Faça uma revisão didática completa do Hibernate 5.1.17 em português brasileiro

Thought for 16 seconds

No relevant results found

Vamos criar uma revisão didática completa do Hibernate 5.1.17 em português brasileiro. O objetivo é fornecer um material educativo abrangente, cobrindo conceitos fundamentais, configuração, mapeamento, operações, consultas, caching, transações, etc. Precisamos adaptar para a versão 5.1.17 (uma versão mais antiga, mas ainda usada em projetos legados). O conteúdo deve ser claro, com exemplos de código, explicações passo a passo.

Vamos estruturar em seções:

1. Introdução ao Hibernate
   - O que é ORM?
   - Hibernate como framework ORM
   - Vantagens
   - Visão geral da arquitetura (SessionFactory, Session, Transaction, Query, etc.)
2. Configuração do ambiente
   - Pré-requisitos (Java, banco de dados, Maven/Gradle)
   - Adicionando dependências (hibernate-core, driver JDBC)
   - Arquivo de configuração hibernate.cfg.xml (ou persistence.xml)
   - Configurações básicas: dialeto, driver, URL, usuário, senha, show_sql, hbm2ddl.auto
3. Mapeamento objeto-relacional
   - Anotações JPA vs arquivos .hbm.xml
   - Anotações principais: @Entity, @Table, @Id, @GeneratedValue, @Column
   - Mapeamento de tipos básicos
   - Mapeamento de relacionamentos: @OneToOne, @OneToMany, @ManyToOne, @ManyToMany
   - Mapeamento de herança (opcional)
   - Exemplos práticos
4. Sessão e operações CRUD
   - Obtendo SessionFactory
   - Abrindo e fechando Session
   - Transações: beginTransaction(), commit(), rollback()
   - Operações: save(), persist(), get(), load(), update(), merge(), delete()
   - Diferenças entre get e load

- Exemplos práticos

5. Consultas com HQL e Criteria
   - HQL (Hibernate Query Language): consultas básicas, parâmetros, joins, funções agregadas
   - Criteria API (legada) e a nova JPA Criteria API (embora na 5.1 ainda tenha a Criteria legada)
   - Exemplos de consultas
   - Named queries
6. Consultas nativas
   - SQL nativo
   - Mapeamento de resultados
7. Gerenciamento de estado das entidades
   - Estados: transient, persistent, detached, removed
   - Ciclo de vida
   - Operações que mudam o estado
8. Cache
   - Cache de primeiro nível (Session)
   - Cache de segundo nível (opcional, configuração com EhCache)
   - Cache de consultas
9. Transações e concorrência
   - Controle de transações programático e declarativo (com @Transactional)
   - Lock otimista e pessimista
   - Versionamento com @Version
10. Boas práticas e dicas
    - Uso de lazy loading
    - Problema N+1 e soluções (join fetch, EntityGraph)
    - Escolha de tipos de coleção
    - Tratamento de exceções
11. Migração de versões (breve menção)
    - Diferenças para versões posteriores

Vou escrever de forma didática, explicando cada conceito e fornecendo trechos de código. Usarei exemplos com entidades como "Cliente", "Pedido", etc.

A linguagem será português brasileiro claro e acessível.

Revisão Didática do Hibernate 5.1.17

1. Introdução ao Hibernate

O que é ORM?

ORM (Object-Relational Mapping) é uma técnica para mapear objetos Java para tabelas de um banco de dados relacional. O objetivo é reduzir a quantidade de código JDBC manual, permitindo que o desenvolvedor trabalhe com objetos e o framework cuide da persistência.

Hibernate como framework ORM

Hibernate é um framework ORM maduro e amplamente utilizado no ecossistema Java. Ele implementa a especificação JPA (Java Persistence API) e oferece recursos adicionais. A versão 5.1.17 é uma versão estável da série 5.1, compatível com JPA 2.1.

Vantagens do Hibernate

- Produtividade: menos código para escrever, pois o mapeamento é declarativo.
- Portabilidade: abstrai as diferenças entre bancos de dados (através do dialeto).
- Gerenciamento de cache: cache de primeiro e segundo níveis.
- Lazy loading: carregamento sob demanda de relacionamentos.
- HQL (Hibernate Query Language): linguagem de consulta orientada a objetos.

Arquitetura simplificada

Os principais componentes são:

- **SessionFactory**: fábrica de sessões, geralmente uma única instância por aplicação. É thread-safe e guarda as configurações e metadados.
- **Session**: representa uma unidade de trabalho com o banco de dados. Não é thread-safe, deve ser usada por uma única thread. Fornece métodos para CRUD e consultas.
- **Transaction**: abstrai a transação subjacente (JDBC, JTA). Usada para demarcar transações.
- **Query**: interface para consultas HQL ou SQL nativo.
- **Criteria**: API para construção de consultas programáticas (legada e JPA).

2. Configuração do ambiente

Pré-requisitos

- JDK 8 ou superior (Hibernate 5.1 requer Java 8).
- Banco de dados relacional (ex: H2, MySQL, PostgreSQL).
- Gerenciador de dependências: Maven ou Gradle.

Adicionando dependências com Maven

No arquivo `pom.xml`, adicione:

```xml
<dependencies>
    <!-- Hibernate core -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.1.17.Final</version>
    </dependency>

    <!-- Driver JDBC do seu banco (exemplo com MySQL) -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.28</version> <!-- ajuste conforme versão do MySQL -->
    </dependency>
</dependencies>
```

Arquivo de configuração `hibernate.cfg.xml`

Crie o arquivo `hibernate.cfg.xml` na pasta `src/main/resources`:

```xml
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Configurações da conexão -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver
</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/meuba
nco</property>
```

```
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">senha</property>

        <!-- Dialeto do banco -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</pro
perty>

        <!-- Opções de exibição e geração do schema -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">update</property> <!-- 'create', 'v
alidate', etc. -->

        <!-- Mapeamento das entidades (classes anotadas) -->
        <mapping class="com.exemplo.entidade.Cliente"/>
        <mapping class="com.exemplo.entidade.Pedido"/>
        <!-- Ou use <mapping resource="arquivo.hbm.xml"/> para mapeamento XML -->
    </session-factory>
</hibernate-configuration>
```

Explicação das propriedades:

- `hibernate.connection.*` : dados da conexão JDBC.
- `hibernate.dialect` : dialeto específico do banco (MySQL, PostgreSQL, Oracle, etc). Hibernate usa isso para gerar SQL adequado.
- `hibernate.show_sql` : exibe o SQL gerado no console.
- `hibernate.format_sql` : formata o SQL para melhor leitura.
- `hibernate.hbm2ddl.auto` : controla a geração automática do esquema:
  - `none` : não faz nada.
  - `validate` : valida se as tabelas existem.
  - `update` : atualiza o esquema (cria/ altera tabelas) – cuidado em produção.
  - `create` : cria o esquema, destruindo dados existentes.
  - `create-drop` : cria ao iniciar e dropa ao finalizar (útil para testes).

3. Mapeamento objeto-relacional

Usando anotações JPA

Hibernate suporta as anotações padrão JPA. Exemplo de uma entidade simples:

```java
package com.exemplo.entidade;

import javax.persistence.*;

@Entity
@Table(name = "clientes")
public class Cliente {

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

    @Column(name = "nome", length = 100, nullable = false)
    private String nome;

    @Column(unique = true)
    private String email;

    private Integer idade;

    // Construtores, getters e setters
    public Cliente() {}

    // getters e setters omitidos para brevidade
}
```

Principais anotações:

- `@Entity` : marca a classe como uma entidade gerenciada pelo Hibernate.
- `@Table` : especifica detalhes da tabela (nome, esquema, etc). Opcional se o nome da tabela for igual ao da classe.
- `@Id` : identifica a chave primária.
- `@GeneratedValue` : estratégia de geração de valor (AUTO, IDENTITY, SEQUENCE, TABLE).
- `@Column` : personaliza a coluna (nome, nullable, length, unique, etc). Se omitido, o nome da coluna é o nome do campo.
- Tipos básicos como `String` , `int` , `Date` são mapeados automaticamente para tipos SQL correspondentes.

Mapeamento de relacionamentos

Exemplo de relacionamento OneToMany / ManyToOne

Considere as entidades `Cliente` e `Pedido`:

```java
@Entity
@Table(name = "pedidos")
public class Pedido {

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

    @Column(name = "data_pedido")
    private LocalDate data;

    @ManyToOne
    @JoinColumn(name = "cliente_id", nullable = false)
    private Cliente cliente;

    // getters e setters
}
```

```java
@Entity
@Table(name = "clientes")
public class Cliente {

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

    private String nome;

    @OneToMany(mappedBy = "cliente") // 'cliente' é o nome do atributo em Pedido
    private List<Pedido> pedidos = new ArrayList<>();

    // getters e setters
}
```

- `@ManyToOne` (lado dono do relacionamento) mapeia uma chave estrangeira. `@JoinColumn` define o nome da coluna FK.

- `@OneToMany(mappedBy = ...)` indica que o relacionamento é bidirecional e o lado inverso é mapeado pelo atributo `cliente` na entidade `Pedido`. O `mappedBy` evita a criação de tabela adicional.

OneToOne

```java
@Entity
public class Endereco {
    @Id @GeneratedValue
    private Long id;

    private String rua;

    @OneToOne(mappedBy = "endereco")
    private Cliente cliente;
}

@Entity
public class Cliente {
    // ...
    @OneToOne
    @JoinColumn(name = "endereco_id")
    private Endereco endereco;
}
```

ManyToMany

Geralmente mapeado com `@JoinTable`:

```java
@Entity
public class Livro {
    @Id @GeneratedValue
    private Long id;
    private String titulo;

    @ManyToMany
    @JoinTable(
        name = "livro_autor",
        joinColumns = @JoinColumn(name = "livro_id"),
        inverseJoinColumns = @JoinColumn(name = "autor_id")
    )
    private List<Autor> autores = new ArrayList<>();
```

```
}

@Entity
public class Autor {
    @Id @GeneratedValue
    private Long id;
    private String nome;

    @ManyToMany(mappedBy = "autores")
    private List<Livro> livros = new ArrayList<>();
}
```

Mapeamento de herança

Hibernate suporta várias estratégias (single table, joined, table per class). Exemplo simples com `@Inheritance`:

```
java

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "tipo_pessoa")
public abstract class Pessoa {
    @Id @GeneratedValue
    private Long id;
    private String nome;
    // getters/setters
}

@Entity
@DiscriminatorValue("F")
public class PessoaFisica extends Pessoa {
    private String cpf;
}

@Entity
@DiscriminatorValue("J")
public class PessoaJuridica extends Pessoa {
    private String cnpj;
}
```

4. Sessão e operações CRUD

Obtendo a SessionFactory

Normalmente cria-se uma única instância de `SessionFactory` para a aplicação. Exemplo com configuração via arquivo:

```java
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Carrega as configurações do hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        getSessionFactory().close();
    }
}
```

Abrindo uma sessão e realizando operações

Salvando uma entidade (persist)

```java
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction transaction = null;
try {
    transaction = session.beginTransaction();

    Cliente cliente = new Cliente();
    cliente.setNome("João Silva");
    cliente.setEmail("joao@email.com");
```

```
        cliente.setIdade(30);

        session.save(cliente); // ou session.persist(cliente)

        transaction.commit();
    } catch (Exception e) {
        if (transaction != null) transaction.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
```

Recuperando uma entidade

- `session.get(Cliente.class, 1L)` : retorna `null` se não encontrar, sempre vai ao banco.
- `session.load(Cliente.class, 1L)` : retorna um proxy, só acessa o banco quando um método getter é chamado (exceto getId()). Lança `ObjectNotFoundException` se não existir.

```
java

Cliente cliente = session.get(Cliente.class, 1L);
if (cliente != null) {
    System.out.println(cliente.getNome());
}
```

Atualizando uma entidade

```
java

// Supondo que cliente já esteja no banco
Cliente cliente = session.get(Cliente.class, 1L);
cliente.setNome("João Souza");
session.update(cliente); // também poderia ser session.merge(cliente)
```

Removendo uma entidade

```
java

Cliente cliente = session.get(Cliente.class, 1L);
if (cliente != null) {
    session.delete(cliente);
}
```

Diferenças entre save, persist, update, merge, saveOrUpdate

- `save()` : insere uma entidade e retorna o ID gerado.
- `persist()` : similar, mas não retorna o ID (é o método JPA).
- `update()` : anexa uma entidade detached ao contexto de persistência, considerando que ela já existe no banco.
- `merge()` : mescla o estado de uma entidade detached para uma entidade persistente (pode retornar uma instância diferente).
- `saveOrUpdate()` : insere ou atualiza baseado no identificador (se é transient ou detached).

Transações

Sempre use transações, mesmo para operações de leitura (em alguns casos pode ser opcional, mas é boa prática). A transação garante atomicidade e consistência.

5. Consultas com HQL e Criteria

HQL (Hibernate Query Language)

HQL é uma linguagem orientada a objetos que opera sobre as entidades e seus atributos.

Consultas básicas

```
java

Session session = HibernateUtil.getSessionFactory().openSession();
try {
    // Listar todos os clientes
    Query<Cliente> query = session.createQuery("from Cliente", Cliente.class);
    List<Cliente> clientes = query.list();

    // Com condição
    Query<Cliente> q2 = session.createQuery("from Cliente c where c.nome like :nom
e", Cliente.class);
    q2.setParameter("nome", "João%");
    List<Cliente> resultado = q2.list();

    // Consulta que retorna um único resultado
    Query<Long> q3 = session.createQuery("select count(c) from Cliente c", Long.cla
```

```
ss);
    Long total = q3.uniqueResult();
} finally {
    session.close();
}
```

Joins em HQL

```
java

// Join explícito
Query<Object[]> query = session.createQuery(
    "select c.nome, p.id from Cliente c join c.pedidos p", Object[].class);
List<Object[]> resultados = query.list();

// Join fetch para evitar LazyInitializationException (carrega os pedidos junto)
Query<Cliente> q = session.createQuery("from Cliente c join fetch c.pedidos where
c.id = :id", Cliente.class);
Cliente c = q.setParameter("id", 1L).uniqueResult();
// Agora c.getPedidos() já está carregado (mesmo que o relacionamento seja lazy)
```

Parâmetros

Há duas formas: posicional (`?1` , `?2`) ou nomeada (`:nome`). Recomenda-se nomeada.

```
java

Query<Cliente> q = session.createQuery("from Cliente where email = :email", Client
e.class);
q.setParameter("email", "joao@email.com");
```

Criteria API (legada do Hibernate)

A API Criteria original do Hibernate (org.hibernate.Criteria) está disponível, mas é considerada legada em favor da JPA Criteria. No entanto, na versão 5.1 ainda é bastante usada.

```
java

Criteria criteria = session.createCriteria(Cliente.class);
criteria.add(Restrictions.eq("email", "joao@email.com"));
criteria.add(Restrictions.gt("idade", 25));
List<Cliente> clientes = criteria.list();
```

```
// Ordenação
criteria.addOrder(Order.asc("nome"));

// Projeções
criteria.setProjection(Projections.rowCount());
Long count = (Long) criteria.uniqueResult();
```

JPA Criteria API

A API Criteria do JPA é mais type-safe, porém mais verbosa. Exemplo:

```
java

CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Cliente> cr = cb.createQuery(Cliente.class);
Root<Cliente> root = cr.from(Cliente.class);
cr.select(root).where(cb.equal(root.get("email"), "joao@email.com"));

Query<Cliente> query = session.createQuery(cr);
List<Cliente> result = query.list();
```

Named queries

Defina a consulta na entidade usando `@NamedQuery`:

```
java

@Entity
@NamedQuery(name = "Cliente.porNome", query = "from Cliente where nome like :nome")
public class Cliente { ... }
```

Depois execute:

```
java

Query<Cliente> q = session.createNamedQuery("Cliente.porNome", Cliente.class);
q.setParameter("nome", "João%");
List<Cliente> lista = q.list();
```

6. Consultas nativas (SQL)

Você pode executar SQL nativo quando necessário:

```
java

SQLQuery<Cliente> query = session.createSQLQuery("select * from clientes where idade > :idade")
        .addEntity(Cliente.class)
        .setParameter("idade", 18);
List<Cliente> clientes = query.list();
```

Ou mapear para um DTO usando `setResultTransformer`:

```
java

SQLQuery<Object[]> query = session.createSQLQuery("select nome, email from clientes");
query.setResultTransformer(Transformers.aliasToBean(ClienteDTO.class));
List<ClienteDTO> lista = query.list();
```

7. Gerenciamento de estado das entidades

As entidades podem estar em um dos seguintes estados:

- **Transient**: objeto criado com `new`, não possui representação no banco e não está associado a uma Session. Exemplo: `Cliente c = new Cliente();`
- **Persistent**: objeto que possui representação no banco e está associado a uma Session. Qualquer alteração feita nele dentro de uma transação será sincronizada com o banco no flush.
- **Detached**: objeto que possui representação no banco, mas não está mais associado a uma Session (a sessão foi fechada ou o objeto foi ejetado). Alterações não são refletidas automaticamente.
- **Removed**: objeto marcado para remoção (após `session.delete` dentro de uma transação, mas antes do commit).

O ciclo de vida é gerenciado pelas operações:

- `save()` / `persist()`: transient → persistent
- `get()` / `load()` / `find()`: retornam objetos persistentes
- `update()` / `merge()` / `saveOrUpdate()`: detached → persistent
- `delete()`: persistent → removed

- `close()` da Session: objetos persistentes tornam-se detached
- `clear()` : limpa a sessão, todos os objetos tornam-se detached
- `evict()` : remove um objeto específico da sessão (detach)

8. Cache

Cache de primeiro nível (L1)

É o cache da Session. Por padrão, sempre ativo e não configurável. Durante uma sessão, se você carregar a mesma entidade duas vezes pelo ID, a segunda chamada não vai ao banco.

```java
Cliente c1 = session.get(Cliente.class, 1L); // consulta banco
Cliente c2 = session.get(Cliente.class, 1L); // retorna do cache L1
assert c1 == c2; // mesma instância
```

Cache de segundo nível (L2)

É opcional, compartilhado entre sessões. Para habilitar, adicione dependências (ex: Ehcache) e configure.

1. Dependência (Maven):

```xml
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>5.1.17.Final</version>
</dependency>
```

2. Configure no `hibernate.cfg.xml` :

```xml
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
```

```
<property name="hibernate.cache.use_query_cache">true</property> <!-- opcional para cache de queries -->
```

3. Anote as entidades que devem ser cacheadas:

```
java

@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Cliente { ... }
```

Estratégias de concorrência: `READ_ONLY`, `READ_WRITE`, `NONSTRICT_READ_WRITE`, `TRANSACTIONAL`.

Cache de consultas

Para armazenar resultados de queries (HQL ou Criteria), ative `hibernate.cache.use_query_cache` e chame `setCacheable(true)` na query:

```
java

Query<Cliente> q = session.createQuery("from Cliente where idade > :idade", Cliente.class)
    .setParameter("idade", 18)
    .setCacheable(true);
List<Cliente> lista = q.list();
```

9. Transações e concorrência

Controle programático de transações

Já vimos o uso de `session.beginTransaction()`, `commit()`, `rollback()`. Em aplicações Java EE/Spring, pode-se usar `@Transactional`.

Lock otimista

Usa uma coluna de versão (`@Version`) para evitar atualizações concorrentes. O Hibernate incrementa a versão automaticamente ao modificar a entidade.

```
java
```

```
@Entity
public class Cliente {
    @Id @GeneratedValue
    private Long id;

    @Version
    private Integer versao;
    // outros atributos
}
```

Se duas transações tentarem atualizar o mesmo registro simultaneamente, a segunda que fizer o commit lançará `OptimisticLockException`.

Lock pessimista

Você pode travar uma entidade explicitamente durante a leitura:

```
java

// Lock pessimista de escrita (SELECT ... FOR UPDATE)
Cliente cliente = session.get(Cliente.class, 1L, LockMode.PESSIMISTIC_WRITE);
```

Há também `LockMode.PESSIMISTIC_READ`, etc.

10. Boas práticas e dicas

Lazy loading

Relacionamentos são lazy por padrão (exceto se anotados com `fetch = FetchType.EAGER`). Isso evita carregar dados desnecessários. Cuidado com `LazyInitializationException` ao acessar a coleção fora da sessão. Soluções:

- Manter a sessão aberta durante a renderização da view (padrão Open Session in View) – cuidado com desempenho.
- Fazer join fetch na consulta para carregar antecipadamente.
- Usar `Hibernate.initialize(proxy)` para forçar carregamento.

Problema N+1

Ocorre quando você executa uma consulta para listar N entidades e depois, ao acessar um relacionamento de cada uma, são disparadas N consultas adicionais. Exemplo:

```java
List<Cliente> clientes = session.createQuery("from Cliente", Cliente.class).list();
for (Cliente c : clientes) {
    System.out.println(c.getPedidos().size()); // dispara uma consulta para cada cliente
}
```

Solução: use `join fetch` na consulta inicial ou configure um batch fetching (`@BatchSize`).

Escolha de tipos de coleção

- Use `List` com índice se precisar de ordenação.
- Use `Set` para evitar duplicatas.
- Em relacionamentos, prefira coleções que suportem lazy loading.

Tratamento de exceções

As exceções do Hibernate geralmente são do tipo `HibernateException` (não verificada). É importante fazer rollback da transação e fechar a sessão em caso de erro.

Pool de conexões

Para produção, configure um pool de conexões (c3p0, HikariCP) nas propriedades do Hibernate. Exemplo com c3p0:

```xml
<property name="hibernate.c3p0.min_size">5</property>
<property name="hibernate.c3p0.max_size">20</property>
<property name="hibernate.c3p0.timeout">300</property>
<property name="hibernate.c3p0.max_statements">50</property>
<property name="hibernate.c3p0.idle_test_period">3000</property>
```

11. Migração para versões posteriores

A versão 5.1.17 é relativamente antiga (lançada em 2017). Versões mais recentes (5.2, 5.3, 5.4, 5.6 e 6.x) trouxeram mudanças significativas:

- Suporte a JPA 2.2, Java 8 time API (LocalDate, etc.) já está na 5.2/5.3.
- API Session unificada com JPA EntityManager.
- Melhorias no Criteria, suporte a streams, etc.
- Novas funcionalidades de multi-tenancy.
- Hibernate 6 requer Jakarta Persistence (javax → jakarta).

A migração pode exigir ajustes, especialmente se usava APIs legadas do Hibernate (como Criteria). Verifique a documentação de migração.

Esta revisão cobriu os principais aspectos do Hibernate 5.1.17 de forma didática. Pratique com exemplos reais para fixar os conceitos. Lembre-se de consultar a documentação oficial do Hibernate 5.1 para detalhes adicionais.