DIM0547 - DESENVOLVIMENTO DE SISTEMAS WEB II

09-12

As regras do jogo

- **Não será aceito uso de IA** no desenvolvimento dos Trabalho ou Provas;
- **Trabalhos copiados** em todo ou em parte de outros estudantes ou da Internet receberão automaticamente nota zero;
- **Códigos que não compilam** serão avaliados, porém com penalidade;
- **Pontos extras** serão adicionados em códigos com uso de boas práticas de programação (**Se necessário para aprovação**);

As regras do jogo

- **Trabalhos entregues fora do prazo não serão considerados;**

- **Atrasos resultam em meia falta;**

As regras do jogo

Faltas justificadas apenas mediante atestado

Qualquer problema, fale comigo, logo, não espero para o final

Avaliação

- Prova escrita;
- Trabalho Prático*;
- Trabalho Prático*;
- Reposição:
  - Prova escrita;
  - Trabalho Prático Correspondente + Algumas coisas;

* Apresentação é obrigatória

* Trabalhos em trio

Aula 13

JPA e Hibernate - parte 01

Java Persistence API (JPA)

O JPA (Java Persistence API) é uma especificação da plataforma Java que define como objetos Java podem ser mapeados para tabelas de banco de dados relacionais.

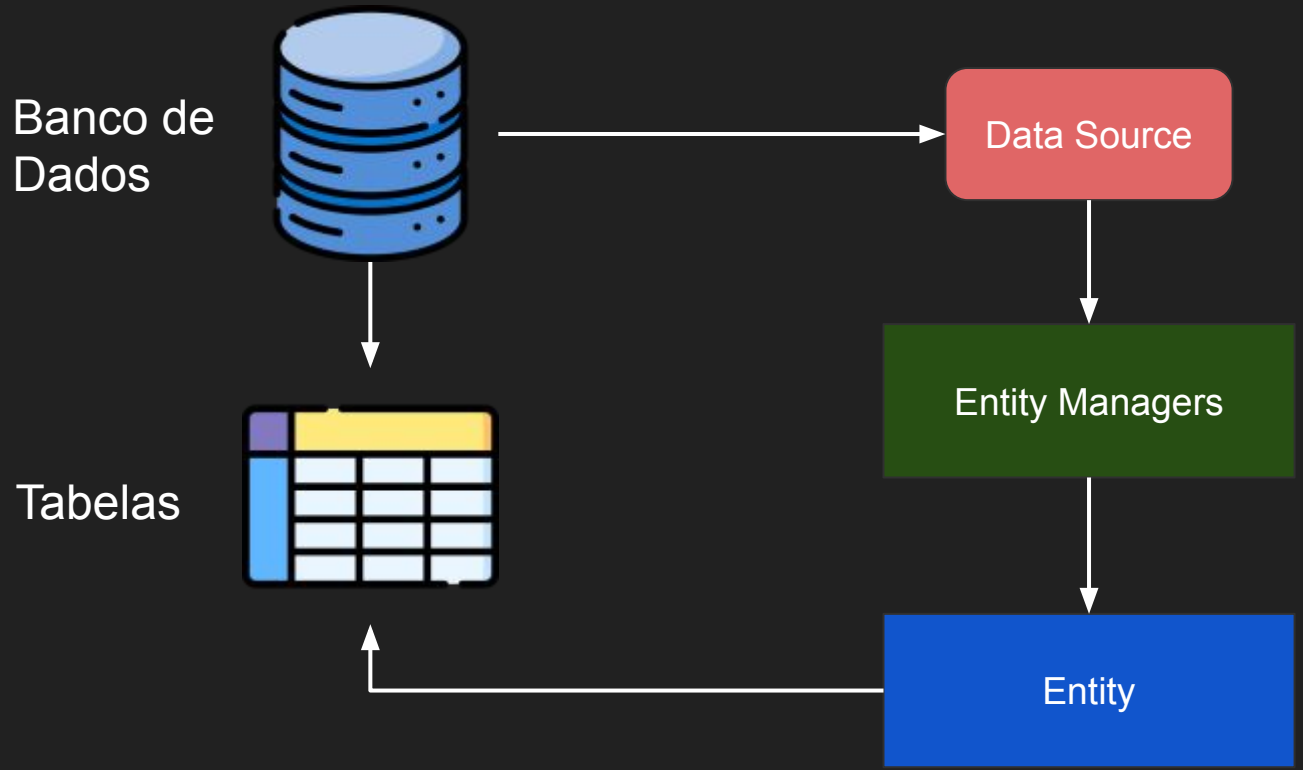
O JPA não é um framework por si só, ele é só uma especificação, um conjunto de regras.

Quem implementa o JPA são frameworks como Hibernate, EclipseLink, OpenJPA, etc

Java Persistence API (JPA)

Ou seja, o JPA diz como deve ser, mas quem implementa é o Hibernate, EclipseLink, OpenJPA

Java Persistence API (JPA)



DataSource

Um DataSource é um objeto que representa a fonte de dados de uma aplicação — normalmente, um banco de dados relacional.

Ele encapsula as informações necessárias para se conectar ao banco

DataSource

Como funciona?

- A aplicação pede uma conexão ao DataSource.
- O DataSource cria (ou pega de um pool) a conexão com o banco.
- A aplicação usa essa conexão para executar SQL.
- Quando termina, devolve a conexão ao DataSource (que pode fechar ou reutilizar).

Entity

Uma Entity (Entidade) é uma classe Java que representa uma tabela no banco de dados.

Permitem trabalhar com Orientação a Objetos sem precisar escrever SQL direto o tempo todo e facilitam operações como CRUD (Create, Read, Update, Delete).

EntityManager

O EntityManager é a interface principal do JPA usada para interagir com o banco de dados.

Ele é o responsável por gerenciar as entidades (objetos anotados com @Entity) e por cuidar do ciclo de vida delas.

Dá pra pensar nele como um "controlador" que sabe:

- Criar, atualizar e deletar registros.
- Consultar dados.
- Controlar quando um objeto Java está sincronizado com o banco.

Spring Data JPA

O Spring Data JPA é um módulo do Spring Framework que facilita o acesso a bancos de dados relacionais usando JPA (Java Persistence API).

Ele abstrai grande parte do trabalho repetitivo de criar DAO (Data Access Objects) e escrever SQL manualmente, permitindo que você trabalhe com repositórios de forma declarativa.

- Facilitar o CRUD (Create, Read, Update, Delete) sem precisar escrever SQL.
- Permitir consultas avançadas usando JPQL ou Query Methods (nomes de métodos que viram queries automaticamente).
- Integrar com o Spring Boot de forma transparente, aproveitando transações, caching, paginação e ordenação.

Requisitos

Colocar a dependência do JPA no pom.xml

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

Requisitos

Colocar a dependência do H2 no pom.xml

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

application.properties

```
// Conexão com o Banco
spring.datasource.url=jdbc:h2:~/test
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Console web do H2
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

application.properties

// Conexão com o Banco

spring.datasource.url=jdbc:h2:~/test → Arquivo criado na raiz do usuário

jdbc:h2:mem:testdb → Banco criado em memória

H2 Database

O H2 Database é um banco de dados relacional (RDBMS) desenvolvido inteiramente em Java.

Ele é open source, leve e pode funcionar de duas formas principais:

- Em memória: os dados ficam só na RAM, desaparecendo quando o processo termina.
- Em arquivo: os dados são armazenados em disco, permanecendo entre execuções

H2 Database

Portabilidade: como é Java, roda em qualquer sistema que tenha a JVM.

Tamanho reduzido: é um banco pequeno, pensado para ser rápido e simples.

SQL padrão: suporta comandos SQL ANSI e até simula dialetos de outros bancos (MySQL, PostgreSQL, Oracle, etc.).

Console web: traz uma interface gráfica acessível via navegador para executar consultas SQL.

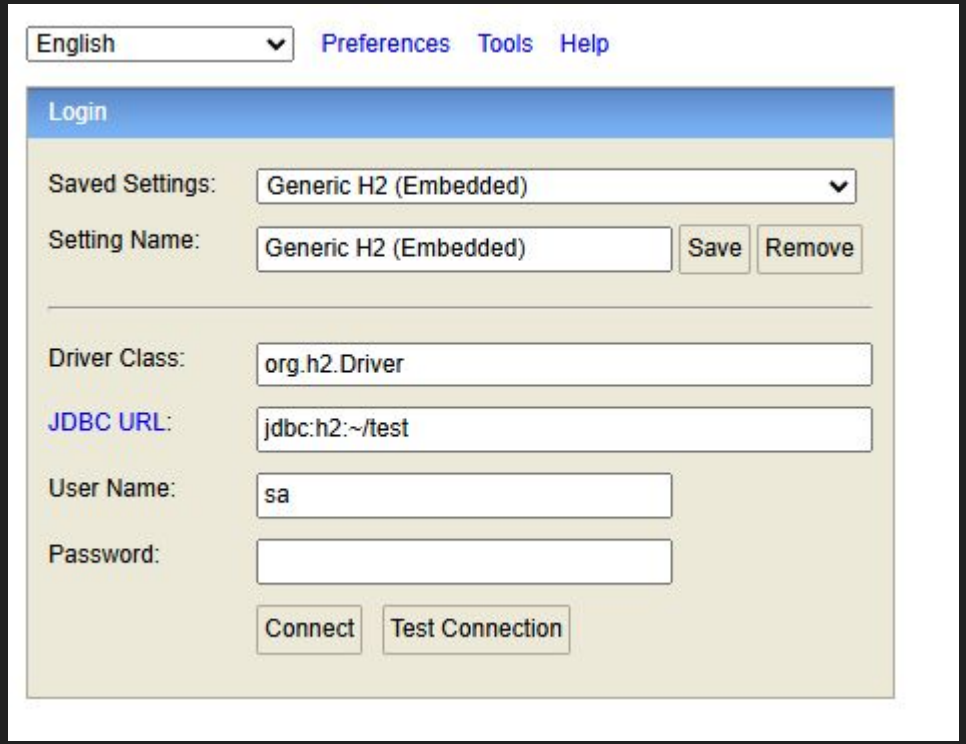
Modo servidor ou embarcado: pode ser usado dentro de uma aplicação (embedded) ou como servidor acessível por rede (Independente da aplicação).

H2 Database - Limitações

- Não é indicado para sistemas de produção que exigem alta concorrência e grandes volumes de dados.
- Sua principal força é a simplicidade, não a robustez para cenários massivos.

Para acessar

http://localhost:<porta>/<path-aplicação>/<path-h2>



jdbc:h2:~/test
INFORMATION_SCHEMA
Users
H2 2.3.232 (2024-08-11)

Run | Run Selected | Auto complete | Clear | SQL statement:

Important Commands

		Displays this Help Page
		Shows the Command History
	Ctrl+Enter	Executes the current SQL statement
	Shift+Enter	Executes the SQL statement defined by the text selection
	Ctrl+Space	Auto complete
		Disconnects from the database

Sample SQL Script

Delete the table if it exists	DROP TABLE IF EXISTS TEST;
Create a new table with ID and NAME columns	CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
Add a new row	INSERT INTO TEST VALUES(1, 'Hello');
Add another row	INSERT INTO TEST VALUES(2, 'World');
Query the table	SELECT * FROM TEST ORDER BY ID;
Change data in a row	UPDATE TEST SET NAME='Hi' WHERE ID=1;
Remove a row	DELETE FROM TEST WHERE ID=2;
Help	HELP ...

Configurando a conexão

O HikariCP é um mecanismo de pool de conexões JDBC com o objetivo de ser extremamente rápido, simples e confiável.

A grande motivação por trás do Hikari é resolver o problema de abrir e fechar conexões diretamente com o banco de dados, algo que é muito custoso em termos de tempo e de processamento. Em vez de criar uma conexão nova a cada requisição, o Hikari mantém um conjunto de conexões já abertas e disponíveis.

Quando a aplicação precisa acessar o banco, ela pega uma dessas conexões, usa o que for necessário e depois devolve ao pool, em vez de fechar. Esse reaproveitamento elimina a sobrecarga de criar conexões repetidamente e faz com que aplicações suportem uma carga muito maior de usuários simultâneos.

@Configuration

A anotação @Configuration no Spring serve para indicar que uma classe é uma classe de configuração — ou seja, uma classe que contém definições de beans que o Spring IoC Container vai gerenciar.

@Value

A anotação @Value é um recurso do Spring que permite injetar valores externos diretamente em variáveis da sua aplicação. Em outras palavras, ela serve como uma ponte entre o código Java e as configurações definidas em arquivos de propriedades, como o application.properties ou o application.yml.

```
@Value("${spring.datasource.url}")

private String springDatasourceUrll;
```

Configuração Hikari

Não é necessário implementar uma configuração de datasource, mas quando for necessário algumas configurações adicionais como mais de uma base é obrigatório criar base de dados separadas.

```java
@Configuration  no usages
public class DataBase {

    @Value("${spring.datasource.url}")  1 usage
    private String springDatasourceUrll;

    @Value("${spring.datasource.driverClassName}")  1 usage
    private String springDatasourceDriverClassName;
    @Value("${spring.datasource.username}")  1 usage
    private String springDatasourceUsername;
    @Value("${spring.datasource.password}")  1 usage
    private String springDatasourcePassword;

    @Bean  no usages
    public DataSource getDataSource() {

        HikariConfig config = new HikariConfig();

        config.setJdbcUrl(springDatasourceUrll);
        config.setDriverClassName(springDatasourceDriverClassName);
        config.setUsername(springDatasourceUsername);
        config.setPassword(springDatasourcePassword);

        config.setMaximumPoolSize(10);
        config.setMinimumIdle(1);
        config.setPoolName("jdbc-h2");

        return new HikariDataSource(config);

    }

}
```

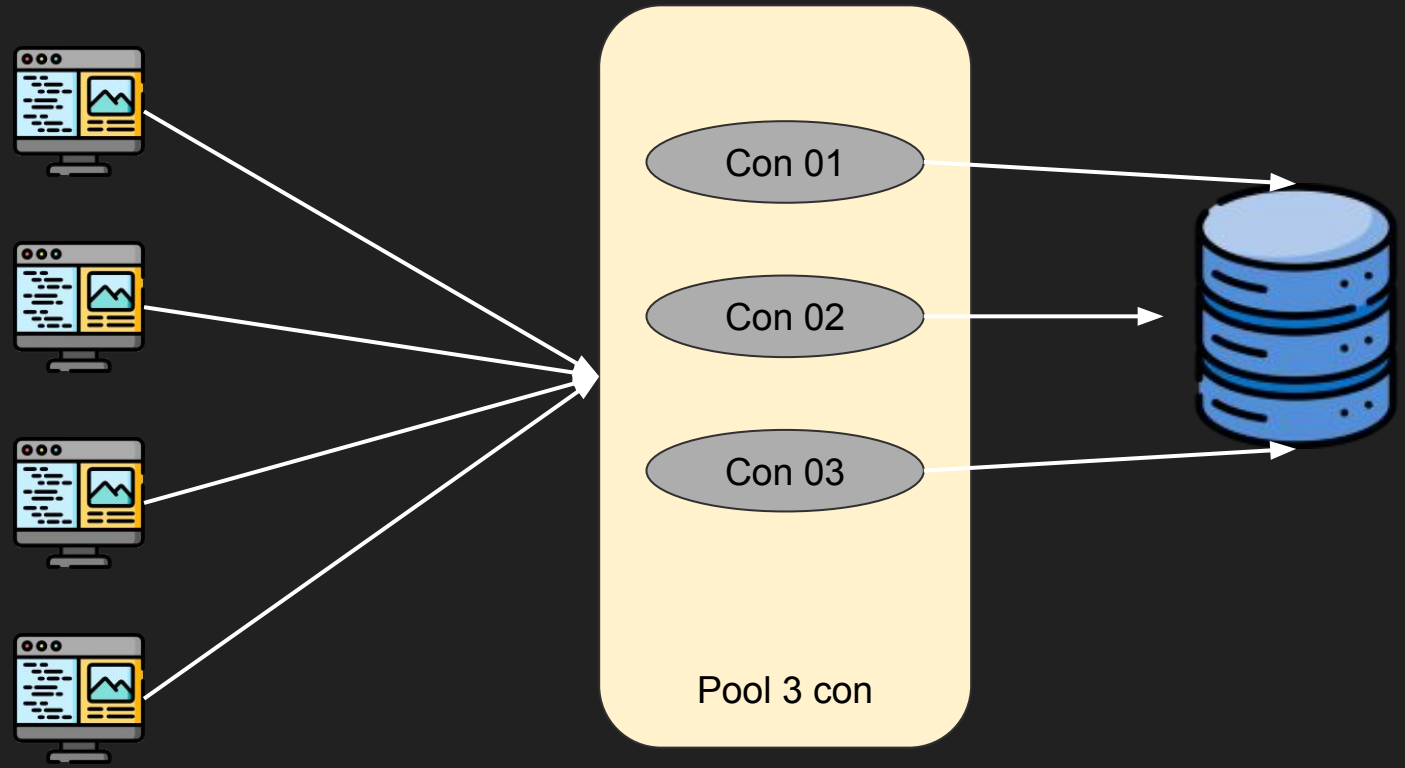
Pool de conexões

Um pool de conexões (connection pool) é um conjunto de conexões pré-abertas com o banco de dados que ficam disponíveis para uso pela aplicação.

Em vez de abrir e fechar uma conexão toda vez que precisa acessar o banco, a aplicação pega uma conexão do pool, usa, e depois devolve para o pool.

O pool gerencia quantas conexões podem existir simultaneamente, evitando sobrecarga no banco.

Pool de conexões



Pool de conexões

Performance:

- Criar uma conexão com o banco é caro (envolve handshake, autenticação, etc.).
- Reusar conexões do pool reduz o tempo de cada operação.

Gerenciamento de recursos:

- Evita que a aplicação abra milhares de conexões simultâneas, o que poderia travar o banco.

Controle de concorrência:

- Permite limitar o número máximo de conexões abertas ao mesmo tempo, garantindo estabilidade.

Pool de conexões - Problemas

Exaustão do pool (Pool Exhaustion): Todas as conexões do pool estão ocupadas e novas requisições precisam esperar ou falham.

Má configuração de tempo de vida da conexão

Idle Timeout muito curto: conexões fecham rápido demais → overhead de abrir novas conexões.

Max Lifetime muito longo: conexões antigas podem ficar inválidas e causar erros aleatórios.

@Entity

Essa anotação marca a classe Java como uma entidade JPA.

Isso significa que o Hibernate (ou outro provedor JPA) vai enxergar a classe como algo que pode ser persistido no banco.

Sem @Entity, a classe é só um POJO qualquer e não será gerenciada pelo contexto de persistência.

@Table

Essa anotação é usada para configurar detalhes da tabela à qual a entidade está associada.

É opcional: se você não usar, o JPA vai assumir que a tabela tem o mesmo nome da classe (Usuario → usuario).

Com @Table, você pode:

- mudar o nome da tabela;
- definir esquema (schema);
- definir restrições únicas (uniqueConstraints);
- criar índices (indexes).

@Id

A anotação @Id no JPA é a mais fundamental quando falamos de mapeamento de entidades. Ela indica qual campo da classe é a chave primária da tabela no banco de dados.

@GeneratedValue

A anotação @GeneratedValue em JPA/Hibernate é usada em conjunto com a anotação @Id para indicar que o valor da chave primária de uma entidade será gerado automaticamente pelo provedor de persistência ou pelo banco de dados.

@GeneratedValue

GenerationType.AUTO: O comportamento é definido pelo provedor JPA (como o Hibernate). Ele escolhe a estratégia mais adequada com base no banco de dados em uso. É a opção padrão, mas pode levar a resultados diferentes dependendo do ambiente.

GenerationType.IDENTITY: Indica que o banco de dados é responsável por gerar o valor da chave primária, geralmente por meio de colunas auto-incremento (AUTO_INCREMENT no MySQL, SERIAL no PostgreSQL). Cada inserção gera automaticamente um novo valor único.

GenerationType.SEQUENCE: Utiliza uma sequência no banco de dados (muito comum em bancos como Oracle ou PostgreSQL). É possível inclusive indicar qual sequência será usada com @SequenceGenerator.

GenerationType.TABLE: Cria uma tabela auxiliar no banco para armazenar e gerenciar os valores de chaves primárias. É menos usado atualmente porque é mais lento, mas era útil para bancos que não tinham suporte nativo a auto-incremento ou sequências.istência ou pelo banco de dados.

GenerationType.UUID: O objetivo ao usar UUID como identificador é gerar IDs globais e não-sequenciais na própria aplicação (ou via provedor

@Column

Você pode definir um nome diferente para a coluna no banco, caso não queira usar o nome do atributo da classe.

Se o campo pode ser null, tamanho, precisão e diversas outras opções.

```
@Column(name = "titulo_completo")
public String titulo;
```

Tipos de Relacionamento

@OneToOne (Um para Um):

- Um objeto A está relacionado com no máximo um objeto B, e vice-versa.

@OneToMany / @ManyToOne (Um para Muitos e Muitos para Um):

- Um para Muitos: Um objeto A pode ter vários objetos B.
- Muitos para Um: Vários objetos B estão associados ao mesmo objeto A.

@ManyToMany (Muitos para Muitos):

- Definição: Um objeto A pode estar relacionado a vários objetos B, e um objeto B pode estar relacionado a vários objetos A.

Tipos de Relacionamento

@OneToOne (Um para Um):

- Um objeto A está relacionado com no máximo um objeto B, e vice-versa.

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

    private String nome;

    @OneToOne
    @JoinColumn(name = "perfil_id") // FK na tabela usuario
    private Perfil perfil;
}

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

    private String descricao;

    @OneToOne(mappedBy = "perfil") // lado inverso
    private Usuario usuario;
}
```

Tipos de Relacionamento

@OneToMany / @ManyToOne (Um para Muitos e Muitos para Um):

- Um para Muitos: Um objeto A pode ter vários objetos B.
- Muitos para Um: Vários objetos B estão associados ao mesmo objeto A.

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

    private String nome;

    @OneToMany(mappedBy = "cliente") // lado inverso
    private List<Pedido> pedidos;
}

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

    private String descricao;

    @ManyToOne
    @JoinColumn(name = "cliente_id") // FK na tabela pedido
    private Cliente cliente;
}
```

Tipos de Relacionamento

@ManyToMany (Muitos para Muitos):

- Definição: Um objeto A pode estar relacionado a vários objetos B, e um objeto B pode estar relacionado a vários objetos A.

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

    private String nome;

    @ManyToMany
    @JoinTable(
        name = "aluno_curso",
        joinColumns = @JoinColumn(name = "aluno_id"),
        inverseJoinColumns = @JoinColumn(name = "curso_id")
    )
    private List<Curso> cursos;
}

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

    private String nome;

    @ManyToMany(mappedBy = "cursos") // lado inverso
    private List<Aluno> alunos;
}
```

@JoinColumn

É a anotação usada no lado dono da relação (owner side).

Serve para indicar qual coluna da tabela vai representar a chave estrangeira (FK) no banco de dados.

Por padrão, o JPA já gera um nome automático (nomeDoAtributo_id), mas com @JoinColumn você pode customizar esse nome e detalhes.

mappedBy

É um atributo da anotação de relacionamento (@OneToMany, @OneToOne, @ManyToMany).

Indica quem é o dono do relacionamento e quem é o lado inverso.

Serve para evitar FK duplicadas no banco de dados.

Gerenciando no base

spring.jpa.hibernate.ddl-auto define como o Hibernate vai lidar com o esquema do banco de dados (tabelas, colunas, constraints) quando a aplicação sobe.

Ou seja, controla se o Hibernate cria, atualiza ou ignora a estrutura das tabelas.

Gerenciando no base

As opções principais

none

- Não faz nada.
- O Hibernate não gera nem altera tabelas.

validate

- Valida se o modelo de entidades Java bate com as tabelas do banco.
- Se algo estiver errado (coluna faltando, tipo errado), a aplicação não sobe.
- Útil quando você quer garantir consistência, mas não quer que o Hibernate altere o banco.

Gerenciando no base

As opções principais

update

- Atualiza automaticamente o esquema do banco para refletir as entidades.
- Cria tabelas/colunas novas se necessário, mas não remove nada.
- Perigoso em produção porque pode deixar o esquema bagunçado com colunas obsoletas.

create

- Igual ao create, mas além disso apaga todas as tabelas quando a aplicação encerra.
- Muito usado em testes automatizados, onde você precisa de um banco limpo em cada execução.

Vendo o SQL gerado

spring.jpa.show-sql=true

Hibernate passa a exibir no console todos os comandos SQL que ele executa — como INSERT, UPDATE, DELETE, SELECT, criação de tabelas, etc.

Vendo o SQL gerado

- O SQL sai sem valores substituídos (aparecem ? nos parâmetros).
- O formato pode ficar pouco legível (tudo em uma linha).
- Pode deixar o log da aplicação inchado

Atividade

Gere uma aplicação com relacionamentos 1:1, 1:n e n:m configure a aplicação para gerar as tabelas e verifique se as tabelas e os relacionamentos foram gerados corretamente

Aula 14

JPA e Hibernate - parte 02

@Transient

A anotação @Transient é usada em entidades JPA para indicar que um atributo não deve ser persistido no banco de dados.

```
@Entity  10 usages
public class Noticia {

    @Transient  no usages
    public String auxiliar;
```

@Repository

Um Repository é uma interface responsável por fazer a comunicação com o banco de dados, de forma simples e sem precisar escrever SQL manualmente.

```java
@Repository  no usages
public interface NoticiaRepository extends JpaRepository<Noticia, Long> {

}
```

@JpaRepository

JpaRepository é uma interface genérica fornecida pelo Spring Data JPA que oferece operações prontas para CRUD, paginação, ordenação e consultas personalizadas em entidades JPA.

```
@Repository  no usages
public interface NoticiaRepository extends JpaRepository<Noticia, Long> {

}
```

Ele elimina a necessidade de escrever código SQL

@JpaRepository

Como primeiro parâmetro deve ser passado a entidade que queremos interagir e o tipo do seu campo de chave primária.

```
@Repository  no usages
public interface NoticiaRepository extends JpaRepository<Noticia, Long> {

}
```

save

O método save() é responsável por persistir (inserir ou atualizar) uma entidade no banco de dados.

O save() retorna a própria entidade persistida, já gerenciada pelo contexto do JPA.

```
Noticia noticia =
        noticiaRepository.save(new Noticia( titulo: "Noticia 01", Situacao.FAZENDO));
```

update

Não existe um método chamado update() no JpaRepository.

O método save() serve tanto para inserir (INSERT) quanto para atualizar (UPDATE), dependendo do estado da entidade.

Se a entidade não tem o id preenchido, ele salva, se a entidade tem id, ele tenta atualizar o registro de id especificado.

Save

O que acontece se eu informar o id manualmente?

Depende:

Save

Se o @GeneratedValue não for definido, a responsabilidade de gerar o ID é da aplicação

- Há a tentativa de inserção.
  - Pode funcionar
  - Pode dar erro de chave duplicada

```
@Entity  15 usages
public class Noticia {

    @Id  5 usages
    public long id;
```

Save

Se o @GeneratedValue for definido, a responsabilidade de gerar o ID é banco

- Pode tentar fazer um INSERT com ID = 10
- O Banco lançará um erro de chave duplicada se já existir esse registro.
- Ou pode gerar um INSERT ignorando o ID manual.
- Pode nem lançar erro e nem inserir o registro no banco

```
@Entity  15 usages
public class Noticia {

    @Id  5 usages
    public long id;
```

findById

Ele busca uma entidade pelo seu ID (chave primária).

Ele devolve um Optional.

```
Optional<Noticia> notica = noticiaRepository.findById(1L);
```

Optional

Optional<T> é uma classe genérica introduzida no Java 8 (pacote java.util), que representa um valor que pode ou não estar presente.

Se o findById não encontrar o usuário, ele retorna null.

A próxima linha tenta acessar user.getNome(), e então ocorre NullPointerException

```
User user = userRepository.findById(1L);
System.out.println(user.getNome());
```

Optional

Optional<T> é uma classe genérica introduzida no Java 8 (pacote java.util), que representa um valor que pode ou não estar presente.

Se o findById não encontrar o usuário, ele retorna null.

A próxima linha tenta acessar user.getNome(), e então ocorre NullPointerException

```
User user = userRepository.findById(1L);
System.out.println(user.getNome());
```

Optional

Optional<T> é uma classe genérica introduzida no Java 8 (pacote java.util), que representa um valor que pode ou não estar presente.

Se o findById não encontrar o usuário, ele retorna null.

A próxima linha tenta acessar user.getNome(), e então ocorre NullPointerException

```
User user = userRepository.findById(1L);
System.out.println(user.getNome());
```

Optional

Optional<T> é uma classe genérica introduzida no Java 8 (pacote java.util), que representa um valor que pode ou não estar presente.

Se o findById não encontrar o usuário, ele retorna null.

A próxima linha tenta acessar user.getNome(), e então ocorre NullPointerException

```
// Verificação Manual - Semelhante a verificar se é null
if (notica.isPresent()) {
    Noticia user = notica.get();
    System.out.println(notica.get().getTitulo());
} else {
    System.out.println("Usuário não encontrado");
}
```

Optional

Optional<T> é uma classe genérica introduzida no Java 8 (pacote java.util), que representa um valor que pode ou não estar presente.

Se o findById não encontrar o usuário, ele retorna null.

A próxima linha tenta acessar user.getNome(), e então ocorre NullPointerException

```
// Maneira 2 - usando orElse
Noticia not = notica.orElse(new Noticia( titulo: "Padrão", Situacao.FAZENDO));
System.out.println(not.getTitulo());
```

Optional

Optional<T> é uma classe genérica introduzida no Java 8 (pacote java.util), que representa um valor que pode ou não estar presente.

Se o findById não encontrar o usuário, ele retorna null.

A próxima linha tenta acessar user.getNome(), e então ocorre NullPointerException

```
// Maneira 3 - lançando exceção personalizada
Noticia notici = notica.orElseThrow(() -> new NegocioException("Usuário não encontrado"));
```

findAll()

O método findAll() é definido na interface JpaRepository (herdada de PagingAndSortingRepository e CrudRepository) e serve para buscar todos os registros de uma entidade no banco de dados.

```
List<Noticia> noticas = noticiaRepository.findAll();
```

findAll()

O método findAll() é definido na interface JpaRepository (herdada de PagingAndSortingRepository e CrudRepository) e serve para buscar todos os registros de uma entidade no banco de dados.

```
List<Noticia> noticas = noticiaRepository.findAll();
```

delete(T entidade)

Remove um objeto específico (instância da entidade) do banco.

Internamente, o JPA verifica o ID do objeto e executa o DELETE correspondente.

```
List<Noticia> noticas = noticiaRepository.findAll();
```

deleteById(IdEntidade)

Remove o registro com o ID especificado do banco de dados.

Atividade

Gere uma aplicação de controle de uma biblioteca, a aplicação deve ter um formulário para cadastro de autor, livros e empréstimos.

Adicione também uma API para que a aplicação forneça suporte a conexão de um aplicativo.

Aula 15

JPA e Hibernate - parte 03

@Version

O que é o Version Lock (ou Optimistic Locking)

Em sistemas com acesso concorrente ao banco, pode acontecer de duas pessoas editarem o mesmo registro ao mesmo tempo.

```
@Entity  18 usages
public class Noticia {

    @Version  2 usages
    public long version;
```

@Version

Você adiciona um campo de versão na entidade, e o JPA automaticamente:

incrementa esse valor a cada atualização (UPDATE), e usa-o para verificar se o registro não foi modificado por outro usuário entre a leitura e a gravação.

```
@Entity  18 usages
public class Noticia {

    @Version  2 usages
    public long version;
```

@Version

Se o registro que está sendo alterado tiver uma versão de lock menor do que a do banco, o banco lança uma exceção.

```
@Entity  18 usages
public class Noticia {

    @Version  2 usages
    public long version;
```

Save com relacionamentos

Quando um objeto é relacionado ele pode ser obrigatório ou não.

Essa definição é escolhida pela propriedade nullable

Por padrão a inserção é opcional

```java
@ManyToOne  no usages
@JoinColumn(name="autor_id")
public Autor autor;
```

```java
// Autor Obrigatório
@ManyToOne  no usages
@JoinColumn(name="autor_id",  nullable=false)
public Autor autor;
```

Save com relacionamentos

Se a informação é obrigatória (nullable=false) e o valor não é informado é lançada uma exceção.

```java
@ManyToOne  no usages
@JoinColumn(name="autor_id")
public Autor autor;
```

```java
// Autor Obrigatório
@ManyToOne  no usages
@JoinColumn(name="autor_id",  nullable=false)
public Autor autor;
```

Save com relacionamentos

A forma mais simples e manual de realizar essa operação é salvar o objeto antes e depois atribuí lo a entidade que irá ser salva.

```
Noticia n = new Noticia( titulo: "Noticia 01", Situacao.FAZENDO);
Autor a = new Autor( nome: "Pedro");

a = autorRepository.save(a);

n.autor = a;
noticiaRepository.save(n);
```

Contexto de persistência

O JPA (ou Hibernate) mantém uma cópia em memória das entidades que você está manipulando — é o chamado contexto de persistência (ou Persistence Context).

Cascade

Ele define como as operações feitas em uma entidade “pai” se propagam (ou não) para suas entidades “filhas” relacionadas

```
Noticia n = new Noticia( titulo: "Noticia 01", Situacao.FAZENDO);
Autor a = new Autor( nome: "Pedro");

a = autorRepository.save(a);

n.autor = a;
noticiaRepository.save(n);
```

Cascade

Imagine o exemplo

```java
@Entity  no usages
public class Pedido {
    @Id  no usages
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToMany(mappedBy = "pedido", cascade = CascadeType.ALL)
    private List<Item> itens;
}
```

Cascade

Tipo	O que faz	Exemplo
`PERSIST`	Ao salvar (persistir) o pai, também salva os filhos.	`entityManager.persist(pedido)` também salva `itens`.
`MERGE`	Ao atualizar o pai, também atualiza os filhos.	`entityManager.merge(pedido)` atualiza `itens`.
`REMOVE`	Ao deletar o pai, também deleta os filhos.	`entityManager.remove(pedido)` deleta `itens`.
`REFRESH`	Atualiza o estado dos filhos com base no banco (reload).	`entityManager.refresh(pedido)` atualiza `itens`.
`DETACH`	Desanexa os filhos do contexto de persistência.	`entityManager.detach(pedido)` desanexa `itens`.
`ALL`	Atalho para aplicar todos os tipos acima.	Equivalente a `cascade = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}`.

Tipos de carregamento

FetchType.EAGER (carregamento imediato)

Os dados relacionados são carregados junto com a entidade principal, no mesmo momento da consulta.

É o comportamento padrão.

```java
@OneToMany(mappedBy = "autor", fetch = FetchType.EAGER)
List<Noticia> noticias;
```

Tipos de carregamento

Os dados relacionados não são carregados imediatamente quando a entidade principal é buscada.

Em vez disso, o Hibernate cria um proxy (um objeto “falso”) e só busca os dados quando o relacionamento é acessado.

É necessário estar dentro de uma transação para o carregamento funcionar

```
@OneToMany(mappedBy = "autor", fetch = FetchType.LAZY)
List<Noticia> noticias;
```

@Transactional

A anotação @Transactional indica ao Spring que o método (ou classe) deve ser executado dentro de uma transação de banco de dados.

```
@OneToMany(mappedBy = "autor", fetch = FetchType.LAZY)
List<Noticia> noticias;
```

EXercício

Gere uma aplicação de controle de uma biblioteca, a aplicação deve ter um formulário para cadastro de autor, livros e empréstimos.

Adicione também uma API para que a aplicação forneça suporte a conexão de um aplicativo.

Adicione um coluna de versionlock nas entidades

Utiliza o cascade para as operações de criar e atualizar

Utiliza transações para não deixar o banco inconcistênte

Aula 16

JPA e Hibernate - parte 04

JPA Query Methods

O Spring Data JPA facilita o acesso ao banco de dados, evitando a escrita manual de SQL, gerando as consultas a partir do nome do método.

```java
public interface ProdutoRepository extends JpaRepository<Produto, Long> {

    List<Produto> findByNome(String nome);
    List<Produto> findByPrecoGreaterThan(Double preco);
    List<Produto> findByNomeContainingIgnoreCase(String nome);
    List<Produto> findByCategoriaNome(String nomeCategoria);

}
```

Operadores Comuns

Operador	Exemplo	Descrição
`Is`, `Equals`	`findByNomeEquals`	Igual
`Between`	`findByPrecoBetween(Double min, Double max)`	Intervalo
`LessThan`, `GreaterThan`	`findByPrecoGreaterThan`	Comparações
`Containing`	`findByNomeContaining`	LIKE %nome%
`StartingWith`, `EndingWith`	`findByNomeStartingWith`	LIKE nome%
`IgnoreCase`	`findByNomeIgnoreCase`	Case insensitive

Operadores Comuns

https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

JPQL (Java Persistence Query Language)

JPQL (Java Persistence Query Language) é uma linguagem de consulta orientada a objetos Java, usada para buscar e manipular entidades gerenciadas pelo JPA (Java Persistence API).

SQLxJPQL

Aspecto	SQL	JPQL
Trabalha com	Tabelas e colunas do banco	Classes e atributos Java (@Entity)
Nomes usados	usuario, nome (nomes físicos)	Usuario, u.nome (nomes de classe e atributo)
Retorno	Linhas e colunas	Objetos Java
Portabilidade	Depende do banco (MySQL, PostgreSQL, etc.)	Funciona em qualquer banco suportado pelo JPA
Onde é usada	JDBC, SQL nativo	@Query no Spring Data, ou EntityManager JPA

@Query

A anotação @Query permite que você escreva manualmente a consulta que o método do repositório vai executar.

```
@Query("SELECT a FROM Autor a ORDER BY nome")
List<Autor> todosOsAutores();
```

@Query

Para Queries mais complexas, é possível usar as aspas triplas para escrever múltiplas linhas

```
@Query(""" no usages
    SELECT a.nome FROM Autor a
        ORDER BY nome
""")
List<String> nomesAutores();
```

@Param - Nomeados

Serve para ligar um parâmetro do método Java ao parâmetro nomeado usado dentro da query do @Query.

```
@Query("SELECT a FROM Autor a WHERE nome like %:pramNome% ORDER BY nome")
List<Autor> nomeContem(@Param("pramNome") String nome);
```

@Param - Posicionais

Serve para ligar um parâmetro do método Java ao parâmetro nomeado usado dentro da query do @Query.

```
@Query("SELECT n FROM Noticia n WHERE autor.id = ?1 AND situacao = ?2")
List<Noticia> findByAutorAndSituacao2(Long autor, Situacao situacao);
```

@Filtros por Entidades Relacionadas

Em JPA, quando uma entidade possui um relacionamento com outra (por exemplo, @ManyToOne, @OneToMany, @OneToOne, @ManyToMany), você pode filtrar registros baseando-se em atributos do objeto relacionado.

```
@Query("SELECT n FROM Noticia n WHERE autor.id = ?1 AND situacao = ?2")
List<Noticia> findByAutorAndSituacao2(Long autor, Situacao situacao);
```

@Filtros por Entidades Relacionadas

Essa abordagem permite trabalhar usando o modelo de objetos, e não a estrutura do banco, como Join.

```
@Query("SELECT n FROM Noticia n WHERE autor.id = ?1 AND situacao = ?2")
List<Noticia> findByAutorAndSituacao2(Long autor, Situacao situacao);
```

SQL Nativo

É possível trablhar com SQL, para isso basta passar a propriedade

nativeQury = true

```
@Query("SELECT n FROM Noticia n WHERE n.autor.nome = :nomeAutor")  no usages
List<Noticia> buscarPorNomeAutor(@Param("nomeAutor") String nomeAutor);

@Query(value = """  no usages
        SELECT n.*
        FROM noticia n
        INNER JOIN autor a ON n.autor_id = a.id
        WHERE a.nome = ?
        """, nativeQuery = true)
List<Noticia> buscarPorNomeAutorNativo(@Param("nomeAutor") String nomeAutor);
```

Desvantagens de JPQL

É uma linguagem de consulta orientada a entidades, não a tabelas, então ela vai ter algumas desvantagens em relação a consultas nativas.

- Menos controle sobre o SQL gerado
  - O JPA monta o SQL internamente e às vezes gera consultas mais complexas ou menos otimizadas.
  - Sem suporte a funções específicas do banco JPQL é padronizado e não reconhece funções nativas
  - Em joins múltiplos, subconsultas aninhadas, filtros dinâmicos, ou consultas que envolvem agregações avançadas (GROUP BY, HAVING, funções de janela etc.), o JPQL pode ficar longo, verboso e limitado.

Pageable

Pageable é uma interface que representa as informações de paginação e ordenação.

page → número da página (começa do 0)

size → quantidade de itens por página

sortBy → nome do campo de ordenação

```
Pageable pageable = PageRequest.of( pageNumber: 0,  pageSize: 10, Sort.by( ...properties: "titulo_completo"));
```

Pageable

Pageable é uma interface que representa as informações de paginação e ordenação.

page → número da página (começa do 0)

size → quantidade de itens por página

sortBy → nome do campo de ordenação

```
Change signature
@Query(value = """  1 usage
        SELECT n.*
        FROM noticia n
        INNER JOIN autor a ON n.autor_id = a.id
        WHERE a.nome = ?
        """, nativeQuery = true)
List<Noticia> buscarPorNomeAutorNativo(@Param("nomeAutor") String nomeAutor, Pageable pageable);
```

Exercício

Gere uma aplicação de controle de uma biblioteca, a aplicação deve ter um formulário para cadastro de autor, livros e empréstimos.

Adicione também uma API para que a aplicação forneça suporte a conexão de um aplicativo.

Adicione métodos de busca personalizados e de páginação.

Aula 17

Atividade

Exercício

Gere uma aplicação de controle de uma biblioteca, a aplicação deve ter um formulário para cadastro de autor, livros e empréstimos.

Adicione também uma API para que a aplicação forneça suporte a conexão de um aplicativo.

Adicione um coluna de versionlock nas entidades

Utiliza o cascade para as operações de criar e atualizar

Utiliza transações para não deixar o banco inconsistente

Adicione métodos de busca personalizados e de páginação.