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

Laboratório: Exemplo 03 – Java com SQLite usando JDBC

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=none
```

A linha *spring.jpa.hibernate.ddl-auto=none* é usada para impedir que o Hibernate altere a estrutura do banco de dados automaticamente. Por padrão, o Spring Boot tenta criar, atualizar ou validar as tabelas com base nas entidades Java, o que poderia modificar a estrutura do banco existente.

No entanto, neste projeto estamos utilizando um banco de dados já existente. Por isso, definimos o valor *none*, o que desativa qualquer ação automática de geração ou modificação do esquema. Dessa forma, o Hibernate apenas acessa as tabelas conforme foram mapeadas nas entidades, sem tentar recriá-las ou alterá-la

Inicialmente, o projeto utilizava a configuração:

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

Essa opção faz com que o Hibernate tente automaticamente atualizar o esquema do banco de dados de acordo com as entidades Java a cada inicialização da aplicação. No entanto, esse comportamento pode causar alterações indesejadas ou perda de controle sobre o banco, especialmente em ambientes de produção.

Para evitar esse risco e garantir que a estrutura do banco seja controlada exclusivamente por scripts SQL explícitos, alteramos a configuração para:

```
spring.jpa.hibernate.ddl-auto=none
```

Com essa mudança, o Hibernate não realiza nenhuma ação automática sobre o esquema do banco de dados. Isso garante maior segurança e previsibilidade na manutenção da base de dados.

Modelo de Dados - Entidades Java com JPA

Neste sistema foram modeladas as seguintes entidades: **Course**, **Department**, **Employee** e **JobHistory**, cada uma mapeada com anotações JPA para persistência em banco de dados relacional via Spring Boot.

Entidade *Course*

- Representa um curso oferecido pela instituição.
- Utiliza a anotação *@Entity*.
- Campos principais:

  - *courseno* – identificador do curso (*@Id*).
  - *cname*, *cdate* – nome e data do curso.
- Relacionamento:
  - *@ManyToMany(mappedBy = "courses")* com a entidade *Employee*, indicando que um curso pode ser ministrado a vários funcionários.

Entidade *Department*

- Representa um departamento da organização.
- Campos principais:
  - *depno* – identificador do departamento.
  - *dname*, *location*, *head* – nome, local e responsável.
- Relacionamento:
  - *@OneToMany(mappedBy = "department")* com *Employee*.

Entidade *Employee*

- Representa um funcionário.
- Campos principais:
  - *empno*, *surname*, *forenames*, *dob*, *address*, *telno*.
- Relacionamentos:
  - *@ManyToOne* com *Department*, através de *@JoinColumn(name = "depno")*.
  - *@OneToMany(mappedBy = "employee")* com *JobHistory*.
  - *@ManyToMany* com *Course*, utilizando *@JoinTable(name = "empcourse")* para mapear a tabela intermediária.

Entidade *JobHistory*

- Representa o histórico de posições ocupadas por um funcionário.
- Campos principais:
  - *position*, *startdate*, *enddate*, *salary*.
- Relacionamento:
  - *@ManyToOne* com *Employee*, via *@JoinColumn(name = "empno")*.

Exemplos adicionais de uso de *@JoinColumn* e *@JoinTable*

- **JoinColumn simples** (FK para uma entidade pai):

```
@ManyToOne
@JoinColumn(name = "category_id")
private Category category;
```

- **JoinTable com colunas adicionais (caso mais complexo)**:

```
@ManyToMany
@JoinTable(
  name = "student_course",
  joinColumns = @JoinColumn(name = "student_id"),
  inverseJoinColumns = @JoinColumn(name = "course_id")
)
private Set<Course> courses;
```

- **Relacionamento um-para-um**:

```
@OneToOne
@JoinColumn(name = "passport_id")
private Passport passport;
```

Roteiro para modelar uma entidade com JPA

1. Anotar a classe com *@Entity*.
2. Definir uma chave primária com *@Id*.
3. Utilizar o construtor padrão (sem argumentos), exigido pelo JPA.
4. Usar *Lombok* para gerar getters, setters, e construtores.
5. Anotar os relacionamentos:
    - *@ManyToOne* com *@JoinColumn*
    - *@OneToMany(mappedBy = "...")*
    - *@ManyToMany* com *@JoinTable*
6. Declarar as coleções com tipos como *Set* ou *List*, inicializadas.
7. Excluir campos sensíveis do *@ToString* para evitar loops infinitos.
8. Se necessário, mapear tabelas intermediárias com colunas extras usando uma entidade auxiliar.

Camada de Repositórios (Repository)

A camada **repository** é responsável por realizar a comunicação entre a aplicação Java e o banco de dados, utilizando a abstração oferecida pelo Spring Data JPA. Cada interface estende a *CrudRepository*, que fornece métodos básicos como *save*, *findById*, *findAll*, *deleteById*, entre outros.

Interface *CourseRepository*

- Estende *CrudRepository<Course, Integer>*.
- Define uma consulta personalizada com a anotação *@Query*, utilizando JPQL.
- Método destacado:

```
@Query("SELECT c FROM Course c WHERE YEAR(c.cdate) = :ano")
List<Course> findByCursosRealizadosEmUmAno(@Param("ano") int ano);
```

- Este método retorna todos os cursos realizados em um determinado ano, passado como parâmetro.
- A utilização de *:ano* representa um parâmetro nomeado.

Interface *DepartmentRepository*

- Interface simples que estende *CrudRepository<Department, Integer>*.
- Não define métodos adicionais, utilizando os métodos padrão do Spring Data JPA.

Interface *EmployeeRepository*

- Estende *CrudRepository<Employee, Integer>*.
- Define dois métodos com consultas personalizadas:

1. Funcionários que fazem aniversário em determinado mês:

```
@Query("SELECT e FROM Employee e WHERE MONTH(e.dob) = ?1")
List<Employee> findByAniversariantesNoMes(int mes);
```

2. Histórico de cargos de um funcionário (ordenado pela data de início, decrescente):

```
@Query("SELECT j FROM JobHistory j WHERE j.employee = ?1
        ORDER BY j.startdate DESC")
List<JobHistory> findByDeCargosNaEmpresa(Employee employee);
```

Roteiro para criação de repositórios com Spring Data

1. Criar uma interface que estende *CrudRepository<T, ID>*, onde:
   - *T* é a classe da entidade;
   - *ID* é o tipo da chave primária.
2. Utilizar os métodos padrão do Spring:
   - *save*, *findById*, *findAll*, *deleteById*, etc.
3. Para consultas personalizadas:
   - Usar a anotação *@Query* com JPQL ou SQL nativo;
   - Definir parâmetros posicionais (*?1*) ou nomeados (*:param*).

Observações sobre *@Query*

- Quando utilizar JPQL, referencie entidades e atributos Java, não os nomes das tabelas:

```
@Query("SELECT e FROM Employee e WHERE e.surname = :nome")
```

- Para SQL nativo, utilize a opção *nativeQuery = true*:

```
@Query(value = "SELECT * FROM employee WHERE MONTH(dob) = ?1",
       nativeQuery = true)
```

Classe Principal da Aplicação - *ExemploJpaApplication*

A classe *ExemploJpaApplication* é o ponto de entrada da aplicação. Ela está anotada com *@SpringBootApplication*, o que permite que o Spring Boot inicialize automaticamente todos os componentes, configurações e beans da aplicação.

Principais componentes utilizados

- *@SpringBootApplication* – Anotação que combina *@Configuration*, *@EnableAutoConfiguration* e *@ComponentScan*.
- *Logger* – Usado para registrar informações de execução, depuração e tratamento de erros.
- *@Autowired* – Injeta automaticamente as dependências dos repositórios.
- *@Bean* com *CommandLineRunner* – Permite executar um trecho de código automaticamente assim que a aplicação for iniciada.

Lógica de execução no método *demo()*

O método *demo()* é um *CommandLineRunner* que contém chamadas de teste às interfaces de repositório, com o objetivo de:

1. Listar todos os cursos disponíveis.
2. Listar os funcionários que fazem aniversário no mês de março.
3. Listar os cursos realizados no ano de 1989.
4. Buscar um funcionário com *empno = 1* e, se encontrado, listar seu histórico de cargos assumidos na empresa.

Trecho de código com destaque

```
@Bean
public CommandLineRunner demo() {
  return (args) -> {
    log.info("Iniciando aplicação");

    // Exemplo: listando aniversariantes
    employeeRepository.findByAniversariantesNoMes(3)
                      .forEach(System.out::println);

    // Exemplo: histórico de cargos
    Optional<Employee> buscaEmp = employeeRepository.findById(1);
    if (buscaEmp.isPresent()) {
      Employee jones = buscaEmp.get();
      List<JobHistory> historico = employeeRepository
                                   .findByDeCargosNaEmpresa(jones);
      // Impressão formatada
    }
  };
}
```

Tratamento de exceções

- A execução do *CommandLineRunner* está encapsulada em um bloco *try-catch*, que garante que falhas durante o acesso ao banco ou à lógica de aplicação sejam tratadas com *log.error*.

Importância do uso de *CommandLineRunner*

Esta abordagem é útil para:

- Testar funcionalidades logo na inicialização;
- Executar carga de dados ou manipulação inicial;
- Demonstrar consultas e interações com o banco de dados em tempo de execução.

Descrição da Base de Dados

O script SQL apresentado tem como objetivo a criação e o povoamento de uma base de dados relacional voltada para o controle de informações de **empregados**, **departamentos**, **cursos** e **histórico de cargos** em uma organização.

Estrutura das Tabelas

A base de dados é composta pelas seguintes tabelas principais:

- **department**: armazena os dados dos departamentos da empresa, incluindo o nome, localização e o número do empregado que atua como chefe do setor.
- **employee**: guarda as informações pessoais dos funcionários, como sobrenome, prenome, data de nascimento, endereço, telefone e o número do departamento ao qual pertencem.
- **course**: contém os dados sobre os cursos oferecidos, como número identificador (*courseno*), nome e data de realização.
- **empcourse**: representa uma relação muitos-para-muitos entre empregados e cursos, indicando quais funcionários participaram de quais cursos.
- **jobhistory**: armazena o histórico de cargos ocupados por cada empregado, incluindo o nome do cargo, datas de início e fim e salário correspondente.

Relacionamentos

A integridade referencial entre as tabelas é garantida por meio de **chaves estrangeiras**, destacando-se:

- A tabela **employee** possui uma chave estrangeira que referencia a tabela **department**.
- A tabela **jobhistory** possui uma chave estrangeira que referencia **employee**.
- A tabela **empcourse** possui duas chaves estrangeiras: uma para **employee** e outra para **course**.

População de Dados

O script também realiza a inserção de dados simulados em todas as tabelas, permitindo testes e análises. Entre os dados inseridos, destacam-se:

- 10 cursos distintos.
- 5 departamentos organizacionais.
- 32 empregados com informações completas.
- Participações em cursos por diversos funcionários.
- Históricos de cargos detalhados para grande parte dos empregados.

Objetivos

A estrutura da base permite aplicações como:

- Consultas sobre o histórico de cargos e salários dos funcionários.
- Levantamento de cursos realizados por cada empregado.
- Análise da movimentação interna nos departamentos.
- Estudos de progressão de carreira e capacitação de pessoal.

A base pode ser utilizada tanto em contextos educacionais (ensino de SQL, modelagem, normalização) quanto em simulações de sistemas empresariais reais.

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.
5. Clique em **OK** para salvar a conexão.
6. Clique com o botão direito na conexão criada e selecione *Open Console* para abrir o console SQL.
7. Agora, abra o arquivo *jobs-schema.sqllab01-mysql-dml-ddl.sql*
8. Clique no ícone de **seta** (Execute) para rodar o script e criar as tabelas e dados.
9. Aguarde a execução e verifique a mensagem de sucesso.

Abra sua IDE e execute a classe *ExemploJpaApplication.java*, ou então utilize a linha de comando com Gradle:

```
./gradlew bootRun
```