados

O seguinte script SQL define a estrutura da tabela `lawyers` e insere alguns dados representando advogados e o número de clientes que cada um atende:

```
CREATE TABLE lawyers(
  register INTEGER PRIMARY KEY,
  name VARCHAR(255),
  customers_number INTEGER
 );

INSERT INTO lawyers(register, name, customers_number)
VALUES
  (1648, 'Marty M. Harrison', 5),
  (2427, 'Jonathan J. Blevins', 15),
  (3365, 'Chelsey D. Sanders', 20),
  (4153, 'Dorothy W. Ford', 16),
  (5525, 'Penny J. Cormier', 6);
```

Saída Esperada

A saída esperada para a base de dados apresentada é a seguinte:

name	customers_number
Chelsey D. Sanders	20
Marty M. Harrison	5
Average	12

Solução do Exercício

A consulta SQL que soluciona o exercício é a seguinte:

```
(SELECT name, customers_number
FROM lawyers
WHERE customers_number = (
    SELECT MAX(customers_number)
```

```
    FROM lawyers
))

UNION ALL

(SELECT name, customers_number
FROM lawyers
WHERE customers_number = (
    SELECT MIN(customers_number)
    FROM lawyers
))

UNION ALL

(SELECT 'Average', ROUND(AVG(customers_number), 0)
FROM lawyers)
```

Explicação Detalhada da Consulta:

1. **Primeira Subconsulta:**
   - `SELECT name, customers_number FROM lawyers WHERE customers_number = (SELECT MAX(customers_number) FROM lawyers)`
   - Esta subconsulta seleciona o nome e o número de clientes do advogado com o maior número de clientes.
2. **Segunda Subconsulta:**
   - `SELECT name, customers_number FROM lawyers WHERE customers_number = (SELECT MIN(customers_number) FROM lawyers)`
   - Esta subconsulta seleciona o nome e o número de clientes do advogado com o menor número de clientes.
3. **Terceira Subconsulta:**
   - `SELECT 'Average', ROUND(AVG(customers_number), 0) FROM lawyers`
   - Esta subconsulta calcula a média (em número inteiro, usando a função `ROUND`) do número de clientes de todos os advogados. Aqui, 'Average' é usado como um rótulo literal para tornar a saída mais apresentável.

Estas subconsultas são combinadas usando o operador `UNION ALL`, que concatena os resultados de cada subconsulta em um único conjunto de resultados.

4.30 URI 2737 Solução alternativa

Agora vamos dar continuidade ao exercício, elaborando uma versão alternativa da solução, usando uma abordagem diferente para o script SQL. Esta abordagem utiliza a ordenação dos registros e a limitação dos resultados para obter os advogados com o máximo e mínimo número de clientes, assim como a média dos clientes.

```
(SELECT name, customers_number
FROM lawyers
ORDER BY customers_number DESC
LIMIT 1)

UNION ALL
```

```
(SELECT name, customers_number
FROM lawyers
ORDER BY customers_number ASC
LIMIT 1)

UNION ALL

(SELECT 'Average', ROUND(AVG(customers_number), 0)
FROM lawyers)
```

1. **Primeira Subconsulta:**
   - `SELECT name, customers_number FROM lawyers ORDER BY customers_number DESC LIMIT 1`
   - Esta subconsulta seleciona o advogado com o maior número de clientes. Ela ordena todos os advogados pelo número de clientes em ordem decrescente (`DESC`) e limita o resultado ao primeiro registro (`LIMIT 1`), ou seja, o advogado com o maior número de clientes.
2. **Segunda Subconsulta:**
   - `SELECT name, customers_number FROM lawyers ORDER BY customers_number ASC LIMIT 1`
   - De forma similar à primeira subconsulta, mas em contraste direto, esta consulta ordena os advogados pelo número de clientes em ordem ascendente (`ASC`) e seleciona o primeiro na lista, que será o advogado com o menor número de clientes.
3. **Terceira Subconsulta:**
   - `SELECT 'Average', ROUND(AVG(customers_number), 0) FROM lawyers`
   - Esta parte da consulta calcula a média do número de clientes entre todos os advogados, utilizando a função `AVG()`. A função `ROUND()` é utilizada para arredondar o valor médio para o número inteiro mais próximo. O uso de `'Average'` como uma string literal serve para etiquetar o resultado, tornando o relatório mais claro e legível.

Essas subconsultas são novamente unidas usando `UNION ALL`, que permite combinar os resultados de múltiplas consultas em um único conjunto de resultados, mantendo todas as linhas de cada subconsulta, incluindo duplicatas. Esta abordagem alternativa, ao contrário da primeira que se baseia em funções agregadas condicionais, emprega ordenação e limitação para obter os registros desejados, que pode ser mais direta e intuitiva em certos contextos de banco de dados, especialmente em casos onde o desempenho não é crítico ou as tabelas não são excessivamente grandes.

4.31 URI 2737 Spring Boot SQL

Continuamos agora o exercício, implementando a consulta SQL no projeto Spring Boot. Abra o projeto "uri2737" fornecido e crie uma base de dados chamada "uri2737" no banco de dados PostgreSQL do seu computador.

Modelo de Domínio

O projeto Spring Boot já possui o modelo de domínio implementado, que reflete o modelo relacional da base de dados:

```
@Entity
@Table(name = "lawyers")
public class Lawyer {

    @Id
    private Long register;
    private String name;
    private Integer customersNumber;

    ...
}
```

Este modelo contém as entidades correspondentes à tabela `lawyers` do banco de dados, com campos para o registro (chave primária), nome e número de clientes.

Projections

Foi necessário implementar uma projection para representar o resultado esperado da consulta SQL:

```
package com.devsuperior.uri2737.projections;

public interface LawyerMinProjection {

    String getName();
    Integer getCustomersNumber();
}
```

Esta projection é utilizada para capturar o nome e o número de clientes dos advogados diretamente da consulta ao banco de dados.

DTO

Foi implementado um DTO para padronizar o resultado da consulta:

```
package com.devsuperior.uri2737.dto;

import com.devsuperior.uri2737.projections.LawyerMinProjection;

public class LawyerMinDTO {

    private String name;
    private Integer customersNumber;

    public LawyerMinDTO() {
    }

    public LawyerMinDTO(String name, Integer customersNumber) {
        this.name = name;
        this.customersNumber = customersNumber;
    }
```

```
    public LawyerMinDTO(LawyerMinProjection projection) {
        name = projection.getName();
        customersNumber = projection.getCustomersNumber();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getCustomersNumber() {
        return customersNumber;
    }

    public void setCustomersNumber(Integer customersNumber) {
        this.customersNumber = customersNumber;
    }

    @Override
    public String toString() {
        return "LawyerMinDTO [name=" + name + ", customersNumber=" +
   customersNumber + "]";
    }
}
```

Consulta SQL

O repositório foi configurado com a consulta SQL necessária para executar o relatório desejado pelo exercício:

```
package com.devsuperior.uri2737.repositories;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import com.devsuperior.uri2737.entities.Lawyer;
import com.devsuperior.uri2737.projections.LawyerMinProjection;

public interface LawyerRepository extends JpaRepository<Lawyer, Long> {

    @Query(nativeQuery = true, value = "SELECT name, customers_number AS
   customersNumber "
            + "FROM lawyers "
            + "WHERE customers_number = ( "
            + "    SELECT MAX(customers_number) "
            + "    FROM lawyers "
            + ") "
            + "UNION ALL "
```

```
            + "SELECT name, customers_number "
            + "FROM lawyers "
            + "WHERE customers_number = ( "
            + "    SELECT MIN(customers_number) "
            + "    FROM lawyers "
            + ") "
            + "UNION ALL "
            + "SELECT 'Average', ROUND(AVG(customers_number), 0) "
            + "FROM lawyers")
    List<LawyerMinProjection> search1();
}
```

A consulta utiliza subconsultas para encontrar o advogado com o maior e menor número de clientes, e para calcular a média do número de clientes, apresentando os resultados em um formato fácil de entender.

Testando a Solução

A classe principal do projeto Spring Boot é configurada para executar e testar a consulta:

```
package com.devsuperior.uri2737;

import java.util.List;
import java.util.stream.Collectors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.devsuperior.uri2737.dto.LawyerMinDTO;
import com.devsuperior.uri2737.projections.LawyerMinProjection;
import com.devsuperior.uri2737.repositories.LawyerRepository;

@SpringBootApplication
public class Uri2737Application implements CommandLineRunner {

    @Autowired
    private LawyerRepository repository;

    public static void main(String[] args) {
        SpringApplication.run(Uri2737Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {

        List<LawyerMinProjection> list = repository.search1();
        List<LawyerMinDTO> result1 = list.stream().map(x -> new LawyerMinDTO(x))
   .collect(Collectors.toList());

        System.out.println("\n*** RESULTADO SQL RAIZ:");
        for (LawyerMinDTO obj : result1) {
            System.out.println(obj);
```

```
        }
    }
}
```

4.32 URI 2990 Elaborando a consulta

Enunciado do Exercício

"Mostrar o CPF, nome dos empregados e o nome do departamento dos empregados que não trabalham em nenhum projeto. O resultado deve estar ordenado por cpf."

Estrutura da Base de Dados

O script para criar e popular as tabelas no banco de dados é:

```
CREATE TABLE empregados (
    cpf VARCHAR(15) PRIMARY KEY,
    enome CHARACTER VARYING (255),
    salary FLOAT,
    cpf_supervisor VARCHAR(15),
    dnumero NUMERIC
);

CREATE TABLE departamentos (
    dnumero NUMERIC PRIMARY KEY,
    dnome CHARACTER VARYING (60),
    cpf_gerente VARCHAR(15) REFERENCES empregados (cpf)
);

CREATE TABLE trabalha (
    cpf_emp VARCHAR(15) REFERENCES empregados(cpf),
    pnumero numeric
);

CREATE TABLE projetos (
    pnumero NUMERIC PRIMARY KEY,
    pnome VARCHAR(45),
    dnumero NUMERIC REFERENCES departamentos (dnumero)
);

INSERT INTO empregados(cpf, enome, salary, cpf_supervisor, dnumero)
VALUES
    ('049382234322', 'João Silva', 2350, '2434332222', 1010),
    ('586733922290', 'Mario Silveira', 3500, '2434332222', 1010),
    ('2434332222', 'Aline Barros', 2350, (null), 1010),
    ('1733332162', 'Tulio Vidal', 8350, (null), 1020),
    ('4244435272', 'Juliana Rodrigues', 3310, (null), 1020),
    ('1014332672', 'Natalia Marques', 2900, (null), 1010);

INSERT INTO departamentos(dnumero, dnome, cpf_gerente)
VALUES
    (1010, 'Pesquisa', '049382234322'),
    (1020, 'Ensino', '2434332222');
```

```
INSERT INTO trabalha (cpf_emp, pnumero)
VALUES
    ('049382234322', 2010),
    ('586733922290', 2020),
    ('049382234322', 2020);

INSERT INTO projetos (pnumero, pnome, dnumero)
VALUES
    (2010, 'Alpha', 1010),
    (2020, 'Beta', 1020);
```

Saída Esperada

O resultado esperado, apresentado em formato de tabela, é:

cpf	enome	dnome
1014332672	Natalia Marques	Pesquisa
1733332162	Tulio Vidal	Ensino
2434332222	Aline Barros	Pesquisa
4244435272	Juliana Rodrigues	Ensino

Solução do Exercício

A consulta SQL proposta para resolver o problema é:

```
SELECT empregados.cpf, empregados.enome, departamentos.dnome
FROM empregados
INNER JOIN departamentos ON empregados.dnumero = departamentos.dnumero
WHERE empregados.cpf NOT IN (
    SELECT empregados.cpf
    FROM empregados
    INNER JOIN trabalha ON trabalha.cpf_emp = empregados.cpf
)
ORDER BY empregados.cpf
```

Explicação Detalhada da Consulta

1. **Join entre Empregados e Departamentos**: O `INNER JOIN` inicial liga as tabelas `empregados` e `departamentos` utilizando a coluna `dnumero`, que relaciona cada empregado ao seu departamento.
2. **Subconsulta para Empregados em Projetos**: A cláusula `WHERE` utiliza uma subconsulta que seleciona os CPFs dos empregados que estão registrados na tabela `trabalha`. A condição `NOT IN` é usada para excluir esses empregados da seleção principal.
3. **Ordenação por CPF**: A cláusula `ORDER BY` assegura que os resultados são ordenados pelo CPF dos empregados, conforme solicitado pelo enunciado.

4.33 URI 2990 Solução alternativa com LEFT JOIN

Após revisar o exercício e entender as consultas anteriores, vamos explorar uma abordagem alternativa para resolver o mesmo problema usando um `LEFT JOIN`. Esta técnica oferece uma maneira diferente de filtrar empregados que não participam de nenhum projeto, e pode ser mais intuitiva em certos contextos.

O script SQL proposto para uma solução alternativa é:

```
SELECT empregados.cpf, empregados.enome, departamentos.dnome
FROM empregados
INNER JOIN departamentos ON empregados.dnumero = departamentos.dnumero
LEFT JOIN trabalha ON trabalha.cpf_emp = empregados.cpf
WHERE trabalha.cpf_emp IS NULL
ORDER BY empregados.cpf
```

Explicação Detalhada do Script

1. **Join entre Empregados e Departamentos**: Inicialmente, realizamos um `INNER JOIN` entre as tabelas `empregados` e `departamentos` utilizando a coluna `dnumero`. Isso garante que cada empregado seja associado ao seu respectivo departamento, fornecendo acesso ao nome do departamento para o resultado final.
2. **Left Join com Trabalha**: Em seguida, aplicamos um `LEFT JOIN` com a tabela `trabalha` , usando a coluna `cpf_emp` para relacionar os empregados com seus projetos. O `LEFT JOIN` é crucial aqui porque ele inclui todos os empregados, mesmo aqueles que não têm correspondência na tabela `trabalha`, ou seja, que não estão vinculados a nenhum projeto.
3. **Condição para Empregados Sem Projetos**: A condição no `WHERE trabalha.cpf_emp IS NULL` filtra apenas os empregados que não possuem entrada na tabela `trabalha`. Como o `LEFT JOIN` inclui todos os empregados e preenche com `NULL` os que não têm projetos, essa condição seleciona efetivamente os empregados que estão fora de qualquer projeto.
4. **Ordenação dos Resultados**: Finalmente, a cláusula `ORDER BY empregados.cpf` organiza os resultados pelo CPF dos empregados, atendendo ao requisito de ordenação do enunciado do exercício.

Vantagens desta Abordagem

- **Clareza**: Utilizar um `LEFT JOIN` com uma condição de `NULL` pode ser mais direto para entender que estamos procurando por empregados sem projetos, especialmente para quem está familiarizado com a semântica de joins.
- **Eficiência**: Em alguns casos, especialmente em bancos de dados otimizados para operações de join, essa consulta pode ser mais eficiente do que usar subconsultas, dependendo da implementação do otimizador de consulta do SGBD.

4.34 URI 2990 Spring Boot SQL e JPQL

Agora vamos dar continuidade ao exercício, elaborando as consultas SQL e JPQL no projeto Spring Boot. Abra o projeto "uri2990" que foi fornecido e crie uma base de dados chamada "uri2990" no banco de dados Postgresql do seu computador.

Modelo de Domínio

O projeto Spring Boot já contém o modelo de domínio implementado, que corresponde ao modelo relacional do banco de dados:

```
@Entity
@Table(name = "departamentos")
public class Departamento {

    @Id
    private Long dnumero;
    private String dnome;

    @OneToMany(mappedBy = "departamento")
    private List<Empregado> empregados = new ArrayList<>();

    @ManyToOne
    @JoinColumn(name = "cpf_gerente")
    private Empregado gerente;

    ...
```

```
@Entity
@Table(name = "empregados")
public class Empregado {

    @Id
    private String cpf;
    private String enome;
    private Double salary;

    @ManyToOne
    @JoinColumn(name = "cpf_supervisor")
    private Empregado supervisor;

    @OneToMany(mappedBy = "supervisor")
    private List<Empregado> supervisionados = new ArrayList<>();

    @ManyToOne
    @JoinColumn(name = "dnumero")
    private Departamento departamento;

    @ManyToMany
    @JoinTable(name = "trabalha",
        joinColumns = @JoinColumn(name = "cpf_emp"),
        inverseJoinColumns = @JoinColumn(name = "pnumero"))
    private Set<Projeto> projetosOndeTrabalha = new HashSet<>();
```

```
    ...
```

Projections

Implementamos uma projection para representar o resultado da consulta SQL:

```
package com.devsuperior.uri2990.projections;

public interface EmpregadoDeptProjection {

    String getCpf();
    String getEnome();
    String getDnome();
}
```

DTO

Implementamos um DTO para representar o resultado da consulta:

```
package com.devsuperior.uri2990.dto;

import com.devsuperior.uri2990.projections.EmpregadoDeptProjection;

public class EmpregadoDeptDTO {

    private String cpf;
    private String enome;
    private String dnome;

    public EmpregadoDeptDTO() {
    }

    public EmpregadoDeptDTO(String cpf, String enome, String dnome) {
        this.cpf = cpf;
        this.enome = enome;
        this.dnome = dnome;
    }

    public EmpregadoDeptDTO(EmpregadoDeptProjection projection) {
        cpf = projection.getCpf();
        enome = projection.getEnome();
        dnome = projection.getDnome();
    }

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
```

```
    }

    public String getEnome() {
        return enome;
    }

    public void setEnome(String enome) {
        this.enome = enome;
    }

    public String getDnome() {
        return dnome;
    }

    public void setDnome(String dnome) {
        this.dnome = dnome;
    }

    @Override
    public String toString() {
        return "EmpregadoDeptDTO [cpf=" + cpf + ", enome=" + enome + ", dnome="
   + dnome + "]";
    }
}
```

Consultas SQL e JPQL

O código do repository é detalhado abaixo, cada consulta serve para identificar empregados que não trabalham em nenhum projeto, utilizando diferentes abordagens:

```
package com.devsuperior.uri2990.repositories;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import com.devsuperior.uri2990.dto.EmpregadoDeptDTO;
import com.devsuperior.uri2990.entities.Empregado;
import com.devsuperior.uri2990.projections.EmpregadoDeptProjection;

public interface EmpregadoRepository extends JpaRepository<Empregado, Long> {

    @Query(nativeQuery = true, value = "SELECT empregados.cpf, empregados.enome,
    departamentos.dnome "
            + "FROM empregados "
            + "INNER JOIN departamentos ON departamentos.dnumero = empregados.
   dnumero "
            + "WHERE empregados.cpf NOT IN ( "
            + "    SELECT empregados.cpf "
            + "    FROM empregados "
            + "    INNER JOIN trabalha ON empregados.cpf = trabalha.cpf_emp "
            + ") "
            + "ORDER BY empregados.cpf")
```

```
    List<EmpregadoDeptProjection> search1();

    @Query("SELECT new com.devsuperior.uri2990.dto.EmpregadoDeptDTO(obj.cpf, obj
    .enome, obj.departamento.dnome) "
            + "FROM Empregado obj "
            + "WHERE obj.cpf NOT IN ( "
            + "    SELECT obj.cpf "
            + "    FROM Empregado obj "
            + "    INNER JOIN obj.projetosOndeTrabalha "
            + ") "
            + "ORDER BY obj.cpf")
    List<EmpregadoDeptDTO> search2();

    @Query(nativeQuery = true, value =

 "SELECT empregados.cpf, empregados.enome, departamentos.dnome "
            + "FROM empregados "
            + "INNER JOIN departamentos ON departamentos.dnumero = empregados.
    dnumero "
            + "LEFT JOIN trabalha ON empregados.cpf = trabalha.cpf_emp "
            + "WHERE pnumero IS NULL "
            + "ORDER BY empregados.cpf")
    List<EmpregadoDeptProjection> search3();
}
```

Testando a Solução

O código abaixo na classe principal do projeto Spring Boot testa as três consultas:

```
package com.devsuperior.uri2990;

import java.util.List;
import java.util.stream.Collectors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.devsuperior.uri2990.dto.EmpregadoDeptDTO;
import com.devsuperior.uri2990.projections.EmpregadoDeptProjection;
import com.devsuperior.uri2990.repositories.EmpregadoRepository;

@SpringBootApplication
public class Uri2990Application implements CommandLineRunner {

    @Autowired
    private EmpregadoRepository repository;

    public static void main(String[] args) {
        SpringApplication.run(Uri2990Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
```

```
        List<EmpregadoDeptProjection> list = repository.search1();
        List<EmpregadoDeptDTO> result1 = list.stream().map(x -> new
    EmpregadoDeptDTO(x)).collect(Collectors.toList());

        System.out.println("\n*** RESULTADO SQL RAIZ:");
        for (EmpregadoDeptDTO obj : result1) {
            System.out.println(obj);
        }
        System.out.println("\n\n");

        List<EmpregadoDeptDTO> result2 = repository.search2();

        System.out.println("\n*** RESULTADO JPQL:");
        for (EmpregadoDeptDTO obj : result2) {
            System.out.println(obj);
        }
        System.out.println("\n\n");

        List<EmpregadoDeptProjection> list3 = repository.search3();
        List<EmpregadoDeptDTO> result3 = list3.stream().map(x -> new
    EmpregadoDeptDTO(x)).collect(Collectors.toList());

        System.out.println("\n*** RESULTADO SQL RAIZ:");
        for (EmpregadoDeptDTO obj : result3) {
            System.out.println(obj);
        }
        System.out.println("\n\n");
    }
}
```

4.35 DSCommerce Consulta de Produtos por Nome

Nesta seção, vamos explicar como implementar uma consulta paginada de produtos no DSCommerce, com a possibilidade de filtrar produtos pelo trecho do nome.

Recapitulação dos Códigos de Entidade e DTO de Produto

Primeiro, vamos recapitular as definições das entidades e DTOs que serão utilizadas nesta consulta:

```
@Entity
@Table(name = "tb_product")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @Column(columnDefinition = "TEXT")
    private String description;
    private Double price;
    private String imgUrl;
```

```
    @ManyToMany
    @JoinTable(name = "tb_product_category",
            joinColumns = @JoinColumn(name = "product_id"),
            inverseJoinColumns = @JoinColumn(name = "category_id"))
    private Set<Category> categories = new HashSet<>();

    @OneToMany(mappedBy = "id.product")
    private Set<OrderItem> items = new HashSet<>();

    ...
}
```

```
public class ProductDTO {

    private Long id;

    @Size(min = 3, max = 80, message = "Nome precisar ter de 3 a 80 caracteres")
    @NotBlank(message = "Campo requerido")
    private String name;

    @Size(min = 10, message = "Descrição precisa ter no mínimo 10 caracteres")
    @NotBlank(message = "Campo requerido")
    private String description;

    @NotNull(message = "Campo requerido")
    @Positive(message = "O preço deve ser positivo")
    private Double price;

    private String imgUrl;

    ...
}
```

Repository: Método de Pesquisa por Nome

No `ProductRepository`, implementamos o método `searchByName` que usa uma consulta JPQL para buscar produtos pelo nome:

```
package com.devsuperior.dscommerce.repositories;

import com.devsuperior.dscommerce.entities.Product;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

public interface ProductRepository extends JpaRepository<Product, Long> {

    @Query("SELECT obj FROM Product obj " +
            "WHERE UPPER(obj.name) LIKE UPPER(CONCAT('%', :name, '%'))")
    Page<Product> searchByName(String name, Pageable pageable);
```

```
}
```

Service: Método FindAll

No `ProductService`, o método `findAll` utiliza o `ProductRepository` para realizar a consulta e transformar o resultado em DTOs:

```
@Service
public class ProductService {

    @Autowired
    private ProductRepository repository;

    @Transactional(readOnly = true)
    public Page<ProductDTO> findAll(String name, Pageable pageable) {
        Page<Product> result = repository.searchByName(name, pageable);
        return result.map(x -> new ProductDTO(x));
    }

    ...
}
```

Controller: Método FindAll

No `ProductController`, definimos o endpoint para acessar a consulta. Os parâmetros `name` e `pageable` são opcionais, permitindo filtrar por nome e paginar os resultados:

```
@RestController
@RequestMapping(value = "/products")
public class ProductController {

    @Autowired
    private ProductService service;

    @GetMapping
    public ResponseEntity<Page<ProductDTO>> findAll(
            @RequestParam(name = "name", defaultValue = "") String name,
            Pageable pageable) {
        Page<ProductDTO> dto = service.findAll(name, pageable);
        return ResponseEntity.ok(dto);
    }

    ...
}
```

Testando a Requisição no Postman

Para testar essa funcionalidade no Postman, você pode acessar as seguintes URLs, dependendo dos parâmetros desejados:

- Sem parâmetros de consulta: `http://localhost:8080/products`
- Com parâmetros de paginação: `http://localhost:8080/products?size=12&page=0&sort=name,desc`
- Filtrando pelo nome do produto: `http://localhost:8080/products?size=12&page=0&sort=name,desc&name=mac`

Cada uma dessas requisições irá retornar uma página de produtos de acordo com os critérios especificados, permitindo um controle refinado sobre os resultados exibidos.

4.36 Evitando consultas lentas muitos-para-muitos

A seguir, será mostrado um projeto Spring Boot com uma solução ineficiente para uma consulta de produtos e categorias, que apresenta o problema das consultas N+1, e em seguida, será apresentada uma solução eficiente, sem o problema das N+1 consultas. Confira o projeto no GitHub:

https://github.com/devsuperior/aulao_nmais1

Modelo de Domínio

Este é um projeto com um modelo de domínio de produtos e categorias, com um relacionamento muitos-para-muitos entre eles:

```
@Entity
@Table(name = "tb_product")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @ManyToMany
    @JoinTable(name = "tb_product_category",
        joinColumns = @JoinColumn(name = "product_id"),
        inverseJoinColumns = @JoinColumn(name = "category_id"))
    private Set<Category> categories = new HashSet<>();
}
```

```
@Entity
@Table(name = "tb_category")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    private String name;

    @ManyToMany(mappedBy = "categories")
    private Set<Product> products = new HashSet<>();
}
```

DTOs para Produto e Categoria

```
public class ProductDTO {

    private Long id;
    private String name;

    private List<CategoryDTO> categories = new ArrayList<>();
}
```

```
public class CategoryDTO {

    private Long id;
    private String name;
}
```

Repository para Produtos

```
public interface ProductRepository extends JpaRepository<Product, Long> {

}
```

Implementação da Consulta na Camada de Serviço

```
@Service
public class ProductService {

    @Autowired
    private ProductRepository repository;

    @Transactional(readOnly = true)
    public Page<ProductDTO> find(PageRequest pageRequest) {
        Page<Product> list = repository.findAll(pageRequest);
        return list.map(x -> new ProductDTO(x));
    }
}
```

Controlador para Produtos

```
@RestController
@RequestMapping(value = "/products")
public class ProductResource {

    @Autowired
    private ProductService service;

    @GetMapping
    public ResponseEntity<Page<ProductDTO>> findAll(
            @RequestParam(value = "page", defaultValue = "0") Integer page,
            @RequestParam(value = "size", defaultValue = "10") Integer size
            ) {

        PageRequest pageRequest = PageRequest.of(page, size);
        Page<ProductDTO> list = service.find(pageRequest);
        return ResponseEntity.ok(list);
    }
}
```

Problema N+1 Consultas

A implementação acima pode resultar em múltiplas consultas ao banco de dados devido ao carregamento lazy das categorias dos produtos, caracterizando o problema das n+1 consultas.

Você pode observar esse problema executando a requisição no Postman para recuperar do banco de dados uma página de produtos, mas se você observar no console do projeto Spring, você verá que várias consultas SQL foram enviadas ao banco de dados, o que é indesejado e causa degradação de performance na aplicação, e tráfego indesejado no banco de dados.

Solução do Problema N+1 Consultas

A solução para o problema das consultas N+1 envolve a otimização do método de busca no repositório para carregar todos os dados necessários em uma ou poucas consultas, reduzindo assim o número total de interações com o banco de dados durante uma operação de busca paginada de produtos e suas categorias associadas.

Modificação do Repository

```
@Query("SELECT obj FROM Product obj JOIN FETCH obj.categories WHERE obj IN :
    products")
List<Product> findProductsCategories(List<Product> products);
```

Neste método `findProductsCategories`, usamos a cláusula `JOIN FETCH` para carregar os produtos juntamente com suas categorias em uma única consulta. O `JOIN FETCH` é essencial para evitar consultas adicionais (lazy loading) quando acessamos as categorias de cada produto. Este método espera uma lista de produtos (`:products`), que são passados como parâmetro. O resultado

é que cada produto na lista já vem com suas categorias carregadas, eliminando a necessidade de consultas subsequentes quando essas categorias são acessadas.

Modificação no Serviço

```
@Transactional(readOnly = true)
public Page<ProductDTO> find(PageRequest pageRequest) {
    Page<Product> page = repository.findAll(pageRequest);
    repository.findProductsCategories(page.getContent());
    return page.map(x -> new ProductDTO(x));
}
```

No serviço `ProductService`, ajustamos o método `find` para fazer uso do novo método do repositório. Primeiro, buscamos uma página de produtos com `repository.findAll(pageRequest)`. Este passo inicialmente carrega apenas os produtos sem suas categorias devido ao comportamento padrão do JPA, que não inclui relacionamentos muitos-para-muitos em uma consulta básica de paginação.

Depois dessa primeira busca, passamos os produtos recuperados como argumento para `findProductsCategories`. Este método carrega efetivamente as categorias para os produtos da página atual, evitando o problema das consultas N+1.

Finalmente, convertemos a página de entidades `Product` em `ProductDTO`, mapeando cada produto para seu respectivo DTO. O método `map` é usado para transformar cada entidade `Product` em um `ProductDTO`, que já inclui as categorias devido ao passo anterior. Este passo é crucial para garantir que os dados são transferidos corretamente do modelo de domínio para o modelo de transferência de dados (DTO), que será usado nas camadas superiores, especialmente na interface de usuário ou em APIs externas.

Essas modificações garantem que todas as operações relacionadas ao carregamento de dados são concluídas eficientemente, reduzindo o número total de consultas ao banco de dados e melhorando a performance da aplicação.

4.37 Evitando Consultas Lentas Muitos-Para-Um com countQuery

Esta seção demonstra como evitar o problema das n+1 consultas em consultas paginadas no Spring, especialmente quando tratamos de buscar entidades com um relacionamento muitos-para-um entre elas. O projeto base para esta seção pode ser obtido no Github no link a seguir:

https://github.com/devsuperior/jpa-queries1

Modelo de Domínio

O projeto de exemplo utiliza um modelo de domínio de empregados e departamentos, com a seguinte estrutura:

```
@Entity
@Table(name = "tb_department")
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    ...
```

```
@Entity
@Table(name = "tb_employee")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private Double salary;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    ...
```

Problema com JOIN FETCH em Consultas Paginadas

Ao tentar realizar uma consulta paginada que inclui um `JOIN FETCH` para trazer os departamentos associados a cada empregado, podemos encontrar problemas relacionados à forma como o JPA calcula a quantidade total de elementos para a paginação. O código problemático seria:

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    @Query(value = "SELECT obj FROM Employee obj JOIN FETCH obj.department")
    Page<Employee> searchAll(Pageable pageable);
}
```

Solução com countQuery

No Spring Data JPA, para contornar esse problema, utilizamos o parâmetro `countQuery`. O `countQuery` é uma SQL adicional que é usada para calcular o número total de registros resultantes da consulta, sem carregar todos os dados, especialmente útil quando a `JOIN FETCH` altera o comportamento padrão de contagem de registros devido à agregação de mais linhas.

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
```

```
    @Query(value = "SELECT obj FROM Employee obj JOIN FETCH obj.department",
            countQuery = "SELECT COUNT(obj) FROM Employee obj JOIN obj.
    department")
    Page<Employee> searchAll(Pageable pageable);
}
```

Implementação do Controlador

O controlador a seguir disponibiliza um endpoint para testar a consulta paginada de empregados:

```
@RestController
@RequestMapping(value = "/employees")
public class EmployeeController {

    @Autowired
    private EmployeeRepository employeeRepository;

    @GetMapping
    public Page<Employee> findAll(Pageable pageable) {
        return employeeRepository.searchAll(pageable);
    }
}
```

Testando a Solução

Para testar essa implementação, você pode usar o Postman para enviar requisições GET para o endpoint `/employees` com parâmetros de paginação, como:

- `http://localhost:8080/employees?page=0&size=10`

Observando o console da aplicação Spring durante a execução da requisição, notaremos que apenas uma única consulta SQL é enviada ao banco de dados para buscar os empregados e seus departamentos, e uma segunda consulta é usada para contar o número total de registros. Esse método evita múltiplas idas ao banco de dados para cada empregado, solucionando o problema das n+1 consultas.

Capítulo 5

Login e controle de acesso

5.1 Visão geral do capítulo

Bem-vindos ao capítulo "Login e controle de acesso". Neste capítulo vamos aprender como incluir segurança ao projeto Spring Boot, e vamos controlar o acesso dos nosso recursos.

Um dos tipos de controle de acesso será por perfil de usuário, onde recursos podem ser ou públicos, ou de acesso a clientes, ou de acesso apenas a administradores. De modo geral, administradores também poderão acessar recursos de clientes.

Além disso, vamos controlar recursos por regras de negócio. Por exemplo, um cliente pode acessar apenas seus próprios pedidos, mas não pode acessar os pedidos de outros clientes.

Vamos implementar acesso por meio de token, e vamos usar o formato JWT para isso.

Ademais, vamos utilizar o padrão OAuth2 para implementar o login e acesso aos recursos, particularmente usando o fluxo password grant, que é aquele no qual o usuário informa suas credenciais (email e senha), e o sistema retorna um token de acesso.

Nossa implementação utilizará o Spring Security, que permite uma implementação flexível do mecanismo de segurança, por meio da implementação de interfaces.

5.2 Material de apoio do capítulo

Para este capítulo, vamos utilizar os seguintes materiais de apoio:

- Slides anexos às aulas na plataforma de ensino
- Projeto DSCommerce ao final do capítulo 5

5.3 Baixando o projeto pronto

Neste momento, caso seja de seu interesse baixar o projeto pronto ao final deste capítulo, e apenas seguir o conteúdo para entendimento rápido, pode fazê-lo.

Basta baixar o projeto no material de apoio e importar em sua IDE favorita. Não esqueça também de baixar a *collection* do Postman, bem como o ambiente do Postman com as variáveis de ambiente para execução do projeto.

5.4 Ideia Geral do Login e Controle de Acesso

O processo de login e controle de acesso em sistemas informatizados é crucial para a segurança da informação e para garantir que apenas usuários autorizados tenham acesso a recursos específicos. Vamos explorar a ideia geral desses processos, dividindo-os em duas partes principais: o login e o acesso a recursos protegidos.

Login

O processo de login é a primeira etapa para acessar um sistema protegido. Ele envolve a seguinte sequência de ações:

1. **Autenticação de Credenciais:** O usuário fornece suas credenciais, como nome de usuário e senha, através de uma interface de login.
2. **Validação:** O sistema verifica as credenciais fornecidas. Se as credenciais estiverem corretas, o sistema gera e retorna um token de acesso. Esse token serve como uma chave temporária que permite ao usuário acessar os recursos protegidos durante um período definido.
3. **Resposta do Sistema:**
   - **Sucesso:** Se as credenciais forem validadas com sucesso, o usuário recebe um token de acesso.
   - **Falha:** Se as credenciais estiverem incorretas, o sistema retorna um erro 401 (Não Autorizado), indicando que o acesso foi negado.

Acesso a Recursos Protegidos

Após realizar o login e obter um token de acesso, o usuário pode tentar acessar recursos protegidos no sistema. O fluxo para acessar um recurso protegido geralmente segue estas etapas:

1. **Requisição do Recurso:** O usuário faz uma solicitação para acessar um recurso protegido. Essa solicitação inclui o token de acesso, que é tipicamente enviado no cabeçalho da requisição.
2. **Verificação do Token:** O sistema verifica o token de acesso para confirmar se é válido e se ainda está dentro do prazo de validade.
3. **Autorização:**
   - **Acesso Concedido:** Se o token é válido, o sistema verifica se o usuário tem permissão para acessar o recurso específico. Se tiver, o sistema retorna o recurso solicitado.
   - **Acesso Negado:** Se o token não for válido ou se o usuário não tiver permissão para acessar o recurso, o sistema pode retornar:
     - **Erro 403 (Proibido):** Indica que o usuário está autenticado, mas não tem permissão para acessar o recurso.
     - **Erro 401 (Não Autorizado):** Indica que o token expirou ou é inválido.

5.5 Visão Geral do OAuth2

O OAuth2 é um protocolo de autorização amplamente adotado que permite que aplicativos de terceiros tenham acesso limitado a um servidor HTTP, em nome de um usuário. É usado para fornecer um acesso seguro e delegado, o que significa que os recursos que o usuário pode acessar em um servidor são disponibilizados para aplicações de terceiros sem expor suas credenciais

de acesso. Vamos explorar os conceitos básicos, como o OAuth2 funciona e seus principais componentes.

Conceitos Básicos

1. **OAuth2:** Protocolo de autorização que permite que um aplicativo obtenha acesso limitado a um serviço HTTP, seja em nome de um usuário ou em nome do próprio aplicativo.
2. **Tokens de Acesso:** São credenciais usadas para acessar recursos protegidos e são emitidos pelo servidor de autorização.
3. **Escopos:** Definem o nível de acesso que o aplicativo tem aos recursos do usuário. Eles são definidos durante o processo de autorização e são incluídos no token de acesso.

Componentes do OAuth2

1. **Cliente:** O aplicativo que deseja acessar os recursos do usuário.
2. **Servidor de Recursos:** O servidor onde os recursos do usuário estão armazenados.
3. **Servidor de Autorização:** O servidor que autentica a identidade do usuário e emite tokens de acesso ao cliente após a autorização adequada.
4. **Proprietário do Recurso:** Geralmente o usuário que concede permissão ao cliente para acessar seus recursos no servidor.

Fluxos de Concessão

OAuth2 define quatro fluxos de concessão para cobrir diversos cenários de aplicação:

1. **Authorization Code:** Usado por aplicativos clientes que são executados em um servidor web. Este é o fluxo mais seguro e comum, pois permite a autenticação do cliente.
2. **Implicit:** Uma versão simplificada e menos segura do fluxo anterior, geralmente usada por aplicativos clientes executados em um navegador usando um script, como JavaScript.
3. **Password Credentials:** Usado por clientes que têm uma relação de confiança com o usuário, como o aplicativo de um serviço que o próprio usuário instalou em seu dispositivo.
4. **Client Credentials:** Usado para controlar o acesso entre aplicações sem a intervenção do usuário, ideal para cenários de servidor para servidor.

Segurança e Considerações

- **Segurança:** OAuth2 usa SSL/TLS para garantir que todos os dados transmitidos, incluindo tokens e credenciais, sejam seguros.
- **Transparência:** Os usuários podem especificar o escopo dos acessos que estão concedendo aos aplicativos clientes.
- **Flexibilidade:** Diversos fluxos de concessão atendem a diferentes tipos de aplicações e necessidades de segurança.

5.6 Login, Credenciais e JWT

Nesta seção, vamos explorar como o processo de login funciona dentro do contexto do OAuth2, usando especificamente o fluxo password grant, que é um dos quatro fluxos de autorização definidos pelo protocolo OAuth2. Este método é frequentemente usado quando a aplicação que

faz a requisição pode ser considerada confiável pelo usuário, como uma aplicação desenvolvida pelo mesmo serviço que fornece os recursos.

Processo de Requisição de Login

O fluxo password grant permite que a aplicação receba um token de acesso diretamente em troca das credenciais do usuário e credenciais do aplicativo. Vejamos os passos e componentes envolvidos:

1. **Requisição de Login:**
   - **Credenciais do Usuário:** Normalmente, são o nome de usuário e senha.
   - **Credenciais da Aplicação:** São o `client_id` e `client_secret`, que identificam a aplicação perante o servidor de autorização.
2. **Formatação da Requisição:**
   - **Corpo da Requisição:** Deve incluir o `username`, `password`, `client_id`, `client_secret`, e também o `grant_type`, que deve ser `password` neste contexto.
   - **Header de Autorização:** As credenciais da aplicação (`client_id` e `client_secret`) podem ser enviadas através de um header de autorização usando o método de codificação básica (Basic Auth). O header seria formatado da seguinte forma:

```
Authorization: Basic <base64 encoded client_id:client_secret>
```

   - **Exemplo de Corpo da Requisição:**

```
POST /oauth/token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=exampleuser&password=123456&client_id=
    client1&client_secret=secret1
```

Resposta da Requisição

Após o envio da requisição com as credenciais corretas, o servidor de autorização responde com um token de acesso, geralmente no formato JWT (JSON Web Token). O JWT é uma maneira compacta e segura de transmitir informações entre partes como um objeto JSON. Este token pode incluir:

- **Header:** Tipicamente consiste no tipo do token (`typ`), que é JWT, e o algoritmo da assinatura (`alg`), como HS256 ou RS256.
- **Payload (Claims):** Contém as declarações (claims) que incluem informações sobre o usuário, a validade do token (`exp`), e outras informações necessárias.
- **Assinatura:** Utilizada para verificar se o token não foi alterado.

Exemplo de Token JWT:

```
{
  "alg": "HS256",
```

```
  "typ": "JWT"
}
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022,
  "exp": 1516242622
}
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
)
```

O token JWT oferece uma forma segura e eficiente de representar as credenciais do usuário para acessar recursos protegidos. O processo de login utilizando o fluxo password grant no OAuth2 é direto, mas deve ser usado com cautela, especialmente em aplicações onde a confiança entre o usuário e a aplicação é absoluta. A manipulação correta do JWT é crucial para a segurança da aplicação, garantindo que os tokens sejam sempre validados e tratados de forma segura.

5.7 Preparando projeto para segurança

Neste momento é preciso baixar o projeto que vamos usar como referência para aprender como incluir segurança em nossos projetos Spring Boot.

Por favor acesse o repositório no Github conforme link a seguir, acesse a pasta `password-grant`, e importe na sua IDE o projeto referência sem segurança, na última versão do Spring Boot. Você pode localizar o projeto sem segurança na versão desejada pelo padrão de nomes, por exemplo: o projeto na versão 3.1.0 do Spring Boot está na subpasta `spring-boot-3-1-0-noauth`.

Além de importar o projeto Spring Boot, importe também a *collection* do Postman e o *environment*, ambos localizados na pasta `password-grant`.

Este projeto será a base para implementar a segurança com Spring Security e OAuth2 que mostraremos nas próximas seções.

Projeto com segurança implementada

Caso você queira, também é possível baixar e executar o projeto com a segurança já implementada. Basta baixar o projeto na versão desejada que não tenha o sufixo `-noauth` em sua subpasta, por exemplo, o projeto da subpasta `spring-boot-3-1-0`.

Passo a passo para executar e testar o projeto

- Importe a coleção e o ambiente do Postman
  - Nota: o caminho da requisição de Login pode ser `/oauth2/token` ou `/oauth/token`, dependendo da versão do Spring Boot. Altere o caminho da requisição, se necessário.
  - Nota: a variável de ambiente `ashost` deve ser definida como `http://localhost:8080` se você estiver executando um projeto que contém tanto o Servidor de Autorização

quanto o Servidor de Recursos.

- Abra o(s) projeto(s) Spring Boot no seu IDE favorito e execute
- Testes no Postman:
  - PARTE 1: não logado
    - Requisição GET /products (deve retornar produtos)
    - Requisição GET /products/1 (deve retornar 401)
    - Requisição POST /products {“name”:“Tablet”} (deve retornar 401)
  - PARTE 2: logado como Alex
    - Defina alex@gmail.com como nome de usuário no ambiente do Postman
    - Requisição de login (deve retornar 200 Ok com token JWT. Esse token será salvo na variável de ambiente ‘token’)
    - Requisição GET /products (deve retornar produtos)
    - Requisição GET /products/1 (deve retornar produto)
    - Requisição POST /products {“name”:“Tablet”} (deve retornar 403)
  - PARTE 3: logado como Maria
    - Defina maria@gmail.com como nome de usuário no ambiente do Postman
    - Requisição de login (deve retornar 200 Ok com token JWT. Esse token será salvo na variável de ambiente ‘token’)
    - Requisição GET /products (deve retornar produtos)
    - Requisição GET /products/1 (deve retornar produto)
    - Requisição POST /products {“name”:“Tablet”} (deve inserir produto)

5.8 Modelo de Dados User-Role

O primeiro passo para implementar nossa segurança é implementar o modelo de domínio de usuários e seus perfis de usuários, pois o acesso será controlado por perfil de usuários.

Modelo de dados User-Role.

A seguir está a implementação das classes de domínio. Omitimos os getters/setters e hashCode/equals para simplificar, considerando que neste momento o leitor está acostumado a criar esses métodos. Criamos também na classe `User` os métodos auxiliares `addRole` e `hasRole` para facilitar a adição de um perfil ao usuário, e a verificação se um usuário possui algum perfil, respectivamente.

```
@Entity
@Table(name = "tb_role")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String authority;

    public Role() {
    }

    public Role(Long id, String authority) {
        this.id = id;
        this.authority = authority;
    }

    ...
```

```
@Entity
@Table(name = "tb_user")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @Column(unique = true)
    private String email;
    private String password;

    @ManyToMany
    @JoinTable(name = "tb_user_role",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<>();

    public User() {
    }

    public User(Long id, String name, String email, String phone, LocalDate
   birthDate, String password) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.password = password;
    }

    public void addRole(Role role) {
        roles.add(role);
    }

    public boolean hasRole(String roleName) {
        for (Role role : roles) {
```

```
            if (role.getAuthority().equals(roleName)) {
                return true;
            }
        }
        return false;
    }

    ...
}
```

Seed da base de dados

Neste momento, vamos também incluir no *seed* da basa de dados os perfis de usuário `ROLE_OPERATOR` e `ROLE_ADMIN`para representar os perfis de usuários operadores comuns e administradores, respectivamente:

```
INSERT INTO tb_role (authority) VALUES ('ROLE_OPERATOR');
INSERT INTO tb_role (authority) VALUES ('ROLE_ADMIN');

INSERT INTO tb_user_role (user_id, role_id) VALUES (1, 1);
INSERT INTO tb_user_role (user_id, role_id) VALUES (2, 1);
INSERT INTO tb_user_role (user_id, role_id) VALUES (2, 2);
```

O prefixo `ROLE_` é padrão do Spring Security, e devemos utilizá-lo no nome de nossos perfis de usuário.

Testando o projeto

Neste momento, é recomendado executar o projeto e conferir no H2 Console se as tabelas `tb_role` e `tb_user_role` foram criadas corretamente com os dados do *seed* inseridos.

5.9 Adicionando Spring Security ao projeto

Agora precisamos adicionar as dependências do Spring Security ao projeto. Inclua as dependências a seguir no `pom.xml` do projeto Spring Boot:

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

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

Depois de adicionar as dependências, se o projeto for executado, todos *endpoints* estarão bloquados por padrão. Este é o comportamento padrão do Spring Security.

Entretanto, para que possamos prosseguir com o desenvolvimento do projeto fazendo testes básicos nos *endpoints*, vamos provisoriamente fazer a liberação de todos *endpoints*, criando uma classe de configuração `SecurityConfig` no projeto, conforme código:

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable());
        http.authorizeHttpRequests(auth -> auth.anyRequest().permitAll());
        return http.build();
    }

}
```

5.10 BCrypt password encoder

Agora vamos acrescentar um componente do tipo `PasswordEncoder` em nosso sistema, pois este componente será utilizado pelo Spring Security, e ele será responsável por criptografar e validar as senhas dos usuários.

Para adicionar o componente no sistema, vamos provisoriamente incluir a definição deste componente na classe `SecurityConfig`, por meio do método `passwordEncoder` anotado com a *annotation* `@Bean`. Repare que o tipo de retorno do método é `PasswordEncoder`, mas a instanciação do objeto é específica do tipo `BCryptPasswordEncoder`, pois vamos usar o padrão de criptografia `BCrypt`.

```
@Configuration
public class SecurityConfig {

    @Bean
    PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable());
        http.authorizeHttpRequests(auth -> auth.anyRequest().permitAll());
        return http.build();
    }

}
```

Testando o BCrypt

É possível testar o funcionamento do algoritmo BCrypt fazendo um pequeno teste na classe principal do projeto. Vamos fazer a classe principal implementar a interface `CommandLineRunner`, de modo que, no método `run`, podemos colocar algum código para executar quando a aplicação Spring iniciar. Por exemplo, o código abaixo faz a aplicação Spring imprimir na tela o código BCrypt equivalente ao texto `123456`:

```
@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

    @Autowired
    private PasswordEncoder encoder;

    public static void main(String[] args) {
        SpringApplication.run(DscommerceApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println(encoder.encode("123456"));
    }
}
```

É importante ressaltar que um mesmo texto pode ter mais de um código BCrypt válido. No caso do texto `123456`, um código BCrypt válido é `$2a$10$BZEayVp6X1Ry93e44/Rnze0hpK5J3ThbAdUm2OzH.GSWjA4zmtGHW`. Execute a aplicação Spring, e observe no console da aplicação que um código BCrypt similar a este será impresso no log do console.

Salvando os códigos BCrypt no seed da base de dados

É importante ressaltar que nunca se deve salvar a senha diretamente no banco de dados. Ao invés disso, salva-se no banco de dados algum *hash* irreversível da senha, como é o caso do código BCrypt.

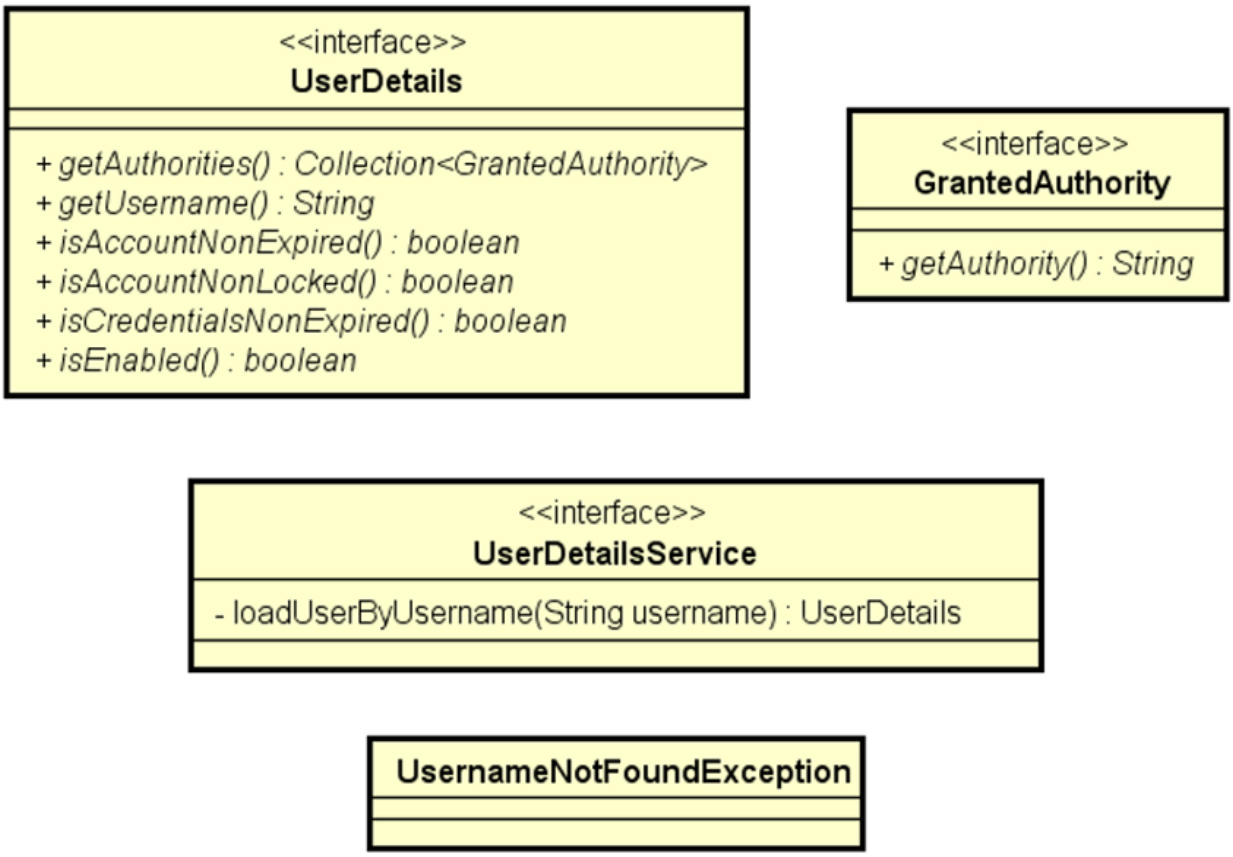
Assim, as senhas salvas no banco de dados devem ser na verdade seus código BCrypt equivalentes. Como as senhas padrão que estamos usando aqui para nossos usuários é `123456`, vamos usar o código BCrypt mencionado anteriormente no valor salvo em nosso *seed*. Deve ficar como mostrado a seguir:

```
INSERT INTO tb_user (name, email, password) VALUES ('Alex', 'alex@gmail.com', '
    $2a$10$BZEayVp6X1Ry93e44/Rnze0hpK5J3ThbAdUm2OzH.GSWjA4zmtGHW');
INSERT INTO tb_user (name, email, password) VALUES ('Maria', 'maria@gmail.com',
    '$2a$10$BZEayVp6X1Ry93e44/Rnze0hpK5J3ThbAdUm2OzH.GSWjA4zmtGHW');
```

5.11 Implementando o checklist do Spring Security PARTE 1

Nesta seção vamos implementar o checklist do Spring Security, ou seja, precisamos implementar os tipos requeridos pelo Spring Security, para que nosso projeto possa executar apropriadamente seu fluxo de segurança. Os tipos básicos são:

- GrantedAuthority: tipo que representa cada perfil de usuário.
- UserDetails: tipo que representa o usuário gerenciado pelo Spring Security.
- UserDetailsService: tipo responsável pela operação de recuperar um usuário `UserDetails` do banco de dados.
- UsernameNotFoundException: exceção lançada caso o usuário não seja encontrado.



Tipos requeridos pelo Spring Security

Nesta seção vamos comerçar implementando `GrantedAuthority` e `UserDetails`, que são interfaces a serem implementadas. Para isto, vamos fazer que estas duas interfaces sejam implementadas pelas nossas classes `Role` e `User`, respectivamente.

Classe Role

```
package com.devsuperior.demo.entities;

import java.util.Objects;
```

```java
import org.springframework.security.core.GrantedAuthority;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Table(name = "tb_role")
public class Role implements GrantedAuthority {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String authority;

    public Role() {
    }

    public Role(Long id, String authority) {
        this.id = id;
        this.authority = authority;
    }

    public Long getId() {
        return id;
    }

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

    @Override
    public String getAuthority() {
        return authority;
    }

    public void setAuthority(String authority) {
        this.authority = authority;
    }

    @Override
    public int hashCode() {
        return Objects.hash(authority);
    }

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

```
    }
}
```

Classe User

```
package com.devsuperior.demo.entities;

import java.time.LocalDate;
import java.util.Collection;
import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Table(name = "tb_user")
public class User implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @Column(unique = true)
    private String email;
    private String password;

    @ManyToMany
    @JoinTable(name = "tb_user_role",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<>();

    public User() {
    }

    public User(Long id, String name, String email, String phone, LocalDate
   birthDate, String password) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.password = password;
    }
```

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

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void addRole(Role role) {
        roles.add(role);
    }

    public boolean hasRole(String roleName) {
        for (Role role : roles) {
            if (role.getAuthority().equals(roleName)) {
                return true;
            }
        }
        return false;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        User user = (User) o;

        return Objects.equals(id, user.id);
    }

    @Override
    public int hashCode() {
```

```
        return id != null ? id.hashCode() : 0;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return roles;
    }

    @Override
    public String getUsername() {
        return email;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

5.12 Implementando o checklist do Spring Security PARTE 2

Prosseguindo com a implementação dos tipos requeridos do Spring Security, vamos implementar agora a interface `UserDetailsService`, que é responsável por buscar um usuário do banco de dados, dado seu nome de usuário, que em nosso caso é o email do usuário.

Liberando acesso ao H2 Console

Antes de implementar a interface `UserDetailsService`, vamos primeiro acrescentar um componente de configuração no sistema para liberar o acesso ao H2 Console no navegador, pois vamos querer verificar se os dados no banco de dados H2 durante nossos testes.

O componente para liberar acesso ao H2 Console será definido pelo tipo `SecurityFilterChain`, o qual pode ser adicionado à nossa classe `SecurityConfig` conforme código a seguir.

```
```java
@Configuration
```
```

```
public class SecurityConfig {

    @Bean
    @Profile("test")
    @Order(1)
    SecurityFilterChain h2SecurityFilterChain(HttpSecurity http) throws
   Exception {

        http.securityMatcher(PathRequest.toH2Console()).csrf(csrf -> csrf.
   disable())
                .headers(headers -> headers.frameOptions(frameOptions ->
   frameOptions.disable()));
        return http.build();
    }

    ...
}
```cd ..

### Consulta do usuário no banco de dados

A operação de busca do usuário tem o intuito de trazer os dados do usuário
   juntamente com os dados de seus perfis de usuário, ou seja, é uma busca que
   precisa recuperar dados relacionados no banco de dados. Para implementar esta
    operação de forma eficiente, optamos por implementar a consulta usando a
   linguagem SQL. Para isto, vamos primeiro definir uma *projection* conforme
   código a seguir:

```java
package com.devsuperior.demo.projections;

public interface UserDetailsProjection {

    String getUsername();
    String getPassword();
    Long getRoleId();
    String getAuthority();
}
```

Em seguida, vamos definir a consulta SQL dentro do nosso `UserRepository`, conforme código a seguir:

```
package com.devsuperior.demo.repositories;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import com.devsuperior.demo.entities.User;
import com.devsuperior.demo.projections.UserDetailsProjection;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
```

```
    @Query(nativeQuery = true, value = """
                SELECT tb_user.email AS username, tb_user.password, tb_role.id
    AS roleId, tb_role.authority
                FROM tb_user
                INNER JOIN tb_user_role ON tb_user.id = tb_user_role.user_id
                INNER JOIN tb_role ON tb_role.id = tb_user_role.role_id
                WHERE tb_user.email = :email
            """)
    List<UserDetailsProjection> searchUserAndRolesByEmail(String email);
}
```

Implementação de UserService

Agora, vamos criar a classe `UserService` dentro do pacote `services`, e vamos implementar a interface `UserDetailsService` nesta classe, conforme código a seguir.

```
package com.devsuperior.demo.services;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.devsuperior.demo.entities.Role;
import com.devsuperior.demo.entities.User;
import com.devsuperior.demo.projections.UserDetailsProjection;
import com.devsuperior.demo.repositories.UserRepository;

@Service
public class UserService implements UserDetailsService {

    @Autowired
    private UserRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
    UsernameNotFoundException {

        List<UserDetailsProjection> result = repository.
    searchUserAndRolesByEmail(username);
        if (result.size() == 0) {
            throw new UsernameNotFoundException("Email not found");
        }

        User user = new User();
        user.setEmail(result.get(0).getUsername());
        user.setPassword(result.get(0).getPassword());
        for (UserDetailsProjection projection : result) {
            user.addRole(new Role(projection.getRoleId(), projection.
    getAuthority()));
```

```
        }

        return user;
    }
}
```

5.13 Checklist OAuth2 JWT password grant

Nesta seção vamos dar continuidade à nossa implementação da segurança no projeto Spring Boot, implementando o Authorization Server e o Resource Server.

Atenção: deste momento em diante, a classe `SecurityConfig` não será mais necessária, e pode ser excluída do projeto. No lugar dela, teremos agora duas classes de configuração: `AuthorizationServerConfig` e `ResourceServerConfig`.

Variáveis de ambiente

Vamos começar incluindo alguns valores de configuração no arquivo `application.properties` do projeto:

```
security.client-id=${CLIENT_ID:myclientid}
security.client-secret=${CLIENT_SECRET:myclientsecret}

security.jwt.duration=${JWT_DURATION:86400}

cors.origins=${CORS_ORIGINS:http://localhost:3000,http://localhost:5173}
```

Estas configurações especificam as seguintes variáveis de ambiente:

- `CLIENT_ID` e `CLIENT_SECRET`: são as credenciais da aplicação. Os valores padrão para testes são `myclientid` e `myclientsecret` respectivamente.
- `JWT_DURATION`: duração em segundos do nosso token JWT. Depois desse tempo, o token expira. Definimos o valor padrão de 86400, correspondente a 24 horas.
- `CORS_ORIGINS`: são as origens da web autorizadas a acessar a aplicação via navegador, separadas por vírgula.

Dependências Maven

Agora vamos definir novas dependências Maven para implementar o Authorization Server e o Resource Server em nossa aplicação. Estas dependências devem ser adicionadas ao arquivo `pom.xml` do projeto:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-authorization-server</artifactId>
</dependency>

<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

Implementação customizada do fluxo password grant

Conforme dito anteriormente, vamos implementar aqui o fluxo "password grant" do OAuth2. Como este fluxo não vem implementado por padrão nas dependências Maven que adicionamos ao projeto, vamos adicionar manualmente algumas classes que prepararmos, as quais possuem uma implementação customizada desse fluxo. Para isto, siga o seguinte passo a passo:

1. Acesse o projeto referência da versão desejada do Spring Boot no nosso repositório Github. Por exemplo, se a versão desejada do Spring Boot for a `3.4.3`, você deve acessar a subpasta correspondente no repositório:

https://github.com/devsuperior/spring-boot-oauth2-jwt-demo/tree/main/password-grant/spring-boot-3-4-3

2. Dentro do projeto, acesse a subpasta `/src/main/java/com/devsuperior/demo/config/customgrant/`, onde estarão contidas as quatro classes que vamos usar:

- CustomPasswordAuthenticationConverter.java
- CustomPasswordAuthenticationProvider.java
- CustomPasswordAuthenticationToken.java
- CustomUserAuthorities.java

3. Dentro do seu projeto Spring Boot, crie um subpacote `config.customgrant`, de forma similar à que está no projeto referência, e copie as quatro classes para esse pacote.

Authorization Server

Atenção: caso não tenha excluído ainda a classe `SecurityConfig` do projeto, faça isso neste momento.

Agora vamos incluir no projeto a implementação do Authorization Server, que será a classe `AuthorizationServerConfig`, dentro do pacote `config`.

Authorization Server é a parte do sistema responsável por processar a autenticação do usuário, ou seja, reconhecer que o usuário é válido, bem como atestar quais são seus perfis de usuário. A requisição de login é processada pelo Authorization Server, bem como verificar se as informações de um usuário logado são válidas.

Por favor, acesse novamente o projeto referência da versão desejada do Spring Boot no nosso repositório Github, e copie a classe `AuthorizationServerConfig.java` do subpacote `config` para seu projeto. Salve a classe em um subpacote `config` no seu projeto, de forma similar ao projeto referência.

Resource Server

Agora vamos incluir no projeto a implementação do Resource Server, que será a classe `ResourceServerConfig`, dentro do pacote `config`.

Resource Server, como o próprio nome sugere, é a parte do sistema responsável por disponibilizar os recursos do sistema. O Resource Server tem a responsabilidade de tratar requisições recebidas pelo sistema e, caso seja um recurso protegido, deverá decidir se o recurso deve ou não ser disponibilizado ao usuário, conforme as políticas de controle de acesso aplicadas àquele usuário.

Por favor, acesse novamente o projeto referência da versão desejada do Spring Boot no nosso repositório Github, e copie a classe `ResourceServerConfig.java` do subpacote `config` para seu projeto. Salve a classe em um subpacote `config` no seu projeto, de forma similar ao projeto referência.

5.14 Requisição de Login

Nesta seção, vamos detalhar como realizar uma requisição de login utilizando o fluxo "password grant" do OAuth2, utilizando a ferramenta Postman. Antes de começar, certifique-se de que o projeto Spring Boot esteja em execução e que o Postman esteja configurado com a *collection* e o *environment* apropriados, conforme detalhado na seção "Preparando projeto para segurança".

Configuração no Postman

1. **Ambiente do Postman:** Verifique se as variáveis de ambiente no Postman estão corretamente configuradas. As variáveis a serem utilizadas são:
   - `host`: http://localhost:8080
   - `client-id`: myclientid
   - `client-secret`: myclientsecret
   - `username`: alex@gmail.com
   - `password`: 123456
2. **Configuração da Requisição:**
   - **URL da Requisição:** Configure o Postman para fazer uma requisição POST para a URL `{{host}}/oauth2/token`.
   - **Body da Requisição:** No corpo da requisição, configure para enviar os dados como `x-www-form-urlencoded`. Adicione os seguintes campos:
     - `username`: `{{username}}`
     - `password`: `{{password}}`
     - `grant_type`: `password`
   - **Header de Autorização:**
     - Utilize o método Basic Auth no Postman, onde você deve inserir `{{client-id}}` como o username e `{{client-secret}}` como o password. O Postman automaticamente irá gerar o header de autorização adequado.

Testando a Requisição

1. **Envio da Requisição:**
   - Após configurar a requisição, clique em "Send" no Postman para enviar a requisição de login ao servidor.
   - Observe a resposta retornada pelo servidor. Se as credenciais estiverem corretas, você receberá um token JWT no corpo da resposta, o qual será utilizado para acessar recursos protegidos na aplicação.

2. **Análise da Resposta:**
    - O token JWT deve estar presente no campo `access_token` da resposta JSON. Este token será usado em requisições subsequentes para autenticação e autorização.
    - Em caso de credenciais incorretas, o servidor responderá com um status HTTP 401 (Unauthorized), indicando que as credenciais fornecidas não são válidas.

5.15 Deixando o Postman top

Vamos nesta seção fazer algumas coisas para deixar nosso Postman mais automatizado e melhor para trabalhar.

Repare que nosso *environment* do Postman possui uma variável de ambiente chamada `token`. Esta variável é usada para armazenar o token de acesso gerado na requisição de login, de modo que outras requisições a recursos protegidos possam utilizar esse token.

Então, para automatizar o processo de pegar o valor do token retornado na resposta da requisição de login, e salvar esse valor na variável `token` do *environment* do Postman, vamos acrescentar o script abaixo na aba "Scripts" da nossa requisição de login:

```
if (responseCode.code >= 200 && responseCode.code < 300) {
   var json = JSON.parse(responseBody);
   postman.setEnvironmentVariable('token', json.access_token);
}
```

Esse script verifica se a requisição ocorreu com sucesso, ou seja, o código de resposta tem que ser 2xx. Em caso afirmativo, o script acessa o corpo da resposta e salva o valor do campo `access_token` na variável `token` do *environment* do Postman.

Com esta automação, basta executar a requisição de login, que o token já estará salvo em nosso *environment*, pronto para ser usado nas outras requisições.

5.16 Analisando o Authorization server

O código fonte da classe `AuthorizationServerConfig` é composto por várias partes, as quais são listadas a seguir.

Obtenção dos valores das variáveis de ambiente

```
@Value("${security.client-id}")
private String clientId;

@Value("${security.client-secret}")
private String clientSecret;

@Value("${security.jwt.duration}")
private Integer jwtDurationSeconds;
```

Injeção de dependência do componente UserDetailsService

```
@Autowired
private UserDetailsService userDetailsService;
```

Configuração do filtro do Autorization Server e do token

```
    @Bean
    @Order(2)
    SecurityFilterChain asSecurityFilterChain(HttpSecurity httpSecurity) throws
   Exception {

        HttpSecurity http = httpSecurity.securityMatcher("/**");

        http.with(OAuth2AuthorizationServerConfigurer.authorizationServer(),
   Customizer.withDefaults());

        // @formatter:off
        http.getConfigurer(OAuth2AuthorizationServerConfigurer.class)
            .tokenEndpoint(tokenEndpoint -> tokenEndpoint
                .accessTokenRequestConverter(new
   CustomPasswordAuthenticationConverter())
                .authenticationProvider(new CustomPasswordAuthenticationProvider
   (authorizationService(), tokenGenerator(), userDetailsService,
   passwordEncoder())));

        http.oauth2ResourceServer(oauth2ResourceServer -> oauth2ResourceServer.
   jwt(Customizer.withDefaults()));
        // @formatter:on

        return http.build();
    }
```

Componentes de configuração do Authorization Server

```
    @Bean
    OAuth2AuthorizationService authorizationService() {
        return new InMemoryOAuth2AuthorizationService();
    }

    @Bean
    OAuth2AuthorizationConsentService oAuth2AuthorizationConsentService() {
        return new InMemoryOAuth2AuthorizationConsentService();
    }

    @Bean
    AuthorizationServerSettings authorizationServerSettings() {
        return AuthorizationServerSettings.builder().build();
    }
```

Componente PasswordEncoder

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Componentes da aplicação cliente

```
@Bean
RegisteredClientRepository registeredClientRepository() {
    // @formatter:off
    RegisteredClient registeredClient = RegisteredClient
        .withId(UUID.randomUUID().toString())
        .clientId(clientId)
        .clientSecret(passwordEncoder().encode(clientSecret))
        .scope("read")
        .scope("write")
        .authorizationGrantType(new AuthorizationGrantType("password"))
        .tokenSettings(tokenSettings())
        .clientSettings(clientSettings())
        .build();
    // @formatter:on

    return new InMemoryRegisteredClientRepository(registeredClient);
}

@Bean
ClientSettings clientSettings() {
    return ClientSettings.builder().build();
}
```

Componentes de configuração de token

```
@Bean
TokenSettings tokenSettings() {
    // @formatter:off
    return TokenSettings.builder()
        .accessTokenFormat(OAuth2TokenFormat.SELF_CONTAINED)
        .accessTokenTimeToLive(Duration.ofSeconds(jwtDurationSeconds))
        .build();
    // @formatter:on
}

@Bean
OAuth2TokenGenerator<? extends OAuth2Token> tokenGenerator() {
    NimbusJwtEncoder jwtEncoder = new NimbusJwtEncoder(jwkSource());
    JwtGenerator jwtGenerator = new JwtGenerator(jwtEncoder);
    jwtGenerator.setJwtCustomizer(tokenCustomizer());
    OAuth2AccessTokenGenerator accessTokenGenerator = new
   OAuth2AccessTokenGenerator();
    return new DelegatingOAuth2TokenGenerator(jwtGenerator, accessTokenGenerator
   );
}
```

```
@Bean
OAuth2TokenCustomizer<JwtEncodingContext> tokenCustomizer() {
    return context -> {
        OAuth2ClientAuthenticationToken principal = context.getPrincipal();
        CustomUserAuthorities user = (CustomUserAuthorities) principal.
   getDetails();
        List<String> authorities = user.getAuthorities().stream().map(x -> x.
   getAuthority()).toList();
        if (context.getTokenType().getValue().equals("access_token")) {
            // @formatter:off
            context.getClaims()
                .claim("authorities", authorities)
                .claim("username", user.getUsername());
            // @formatter:on
        }
    };
}

@Bean
JwtDecoder jwtDecoder(JWKSource<SecurityContext> jwkSource) {
    return OAuth2AuthorizationServerConfiguration.jwtDecoder(jwkSource);
}

@Bean
JWKSource<SecurityContext> jwkSource() {
    RSAKey rsaKey = generateRsa();
    JWKSet jwkSet = new JWKSet(rsaKey);
    return (jwkSelector, securityContext) -> jwkSelector.select(jwkSet);
}

private static RSAKey generateRsa() {
    KeyPair keyPair = generateRsaKey();
    RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
    RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
    return new RSAKey.Builder(publicKey).privateKey(privateKey).keyID(UUID.
   randomUUID().toString()).build();
}

private static KeyPair generateRsaKey() {
    KeyPair keyPair;
    try {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048);
        keyPair = keyPairGenerator.generateKeyPair();
    } catch (Exception ex) {
        throw new IllegalStateException(ex);
    }
    return keyPair;
}
```

5.17 Analisando o Resource server

O código fonte da classe `ResourceServerConfig` hé composto por várias partes, as quais são listadas a seguir.

Obtenção dos valores das variáveis de ambiente

```
@Value("${cors.origins}")
private String corsOrigins;
```

Configuração do filtro para liberar H2 Console no perfil test

```
@Bean
@Profile("test")
@Order(1)
public SecurityFilterChain h2SecurityFilterChain(HttpSecurity http) throws
    Exception {

     http.securityMatcher(PathRequest.toH2Console()).csrf(csrf -> csrf.disable())
             .headers(headers -> headers.frameOptions(frameOptions ->
    frameOptions.disable()));
     return http.build();
}
```

Configuração do filtro para CSRF, requisições, token e CORS

```
@Bean
@Order(3)
public SecurityFilterChain rsSecurityFilterChain(HttpSecurity httpSecurity)
    throws Exception {
     HttpSecurity http = httpSecurity.securityMatcher("/**");
     http.csrf(csrf -> csrf.disable());
     http.authorizeHttpRequests(authorize -> authorize.anyRequest().permitAll());
     http.oauth2ResourceServer(oauth2ResourceServer -> oauth2ResourceServer.jwt(
    Customizer.withDefaults()));
     http.cors(cors -> cors.configurationSource(corsConfigurationSource()));
     return http.build();
}
```

Componente de verificação de token

```
@Bean
JwtAuthenticationConverter jwtAuthenticationConverter() {
     JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new
    JwtGrantedAuthoritiesConverter();
     grantedAuthoritiesConverter.setAuthoritiesClaimName("authorities");
     grantedAuthoritiesConverter.setAuthorityPrefix("");

     JwtAuthenticationConverter jwtAuthenticationConverter = new
    JwtAuthenticationConverter();
     jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(
    grantedAuthoritiesConverter);
     return jwtAuthenticationConverter;
}
```

Componentes de configuração de CORS

```
@Bean
CorsConfigurationSource corsConfigurationSource() {

    String[] origins = corsOrigins.split(",");

    CorsConfiguration corsConfig = new CorsConfiguration();
    corsConfig.setAllowedOriginPatterns(Arrays.asList(origins));
    corsConfig.setAllowedMethods(Arrays.asList("POST", "GET", "PUT", "DELETE", "
   PATCH"));
    corsConfig.setAllowCredentials(true);
    corsConfig.setAllowedHeaders(Arrays.asList("Authorization", "Content-Type"))
   ;

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource
   ();
    source.registerCorsConfiguration("/**", corsConfig);
    return source;
}

@Bean
FilterRegistrationBean<CorsFilter> filterRegistrationBeanCorsFilter() {
    FilterRegistrationBean<CorsFilter> bean = new FilterRegistrationBean<>(
            new CorsFilter(corsConfigurationSource()));
    bean.setOrder(Ordered.HIGHEST_PRECEDENCE);
    return bean;
}
```

5.18 Controle de acesso por perfil e rota

Nesta seção vamos tratar uma questão fundamental sobre o controle de acesso aos recursos. Basicamente, precisamos responder à questão: quem pode acessar o quê?

Primeiramente, vamos recordar quem são nossos usuários do projeto referência, quais são os perfis de usuário de cada um, e quais são as regras de negócio para controle de acesso:

Usuários e perfis

Conforme pode-se conferir no *seed* da nossa base de dados, os usuários e perfis de acesso do nosso projeto referência são:

- alex@gmail.com, perfis: [ROLE_OPERATOR]
- maria@gmail.com, perfis: [ROLE_OPERATOR, ROLE_ADMIN]

Regras de controle de acesso

Cada um dos endpoints do projeto tem as seguintes regras de acesso:

- `GET /products`
  - Acesso público (não precisar estar logado)
- `GET /products/{id}`
  - Acesso autorizado para perfis: ROLE_OPERATOR, ROLE_ADMIN
- `POST /products`

– Acesso autorizado somente para perfil: ROLE_ADMIN

Prática adotada em nossa implementação

Se você reparar no código do Resource Server, nós definimos que todos endpoints por padrão são liberados, por meio do método `authorizeHttpRequests`, que depois recebe como argumento a função `authorize -> authorize.anyRequest().permitAll()`:

```
@Bean
@Order(3)
public SecurityFilterChain rsSecurityFilterChain(HttpSecurity httpSecurity)
    throws Exception {
    HttpSecurity http = httpSecurity.securityMatcher("/**");

    ...

    http.authorizeHttpRequests(authorize -> authorize.anyRequest().permitAll());

    ...
}
```

Esta definição dentro do filtro do ResourceServer tem efeito global no sistema. Sendo assim, para simplificar nossa lógica de controle de acesso, decidimos definir globalmente que todos endpoints estão liberados e, aqueles endpoints que tiverem que ser protegidos, serão assim definidos individualmente, conforme vamos mostrar a seguir.

Definindo controle de acesso em cada endpoint

Para especificar o controle de acesso em cada endpoint, vamos utilizar a annotation `@PreAuthorize`, por meio da qual é possível configurar o controle de acesso ao endpoint com base nos perfis de usuário. Vamos então fazer a configuração de cada endpoint a seguir:

Endpoint GET /products

Nossa regra de negócio define que este endpoint é de acesso público. Assim, como nossa configuração global já define os endpoints como liberados por padrão, este endpoint não precisa de configurações adicionais:

```
@GetMapping
public ResponseEntity<List<ProductDTO>> findAll() {
    List<ProductDTO> list = productService.findAll();
    return ResponseEntity.ok(list);
}
```

Endpoint GET /products/{id}

Nossa regra de negócio define que este endpoint pode ser acessado por ambos perfis ROLE_OPERATOR e ROLE_ADMIN. Assim, vamos sobrescrever a configuração global

acrescentando a annotation `@PreAuthorize`, especificando estes perfis como autorizados a acessar o endpoint:

```
@PreAuthorize("hasAnyRole('ROLE_ADMIN', 'ROLE_OPERATOR')")
@GetMapping(value = "/{id}")
public ResponseEntity<ProductDTO> findById(@PathVariable Long id) {
    ProductDTO dto = productService.findById(id);
    return ResponseEntity.ok(dto);
}
```

Endpoint POST /products

Nossa regra de negócio define que este endpoint pode ser acessado somente pelo perfil ROLE_ADMIN. Assim, vamos sobrescrever a configuração global acrescentando a annotation `@PreAuthorize`, especificando este perfil como autorizado a acessar o endpoint:

```
@PreAuthorize("hasRole('ROLE_ADMIN')")
@PostMapping
public ResponseEntity<ProductDTO> insert(@RequestBody ProductDTO dto) {
    dto = productService.insert(dto);
    URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
            .buildAndExpand(dto.getId()).toUri();
    return ResponseEntity.created(uri).body(dto);
}
```

Testando a aplicação

Neste momento você já pode executar a aplicação e testar os endpoints. A seguir apresentamos as orientações de como fazer.

Preparando o Postman

Primeiro é preciso que cada requisição no Postman esteja devidamente configurada para passar o token de acesso no cabeçado `Authorization` da requisição, se for uma requisição a um recurso protegido. Para cada requisição da sua *collection* Postman, acesse a aba `Authorization` para configurar o token, como se segue:

- Requisição `GET /products`: como este é um recurso público, ou seja, que não precisa de usuário logado, na aba `Authorization`, no campo `Type` selecione o valor "No Auth".
- Requisição `GET /products{id}`: como este é um recurso protegido, na aba `Authorization`, no campo `Type` selecione o valor "Bearer Token", e no campo `Token` digite o valor `{{token }}` para que o valor do token seja obtido a partir da variável `token` do *environment* do Postman.
- Requisição `POST /products`: como este é um recurso protegido, na aba `Authorization`, no campo `Type` selecione o valor "Bearer Token", e no campo `Token` digite o valor `{{token }}` para que o valor do token seja obtido a partir da variável `token` do *environment* do Postman.

Pronto. Agora é possível testar cada um dos cenários de teste, conforme especificamos anteriormente:

- Testes no Postman:
  - PARTE 1: não logado
    - Requisição GET /products (deve retornar produtos)
    - Requisição GET /products/1 (deve retornar 401)
    - Requisição POST /products {“name”:“Tablet”} (deve retornar 401)
  - PARTE 2: logado como Alex
    - Defina alex@gmail.com como nome de usuário no ambiente do Postman
    - Requisição de login (deve retornar 200 Ok com token JWT. Esse token será salvo na variável de ambiente ‘token’)
    - Requisição GET /products (deve retornar produtos)
    - Requisição GET /products/1 (deve retornar produto)
    - Requisição POST /products {“name”:“Tablet”} (deve retornar 403)
  - PARTE 3: logado como Maria
    - Defina maria@gmail.com como nome de usuário no ambiente do Postman
    - Requisição de login (deve retornar 200 Ok com token JWT. Esse token será salvo na variável de ambiente ‘token’)
    - Requisição GET /products (deve retornar produtos)
    - Requisição GET /products/1 (deve retornar produto)
    - Requisição POST /products {“name”:“Tablet”} (deve inserir produto)

5.19 Adicionando segurança ao DSCommerce

Agora que já aprendemos todo processo de incluir segurança em um projeto Spring, vamos aplicar este conhecimento em nosso projeto DSCommerce.

Neste momento, preciso que você esteja com seu projeto DSCommerce preparado, no estado em que ele se encontra no final do capítulo passado.

Agora, vamos começar por implementar o modelo de domínio de usuários e seus perfis de usuários, conforme já aprendemos.

Como nosso projeto DSCommerce já possui a classe `User`, vamos focar em implementar o restante. A seguir apresentamos o que precisa ser implementado:

Classe Role

Vamos implementar a entidade Role para representar um perfil de usuário.

```
@Entity
@Table(name = "tb_role")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String authority;

    public Role() {
```

```
    }

    public Role(Long id, String authority) {
        this.id = id;
        this.authority = authority;
    }

    ...
}
```

Classe User

Vamos acrescentar o relacionamento entre usuário e role na classe `User`, e vamos acrescentar os métodos `addRole`, `hasRole` e `getRoles`.

```
@Entity
@Table(name = "tb_user")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @Column(unique = true)
    private String email;
    private String phone;
    private LocalDate birthDate;
    private String password;

    @OneToMany(mappedBy = "client")
    private List<Order> orders = new ArrayList<>();

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "tb_user_role",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<>();

    public User() {
    }

    public User(Long id, String name, String email, String phone, LocalDate
   birthDate, String password) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.phone = phone;
        this.birthDate = birthDate;
        this.password = password;
    }

    public void addRole(Role role) {
        roles.add(role);
```

```
    }

    public boolean hasRole(String roleName) {
        for (Role role : roles) {
            if (role.getAuthority().equals(roleName)) {
                return true;
            }
        }
        return false;
    }

    public Set<Role> getRoles() {
        return roles;
    }

    ...
}
```

Seeding da base de dados

Vamos acrescentar o *seeding* dos perfis de usuário do DSCommerce, que serão `ROLE_CLIENT` (clientes) e `ROLE_ADMIN` (administradores de sistema). Vamos definir que Maria será apenas cliente, e que Alex será cliente e administrador.

Já vamos também mudar o valor padrão da senha de ambos usuários, para o código BCrypt equivalente à senha `123456`, conforme explicamos na seção sobre BCrypt password encoder.

```
INSERT INTO tb_user (name, email, phone, password, birth_date) VALUES ('Maria
    Brown', 'maria@gmail.com', '988888888', '$2a$10$BZEayVp6X1Ry93e44/
    RnzeOhpK5J3ThbAdUm2OzH.GSWjA4zmtGHW', '2001-07-25');
INSERT INTO tb_user (name, email, phone, password, birth_date) VALUES ('Alex
    Green', 'alex@gmail.com', '977777777', '$2a$10$BZEayVp6X1Ry93e44/
    RnzeOhpK5J3ThbAdUm2OzH.GSWjA4zmtGHW', '1987-12-13');

INSERT INTO tb_role (authority) VALUES ('ROLE_CLIENT');
INSERT INTO tb_role (authority) VALUES ('ROLE_ADMIN');

INSERT INTO tb_user_role (user_id, role_id) VALUES (1, 1);
INSERT INTO tb_user_role (user_id, role_id) VALUES (2, 1);
INSERT INTO tb_user_role (user_id, role_id) VALUES (2, 2);
```

Testando o projeto

Neste momento, é recomendado executar o projeto e conferir no H2 Console se as tabelas `tb_role` e `tb_user_role` foram criadas corretamente com os dados do *seed* inseridos.

5.20 Adicionando UserDetails e GrantedAuthority

Agora vamos implementar as interfaces `GrantedAuthority` e `UserDetails` do Spring Security, conforme aprendemos anteriormente.

Dependências Maven

Primeiramente, acrescente todas as dependências de segurança ao projeto:

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

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-authorization-server</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

Classes User e Role atualizadas

A seguir mostramos os trechos de código das classes `Role` e `User` onde implementamos as interfaces `GrantedAuthority` e `UserDetails`, respectivamente.

Classe Role

```
@SuppressWarnings("serial")
@Entity
@Table(name = "tb_role")
public class Role implements GrantedAuthority {

    ...

    @Override
    public String getAuthority() {
        return authority;
    }

    ...
```

Classe User

```
@SuppressWarnings("serial")
@Entity
@Table(name = "tb_user")
public class User implements UserDetails {
```

```
    ...

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return roles;
    }

    @Override
    public String getUsername() {
        return email;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

5.21 Adicionando UserDetailsService

Agora vamos implementar a interface `UserDetailsService` conforme aprendemos anteriormente. A seguir apresentamos os códigos fonte.

Interface UserDetailsProjection

```
package com.devsuperior.dscommerce.projections;

public interface UserDetailsProjection {

    String getUsername();
    String getPassword();
    Long getRoleId();
    String getAuthority();
}
```

Interface UserRepository

```
public interface UserRepository extends JpaRepository<User, Long> {

    @Query(nativeQuery = true, value = """
                SELECT tb_user.email AS username, tb_user.password, tb_role.id
    AS roleId, tb_role.authority
                FROM tb_user
                INNER JOIN tb_user_role ON tb_user.id = tb_user_role.user_id
                INNER JOIN tb_role ON tb_role.id = tb_user_role.role_id
                WHERE tb_user.email = :email
            """)
    List<UserDetailsProjection> searchUserAndRolesByEmail(String email);
}
```

Classe UserService

```
@Service
public class UserService implements UserDetailsService {

    @Autowired
    private UserRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
    UsernameNotFoundException {

        List<UserDetailsProjection> result = repository.
    searchUserAndRolesByEmail(username);
        if (result.size() == 0) {
            throw new UsernameNotFoundException("Email not found");
        }

        User user = new User();
        user.setEmail(result.get(0).getUsername());
        user.setPassword(result.get(0).getPassword());
        for (UserDetailsProjection projection : result) {
            user.addRole(new Role(projection.getRoleId(), projection.
    getAuthority()));
        }

        return user;
    }
}
```

5.22 Adicionando OAuth2, JWT, password grant

Agora vamos fazer todo passo a passo para incluir o Authorization Server e o Resource Server com o fluxo *password grant*, conforme já aprendemos.

Variáveis de ambiente

Confira como deve ficar o arquivo `application.properties` do projeto DSCommerce:

```
spring.profiles.active=test
spring.jpa.open-in-view=false

security.client-id=${CLIENT_ID:myclientid}
security.client-secret=${CLIENT_SECRET:myclientsecret}

security.jwt.duration=${JWT_DURATION:86400}

cors.origins=${CORS_ORIGINS:http://localhost:3000,http://localhost:5173}
```

Implementação customizada do fluxo password grant

Vamos fazer novamente o passo a passo para incluir o código de implementação customizada do fluxo *password grant* no projeto, só que agora para o projeto DSCommerce.

O processo é similar ao que fizemos no projeto referência, com a diferença que será necessário editar o nome do pacote para o pacote correto do DSCommerce, por exemplo: troque o nome do pacote `com.devsuperior.demo.config.customgrant` para `com.devsuperior.dscommerce.config.customgrant`.

1. Acesse o projeto referência da versão desejada do Spring Boot no nosso repositório Github. Por exemplo, se a versão desejada do Spring Boot for a `3.4.3`, você deve acessar a subpasta correspondente no repositório:

https://github.com/devsuperior/spring-boot-oauth2-jwt-demo/tree/main/password-grant/spring-boot-3-4-3

2. Dentro do projeto, acesse a subpasta `/src/main/java/com/devsuperior/demo/config/customgrant/`, onde estarão contidas as quatro classes que vamos usar:

- CustomPasswordAuthenticationConverter.java
- CustomPasswordAuthenticationProvider.java
- CustomPasswordAuthenticationToken.java
- CustomUserAuthorities.java

3. Dentro do seu projeto Spring Boot, crie um subpacote `config.customgrant`, de forma similar à que está no projeto referência, e copie as quatro classes para esse pacote.

Authorization Server

Agora vamos incluir no projeto a implementação do Authorization Server, que será a classe `AuthorizationServerConfig`, dentro do pacote `config`.

Por favor, acesse novamente o projeto referência da versão desejada do Spring Boot no nosso repositório Github, e copie a classe `AuthorizationServerConfig.java` do subpacote `config` para seu projeto. Salve a classe em um subpacote `config` no seu projeto, de forma similar ao projeto referência. Não esqueça de editar nome do pacote da classe para `com.devsuperior.dscommerce.config`.

Resource Server

Agora vamos incluir no projeto a implementação do Resource Server, que será a classe `ResourceServerConfig`, dentro do pacote `config`.

Por favor, acesse novamente o projeto referência da versão desejada do Spring Boot no nosso repositório Github, e copie a classe `ResourceServerConfig.java` do subpacote `config` para seu projeto. Salve a classe em um subpacote `config` no seu projeto, de forma similar ao projeto referência. Não esqueça de editar nome do pacote da classe para `com.devsuperior.dscommerce.config`.

Requisição de login

Na sua `collection` Postman do projeto DSCommerce, prepare uma requisição de login de forma similar à que aprendemos anteriormente no projeto referência. Prepare também o `environment` do Postman conforme nos aprendemos no projeto referência.

Depois, execute o projeto DSCommerce e execute a requisição de login. Neste momento o projeto DSCommerce deve ser capaz de executar a requisição com sucesso e retornar um token.

5.23 Adicionando controle de acesso por perfil e rota

Agora vamos adicionar o controle de acesso aos nossos endpoints de produto do DSCommerce.

Regras de negócio para controle de acesso

Os endpoints `GET /products{id}` e `GET /products` devem ser públicos, ou seja, não se deve exigir usuário logado para acesso ao recurso.

Os endpoints `POST /products`, `PUT /products` e `DELETE /products` devem ser acessados somente por usuários com perfil `ROLE_ADMIN`.

Preparando as requisições no Postman

Conforme aprendemos anteriormente, é preciso passar o token de acesso como cabeçalho `Authorization` nas requisições a recursos protegidos. Assim, edite cada uma das requisições no Postman na aba "Authorization", conforme regras de negócio do sistema:

- Requisições `GET /products{id}` e `GET /products`: campo `Type` com valor "No Auth".
- Requisições `POST /products`, `PUT /products` e `DELETE /products`: campo `Type` com valor "Bearer Token" e campo `Token` com valor `{{token}}`.

Controle de acesso em ProductController

Agora vamos usar a annotation `@PreAuthorize` para definir os controles de acesso na classe `ProductController` conforme as regras de negócio.

Vale ressaltar que, como os endpoints `GET` são públicos, não foi preciso adicionar controle de acesso a estes endpoints.

A seguir mostramos os trechos de código de ProductController onde definirmos os controles de acesso.

```
@RestController
@RequestMapping(value = "/products")
public class ProductController {

    ...

    @PreAuthorize("hasRole('ROLE_ADMIN')")
    @PostMapping
    public ResponseEntity<ProductDTO> insert(@Valid @RequestBody ProductDTO dto)
    {
        dto = service.insert(dto);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(dto.getId()).toUri();
        return ResponseEntity.created(uri).body(dto);
    }

    @PreAuthorize("hasRole('ROLE_ADMIN')")
    @PutMapping(value = "/{id}")
    public ResponseEntity<ProductDTO> update(@PathVariable Long id, @Valid
   @RequestBody ProductDTO dto) {
        dto = service.update(id, dto);
        return ResponseEntity.ok(dto);
    }

    @PreAuthorize("hasRole('ROLE_ADMIN')")
    @DeleteMapping(value = "/{id}")
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        service.delete(id);
        return ResponseEntity.noContent().build();
    }
}
```

Testando os endpoints

Execute o projeto e faça os seguintes testes para conferir os comportamentos desejados:

- Requisições `GET /products{id}` e `GET /products`: devem retornar os produtos, com ou sem usuários logados. Para simular usuários não logados, basta apagar o valor da variável `token` no *environment* do Postman.
- Requisições `POST /products`, `PUT /products` e `DELETE /products`
  - Devem retornar 401 quando forem chamadas sem usuário logado.
  - Devem retornar 403 quando o usuário logado for `maria@gmail.com`, pois este usuário não tem o perfil `ROLE_ADMIN`.
  - Devem executar com sucesso quando o usuário for `alex@gmail.com`, pois este usuário tem o perfil `ROLE_ADMIN`.

5.24 Obtendo o usuário logado

Nesta seção vamos mostrar a implementação de uma operação muito útil, que é a obtenção dos dados do usuário logado.

Método findByEmail no repository

Primeiro, vamos acrescentar um método simples `findByEmail` de busca de usuário por email, utilizando um *query method* do Spring Data JPA:

```
public interface UserRepository extends JpaRepository<User, Long> {

    ...

    Optional<User> findByEmail(String email);
}
```

Método authenticated em UserService

Agora vamos criar um método `authenticated` em UserService. Este método usa alguns recursos do Spring Security para obter o "username", ou seja, o email do usuário logado, e depois usa nosso método `findByEmail` para buscar o usuário do banco de dados na forma de uma entidade, ou seja, um objeto da classe `User`:

```
@Service
public class UserService implements UserDetailsService {

    ...

    protected User authenticated() {
        try {
            Authentication authentication = SecurityContextHolder.getContext().
   getAuthentication();
            Jwt jwtPrincipal = (Jwt) authentication.getPrincipal();
            String username = jwtPrincipal.getClaim("username");
            return repository.findByEmail(username).get();
        }
        catch (Exception e) {
            throw new UsernameNotFoundException("Invalid user");
        }
    }
}
```

Classe UserDTO

Para que possamos criar um endpoint para retornar os dados do usuário logado, e customizar os dados que queremos retornar, vamos criar uma classe `UserDTO` conforme código a seguir.

```java
package com.devsuperior.dscommerce.dto;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

import org.springframework.security.core.GrantedAuthority;

import com.devsuperior.dscommerce.entities.User;

public class UserDTO {

    private Long id;
    private String name;
    private String email;
    private String phone;
    private LocalDate birthDate;
    private List<String> roles = new ArrayList<>();

    public UserDTO(Long id, String name, String email, String phone, LocalDate
   birthDate) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.phone = phone;
        this.birthDate = birthDate;
    }

    public UserDTO(User entity) {
        id = entity.getId();
        name = entity.getName();
        email = entity.getEmail();
        phone = entity.getPhone();
        birthDate = entity.getBirthDate();
        for (GrantedAuthority role : entity.getAuthorities()) {
            roles.add(role.getAuthority());
        }
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    public String getPhone() {
        return phone;
    }

    public LocalDate getBirthDate() {
        return birthDate;
    }
```

```
    public List<String> getRoles() {
        return roles;
    }
}
```

Método getMe em UserService

Agora, de volta à classe `UserService` vamos criar um método `getMe` que será responsável por buscar o usuário logado no sistema e retornar o objeto tipo `UserDTO` a partir deste usuário logado.

```
@Service
public class UserService implements UserDetailsService {

    ...

    @Transactional(readOnly = true)
    public UserDTO getMe() {
        User entity = authenticated();
        return new UserDTO(entity);
    }
}
```

Classe UserController

Agora, para que possamos disponibilizar um *endpoint* para retornar os dados do usuário logado, vamos implementar este *endpoint* em uma nova classe `UserController` dentro do subpacote `controllers`, conforme código a seguir. Repare que estamos usando a annotation `@PreAuthorize` para indicar que este *endpoint* pode ser acessado por qualquer usuário logado, independente do perfil.

```
package com.devsuperior.dscommerce.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.devsuperior.dscommerce.dto.UserDTO;
import com.devsuperior.dscommerce.services.UserService;

@RestController
@RequestMapping(value = "/users")
public class UserController {

    @Autowired
    private UserService service;
```

```
    @PreAuthorize("hasAnyRole('ROLE_ADMIN', 'ROLE_CLIENT')")
    @GetMapping(value = "/me")
    public ResponseEntity<UserDTO> getMe() {
        UserDTO dto = service.getMe();
        return ResponseEntity.ok(dto);
    }
}
```

Testando a requisição

Acrescente uma requisição no Postman `GET /users/me`. Defina o cabeçalho `Authorization` com o tipo "Bearer Token" e o token `{{token}}`. Com o projeto Spring Boot em execução, execute a requisição de login para o usuário `maria@gmail.com`, depois execute a requisição `GET /users/me`. O resultado do corpo da resposta deve ser como mostrado a seguir.

```
{
    "id": 1,
    "name": "Maria Brown",
    "email": "maria@gmail.com",
    "phone": "988888888",
    "birthDate": "2001-07-25",
    "roles": [
        "ROLE_CLIENT"
    ]
}
```

5.25 Consultar catálogo ProductMinDTO

Nesta seção vamos refinar a implementação do endpoint `GET /products`, ou seja, nossa busca paginada de produtos. Isso porque na listagem de produtos não precisamos retornar todos dados do produto, mas somente seu id, nome, preço e imagem.

Classe ProductMinDTO

Vamos criar um DTO para customizar o retorno da nossa busca paginada de produtos.

```
package com.devsuperior.dscommerce.dto;

import com.devsuperior.dscommerce.entities.Product;

public class ProductMinDTO {

    private Long id;
    private String name;
    private Double price;
    private String imgUrl;

    public ProductMinDTO(Long id, String name, Double price, String imgUrl) {
```

```
        this.id = id;
        this.name = name;
        this.price = price;
        this.imgUrl = imgUrl;
    }

    public ProductMinDTO(Product entity) {
        id = entity.getId();
        name = entity.getName();
        price = entity.getPrice();
        imgUrl = entity.getImgUrl();
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public Double getPrice() {
        return price;
    }

    public String getImgUrl() {
        return imgUrl;
    }
}
```

Classe ProductService

Agora vamos atualizar o método `findAll` na classe `ProductService` para retornar uma página de objetos `ProductMinDTO`:

```
@Service
public class ProductService {

    ...

    @Transactional(readOnly = true)
    public Page<ProductMinDTO> findAll(String name, Pageable pageable) {
        Page<Product> result = repository.searchByName(name, pageable);
        return result.map(x -> new ProductMinDTO(x));
    }

    ...
}
```

Classe ProductController

Vamos atualizar também o controlador para retornar uma página de objetos `ProductMinDTO`:

```java
@RestController
@RequestMapping(value = "/products")
public class ProductController {

    ...

    @GetMapping
    public ResponseEntity<Page<ProductMinDTO>> findAll(
            @RequestParam(name = "name", defaultValue = "") String name,
            Pageable pageable) {
        Page<ProductMinDTO> dto = service.findAll(name, pageable);
        return ResponseEntity.ok(dto);
    }

    ...
}
```

ProductDTO com categorias

Agora vamos aprimorar a implementação da inserção e atualização de produtos, pois nosso caso de uso define que também é preciso informar as categorias do produto na hora de inserir ou atualizar um produto.

Classe CategoryDTO

Vamos começar implementando a classe `CategoryDTO`.

```java
package com.devsuperior.dscommerce.dto;

import com.devsuperior.dscommerce.entities.Category;

public class CategoryDTO {

    private Long id;
    private String name;

    public CategoryDTO() {
    }

    public CategoryDTO(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public CategoryDTO(Category entity) {
        id = entity.getId();
        name = entity.getName();
    }

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

```
    public String getName() {
        return name;
    }
}
```

Classe ProductDTO

Agora, vamos atualizar o código da nossa classe `ProductDTO`. Repare que incluímos uma lista `categories` do tipo `List<CategoryDTO>`, incluímos o método `getCategories`, e incluímos no construtor `public ProductDTO(Product entity)` uma lógica para instanciar as categorias do DTO a partir das categorias de uma entidade `entity`.

```
package com.devsuperior.dscommerce.dto;

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

import com.devsuperior.dscommerce.entities.Category;
import com.devsuperior.dscommerce.entities.Product;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Positive;
import jakarta.validation.constraints.Size;

public class ProductDTO {

    private Long id;

    @Size(min = 3, max = 80, message = "Nome precisar ter de 3 a 80 caracteres")
    @NotBlank(message = "Campo requerido")
    private String name;

    @Size(min = 10, message = "Descrição precisa ter no mínimo 10 caracteres")
    @NotBlank(message = "Campo requerido")
    private String description;

    @NotNull(message = "Campo requerido")
    @Positive(message = "O preço deve ser positivo")
    private Double price;

    private String imgUrl;

    @NotEmpty(message = "Deve ter pelo menos uma categoria")
    private List<CategoryDTO> categories = new ArrayList<>();

    public ProductDTO() {
    }

    public ProductDTO(Long id, String name, String description, Double price,
   String imgUrl) {
        this.id = id;
        this.name = name;
```

```
        this.description = description;
        this.price = price;
        this.imgUrl = imgUrl;
    }

    public ProductDTO(Product entity) {
        id = entity.getId();
        name = entity.getName();
        description = entity.getDescription();
        price = entity.getPrice();
        imgUrl = entity.getImgUrl();
        for (Category cat : entity.getCategories()) {
            categories.add(new CategoryDTO(cat));
        }
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public Double getPrice() {
        return price;
    }

    public String getImgUrl() {
        return imgUrl;
    }

    public List<CategoryDTO> getCategories() {
        return categories;
    }
}
```

Classe ProductService

Como agora nosso `ProductDTO` é um objeto que contém não só os dados básicos do produto, mas também uma lista com as categorias do produto, na classe `ProductService` foi necessário atualizar a forma como copiamos os dados do DTO para a entidade que será salva no banco de dados.

```
@Service
public class ProductService {

    ...

    @Transactional
```

```
    public ProductDTO insert(ProductDTO dto) {
        Product entity = new Product();
        copyDtoToEntity(dto, entity);
        entity = repository.save(entity);
        return new ProductDTO(entity);
    }

    @Transactional
    public ProductDTO update(Long id, ProductDTO dto) {
        try {
            Product entity = repository.getReferenceById(id);
            copyDtoToEntity(dto, entity);
            entity = repository.save(entity);
            return new ProductDTO(entity);
        }
        catch (EntityNotFoundException e) {
            throw new ResourceNotFoundException("Recurso não encontrado");
        }
    }

    private void copyDtoToEntity(ProductDTO dto, Product entity) {
        entity.setName(dto.getName());
        entity.setDescription(dto.getDescription());
        entity.setPrice(dto.getPrice());
        entity.setImgUrl(dto.getImgUrl());

        entity.getCategories().clear();
        for (CategoryDTO catDto : dto.getCategories()) {
            Category cat = new Category();
            cat.setId(catDto.getId());
            entity.getCategories().add(cat);
        }
    }
}
```

Testando as requisições de salvar e atualizar

A partir de agora, no corpo das requisições para salvar e alterar produtos, precisamos informar uma lista `categories` com pelo menos uma categoria, conforme no JSON exemplo a seguir. Faça os testes nas requisições `POST /products` e `PUT /products` utilizando esse JSON e verifique se o produto é corretamente inserido e atualizado, respectivamente.

```
{
    "name": "Meu produto",
    "description": "Lorem ipsum, dolor sit amet",
    "imgUrl": "https://site.com/image.jpg",
    "price": 50.0,
    "categories": [
        {
            "id": 2
        },
        {
            "id": 3
        }
```

```
    ]
}
```

5.26 OrderDTO, busca de pedido por id

Nesta seção vamos implementar a busca de um pedido por id.

Representação dos dados

Em primeiro lugar, precisamos entender a estrutura dos dados de um pedido disponibilizada por nossa API. O JSON a seguir apresenta esta estrutura.

```
{
  "id": 1,
  "moment": "2022-07-25T13:00:00Z",
  "status": "PAID",
  "client": {
    "id": 1,
    "name": "Maria Brown"
  },
  "payment": {
    "id": 1,
    "moment": "2022-07-25T15:00:00Z"
  },
  "items": [
    {
      "productId": 1,
      "name": "The Lord of the Rings",
      "price": 90.5,
      "quantity": 2,
      "subTotal": 181.0
    },
    {
      "productId": 3,
      "name": "Macbook Pro",
      "price": 1250.0,
      "quantity": 1,
      "subTotal": 1250.0
    }
  ],
  "total": 1431.0
}
```

Repare que nosso objeto JSON possui objetos aninhados para os campos `client`, `payment` e `items`, onde este último é uma lista de objetos que representam um item do pedido. Vamos começar então implementando classes DTO para representar esses objetos aninhados. Repare como vamos definindo os nomes dos campos das classes para corresponder com os nomes dos campos mostrados no JSON mostrado anteriormente.

Classe ClientDTO

```
package com.devsuperior.dscommerce.dto;

import com.devsuperior.dscommerce.entities.User;

public class ClientDTO {

    private Long id;
    private String name;

    public ClientDTO(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public ClientDTO(User entity) {
        id = entity.getId();
        name = entity.getName();
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

Classe PaymentDTO

```
package com.devsuperior.dscommerce.dto;

import java.time.Instant;

import com.devsuperior.dscommerce.entities.Payment;

public class PaymentDTO {

    private Long id;
    private Instant moment;

    public PaymentDTO(Long id, Instant moment) {
        this.id = id;
        this.moment = moment;
    }

    public PaymentDTO(Payment entity) {
        id = entity.getId();
        moment = entity.getMoment();
    }

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

```
    public Instant getMoment() {
        return moment;
    }
}
```

Classe OrderItemDTO

```
package com.devsuperior.dscommerce.dto;

import com.devsuperior.dscommerce.entities.OrderItem;

public class OrderItemDTO {

    private Long productId;
    private String name;
    private Double price;
    private Integer quantity;
    private String imgUrl;

    public OrderItemDTO() {
    }

    public OrderItemDTO(Long productId, String name, Double price, Integer
   quantity, String imgUrl) {
        this.productId = productId;
        this.name = name;
        this.price = price;
        this.quantity = quantity;
        this.imgUrl = imgUrl;
    }

    public OrderItemDTO(OrderItem entity) {
        productId = entity.getProduct().getId();
        name = entity.getProduct().getName();
        price = entity.getPrice();
        quantity = entity.getQuantity();
        imgUrl = entity.getProduct().getImgUrl();
    }

    public Long getProductId() {
        return productId;
    }

    public String getName() {
        return name;
    }

    public Double getPrice() {
        return price;
    }

    public Integer getQuantity() {
        return quantity;
    }

    public Double getSubTotal() {
```

```
        return price * quantity;
    }

    public String getImgUrl() {
        return imgUrl;
    }
}
```

Agora que as três classes DTO auxiliares estão implementadas, vamos implementar a classe `OrderDTO` para representar todo o objeto do pedido conforme JSON apresentado anteriormente.

Classe OrderDTO

```
package com.devsuperior.dscommerce.dto;

import java.time.Instant;
import java.util.ArrayList;
import java.util.List;

import com.devsuperior.dscommerce.entities.Order;
import com.devsuperior.dscommerce.entities.OrderItem;
import com.devsuperior.dscommerce.entities.OrderStatus;

import jakarta.validation.constraints.NotEmpty;

public class OrderDTO {

    private Long id;
    private Instant moment;
    private OrderStatus status;

    private ClientDTO client;

    private PaymentDTO payment;

    private List<OrderItemDTO> items = new ArrayList<>();

    public OrderDTO() {
    }

    public OrderDTO(Long id, Instant moment, OrderStatus status, ClientDTO
   client, PaymentDTO payment) {
        this.id = id;
        this.moment = moment;
        this.status = status;
        this.client = client;
        this.payment = payment;
    }

    public OrderDTO(Order entity) {
        this.id = entity.getId();
        this.moment = entity.getMoment();
        this.status = entity.getStatus();
        this.client = new ClientDTO(entity.getClient());
        this.payment = (entity.getPayment() == null) ? null : new PaymentDTO(
   entity.getPayment());
```

```
        for (OrderItem item : entity.getItems()) {
            OrderItemDTO itemDto = new OrderItemDTO(item);
            items.add(itemDto);
        }
    }

    public Long getId() {
        return id;
    }

    public Instant getMoment() {
        return moment;
    }

    public OrderStatus getStatus() {
        return status;
    }

    public ClientDTO getClient() {
        return client;
    }

    public PaymentDTO getPayment() {
        return payment;
    }

    public List<OrderItemDTO> getItems() {
        return items;
    }

    public Double getTotal() {
        double sum = 0.0;
        for (OrderItemDTO item : items) {
            sum += item.getSubTotal();
        }
        return sum;
    }
}
```

Funcionalidade de buscar pedido por id

Para implementar a funcionalidade de buscar pedido por id, primeiro precisamos implementar o repository, conforme código a seguir.

Interface OrderRepository

```
package com.devsuperior.dscommerce.repositories;

import org.springframework.data.jpa.repository.JpaRepository;

import com.devsuperior.dscommerce.entities.Order;

public interface OrderRepository extends JpaRepository<Order, Long> {

}
```

Depois, vamos implementar a classe `OrderService`, para buscar os dados do banco de dados e gerar como resposta o DTO completo com todos objetos aninhados.

Classe OrderService

```
package com.devsuperior.dscommerce.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.devsuperior.dscommerce.dto.OrderDTO;
import com.devsuperior.dscommerce.entities.Order;
import com.devsuperior.dscommerce.repositories.OrderRepository;
import com.devsuperior.dscommerce.services.exceptions.ResourceNotFoundException;

@Service
public class OrderService {

    @Autowired
    private OrderRepository repository;

    @Transactional(readOnly = true)
    public OrderDTO findById(Long id) {
        Order order = repository.findById(id).orElseThrow(
                () -> new ResourceNotFoundException("Recurso não encontrado"));
        return new OrderDTO(order);
    }
}
```

Por fim, vamos implementar a classe `OrderController` responsável por disponibilizar o *endpoint* para buscar um pedido por id. Repare que já fizemos o controle de acesso básico a este endpoint, definindo que só pode ser acessado por um usuário logado, seja cliente ou administrador.

```
package com.devsuperior.dscommerce.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.devsuperior.dscommerce.dto.OrderDTO;
import com.devsuperior.dscommerce.services.OrderService;

@RestController
@RequestMapping(value = "/orders")
public class OrderController {
```

```
    @Autowired
    private OrderService service;

    @PreAuthorize("hasAnyRole('ROLE_ADMIN', 'ROLE_CLIENT')")
    @GetMapping(value = "/{id}")
    public ResponseEntity<OrderDTO> findById(@PathVariable Long id) {
        OrderDTO dto = service.findById(id);
        return ResponseEntity.ok(dto);
    }
}
```

Testando o endpoint

Agora podemos testar nosso *endpoint* no Postman. Crie uma requisição na *collection* do projeto, para acessar o caminho `/orders/{id}`. Defina o cabeçalho `Authorization` com o tipo "Bearer Token" e o token com o valor `{{token}}`. Ao executar a requisição, deverá ser retornado o id do produto informado na URL.

5.27 Salvando um pedido

Nesta seção vamos mostrar a implementação da funcionalidade de salvar um pedido. Conforme especificação do sistema, os dados necessários para salvar um pedido correspondem a uma lista de produtos e suas respectivas quantidades, conforme JSON exemplo a seguir:

```
{
  "items": [
    {
      "productId": 1,
      "quantity": 2
    },
    {
      "productId": 5,
      "quantity": 1
    }
  ]
}
```

Repare, entretanto, que essa estrutura de dados está contida no tipo `ProductDTO` que criamos anteriormente. Assim, podemos utilizar o próprio tipo `ProductDTO` para receber os dados dos produtos e quantidades para salvar um novo pedido. A única modificação que faremos na classe `ProductDTO` será acrescentar uma validação para não aceitar uma lista de itens vazia:

```
public class OrderDTO {

    ...

    @NotEmpty(message = "Deve ter pelo menos um item")
    private List<OrderItemDTO> items = new ArrayList<>();
```

```
    ...
```

Implementando a funcionalidade de salvar um pedido

Para implementar a funcionalidade de salvar um pedido, primeiro vamos precisar implentar um repository para itens de pedido, pois além do registro do pedido, vamos precisar também salvar cada um dos itens do pedido. O código é como mostrado a seguir.

Interface OrderItemRepository

```
package com.devsuperior.dscommerce.repositories;

import org.springframework.data.jpa.repository.JpaRepository;

import com.devsuperior.dscommerce.entities.OrderItem;
import com.devsuperior.dscommerce.entities.OrderItemPK;

public interface OrderItemRepository extends JpaRepository<OrderItem,
    OrderItemPK> {

}
```

Agora vamos apresentar o código fonte atualizado da classe `OrderService`, com o novo método `insert` para salvar um pedido. A explicação do código será apresentada em seguida.

```
@Service
public class OrderService {

    @Autowired
    private OrderRepository repository;

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private OrderItemRepository orderItemRepository;

    @Autowired
    private UserService userService;

    @Transactional(readOnly = true)
    public OrderDTO findById(Long id) {
        Order order = repository.findById(id).orElseThrow(
                () -> new ResourceNotFoundException("Recurso não encontrado"));
        return new OrderDTO(order);
    }

    @Transactional
    public OrderDTO insert(OrderDTO dto) {

        Order order = new Order();
```

```
        order.setMoment(Instant.now());
        order.setStatus(OrderStatus.WAITING_PAYMENT);

        User user = userService.authenticated();
        order.setClient(user);

        for (OrderItemDTO itemDto : dto.getItems()) {
            Product product = productRepository.getReferenceById(itemDto.
    getProductId());
            OrderItem item = new OrderItem(order, product, itemDto.getQuantity()
    , product.getPrice());
            order.getItems().add(item);
        }

        repository.save(order);
        orderItemRepository.saveAll(order.getItems());

        return new OrderDTO(order);
    }
}
```

O método `insert` prepara um objeto `Order`, associado a um ou mais objetos `OrderItem`, e depois salva todos objetos no banco de dados.

Repare que os dados do objeto order são iniciados com os valores apropriados:

- Campo `moment` é iniciado com o instante atual.
- Campo `status` é iniciado com o valor `WAITING_PAYMENT`.
- Campo `client` é iniciado com o usuário logado.

Além disso, os itens do pedido são instanciados conforme valores advindos do objeto `dto`: o id de cada produto e sua respectiva quantidade.

Por fim, os objetos são salvos no banco de dados, e o novo objeto salvo `order` é utilizado para retorna o objeto `OrderDTO` resultante.

Método insert em OrderController

Por fim, apresentamos a implementação do método `insert` na classe `OrderController`, que é bastante similar à implementação da inserção de um produto conforme já aprendemos. Repare que incluímos o controle de acesso, de forma que somente usuários com o perfil de cliente podem inserir pedidos.

```
@RestController
@RequestMapping(value = "/orders")
public class OrderController {

    ...

    @PreAuthorize("hasRole('ROLE_CLIENT')")
    @PostMapping
    public ResponseEntity<OrderDTO> insert(@Valid @RequestBody OrderDTO dto) {
        dto = service.insert(dto);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
```

```
                .buildAndExpand(dto.getId()).toUri();
        return ResponseEntity.created(uri).body(dto);
    }
}
```

Testando a requisição

Para testar a requisição, basta criar uma nova requisição com acesso protegido de maneira similar a que fizemos anteriormente. A requisição será uma operação `POST` no caminho `/orders`, e o corpo da requisição será uma lista de objetos contendo id do produto e quantidade, conforme mostrado no início desta seção.

5.28 Controle de acesso programático self ou admin

Nesta seção vamos explorar o controle de acesso programático, ou seja, um controle de acesso a recursos que pode ser implementado em nível de aplicação.

Um exemplo desse controle é no *endpoint* que recupera um pedido por id, porque um usuário que tem perfil de cliente pode acessar por id somente os pedidos que pertencem a ele próprio, mas não podem acessar por id os pedidos de outros clientes. Além disso, se o usuário tiver perfil de administrador, ele poderá acessar qualquer pedido.

Desta forma, para implementar esta regra de negócio, precisamos implementar um controle de acesso mais refinado do que somente restringir o *endpoint* por perfil de usuário.

Criando uma exceção customizada

Vamos começar implementando nossa exceção customizada `ForbiddenException` para acesso negado a um recurso.

Classe ForbiddenException

```
package com.devsuperior.dscommerce.services.exceptions;

@SuppressWarnings("serial")
public class ForbiddenException extends RuntimeException {

    public ForbiddenException(String msg) {
        super(msg);
    }
}
```

Vamos também implementar um método específico para tratar esta exceção em nossa classe `ControllerExceptionHandler`.

Classe ControllerExceptionHandler

```
@ControllerAdvice
public class ControllerExceptionHandler {
```

```
    ...

    @ExceptionHandler(ForbiddenException.class)
    public ResponseEntity<CustomErrorDTO> forbidden(ForbiddenException e,
   HttpServletRequest request) {
        HttpStatus status = HttpStatus.FORBIDDEN;
        CustomErrorDTO err = new CustomErrorDTO(Instant.now(), status.value(), e
   .getMessage(), request.getRequestURI());
        return ResponseEntity.status(status).body(err);
    }
}
```

Classe AuthService para controle de acesso

Vamos agora criar uma classe `AuthService` responsável por implementar regras de negócio de controle de acesso.

Em nosso caso, conforme já explicado, para verificar se um usuário está autorizado a acessar um pedido por id, precisamos verificar se este usuário é o dono do pedido, ou então é um usuário administrador. Para isto, vamos criar um método `validateSelfOrAdmin` que faz esse teste, e lança uma exceção `ForbiddenException` em caso negativo. O código é mostrado a seguir.

Classe AuthService

```
package com.devsuperior.dscommerce.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.devsuperior.dscommerce.entities.User;
import com.devsuperior.dscommerce.services.exceptions.ForbiddenException;

@Service
public class AuthService {

    @Autowired
    private UserService userService;

    public void validateSelfOrAdmin(long userId) {
        User me = userService.authenticated();
        if (!me.hasRole("ROLE_ADMIN") && !me.getId().equals(userId)) {
            throw new ForbiddenException("Access denied");
        }
    }
}
```

Acrescentando o controle de acesso em OrderService

Vamos agora incluir a verificação de controle de acesso em nosso método `findById` da classe `OrderService`.

```
@Service
public class OrderService {

    ...

    @Autowired
    private AuthService authService;

    @Transactional(readOnly = true)
    public OrderDTO findById(Long id) {
        Order order = repository.findById(id).orElseThrow(
                () -> new ResourceNotFoundException("Recurso não encontrado"));
        authService.validateSelfOrAdmin(order.getClient().getId());
        return new OrderDTO(order);
    }

    ...
}
```

Testando o controle de acesso

Para testar se nosso controle de acesso programático está funcionando, basta explorar os seguintes cenários:

- Fazer login com o usuário `maria@gmail.com`. Como este usuário é somente cliente, ele poderá acessar somente seus próprios pedidos, que são os pedidos 1 e 3. Se tentar acessar o pedido 2 com este usuário, a resposta deverá ser 403.
- Fazer login com o usuário `alex@gmail.com`. Como este usuário é administrador, ele poderá acessar todos os pedidos, ou seja, os pedidos 1, 2 e 3.

5.29 Endpoint para buscar categorias

Nesta seção vamos acrescentar um endpoint para buscar as categorias de produtos. O procedimento é similar ao que já foi discutido anteriormente.

Implementações

Interface CategoryRepository

```
package com.devsuperior.dscommerce.repositories;

import org.springframework.data.jpa.repository.JpaRepository;

import com.devsuperior.dscommerce.entities.Category;

public interface CategoryRepository extends JpaRepository<Category, Long> {

}
```

Classe CategoryService

```
package com.devsuperior.dscommerce.services;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.devsuperior.dscommerce.dto.CategoryDTO;
import com.devsuperior.dscommerce.entities.Category;
import com.devsuperior.dscommerce.repositories.CategoryRepository;

@Service
public class CategoryService {

    @Autowired
    private CategoryRepository repository;

    @Transactional(readOnly = true)
    public List<CategoryDTO> findAll() {
        List<Category> result = repository.findAll();
        return result.stream().map(x -> new CategoryDTO(x)).toList();
    }
}
```

Classe CategoryController

```
package com.devsuperior.dscommerce.controllers;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.devsuperior.dscommerce.dto.CategoryDTO;
import com.devsuperior.dscommerce.services.CategoryService;

@RestController
@RequestMapping(value = "/categories")
public class CategoryController {

    @Autowired
    private CategoryService service;

    @GetMapping
    public ResponseEntity<List<CategoryDTO>> findAll() {
        List<CategoryDTO> list = service.findAll();
        return ResponseEntity.ok(list);
    }
}
```

Testando o endpoint

O *endpoint* para buscar as categorias é um endpoint público, que não exige um usuário logado. Assim, basta criar uma nova requisição na *collection* do Postman, com o método `GET` no caminho `categories`. Ao executar esta requisição, deve ser retornada uma resposta cujo corpo possua uma lista de categorias, similar ao JSON mostrado a seguir.

```
[
    {
        "id": 1,
        "name": "Livros"
    },
    {
        "id": 2,
        "name": "Eletrônicos"
    },
    {
        "id": 3,
        "name": "Computadores"
    }
]
```

5.30 Ajustes nas Validações de Dados

Para finalizarmos os últimos ajustes no código do projeto, vamos acrescentar uma pequena melhoria no nosso DTO que representa os erros de validação.

O problema que queremos tratar aqui é que um mesmo campo pode ter mais de uma tratativa de erro, conforme nós pudemos verificar na classe `ProductDTO`:

```
public class ProductDTO {

    private Long id;

    @Size(min = 3, max = 80, message = "Nome precisar ter de 3 a 80 caracteres")
    @NotBlank(message = "Campo requerido")
    private String name;

    @Size(min = 10, message = "Descrição precisa ter no mínimo 10 caracteres")
    @NotBlank(message = "Campo requerido")
    private String description;

    @NotNull(message = "Campo requerido")
    @Positive(message = "O preço deve ser positivo")
    private Double price;

    private String imgUrl;

    ...
```

Conforme podemos observar, alguns campos possuem mais de uma tratativa de erro. Porém, na

resposta de nossas requisições, queremos mostrar apenas uma mensagem de erro por vez para cada campo. Assim, para assegurar esse comportamento em nossa resposta, vamos recapitular nossa classe `ValidationError`, responsável por carregar os dados de validação:

```
public class ValidationError extends CustomError {

    private List<FieldMessage> errors = new ArrayList<>();

    public ValidationError(Instant timestamp, Integer status, String error,
    String path) {
        super(timestamp, status, error, path);
    }

    public List<FieldMessage> getErrors() {
        return errors;
    }

    public void addError(String fieldName, String message) {
        errors.add(new FieldMessage(fieldName, message));
    }
}
```

Para assegurar que apenas um erro seja armazenado para cada campo do objeto, vamos acrescentar um comando no método `addError` para remover algum possível erro do mesmo campo, antes de adicionar um novo erro para este campo:

```
public void addError(String fieldName, String message) {
    errors.removeIf(x -> x.getFieldName().equals(fieldName));
    errors.add(new FieldMessage(fieldName, message));
}
```

Esta é uma forma de assegurar que apenas um erro para cada campo esteja presente no objeto `ValidationError`, e consequentemente na resposta da requisição.

Capítulo 6

Homologação e implantação com CI/CD

6.1 Visão geral do capítulo

Bem-vindos ao capítulo "Homologação e implantação com CI/CD". Neste capítulo vamos aprender como duas coisas:

- **Homologação local com banco de dados real**: até o momento, estamos utilizando apenas o banco de dados H2 para fazer nossos testes durante o desenvolvimento. Porém, antes de implantar a aplicação na nuvem, é preciso homologar se realmente a aplicação está funcionando com o mesmo banco de dados que será utilizado na nuvem, no caso o banco de dados Postgresql.
- **Implantação na nuvem com CI/CD usando PaaS**: também vamos aprender o procedimento para implantar a aplicação na nuvem, com o banco de dados. Vamos implantar nossa aplicação usando um serviço de *PaaS*, ou seja "Platform as a Service. *Estes serviços, tais como Heroku e Railway, oferecem uma infraestrutura preparada que facilita a implantação e manutenção da aplicação na nuvem, especialmente para iniciantes. Também vamos contar com uma estrutura de* CI/CD*, ou seja, integração contínua e entrega contínua, que permitirá a gestão automatizada de mudanças, de modo que, ao registrar um novo* commit* em nosso repositório Git, o processo de implantação da nova versão na nuvem ocorrerá de forma automatizada.

6.2 Aviso conteúdo opcional sobre implantação

Esta seção é apenas para avisar que o conteúdo deste capítulo pode ser opcional neste momento, caso você não esteja disposto a gastar dinheiro, ou usar um cartão de crédito, com a implantação.

Isso porque os serviços de hospedagem geralmente possuem custos, e requerem um cartão de crédito internacional válido. Sendo assim, como o projeto que estamos fazendo aqui é um projeto de estudo para portfolio, talvez não valha a pena arcar com custos neste momento.

Assim, queremos deixar claro que o conteúdo deste capítulo é opcional, e uma opção de estudo no momento possa ser apenas consumir o conteúdo para aprendizado, porém focar em

deixar o projeto como portfolio para seu currículo no seu Github, com um bom README de apresentação.

Aproveitamos para sugerir um conteúdo nosso do Youtube sobre como fazer um README para portfolio no Github:

https://youtu.be/jIa8R69pKh8

6.3 Sobre os Serviços de Implantação

Esta seção proporciona uma visão geral sobre os principais serviços de cloud para hospedagem de sistemas na nuvem, explorando as características de serviços de cloud "completos" e serviços PaaS (Platform as a Service).

Serviços de Cloud "Completos"

Os serviços de cloud completos oferecem uma gama extensiva de recursos que cobrem praticamente todas as necessidades de infraestrutura de TI, desde servidores e armazenamento até redes e inteligência artificial. Eles são ideais para empresas que precisam de soluções altamente escaláveis e flexíveis. Os principais players nesta categoria incluem:

1. **Amazon Web Services (AWS)**:
   - **Características**: Oferece uma extensa gama de serviços incluindo computação, armazenamento, bases de dados, análise, redes, mobile, ferramentas de desenvolvedor, ferramentas de gestão, IoT, segurança e aplicações empresariais.
   - **Vantagens**: Alta escalabilidade, presença global com múltiplas regiões e zonas de disponibilidade, e um amplo ecossistema de serviços integrados.
2. **Microsoft Azure**:
   - **Características**: Fornece mais de 200 produtos e serviços de cloud projetados para ajudar a trazer novas soluções para a vida — para resolver os desafios de hoje e criar o futuro.
   - **Vantagens**: Integração forte com produtos Microsoft, como Office 365 e Windows, e suporte para uma variedade de frameworks e linguagens de programação.
3. **Google Cloud Platform (GCP)**:
   - **Características**: Oferece serviços em computação, armazenamento de dados, machine learning, e análise de dados, tudo integrado com a segurança avançada do Google.
   - **Vantagens**: Alta qualidade em serviços de contêineres e machine learning com a facilidade de integração com ferramentas e serviços Google.

Serviços de Cloud PaaS

Os serviços PaaS são plataformas de desenvolvimento e implantação que simplificam a codificação, o teste, o deployment e a manutenção de aplicações. Eles são ideais para desenvolvedores que querem se concentrar no desenvolvimento de aplicações sem se preocupar com a infraestrutura subjacente. Os principais serviços PaaS incluem:

1. **Heroku**:
   - **Características**: Permite aos desenvolvedores construir, rodar e operar aplicações inteiramente na cloud.

- **Vantagens**: Fácil de usar, suporta várias linguagens de programação populares como Ruby, Java, PHP, Python, Node, Go, Scala e Clojure.

2. **Railway**:
   - **Características**: Plataforma que oferece uma experiência simplificada para lançar rapidamente aplicações e bancos de dados.
   - **Vantagens**: Integração fácil com GitHub, provisionamento rápido e interface intuitiva.
3. **Netlify**:
   - **Características**: Foca na hospedagem e automação de front-ends modernos, com suporte robusto para JAMstack.
   - **Vantagens**: Deploy automático a partir de sistemas de controle de versão, funcionalidades de pré-visualização ao vivo, e alto desempenho para aplicações estáticas.
4. **Firebase**:
   - **Características**: Plataforma do Google que oferece uma solução completa para o desenvolvimento de aplicações móveis e web que necessitam de backend, análise e recursos de ligação com o cliente.
   - **Vantagens**: Integração com outros serviços Google, autenticação simplificada, e uma excelente base de dados em tempo real.

Escolher entre um serviço de cloud completo e um PaaS depende das necessidades específicas do projeto e da equipe de desenvolvimento. Enquanto os serviços completos oferecem controle total sobre os recursos e a configuração, os serviços PaaS proporcionam conveniência e eficiência, especialmente para projetos com requisitos de infraestrutura mais simples ou para equipes que desejam evitar a complexidade da gestão de servidores. Ambos os tipos de serviços têm o potencial de oferecer soluções robustas e escaláveis para uma variedade de necessidades de desenvolvimento e implantação de aplicações.

6.4 Preparando projeto DSList para o estudo de caso com Railway

Nesta seção e nas subsequentes, vamos iniciar um estudo de caso de implantação de uma pequena aplicação na plataforma Railway, que é um serviço *PaaS* que tipicamente oferece uma cota gratuita de tempo mensal, o que pode ser uma opção viável para testar pequenos projetos de estudo.

Nesta seção vamos preparar o projeto Spring Boot utilizado neste estudo de caso. Será o projeto DSList: um pequeno sistema de lista de jogos.

Passos para baixar o projeto e ajustar a versão do mesmo

Primeiramente, vamos clonar o projeto para nosso computador:

```
git clone git@github.com:devsuperior/dslist-backend.git
```

Agora, dentro da pasta do projeto, vamos fazer o comando Git para deixar o projeto somente até o commit `SQL, projection, get games by list`:

```
git reset --hard cc7be51
```

Neste momento, você pode conferir o histórico de commits do projeto fazendo o seguinte comando:

```
git log --oneline
```

Passos para salvar o projeto no seu Github

Agora que o projeto já está no seu computador local na versão desejada, precisamos salvar o projeto no seu Github. Para isto, crie um repositório vazio no seu Github e execute o comando padrão `git remote` para associar o repositório remoto ao seu projeto local, porém trocando o comando `add` pelo comando `set-url`. O exemplo a seguir mostra como deve ser dado o comando. Lembre-se de trocar `meuusuario` e `meuprojeto` pelo seu nome de usuário e pelo nome do seu projeto no Github, respectivamente:

```
git remote set-url origin git@github.com:meuusuario/meuprojeto.git
```

Agora que seu projeto local está devidamente associado com o repositório no Github, podemos salvar o projeto no Github com o comando:

```
git push -u origin main
```

Testando o projeto DSList

Vamos agora testar o projeto. Primeiramente, importe o projeto na sua IDE, e execute o mesmo.

Em seguida, importe a *collection* do Postman do projeto. A *collection* pode ser obtida na pasta principal do projeto referência no Github:

https://github.com/devsuperior/dslist-backend

Atenção: a *collection* Postman oferecida possui cinco *endpoints*, sendo quatro `GET` e um `POST`. O *endpoint* `POST` não será usado nesta versão do projeto.

Execute cada um dos *endpoints* `GET` da *collection* e verifique se os dados são devidamente retornados pela API.

6.5 Perfis de Projeto

Nesta seção, exploraremos o conceito de perfis de projeto no contexto de um projeto Spring Boot, uma funcionalidade muito útil para gerenciar diferentes configurações de ambiente dentro

do mesmo projeto.

Visão Geral

Perfis de projeto são uma maneira de segmentar as configurações do aplicativo para diferentes ambientes, facilitando a gestão de variáveis específicas de cada ambiente, como configurações de banco de dados, variáveis de ambiente e customizações de comportamento. No Spring Boot, esses perfis são definidos no `application.properties` ou `application.yml`, permitindo ativar ou desativar certos beans e configurações baseadas no perfil ativo.

Exemplos de Perfis

Perfil `test`

- **Descrição**: Utilizado principalmente durante o desenvolvimento e testes.
- **Configuração típica**:
  - Utilização do banco de dados H2, que é um banco de dados em memória.
  - Configurações simplificadas que não necessitam de setup externo.
  - Esse perfil facilita a execução de testes automatizados e verificações rápidas de integridade e funcionalidade do código.

Perfil `dev`

- **Descrição**: Usado para desenvolvimento contínuo e homologação em um ambiente local.
- **Configuração típica**:
  - Conexão com uma instância local do PostgreSQL.
  - Configurações podem incluir detalhes mais próximos ao ambiente de produção, mas ainda em um cenário controlado.

Perfil `prod`

- **Descrição**: Configurações destinadas ao ambiente de produção.
- **Configuração típica**:
  - Conexão com o banco de dados PostgreSQL em um ambiente de nuvem.
  - Configurações de segurança e performance otimizadas para carga alta.

Os perfis de projeto permitem flexibilidade para gerenciar as diferenças entre os ambientes de desenvolvimento, teste e produção de uma aplicação Spring Boot. Com o uso adequado dos perfis, é possível manter uma base de código comum e, ao mesmo tempo, garantir que cada ambiente funcione com as configurações apropriadas sem necessidade de alterações manuais contínuas nas configurações de desenvolvimento.

6.6 Preparando Postgresql e pgAdmin com Docker

Nesta seção, vamos aprender a configurar e utilizar o Postgresql e o pgAdmin em containers Docker usando Docker Compose. Esta abordagem facilita o gerenciamento das versões do banco de dados e das ferramentas de administração, além de manter o ambiente de desenvolvimento limpo e padronizado.

Docker Compose para Postgresql e pgAdmin

O Docker Compose é uma ferramenta que permite definir e rodar multi-containers Docker. Vamos utilizar um arquivo `docker-compose.yml` para definir os serviços necessários para nosso ambiente de desenvolvimento com Postgresql e pgAdmin. Abaixo está o script Docker Compose explicado:

```
version: "3.7"
services:
  # Postgres Server
  pg-docker:
    image: postgres:14-alpine
    container_name: dev-postgresql
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_PASSWORD: 1234567
    ports:
      - 5433:5432
    volumes:
      - ./.data/postgresql/data:/var/lib/postgresql/data
    networks:
      - dev-network
  # pgAdmin
  pgadmin-docker:
    image: dpage/pgadmin4
    container_name: dev-pgadmin
    environment:
      PGADMIN_DEFAULT_EMAIL: me@example.com
      PGADMIN_DEFAULT_PASSWORD: 1234567
    ports:
      - 5050:80
    volumes:
      - ./.data/pgadmin:/var/lib/pgadmin
    depends_on:
      - pg-docker
    networks:
      - dev-network
networks:
  dev-network:
    driver: bridge
```

Descrição dos Serviços

- **Postgres Server (`pg-docker`)**:
  - Utiliza a imagem `postgres:14-alpine`.
  - Configura a base de dados com o nome `mydatabase` e senha `1234567`.
  - Mapeia a porta `5433` do host para `5432` do container, permitindo acesso ao banco de dados do host.
  - Volumes são usados para persistir os dados do Postgres fora do ciclo de vida do container.
- **pgAdmin (`pgadmin-docker`)**:
  - Utiliza a imagem `dpage/pgadmin4`.
  - Define o email e senha padrão para login no pgAdmin.

- Mapeia a porta `5050` do host para `80` do container, permitindo acesso via navegador.
- Configura volumes para persistir configurações do pgAdmin.
- Depende do serviço `pg-docker`, garantindo que o Postgres esteja disponível antes do pgAdmin iniciar.

Passo a Passo para Subir a Estrutura

1. **Crie um diretório para os dados**: Certifique-se de criar o diretório `.data` na raiz do seu projeto, que será usado para armazenar os dados persistentes do Postgres e do pgAdmin.
2. **Inicie os serviços**: No diretório onde o arquivo `docker-compose.yml` está localizado, execute o comando:

   ```
   docker-compose up -d
   ```

   Este comando baixa as imagens necessárias, cria e inicia os containers em modo desanexado.
3. **Acessar o pgAdmin**:
   - Abra um navegador e visite `http://localhost:5050`.
   - Use o email `me@example.com` e a senha `1234567` para entrar.
4. **Conectar ao Postgresql pelo pgAdmin**:
   - Crie uma nova conexão de servidor no pgAdmin.
   - No campo hostname, use `dev-postgresql`, que é o nome do serviço Postgres definido no Docker Compose.
   - A porta será `5432`, o nome do usuário padrão é `postgres`, e a senha é `1234567`.
   - O nome do banco de dados é `mydatabase`.

Testando a Configuração

Verifique se consegue conectar ao banco de dados pelo pgAdmin e execute comandos SQL básicos para testar a conectividade. Se tudo estiver configurado corretamente, você poderá gerenciar o banco de dados Postgresql através do pgAdmin rodando em containers Docker.

6.7 Perfis de projeto dev e prod

Agora que o banco de dados Postgresql está executando localmente, vamos preparar os perfis `dev` e `prod` em nosso projeto Spring Boot.

Perfil dev

Vamos começar criando o perfil `dev`, que será o perfil de projeto que utilizaremos para homologar localmente a aplicação com o banco de dados Postgresql local.

Dentro da pasta `src/main/resources` do projeto, crie um arquivo `application-dev.properties` com o seguinte conteúdo:

application-dev.properties

```
#spring.jpa.properties.jakarta.persistence.schema-generation.create-source=
    metadata
#spring.jpa.properties.jakarta.persistence.schema-generation.scripts.action=
    create
#spring.jpa.properties.jakarta.persistence.schema-generation.scripts.create-
    target=create.sql
#spring.jpa.properties.hibernate.hbm2ddl.delimiter=;

spring.datasource.url=jdbc:postgresql://localhost:5433/dslist
spring.datasource.username=postgres
spring.datasource.password=1234567

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.hibernate.ddl-auto=none
```

Repare no padrão de nome do arquivo de configuração do perfil do projeto, que possui o sufixo `dev`, conforme o nome do perfil. O próprio framework reconhece o nome do perfil de projeto pelo padrão de nome do arquivo.

Sobre o código de configuração, as primeiras quatro linhas de configuração estão comentadas com o símbolo `#`, pois são configurações para gerar o script SQL, que serão utilizadas mais adiante.

As três linhas seguintes são configurações de conexão com a base de dados Postgresql, que definem as URL de conexão, o nome de usuário e a senha. Repare que a porta utilizada na URL de conexão é a `5433`, pois nossa aplicação Spring Boot vai realizar um acesso externo ao container do servidor do Postgresql.

As três últimas linhas são configurações adicionais do banco de dados Postgresql:

1. **spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect**
   - Esta linha especifica o dialeto do banco de dados que o Hibernate deve usar. Os dialetos do Hibernate são configurações que permitem que o Hibernate traduza suas consultas de entidades para o SQL específico que é compreendido pelo banco de dados em uso, neste caso, o PostgreSQL. O uso do dialeto correto assegura que o Hibernate possa gerar SQL que é otimizado para o tipo de banco de dados que você está utilizando.
2. **spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true**
   - Esta configuração é um pouco específica e trata de um problema que pode ocorrer quando o Hibernate tenta lidar com tipos de dados LOB (Large Objects, como `BLOB` e `CLOB`) em um banco de dados PostgreSQL. O parâmetro `non_contextual_creation` quando definido como `true`, instrui o Hibernate a evitar certas operações contextuais ao criar LOBs. Isso é útil para evitar exceções devido a uma tentativa de acesso a métodos LOB em um contexto JDBC inadequado.
3. **spring.jpa.hibernate.ddl-auto=none**
   - Esta propriedade controla o comportamento do Hibernate em relação ao schema do banco de dados durante a inicialização e o fechamento da aplicação. Os valores comuns para essa configuração incluem `none`, `update`, `create`, `create-drop`, entre outros.

O valor `none` especificado aqui indica que o Hibernate não deve fazer nenhuma alteração no esquema do banco de dados automaticamente. Isso significa que nenhum esquema será gerado nem modificado ao iniciar ou parar a aplicação, o que é útil em ambientes de produção onde as alterações de esquema devem ser controladas de forma mais rigorosa.

Perfil prod

Agora vamos preparar o perfil `prod`, que será o perfil do projeto para executar em produção, online na nuvem.

Dentro da pasta `src/main/resources` do projeto, crie um arquivo `application-prod.properties` com o seguinte conteúdo:

application-prod.properties

```
spring.datasource.url=${DB_URL}
spring.datasource.username=${DB_USERNAME}
spring.datasource.password=${DB_PASSWORD}

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.hibernate.ddl-auto=none
```

Repare que, no arquivo de configuração do perfil de produção, nós não utilizamos os valores reais de conexão ao banco de dados, mas sim variáveis de ambiente `${DB_URL}`. `${DB_USERNAME}` e `${DB_PASSWORD}`. Os valores destas variáveis serão configurados dentro do ambiente de execução na nuvem na plataforma onde a aplicação será implantada.

Arquivo system.properties

Vamos também criar um arquivo chamado `system.properties` na pasta raiz do projeto Spring Boot. Nós vamos criar este arquivo, pois ele é eventualmente necessário por algumas plataformas *PaaS*. O conteúdo do arquivo deverá ser conforme mostrado a seguir. Você pode mudar a versão do Java conforme seu projeto:

```
java.runtime.version=21
```

6.8 Script SQL para criar base de dados e seed

Nesta seção vamos aprender como gerar um script completo para criar e popular a base de dados Postgresql.

Passos para gerar o script SQL

Primeiramente, vamos preparar nossa aplicação para executar no perfil `dev`. Para isto, edite o arquivo `application.properties`, definindo o perfil `dev` como padrão:

application.properties

```
spring.profiles.active=${APP_PROFILE:dev}
spring.jpa.open-in-view=false

cors.origins=${CORS_ORIGINS:http://localhost:5173,http://localhost:3000}
```

Em seguida, descomente as quatro primeiras linhas do arquivo `application-dev.properties`:

application-dev.properties

```
spring.jpa.properties.jakarta.persistence.schema-generation.create-source=
    metadata
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.action=
    create
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.create-
    target=create.sql
spring.jpa.properties.hibernate.hbm2ddl.delimiter=;

spring.datasource.url=jdbc:postgresql://localhost:5433/dslist
spring.datasource.username=postgres
spring.datasource.password=1234567

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.hibernate.ddl-auto=none
```

Neste momento, acesse seu pgAdmin e crie uma base de dados vazia de nome `dslist`, conforme especificado na URL de conexão.

Depois disso, execute o projeto Spring Boot, e observe que será criado um script SQL de nome `create.sql` na pasta do projeto Spring Boot. O código desse script conterá todos comandos para criar a base de dados e inserir os dados:

```
create table tb_belonging (position integer, game_id bigint not null, list_id
    bigint not null, primary key (game_id, list_id));
create table tb_game (game_year integer, score float(53), id bigint generated by
     default as identity, genre varchar(255), img_url varchar(255),
    long_description TEXT, platforms varchar(255), short_description TEXT, title
    varchar(255), primary key (id));
create table tb_game_list (id bigint generated by default as identity, name
    varchar(255), primary key (id));

alter table if exists tb_belonging add constraint FKrchwdikeu66uky1hf75ym1kh
    foreign key (list_id) references tb_game_list;
alter table if exists tb_belonging add constraint FK2slybclee7wdfxhfltbvqkgpg
    foreign key (game_id) references tb_game;

INSERT INTO tb_game_list (name) VALUES ('Aventura e RPG');
INSERT INTO tb_game_list (name) VALUES ('Jogos de plataforma');

INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
```

```
    short_description, long_description) VALUES ('Mass Effect Trilogy', 4.8,
    2012, 'Role-playing (RPG), Shooter', 'XBox, Playstation, PC', 'https://raw.
    githubusercontent.com/devsuperior/java-spring-dslist/main/resources/1.png', '
    Lorem ipsum...');
INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
    short_description, long_description) VALUES ('Red Dead Redemption 2', 4.7,
    2018, 'Role-playing (RPG), Adventure', 'XBox, Playstation, PC', 'https://raw.
    githubusercontent.com/devsuperior/java-spring-dslist/main/resources/2.png', '
    Lorem ipsum...');
INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
    short_description, long_description) VALUES ('The Witcher 3: Wild Hunt', 4.7,
     2014, 'Role-playing (RPG), Adventure', 'XBox, Playstation, PC', 'https://raw
    .githubusercontent.com/devsuperior/java-spring-dslist/main/resources/3.png',
    'Lorem ipsum...');
INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
    short_description, long_description) VALUES ('Sekiro: Shadows Die Twice',
    3.8, 2019, 'Role-playing (RPG), Adventure', 'XBox, Playstation, PC', 'https
    ://raw.githubusercontent.com/devsuperior/java-spring-dslist/main/resources/4.
    png', 'Lorem ipsum...');
INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
    short_description, long_description) VALUES ('Ghost of Tsushima', 4.6, 2012,
    'Role-playing (RPG), Adventure', 'XBox, Playstation, PC', 'https://raw.
    githubusercontent.com/devsuperior/java-spring-dslist/main/resources/5.png', '
    Lorem ipsum...');
INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
    short_description, long_description) VALUES ('Super Mario World', 4.7, 1990,
    'Platform', 'Super Ness, PC', 'https://raw.githubusercontent.com/devsuperior/
    java-spring-dslist/main/resources/6.png', 'Lorem ipsum...');
INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
    short_description, long_description) VALUES ('Hollow Knight', 4.6, 2017, '
    Platform', 'XBox, Playstation, PC', 'https://raw.githubusercontent.com/
    devsuperior/java-spring-dslist/main/resources/7.png', 'Lorem ipsum...');
INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
    short_description, long_description) VALUES ('Ori and the Blind Forest', 4,
    2015, 'Platform', 'XBox, Playstation, PC', 'https://raw.githubusercontent.com
    /devsuperior/java-spring-dslist/main/resources/8.png', 'Lorem ipsum...');
INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
    short_description, long_description) VALUES ('Cuphead', 4.6, 2017, 'Platform'
    , 'XBox, Playstation, PC', 'https://raw.githubusercontent.com/devsuperior/
    java-spring-dslist/main/resources/9.png', 'Lorem ipsum...');
INSERT INTO tb_game (title, score, game_year, genre, platforms, img_url,
    short_description, long_description) VALUES ('Sonic CD', 4, 1993, 'Platform',
     'Sega CD, PC', 'https://raw.githubusercontent.com/devsuperior/java-spring-
    dslist/main/resources/10.png', 'Lorem ipsum...');

INSERT INTO tb_belonging (list_id, game_id, position) VALUES (1, 1, 0);
INSERT INTO tb_belonging (list_id, game_id, position) VALUES (1, 2, 1);
INSERT INTO tb_belonging (list_id, game_id, position) VALUES (1, 3, 2);
INSERT INTO tb_belonging (list_id, game_id, position) VALUES (1, 4, 3);
INSERT INTO tb_belonging (list_id, game_id, position) VALUES (1, 5, 4);
INSERT INTO tb_belonging (list_id, game_id, position) VALUES (2, 6, 0);
INSERT INTO tb_belonging (list_id, game_id, position) VALUES (2, 7, 1);
INSERT INTO tb_belonging (list_id, game_id, position) VALUES (2, 8, 2);
INSERT INTO tb_belonging (list_id, game_id, position) VALUES (2, 9, 3);
INSERT INTO tb_belonging (list_id, game_id, position) VALUES (2, 10, 4);
```

Agora é possível executar este script SQL na base de dados `dslist`. No pgAdmin, clique com o botão direito na base de dados `dslist` e selecione "Query tool" para abrir uma janela de execução de consultas SQL. Copie o script no conteúdo dessa janela e execute o script. Depois disso, clique novamente com o botão direito na base de dados `dslist` e selecione "Refresh". As tabelas `tb_belonging`, `tb_game` e `tb_game_list` deverão aparecer na base de dados.

6.9 Homologação local com Postgresql

Depois de criar a base de dados no Postgresql local, neste momento já podemos fazer a homologação manual do sistema, ou seja, vamos executar o sistema localmente e testar todas requisições no Postman, para homologar que todas consultas ao banco de dados também estejam funcionando com o Postgresql.

Particularmente neste projeto DSList, ocorreu um erro interessante ao tentar executar uma consulta de jogos por lista. Vamos detalhar a seguir o que ocorreu.

A consulta SQL original do projeto, contida no componente `GameRepository` do projeto, era conforme mostrado a seguir:

```
public interface GameRepository extends JpaRepository<Game, Long> {

    @Query(nativeQuery = true, value = """
            SELECT tb_game.id, tb_game.title, tb_game.game_year AS `year`,
   tb_game.img_url AS imgUrl,
            tb_game.short_description AS shortDescription, tb_belonging.position
            FROM tb_game
            INNER JOIN tb_belonging ON tb_game.id = tb_belonging.game_id
            WHERE tb_belonging.list_id = :listId
            ORDER BY tb_belonging.position
                """)
    List<GameMinProjection> searchByList(Long listId);
}
```

Repare que, na consulta utilizamos uma renomeação de campo com a cláusula `AS`, atribuindo o nome do campo para `year` entre crases. Esta renomeação funciona corretamente no banco de dados H2, porém gerou um erro ao executar no Postgresql.

Sendo assim, foi preciso mudar esta consulta, atribuindo um nome diferente de `year`, sem crases. A consulta corrigida ficou como mostrado a seguir:

```
public interface GameRepository extends JpaRepository<Game, Long> {

    @Query(nativeQuery = true, value = """
            SELECT tb_game.id, tb_game.title, tb_game.game_year AS gameYear,
   tb_game.img_url AS imgUrl,
            tb_game.short_description AS shortDescription, tb_belonging.position
            FROM tb_game
            INNER JOIN tb_belonging ON tb_game.id = tb_belonging.game_id
            WHERE tb_belonging.list_id = :listId
            ORDER BY tb_belonging.position
                """)
```

```
    List<GameMinProjection> searchByList(Long listId);
}
```

Desta forma, durante o processo de homologação manual, pudemos fazer as adequações necessárias para assegurar que todo o sistema executasse corretamente no banco de dados Postgresql.

Após realizar a homologação manual de todos endpoints, deve-se fazer os seguintes passos para finalizar o procedimento:

1. Voltar o perfil padrão do projeto para `test` no arquivo `application.properties`:

```
spring.profiles.active=${APP_PROFILE:test}
spring.jpa.open-in-view=false

cors.origins=${CORS_ORIGINS:http://localhost:5173,http://localhost:3000}
```

2. Deletar o arquivo `create.sql` da pasta principal do projeto. **Atenção:** salve o arquivo em outra pasta, pois esse script será útil depois para iniciar a base de dados no ambiente de produção.
3. Comente novamente as quatro configurações para gerar a base de dados no arquivo `application-dev-properties`:

```
#spring.jpa.properties.jakarta.persistence.schema-generation.create-source=
    metadata
#spring.jpa.properties.jakarta.persistence.schema-generation.scripts.action=
    create
#spring.jpa.properties.jakarta.persistence.schema-generation.scripts.create-
    target=create.sql
#spring.jpa.properties.hibernate.hbm2ddl.delimiter=;

spring.datasource.url=jdbc:postgresql://localhost:5433/dslist
spring.datasource.username=postgres
spring.datasource.password=1234567

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.hibernate.ddl-auto=none
```

4. Salve um novo commit do projeto, e salve o projeto no Github.

6.10 Processo de Deploy no Railway

Railway é uma plataforma de deploy rápido que simplifica o processo de hospedar e gerenciar aplicações na nuvem. Vamos explorar como utilizar o Railway para implantar uma aplicação Spring Boot e um banco de dados Postgresql.

Pré-requisitos e Criação de Conta no Railway

Antes de começarmos, você precisa criar uma conta no Railway. Siga os passos abaixo para configurar sua conta:

1. **Acesse o site**: Vá para railway.app e clique em "Start for Free".
2. **Autenticação**: Você pode se registrar usando uma conta GitHub, GitLab, ou Bitbucket para autenticação rápida. Alternativamente, você pode usar um endereço de e-mail.
3. **Confirmação de E-mail**: Se você se registrar com um e-mail, será necessário confirmar o endereço através de um link enviado para o seu e-mail antes de prosseguir.

Criando uma Instância de Banco de Dados Postgresql

Após criar sua conta no Railway, siga estes passos para configurar um banco de dados Postgresql:

1. **Dashboard**: No painel de controle do Railway, clique em "New Project".
2. **Selecionar Banco de Dados**: Escolha "Provision a Database" e selecione Postgresql da lista de opções disponíveis.
3. **Configuração do Projeto**: Dê um nome ao seu projeto e selecione a região mais próxima para hospedar seu banco de dados.
4. **Criação do Projeto**: Clique em "Create" para provisionar o banco de dados Postgresql.

Configurando o Banco de Dados

Após a criação, você pode acessar o banco de dados através do painel do Railway, onde informações como Host, Port, Username, Password e Database estão disponíveis.

- **Executar Script SQL**: Para criar as tabelas e configurar o banco de dados, você pode conectar-se ao banco de dados usando o pgAdmin e executar o script `create.sql`.

Implantando uma Aplicação Spring Boot

Para implantar uma aplicação Spring Boot que está no GitHub, siga estes passos:

1. **Criar Novo Projeto**: No Railway, crie um novo projeto e escolha a opção "Deploy from GitHub".
2. **Configuração do Repositório**: Conecte sua conta GitHub e selecione o repositório que contém sua aplicação Spring Boot.
3. **Configurações de Deploy**:
   - Defina as variáveis de ambiente necessárias para a aplicação, como `DB_URL`, `DB_USERNAME`, e `DB_PASSWORD`, usando os dados fornecidos pelo serviço de banco de dados do Railway. O nome destas variáveis devem ser os mesmos configurados no arquivo `application-prod.properties`.
   - Certifique-se de configurar o perfil de produção no Spring Boot para usar as configurações corretas.
4. **Deploy**: Após configurar, clique em "Deploy" para iniciar o processo de deploy. Railway irá construir e executar sua aplicação baseada na configuração do seu repositório.

Acesso e Gerenciamento

Após o deploy, o Railway fornece uma URL pública onde você pode acessar sua aplicação. Além disso, você pode gerenciar sua aplicação através do painel do Railway, onde é possível visualizar logs, reiniciar serviços, e modificar configurações.

Este processo permite que você rapidamente configure, teste e implante aplicações sem a necessidade de gerenciar a infraestrutura subjacente, tornando o Railway uma opção conveniente para desenvolvedores que desejam focar mais no desenvolvimento do que na administração do sistema.

6.11 Teste de CI CD adicionando CORS

Para finalizar nosso estudo de caso com Railway, vamos testar o processo de CI/CD do Railway integrado com Github.

Vamos primeiro fazer uma modificação no projeto. Em um subpacote `config`, crie uma classe `WebConfig` conforme código a seguir. Esta é uma configuração básica de CORS, para liberar o backend para ser acessado por aplicações frontend nos navegadores:

```
package com.devsuperior.dslist.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig {

    @Value("${cors.origins}")
    private String corsOrigins;

    @Bean
    WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**").allowedMethods("*").allowedOrigins(
   corsOrigins);
            }
        };
    }

}
```

Agora, salve um novo commit e envie a modificação para o Github. Observe no painel do projeto no Railway se um novo deploy é automaticamente iniciado.

Se tudo funcionar corretamente, um novo deploy será executado no Railway e a aplicação Spring Boot estará atualizada conforme as modificações realizadas.

6.12 Preparando projeto DSCommerce para o estudo de caso com Heroku

Nesta seção e nas próximas, vamos realizar um estudo de caso de implantação no Heroku.

Vale ressaltar novamente que, como o Heroku é uma plataforma que exige cartão de crédito internacional e possui custo, caso você não tenha disponibilidade no momento para realizar o procedimento na prática, você pode apenas consumir este conteúdo para conhecimento neste momento.

Para iniciarmos nosso estudo de caso, é preciso que seu projeto DSCommerce esteja preparado na sua IDE, e devidamente salvo no Github.

Como o projeto DSCommerce já foi largamente abordado neste curso, vamos presumir que você já tem o projeto preparado no seu computador. Vamos apenas te fazer a recomendação que você execute o projeto e teste todos endpoints no Postman conforme aprendemos nos capítulos anteriores. Feito isso, você estará pronto para seguir com o estudo de caso nas seções seguintes.

6.13 Perfil dev de homologação local

Dependências Maven do Posgresql

O primeiro passo de nosso estudo de caso é adicionar as dependências do Postgresql no projeto Spring Boot. Adicione a seguinte dependência no arquivo `pom.xml` do projeto DSCommerce:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

Perfil dev

Em seguida, vamos adicionar o perfil `dev` ao projeto Spring Boot. Na pasta `src/main/resources`, crie o arquivo `application-dev.properties` com o seguinte conteúdo:

```
#spring.jpa.properties.javax.persistence.schema-generation.create-source=
    metadata
#spring.jpa.properties.javax.persistence.schema-generation.scripts.action=create
#spring.jpa.properties.javax.persistence.schema-generation.scripts.create-target
    =create.sql
#spring.jpa.properties.hibernate.hbm2ddl.delimiter=;

spring.datasource.url=jdbc:postgresql://localhost:5433/dscommerce
spring.datasource.username=postgres
spring.datasource.password=1234567

spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.hibernate.ddl-auto=update
```

Agora, acesse seu pgAdmin e crie uma base de dados local chamada `dscommerce` conforme o nome da base de dados especificado na URL de conexão do arquivo `application-dev.properties`.

Mude o perfil padrão de execução da aplicação para `dev` no arquivo `application.properties` e execute o projeto. O projeto neste momento deve executar sem gerar nenhum erro no console da IDE.

6.14 Script SQL de criação da base de dados

Prosseguindo com nosso estudo de caso com Heroku, devemos agora criar a base de dados do DSCommerce no banco de dados Postgresql, bem como povoar os dados das tabelas.

Para isto, você deve fazer o procedimento de geração do script SQL a partir do projeto Spring Boot, conforme foi mostrado no estudo de caso do Railway.

Assim, neste momento faça o procedimento de geração do script SQL do projeto DSCommerce, o qual vai gerar o arquivo `create.sql` na pasta raiz do projeto DSCommerce.

Depois disso, acesse a base de dados `dscommerce` usando o pgAdmin, e execute o script `create.sql` nessa base de dados para criar as tabelas e os dados.

6.15 Criando Projeto e Base de Dados no Heroku

Heroku é uma plataforma como serviço (PaaS) que permite aos desenvolvedores construir, executar e operar aplicações inteiramente na nuvem. Nesta seção, vamos guiar você através do processo de criação de um projeto no Heroku e de provisionamento de uma base de dados PostgreSQL.

Pré-requisitos e Criação de Conta no Heroku

Antes de começar, você precisará de uma conta no Heroku:

1. **Acesse o site**: Visite https://www.heroku.com e clique em "Sign up".
2. **Preencha o formulário de cadastro**: Insira as informações solicitadas, incluindo endereço de e-mail, nome completo, país e a finalidade principal para usar o Heroku.
3. **Verificação de E-mail**: Após se cadastrar, verifique seu e-mail para ativar sua conta através de um link enviado pela Heroku.

Passo a Passo para Criação de Projeto e Base de Dados

1. Criar um Projeto no Heroku

Depois de logar na sua conta Heroku, siga estes passos para criar um novo projeto:

1. **Dashboard**: No painel de controle, clique em "New" e selecione "Create new app".
2. **Nome do Aplicativo**: Escolha um nome para seu aplicativo. Este nome precisa ser único em toda a plataforma Heroku.
3. **Escolha da Região**: Selecione a região geográfica mais próxima para hospedar seu aplicativo.
4. **Criação do Aplicativo**: Clique em "Create app".

2. Provisionar uma Base de Dados PostgreSQL no Projeto Heroku

Para adicionar um banco de dados PostgreSQL ao seu projeto:

1. **Acesse o Dashboard do Aplicativo**: Vá para o painel de controle do seu aplicativo no Heroku.
2. **Adicionar Add-ons**: Clique em "Resources" e no campo de busca sob "Add-ons", digite e selecione "Heroku Postgres".
3. **Escolha do Plano**: Selecione um plano adequado. Para testes, você pode começar com o plano gratuito.
4. **Provisionamento**: Clique em "Provision" para adicionar o PostgreSQL ao seu projeto.

3. Conectar à Base de Dados PostgreSQL do Projeto Heroku Usando o pgAdmin

Depois de provisionar sua base de dados:

1. **Obtenha as Credenciais de Conexão**:
   - Acesse "Settings" no dashboard do Heroku.
   - Clique em "Reveal Config Vars".
   - Localize a variável `DATABASE_URL` que contém todas as informações necessárias para a conexão.
2. **Configure o pgAdmin**:
   - Abra o pgAdmin e crie uma nova conexão.
   - Use os dados extraídos de `DATABASE_URL` para configurar o host, porta, nome do usuário, senha e banco de dados.
   - Conecte-se ao banco de dados PostgreSQL.

Executando o Script SQL

Com a base de dados configurada e acessível via pgAdmin, agora você deve executar o script SQL do arquivo `create.sql` para criar as tabelas necessárias no banco de dados. Isso pode ser feito facilmente através da interface do pgAdmin:

- **Abra o Query Tool no pgAdmin**.
- **Carregue e execute o arquivo** `create.sql` para configurar sua base de dados conforme definido nas especificações do seu projeto.

6.16 Preparando o Projeto para Implantação no Heroku

A implantação de um projeto Spring Boot no Heroku requer algumas configurações prévias, incluindo a instalação do Heroku CLI, configuração do projeto para trabalhar com o Git e ajustes nos arquivos de propriedades do Spring Boot para conectar-se ao banco de dados PostgreSQL no Heroku.

Instalando o Heroku CLI

O Heroku Command Line Interface (CLI) é uma ferramenta essencial para criar e gerenciar aplicações no Heroku. Vamos começar com a instalação do Heroku CLI no Windows:

1. **Download**: Visite o site oficial do Heroku https://devcenter.heroku.com/articles/heroku-cli e baixe o instalador para Windows.

2. **Instalação**: Execute o arquivo baixado e siga as instruções de instalação. Certifique-se de incluir o Heroku CLI no PATH do sistema.
3. **Verificação da Instalação**: Abra o terminal (cmd ou PowerShell) e digite `heroku --version` para verificar se o CLI foi instalado corretamente.

Associando o Projeto Spring Boot do Git Local com o Projeto no Heroku

Após a instalação do CLI, associe seu projeto Spring Boot local, que já deve estar versionado com Git, ao projeto no Heroku:

1. **Login no Heroku CLI**: Abra o terminal e execute `heroku login`. Siga as instruções para logar no Heroku através do navegador.
2. **Criação ou Acesso ao Projeto no Heroku**:
   - Se você já criou um projeto no Heroku (via web), navegue até o diretório do seu projeto local.
   - Conecte seu repositório local ao Heroku usando `heroku git:remote -a nome-do-seu-app`.
3. **Criação de Novo Projeto via CLI**:
   - Se preferir criar um novo projeto diretamente via CLI, use `heroku create nome-do-seu-app`.
   - Isso criará um novo projeto e automaticamente adicionará o remote `heroku` ao seu repositório Git.

Configurando o arquivo `application-prod.properties`

Para que o Spring Boot se conecte ao PostgreSQL no Heroku, é necessário configurar as propriedades de produção:

1. **Criação do Arquivo de Propriedades**:
   - Crie um arquivo chamado `application-prod.properties` na pasta `src/main/resources` do seu projeto Spring Boot.
2. **Configuração do DataSource**:
   - Adicione a seguinte linha ao arquivo: `spring.datasource.url=${DATABASE_URL}`.
   - Esta configuração extrai a URL do banco de dados diretamente da variável de ambiente `DATABASE_URL` configurada automaticamente pelo Heroku.

Commit das Alterações no Repositório Git

Com as configurações realizadas, faça um commit das alterações para preparar o projeto para implantação:

1. **Adicione as Alterações ao Git**:
   - Use `git add .` para adicionar todas as alterações recentes ao índice do Git.
2. **Commit das Alterações**:
   - Execute `git commit -m "Preparar projeto para implantação no Heroku"` para salvar as alterações no seu repositório local.

6.17 Implantando a aplicação

Configurar variáveis de ambiente no Heroku

Antes de enviar a aplicação para o Heroku, precisamos primeiro configurar as variáveis de ambiente do projeto Spring Boot dentro do ambiente de execução do projeto Heroku.

Para isto, acesse a aba `Settings` do projeto Heroku e, dentro da seção "Config Vars", configure cada uma das variáveis de ambiente especificadas no arquivo `application.properties` do projeto Spring Boot, bem como alguma eventual variável de ambiente específica do perfil `prod` que esteja especificada no arquivo `application-prod.properties`. Vale ressaltar que, dentre as variáveis de ambiente que você for configurar na seção "Config Vars" do seu projeto Heroku, o valor da variável de ambiente `APP_PROFILE` deve ser `prod`, para que o projeto no Heroku execute no perfil `prod`.

Realizar o deploy no Heroku

Agora que o projeto Spring Boot local está devidamente preparado, bem como o projeto Heroku, podemos enviar nossa aplicação Spring Boot para implantação no Heroku, usando o sistema Git.

Vale lembrar que nós já associamos nosso projeto Spring Boot local com o projeto remoto no Heroku usando o Heroku CLI. Você pode inclusive conferir essa associação executando o seguinte comando o Git:

```
git remote -v
```

Se tudo estiver correto, você verá duas associações: o *remote* `origin` apontando para o repositório remoto do Github, e o *remote* `heroku` apontando para o repositório do Heroku.

Basta agora executar o seguinte comando para enviar o projeto para o Heroku:

```
git push heroku main
```

Observação sobre monorepositório: se seu projeto Spring Boot estiver armazenado em um monorepositório Git, ou seja, se o projeto Spring Boot for uma subpasta em um repositório com vários projetos (por exemplo: no mesmo repositório temos projetos de backend e frontend nas subpastas `backend` e `frontend` respectivamente), você deverá executar o seguinte comando para enviar o projeto para o Heroku, especificando a subpasta:

```
git subtree push --prefix backend heroku main
```

Se a implantação no Heroku estiver funcionado corretamente, você verá um log de implantação no terminal, finalizando com as informações de "BUILD SUCCESS".

6.18 Testando a aplicação no Heroku

Agora que a aplicação foi implantada como um projeto Heroku, precisamos testar a aplicação.

Abra o painel do Heroku e acesse sua aplicação. No *dashboard* da aplicação, haverá um botão "Open app" no canto da tela. Ao clicar nesse botão, a aplicação Spring Boot no Heroku será aberta em uma nova aba do navegador. Se a aplicação Spring Boot estiver corretamente implantada, deverá aparecer a mensagem padrão de erro da aplicação Spring Boot "Whitelabel Error Page" na tela do navegador. Observe a URL da aplicação Spring Boot na barra de navegação do navegador.

Uma vez validado no navegador que a aplicação está executando no Heroku, você pode pegar a URL da aplicação que aparece na barra de navegação do navegador, e usar esse valor de URL na variável de ambiente `host` do seu ambiente do Postman.

Depois de atualizar o valor da variável `host`, faça o teste de cada um dos endpoints para conferir se a aplicação Heroku está respondendo apropriadamente.

Parabéns! Agora você está com seu projeto backend implantado na nuvem, com a API respondendo às requisições.