Análise e Desenvolvimento de Sistemas
Programação Java
Profª. Ana Luiza Scharf
ana.scharf@ifsc.edu.br

Laboratório: Exemplo 04

Introdução

Grande parte desse texto abaixo é de autoria do Professor Emerson Ribeiro de Mello. Link do repositório Link aqui

Como criar o projeto Java

O Spring Boot permite a criação simplificada de aplicações isoladas, ideais durante a etapa de desenvolvimento, bem como para aplicações de produção baseadas no framework Spring.

Com o Spring Boot não é necessário fazer qualquer configuração em arquivos XML, algo típico com JPA, pois algumas configurações ficariam no arquivo *persistence.xml*. No Spring Boot, toda configuração pode ser feita diretamente no código Java e por meio de arquivos de propriedades (properties file) para configurações de conexão com o banco de dados, entre outras.

Para cada um dos exemplos disponíveis neste repositório foi usado o Spring Initializr para criar o esqueleto do projeto. Se deseja criar um projeto como foi criado aqui, siga os passos abaixo:

- Gerar o projeto em https://start.spring.io/
- Configurações:
  - Project: *gradle - groovy*
  - Language: *Java*
  - Spring Boot: *3.5.3*
  - Project metadata:
    - group: *ads.bcd*
    - packaging: *jar*
    - java: *17* (Obs: depende da versão do java do instalada)
  - Dependências:
    - Spring Data JPA
    - MySQL Driver
    - Spring Boot DevTools
- Os demais campos e deixa em branco
- Baixe o arquivo *.ZIP* contendo o projeto Gradle, descompacte-o em uma pasta (de preferência com o nome *Exemplo0X*), e abra essa pasta com o Visual Studio Code ou IntelliJ.

O Spring Boot DevTools inclui um conjunto de ferramentas para tornar mais agradável a experiência de desenvolvimento. De forma resumida, ele irá reiniciar automaticamente a aplicação sempre que notar alguma alteração nos arquivos contidos no classpath. Se não desejar tal comportamento, você pode remover o Spring Boot DevTools da lista de dependências no arquivo *build.gradle*.

Vamos seguir o exemplo disponível no sigaa com todos os arquivos.

Servidor MySQL

Para executar esse exemplo, é necessário que tenha um servidor MySQL disponível. Você pode subir um rapidamente dentro de um contêiner com o Docker. **Dica: no Windows, o Docker precisa estar rodando — ou seja, é necessário abrir a ferramenta antes de usar.** .Basta executar o comando abaixo:

```
docker run -d --rm -p 3306:3306 -e MYSQL_ROOT_PASSWORD=senhaRoot \
-e MYSQL_DATABASE=bcd -e MYSQL_USER=aluno -e MYSQL_PASSWORD=aluno \
-e MYSQL_ROOT_HOST='%' --name meumysql mysql/mysql-server:latest
```

Cabe lembrar que sempre que o contêiner for parado, ele será excluído (opção *–rm*) e todos os dados serão perdidos. Se quiser que os dados continuem mesmo depois da parada e exclusão do contêiner, passe o parâmetro *-v $(pwd)/db_data:/var/lib/mysql*, que fará o mapeamento do diretório usado pelo MySQL no contêiner para um diretório no computador hospedeiro.

Configuração do Spring para conexão com o banco de dados MySQL

O projeto criado terá o arquivo *src/main/resources/application.properties*, onde são colocadas informações de configuração da aplicação, incluindo os dados de conexão com o banco de dados MySQL.

Edite o arquivo e configure as seguintes propriedades:

```
spring.datasource.url=jdbc:mysql://localhost:3306/bcd
spring.datasource.username=aluno
spring.datasource.password=aluno
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create
```

ANa configuração do *application.properties* do projeto, utilizamos a seguinte propriedade:

```
spring.jpa.hibernate.ddl-auto=create
```

Esse parâmetro informa ao Hibernate que ele deve **criar automaticamente todas as tabelas do banco de dados** com base nas entidades JPA a cada vez que a aplicação for iniciada. No modo `create`, o banco de dados é **zerado e recriado**, ou seja, todas as tabelas existentes são apagadas e criadas novamente. Isso é útil durante o desenvolvimento e testes iniciais, pois garante que a estrutura do banco esteja sempre sincronizada com o modelo de dados da aplicação.

Atenção: essa configuração **não deve ser usada em produção**, pois apaga os dados existentes.

Dependência `net.datafaker:datafaker:2.0.2`

Nesta implementação, utilizamos a biblioteca `Datafaker` na versão 2.0.2, adicionada no arquivo de build do Gradle com a seguinte linha:

```
implementation 'net.datafaker:datafaker:2.0.2'
```

A biblioteca `Datafaker` é uma ferramenta poderosa para gerar dados falsos realistas, como nomes, endereços, números de telefone, e muito mais. Isso é especialmente útil para popular bancos de dados de teste, criar dados de exemplo para desenvolvimento e simulações, garantindo variedade e realismo nos dados gerados automaticamente.

Uso típico: ajuda a criar cenários de testes robustos sem precisar inserir dados manualmente.

Enumerações e Conversores no Projeto

Para representar conjuntos fixos de constantes relacionadas ao domínio da aplicação, utilizamos **enums** em Java. No projeto, as enums são importantes para garantir a integridade dos dados e facilitar o uso e manutenção do código.

Enumeração `Idiomas`

A enumeração `Idiomas` representa os idiomas suportados no sistema, com os valores:

- `PT` — Português
- `EN` — Inglês
- `ES` — Espanhol

Essa enum é simples e não possui atributos adicionais, servindo como lista restrita de idiomas possíveis.

Enumeração `Situacao`

A enumeração `Situacao` é mais elaborada e contém dois atributos: um código inteiro e uma descrição textual. As situações possíveis são:

- `ANALISE` (código 1, nome “Em análise”)
- `APROVADO` (código 2, nome “Aprovado”)
- `TRANSITO` (código 3, nome “Em trânsito”)
- `ENTREGUE` (código 4, nome “Entregue”)

Além dos atributos, a enum `Situacao` possui um método estático `of(int codSituacao)` que permite obter a enum correspondente a partir de seu código inteiro. Isso facilita a conversão entre dados armazenados (como no banco) e a enumeração usada na aplicação.

O método `toString()` foi sobrescrito para retornar o nome descritivo, melhorando a legibilidade quando o objeto for convertido para texto.

Conversor JPA: `SituacaoConverter`

Para que o JPA saiba como persistir a enum `Situacao` no banco de dados, foi implementado um conversor que mapeia a enum para seu código inteiro e vice-versa.

O `SituacaoConverter` implementa a interface `AttributeConverter<Situacao, Integer>` e é anotado com `@Converter(autoApply = true)`, o que permite que ele seja aplicado automaticamente a todos os atributos do tipo `Situacao` nas entidades.

Esse conversor garante que no banco seja armazenado o código inteiro da situação, mantendo o banco mais simples e eficiente, enquanto na aplicação é possível trabalhar com a enum, com toda sua segurança e clareza.

Benefícios do uso de Enums e Conversores

- **Segurança de tipo**: evita valores inválidos para atributos restritos.
- **Clareza**: melhora a legibilidade e a manutenção do código.
- **Facilidade na persistência**: o conversor simplifica o armazenamento e a recuperação dos dados no banco, fazendo a conversão automática entre código e enum.

Dessa forma, as enums e seus conversores ajudam a garantir a consistência dos dados e a organização do código, seguindo boas práticas no desenvolvimento de sistemas Java com JPA.

Entidades e seus Relacionamentos

1. Livro

A entidade **Livro** representa um título genérico, como "Dom Casmurro", independentemente da edição.

- Mapeia a tabela `livro`.
- Contém os atributos `idLivro`, `titulo`, `descricao` (como `LONGTEXT`) e `idioma` (como `VARCHAR` via enum).
- Relaciona-se com **Autor** em uma relação **N:N** usando a tabela intermediária `livro_autores`.
- Relaciona-se com **Edicao** em uma relação **1:N**, permitindo que um livro possua diversas edições.

2. Autor

Autor representa os escritores dos livros.

- Mapeia a tabela `autor`.
- Contém `idAutor`, `nome` e `sobrenome`.
- Participa de uma relação **N:N** com `Livro`, sendo a navegação bidirecional.

3. Edicao

A entidade **Edicao** detalha uma versão específica de um livro (ex: 2ª edição).

- Mapeia a tabela `edicao`.
- Usa chave composta (`numero`, `idLivro`), representada pela classe `EdicaoId`.
- Contém dados como `isbn13`, `dataPublicacao`, `preco` e `totalDePaginas`.
- A dimensão física (largura, altura, profundidade) é encapsulada na classe `Dimensao`, marcada como `@Embeddable`.
- Tem relações:
  - **N:1** com **Livro**.
  - **N:1** com **Editora**.
  - **1:N** com **ItemDoPedido**.

4. Editora

A entidade **Editora** representa a empresa responsável por publicar as edições.

- Mapeia a tabela `editora`.
- Possui `idEditora`, `nome` e `cidade`.
- Relaciona-se com **Edicao** em uma relação **1:N**, ou seja, uma editora pode publicar várias edições.

5. Cliente

Cliente representa o consumidor que realiza pedidos.

- Mapeia a tabela `cliente`.
- Contém dados pessoais: `idCliente`, `cpf`, `nome`, `endereco`, `dataNascimento` e `email`.
- O campo `email` tem validação manual com expressão regular.
- Tem relacionamento **1:N** com **Pedido**.

6. Pedido

Pedido representa a compra realizada por um cliente.

- Mapeia a tabela `pedido`.
- Tem os atributos: `idPedido`, `data`, `situacao` e referência ao `cliente`.
- Relacionamento **N:1** com **Cliente** e **1:N** com **ItemDoPedido**.

7. ItemDoPedido

Essa classe representa os itens de um pedido (livros adquiridos com quantidade e preço no momento da compra).

- Mapeia a tabela `item_do_pedido`.
- É uma entidade intermediária entre **Pedido** e **Edicao**.
- Possui uma chave composta (pedido + edição), definida na classe `ItemDoPedidoId`.
- Armazena `preco` e `quantidade`, o que caracteriza um relacionamento **muitos-para-muitos com atributos**.

8. Classes de Suporte

- **Dimensao**:
  - É uma classe embutida em `Edicao` (`@Embeddable`).
  - Seus campos (`largura`, `altura`, `profundidade`) são incorporados na tabela `edicao`.
- **EdicaoId**:
  - Representa a chave composta da entidade `Edicao`.
- **ItemDoPedidoId**:
  - Representa a chave composta da entidade `ItemDoPedido`, contendo um `pedido` e uma instância de `EdicaoId`.

Repositórios com Spring Data JPA

Nesta aplicação, o acesso aos dados é feito por meio do uso da biblioteca **Spring Data JPA**, que permite abstrair a implementação das operações de banco de dados usando interfaces.

Cada entidade do modelo possui um **repositório** correspondente que estende a interface `CrudRepository`, fornecendo automaticamente os métodos básicos como: `save`, `findById`, `delete`, `findAll`, entre outros.

1. CrudRepository e Generics

A interface `CrudRepository<T, ID>` é parametrizada com dois tipos:

- **T** — o tipo da entidade (ex: `Livro`, `Cliente`, etc.);
- **ID** — o tipo da chave primária da entidade (ex: `Integer` ou classe composta como `EdicaoId`).

2. Repositórios Criados

- **AutorRepository**:
  - Interface que gerencia a entidade `Autor`.
  - Usa `Integer` como chave primária.
- **ClienteRepository**:
  - Responsável pelo acesso à entidade `Cliente`.
  - Permite encontrar, salvar ou deletar clientes.
- **EditoraRepository**:
  - Interface para acesso aos dados da entidade `Editora`.
- **LivroRepository**:
  - Permite manipular registros da entidade `Livro`.

- **PedidoRepository**:
  - Gerencia os pedidos cadastrados.
  - É possível buscar pedidos por ID ou listar todos.
- **EdicaoRepository**:
  - Repositório da entidade `Edicao`, que possui chave primária composta.
  - Utiliza a classe `EdicaoId` como identificador.
- **ItemDoPedidoRepository**:
  - Gerencia a entidade associativa `ItemDoPedido`, também com chave composta.
  - Utiliza a classe `ItemDoPedidoId`, que contém referências ao pedido e à edição.

3. Benefícios do uso de repositórios

- Elimina a necessidade de escrever SQL para operações básicas.
- Permite extensões com métodos de consulta personalizados (ex: `findByNome`).
- Facilita testes e manutenção ao seguir o princípio de separação de responsabilidades.

Execução da Aplicação e População de Dados

A pasta raiz da aplicação contém as duas classes principais responsáveis por inicializar e executar o sistema: `LivrariaApplication` e `LivrariaRunner`.

1. Classe `LivrariaApplication`

Esta é a classe principal da aplicação Spring Boot. Ela está anotada com `@SpringBootApplication`, o que a torna o ponto de entrada da aplicação. Ao ser executada, ela dispara o mecanismo de auto-configuração e escaneamento de componentes da aplicação.

- O método `main` chama `SpringApplication.run()`, que inicializa todo o contexto da aplicação.
- A anotação `@Slf4j` permite o uso do logger para registrar informações no console, como mensagens de inicialização ou término.

2. Classe `LivrariaRunner`

Essa classe está anotada com `@Component`, sendo automaticamente gerenciada pelo Spring. Por implementar a interface `CommandLineRunner`, ela executa seu método `run()` assim que a aplicação é iniciada.

Principais responsabilidades dessa classe:

- **Povoar o banco de dados com dados fictícios:**
  - Usa a biblioteca `Faker` para gerar nomes, datas, descrições, entre outros dados aleatórios.
  - Cria entidades como `Cliente`, `Autor`, `Editora`, `Livro`, `Edicao`, `Pedido` e `ItemDoPedido`.
  - Salva essas entidades por meio dos repositórios JPA.
- **Listar dados de todas as tabelas:**
  - Chama o método `findAll()` de cada repositório e imprime no console todos os registros.
- **Alterar a situação de um pedido:**
  - Busca um pedido específico (ID = 1).
  - Altera o status do pedido de `ANALISE` para `APROVADO`.
  - Atualiza o registro no banco com o método `save()`.

3. Conexão com o Diagrama Entidade-Relacionamento

O código reflete diretamente o modelo representado no diagrama relacional:

- O relacionamento entre livros, autores e edições é respeitado.
- As edições estão associadas a uma editora e ao seu respectivo livro.
- Os pedidos estão associados a clientes e contêm itens que representam edições específicas.
- As chaves compostas, como em `EdicaoId` e `ItemDoPedidoId`, são utilizadas corretamente para modelar relacionamentos complexos.

4. Considerações Finais

O uso de `Spring Boot`, `Spring Data JPA`, `Lombok` e `Faker` permite criar uma aplicação robusta com persistência automática de dados, testes realistas e código limpo. O sistema é modular, seguindo boas práticas de desenvolvimento orientado a objetos e arquitetura em camadas.

Testes Automatizados com Spring Boot

Para garantir que a aplicação funcione corretamente em diferentes cenários, é possível criar testes automatizados com a ajuda do framework **JUnit 5** em conjunto com o **Spring Boot**.

1. Classe `PopulatingDatabaseTest`

A classe `PopulatingDatabaseTest` é uma classe de teste da aplicação. Ela está anotada com:

- **@SpringBootTest**: indica que os testes serão executados dentro do contexto da aplicação Spring Boot.
- **@Autowired**: injeta os repositórios para uso direto nos métodos de teste.
- **@Test**: marca os métodos que devem ser executados como testes.
- **@Order(n)**: define a ordem de execução dos testes (útil quando há dependências entre eles).

Executando o Projeto

Conectar o IntelliJ ao MySQL via Docker e executar o script *jobs-schema.sqllab01-mysql-dml-ddl.sql*

1. No IntelliJ, abra a aba **Database** (geralmente na lateral direita).
2. Clique no botão **+** e escolha *Data Source > MySQL*.
3. Preencha os dados da conexão:
   - **Host**: *127.0.0.1*
   - **Port**: *3306*
   - **User**: *aluno* (ou seu usuário do MySQL)
   - **Password**: *aluno*
   - **Database**:

   Observação: as variáveis *user* e *password* devem ser as mesmas do arquivo *database.properties*.
4. Clique em **Test Connection** para verificar se está tudo ok.

Abra o IDE e execute a classe *PopulatingDatabaseTest.java* e rode.