Guia Prático: Spring Data JPA

1. Introdução ao Spring Data JPA

O Spring Data JPA simplifica o acesso a dados, fornecendo uma abstração poderosa para trabalhar com bancos de dados relacionais utilizando a JPA (Java Persistence API). Ele usa a implementação padrão **SimpleJpaRepository** para todas as interfaces que estendem **JpaRepository**, eliminando a necessidade de criar implementações manuais.

2. Tipos de Cascade no Spring Data JPA

O comportamento de cascata determina como as operações realizadas em uma entidade principal são propagadas para as entidades relacionadas.

2.1. CascadeType.PERSIST

Usado para propagar operações de **persistência** (salvar) da entidade principal para as entidades relacionadas.

Exemplo: Se uma entidade Livro tem um relacionamento com Autor e o tipo de cascata **PERSIST** estiver configurado, ao salvar um livro, os autores associados serão salvos automaticamente.

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

   @ManyToOne(cascade = CascadeType.PERSIST)
private Autor autor; }
```

2.2. CascadeType.MERGE

Atualiza entidades existentes (baseando-se na **chave primária**).
Útil quando a entidade principal e suas entidades relacionadas já existem no banco e precisam de atualização.

1. Diferença entre persist e merge com Entidades Relacionadas 2.
persist salva novas entidades:

a. Quando você chama persist em uma entidade principal (pai), ele também insere as entidades relacionadas (filhas) **se forem novas** e a relação estiver configurada com CascadeType.PERSIST.

3. merge atualiza entidades existentes:

a. Ao usar merge, ele **procura por IDs** nas entidades relacionadas:

i. Se encontrar, ele atualiza esses registros.
ii. Se não encontrar, cria novos registros no banco.

```
Session session = ...;
Transaction transaction = session.beginTransaction();

Person detachedPerson = new Person();
detachedPerson.setId(1L); // ID existente
detachedPerson.setName("Maria"); // Nome alterado

Address detachedAddress = new Address();
detachedAddress.setId(2L); // ID existente
detachedAddress.setStreet("Rua B"); // Rua alterada
detachedAddress.setPerson(detachedPerson);

detachedPerson.setAddresses(Arrays.asList(detachedAddress));

// Atualiza Person e Address
Person managedPerson = (Person) session.merge(detachedPerson); transaction.commit();
```

Comportamento:

- O merge procura por registros no banco com os IDs fornecidos:
  - Atualiza o nome de person para "Maria".
  - Atualiza o endereço para "Rua B".

- Gera o SQL:

```
SELECT * FROM Person WHERE id = 1; -- Busca entidade principal
SELECT * FROM Address WHERE id = 2; -- Busca entidade relacionada
UPDATE Person SET name = 'Maria' WHERE id = 1;
UPDATE Address SET street = 'Rua B' WHERE id = 2;
```

2.3. CascadeType.REMOVE

Usado para propagar operações de **remoção** da entidade principal para as entidades relacionadas.

Exemplo: Ao remover um Pedido, todos os itens associados também são removidos automaticamente.

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

   @OneToMany(cascade = CascadeType.REMOVE)
   private List<Item> itens;
```

}

Importante:

- **Não exclui entidades associadas se elas ainda estiverem vinculadas a outras entidades.**
- **Exemplo**: ○ Uma entidade Actor está associada tanto a Film quanto a Series. ○ Se você remover um Film, a relação entre o filme e os atores será excluída da tabela de junção. Contudo, o Actor não será excluído se estiver associado a uma Series. Nesse caso, o relacionamento entre esses atores e a série vai continuar existindo.

2.4. CascadeType.REFRESH

Usado para sincronizar o estado da entidade com o banco de dados, descartando quaisquer alterações locais não salvas.

Exemplo: No caso de um sistema de e-commerce, ao processar um pedido enquanto o administrador altera o preço de um item, o **REFRESH** garante que o preço mais atualizado do banco de dados seja usado, evitando inconsistências financeiras.

entityManager.refresh(entidade);

2.5. CascadeType.DETACH

Usado para desassociar a entidade principal e suas entidades relacionadas do contexto de persistência.

Exemplo: Se uma entidade Contrato for desassociada, seus Anexos relacionados também serão desassociados automaticamente.

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

   @OneToMany(cascade = CascadeType.DETACH)
private List<Anexo> anexos; }
```

3. Serialização e Desserialização

3.1. Serialização

É o processo de converter um objeto em uma sequência de bytes, permitindo que ele seja armazenado ou transmitido. Para isso, é necessário implementar a interface Serializable.

```
public class Produto implements Serializable {
private static final long serialVersionUID = 1L;
private Long id;     private String nome;
```

}

Exemplos muito comuns no uso de serialização são nas entidades e DTO's.

Entidades (@Entity)

- **Deve implementar Serializable?**
- Sim, em muitos casos.

Por quê?

1. **Exigência do JPA/Hibernate**:
    a. Algumas implementações JPA recomendam ou até exigem que as entidades sejam serializáveis para funcionar corretamente em contextos como cache distribuído, replicação e serialização em sessões de aplicativos web.
    b. A especificação JPA sugere, mas não obriga.
2. **Desempenho e Transferência de Estado**:
    a. Entidades precisam ser serializáveis em cenários onde o estado da sessão é transferido para outra máquina (exemplo: aplicações distribuídas).
3. **Boas práticas**:
    a. Implementar Serializable evita problemas em operações de persistência, mesmo que não seja estritamente necessário em todas as situações.

DTOs (Data Transfer Objects)

- **Deve implementar Serializable?**

Sim, **se o DTO for usado para transferir dados entre sistemas** (exemplo: entre uma aplicação cliente e um servidor via rede).

Por quê?

1. **Transferência de Dados**:
    a. DTOs frequentemente são usados para transportar informações, e a serialização é necessária em protocolos que exigem a conversão do objeto para um formato transferível (como Java RMI ou serialização HTTP).
2. **Cache e Sessão**:
    a. DTOs podem ser armazenados em cache ou em sessões, o que exige que sejam serializáveis.

3.2. Desserialização

É o processo inverso, no qual os bytes serializados são convertidos de volta em um objeto Java.

3.3. Uso da Anotação @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)

Utilizada para controlar o acesso de campos em operações de serialização e desserialização.

Exemplo: Um campo anotado com WRITE_ONLY será considerado apenas para entrada de dados (como requisições JSON) e não será exibido nas respostas.

```
public class Usuario {
   private String nome;

   @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
private String senha; }
```

4. JSON IGNORE

@JsonIgnore para Evitar Serialização Cíclica:

a. Em relações bidirecionais, use @JsonIgnore para evitar erros como StackOverflowError.

```
@ManyToOne @JsonIgnore
private Disciplina disciplina;
```

5. Execução de Scripts SQL no Spring Boot

O Spring Boot reconhece automaticamente os seguintes arquivos para executar scripts SQL:

- **schema.sql**: usado para criar o esquema do banco de dados.
- **data.sql**: usado para inserir ou manipular dados no banco de dados.

Especificando um nome de arquivo diferente

Se desejar usar outro nome, como example.sql, você precisa configurar explicitamente no arquivo application.properties ou application.yml:

Usando application.properties:

```
properties spring.sql.init.schema-locations=classpath:example.sql
```

Usando application.yml:

```
spring:
sql:
init:
    schema-locations: classpath:example.sql
```

6. Relacionamentos Many-to-Many

Ao modelar relacionamentos **ManyToMany**, é sempre recomendável criar uma tabela intermediária explicitamente. Isso permite maior controle sobre o relacionamento e facilita a adição de informações adicionais, como a data de associação.**Exemplo:**

```
@Entity public class Student implements
Serializable {     private static final long
serialVersionUID = 1L;

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

   private String name;

   @ManyToMany
   @JoinTable(
      name = "student_course", // Nome da tabela intermediária
      joinColumns = @JoinColumn(name = "student_id"), // Chave estrangeira da entidade
atual
      inverseJoinColumns = @JoinColumn(name = "course_id") // Chave estrangeira da outra
entidade
   )
   private Set<Course> courses = new HashSet<>();
}

@Entity
public class Course implements Serializable {
private static final long serialVersionUID = 1L;

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

   private String title;

   @ManyToMany(mappedBy = "courses") // Referência ao lado mapeado da relação
private Set<Student> students = new HashSet<>(); }
```

Explicação

1. **Anotação @ManyToMany:**
    a. Configura o relacionamento entre as entidades.
    b. Em Student, usamos @JoinTable para personalizar o nome da tabela intermediária (student_course) e das colunas.

2. **Tabela Intermediária:**
    a. O JPA cria uma tabela chamada student_course com duas colunas: student_id e course_id.
3. **Direção do Relacionamento:**
    a. Student é o **lado proprietário** (define a tabela intermediária).
    b. Course é o lado inverso (usa mappedBy para se referir ao lado proprietário).

7. Herança no JPA

7.1. Estratégias de Herança

a) SINGLE_TABLE (Padrão):

O JPA cria uma única tabela para armazenar os dados da superclasse e de todas as subclasses. Um campo discriminador é usado para identificar a classe da instância.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "user_type") public
class Usuario { /* atributos */ }

@Entity
public class Cliente extends Usuario { /* atributos */ }

@Entity
public class Admin extends Usuario { /* atributos */ }
```

b) JOINED:

Cria uma tabela para a superclasse com atributos comuns e tabelas separadas para cada subclasse, contendo apenas seus atributos específicos.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED) public
class Usuario { /* atributos */ }
```

c) *TABLE_PER_CLASS:*

Cria uma tabela para cada subclasse contendo todos os atributos herdados da superclasse.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Usuario { /* atributos */ }
```

7.2. Uso de @MappedSuperclass

Se não for necessário criar uma tabela para a superclasse, use a anotação

@MappedSuperclass. Isso permite que os atributos da superclasse sejam herdados, mas impede que ela seja tratada como uma entidade.

```
@MappedSuperclass
public abstract class BaseEntity {
private LocalDate createdDate;
private LocalDate updatedDate; }
```

8. Auditoria com Spring Data JPA

Configuração Básica

Para rastrear quem criou ou modificou uma entidade, implemente a interface AuditorAware:

```
@Component
public class AuditorAwareImpl implements AuditorAware<String> {
   @Override
   public Optional<String> getCurrentAuditor() {
return
Optional.ofNullable(SecurityContextHolder.getContext().getAuthentication().getName());
   }
}
```

Ative a auditoria na configuração principal do Spring Boot:

```
@SpringBootApplication
@EnableJpaAuditing(auditorAwareRef = "auditorAwareImpl") public
class MyApplication { }
```

Adicione a auditoria às entidades usando as anotações apropriadas:

```
@EntityListeners(AuditingEntityListener.class)
@Entity
public class Pedido {
@CreatedBy
   private String criadoPor;

   @LastModifiedBy
   private String modificadoPor;

   @CreatedDate
   private LocalDateTime criadoEm;

   @LastModifiedDate
   private LocalDateTime modificadoEm;
```

Nota:

- Sem implementar AuditorAware, apenas as anotações @CreatedDate e @LastModifiedDate funcionarão.
- Para @CreatedBy e @LastModifiedBy, é necessária uma implementação de AuditorAware.

9. Validações no Spring

9.1. Validação de Objetos com @Valid

Usada para validar objetos completos, como DTOs.

```
@PostMapping("/usuarios")
public ResponseEntity<String> criarUsuario(@Valid @RequestBody UsuarioDto
usuarioDto) {     return ResponseEntity.ok("Usuário criado"); }
```

9.2. Validação de Parâmetros com @Validated

Usada para validar parâmetros individuais, como @RequestParam ou @PathVariable. Deve ser anotada na classe ou no método.

```
@RestController @Validated
public class UsuarioController {

   @GetMapping("/buscar")
   public ResponseEntity<String> buscarUsuario(
         @RequestParam @Pattern(regexp = "^[0-9]{10}$", message = "ID deve ter 10
dígitos") String id) {         return ResponseEntity.ok("Usuário encontrado");
   }
}
```

10. Contexto de Persistência no Spring Data JPA

O contexto de persistência monitora automaticamente as alterações feitas nas entidades carregadas. Se você modificar uma entidade que já está sendo gerenciada pelo JPA, as alterações serão salvas automaticamente sem a necessidade de chamar explicitamente o método save.

Exemplo:

```
@Override
public LoansDto update(LoansDto loansDto) {
   Loans loans = loansRepository.findLoansByLoanNumber(loansDto.getLoanNumber())
.orElseThrow(() -> new ResourceNotFoundException("Loans", "Loan Number",
loansDto.getLoanNumber()));
   LoansMapper.loanDtoToLoan(loansDto, loans);
loansRepository.save(loans); // O save aqui é opcional
return LoansMapper.loanToLoanDto(loans); }
```

11. Observação Importante sobre JPQL

Para que o JPQL funcione corretamente, a entidade deve estar anotada com @Table. Sem essa anotação, o JPQL pode não ser capaz de mapear adequadamente a entidade para uma tabela do banco de dados.

12. Mapeando Listas de Valores em uma Entidade

Para mapear listas diretamente como atributos de uma entidade sem criar uma nova tabela de entidade, usamos a anotação @ElementCollection.

Exemplo: Criando uma tabela de atores relacionados a um filme:

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

   @ElementCollection
   @CollectionTable(name = "media_actors", joinColumns = @JoinColumn(name =
"filme_id"))
   @Column(name = "atores")
   private List<String> atores;

   // Getters e Setters }
```

Explicação:

- **@ElementCollection**: Indica que o atributo é uma coleção de valores simples ou objetos embutidos, não uma entidade.
- **@CollectionTable**: Define a tabela que armazenará os valores da coleção, associando-a à entidade principal.
- **@Column**: Nome da coluna que armazenará os valores individuais da coleção.

13. Configuração spring.jpa.hibernate.ddl-auto

A configuração spring.jpa.hibernate.ddl-auto controla como o Hibernate deve lidar com o esquema do banco de dados ao iniciar a aplicação.

Os valores comuns incluem:

- **none**: Não realiza nenhuma ação.
- **create**: Cria o esquema do zero, removendo o existente.
- **create-drop**: Cria o esquema e o remove ao finalizar a aplicação.
- **update**: Atualiza o esquema existente (mas não altera ou remove colunas).

- **validate**: Valida o esquema contra as entidades mapeadas (sem alterar o banco).

Nota:

Esta configuração não afeta diretamente os dados salvos nas tabelas; ela apenas gerencia o **esquema** (tabelas, colunas, índices, etc.).

14. Uso de Classes Pai e Herança

Ao projetar classes que compartilham comportamentos ou atributos, escolha a abordagem mais adequada:

1. **Classe Pai Concreta**:

Se as classes filhas compartilham a mesma implementação de métodos, pode ser usada uma classe pai concreta que implemente esses métodos.

2. **Classe Abstrata ou Interface**:

Se as classes filhas têm implementações diferentes para os mesmos métodos, use uma classe abstrata ou interface.

15. Uso de final em Atributos

O modificador final é usado para indicar que o valor de um atributo não pode ser alterado após sua inicialização.

Recomendado para:

- Atributos que representam constantes ou propriedades imutáveis, como o modelo de um carro.

Exemplo:

```
public class Carro {
   private final String modelo;

   public Carro(String modelo) {
      this.modelo = modelo;
   }

   public String getModelo() {
      return modelo;
   }
}
```

16. Paginação e Ordenação

O Spring Data JPA oferece suporte nativo a paginação e ordenação com o uso de Pageable.

Exemplo:

```
@Repository
public interface ProdutoRepository extends JpaRepository<Produto, Long> {
Page<Produto> findByNomeContaining(String nome, Pageable pageable); }
```

Uso no Serviço:

```
Pageable pageable = PageRequest.of(0, 10, Sort.by("nome").ascending());
Page<Produto> produtos = produtoRepository.findByNomeContaining("Livro", pageable);
```

17. Anotação @Embedded

A anotação @Embedded é útil para reutilizar atributos comuns em diferentes entidades sem criar uma hierarquia de herança.

Características:

- **Composição de Entidades**: Permite incluir um objeto com atributos comuns em uma entidade.
- **Sem Identidade Própria**: O objeto embutido não possui chave primária própria e é parte da entidade principal.
- **Reutilizável**: Pode ser usado em várias entidades, reduzindo a duplicação de código.

Exemplo:

```
@Embeddable
public class Endereco {
private String rua;
private String cidade;
   private String estado;

   // Getters e Setters
}

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

   @Embedded
   private Endereco endereco;

   // Getters e Setters
}
```

18. Recuperando Relacionamentos com JPA

No JPA, é possível recuperar entidades associadas devido à presença de chaves estrangeiras, mesmo que o relacionamento não seja explicitamente salvo.

Exemplo:

Considere duas entidades: Disciplina e Assunto.

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

   @OneToMany(mappedBy = "disciplina", cascade = CascadeType.ALL)
private List<Assunto> assuntos;
}

@Entity
public class Assunto {
   @Id
   @GeneratedValue
private Long id;

   @ManyToOne
   @JoinColumn(name = "disciplina_id")
private Disciplina disciplina;
}
```

1. Crie uma Disciplina:

```
Disciplina disciplina = new Disciplina();
disciplinaRepository.save(disciplina);
```

2. Crie um Assunto associado à Disciplina: Assunto assunto = new Assunto();

```
    assunto.setDisciplina(disciplina);
assuntoRepository.save(assunto);
```

3. Recupere a Disciplina e seus Assuntos:

```
Disciplina disciplina = disciplinaRepository.findById(1L).get();
List<Assunto> assuntos = disciplina.getAssuntos(); // Recupera os assuntos associados
```

Por que isso funciona?

A associação é gerenciada pela chave estrangeira na tabela Assunto, permitindo que o Hibernate resolva a relação automaticamente.

19. FetchType: Lazy vs Eager

O FetchType define como o JPA deve carregar os relacionamentos das entidades.

1. **Lazy (Padrão para coleções):**

2. Relacionamentos são carregados somente quando acessados pela primeira vez. Isso melhora o desempenho inicial ao evitar consultas desnecessárias.

```
@OneToMany(mappedBy = "disciplina", fetch = FetchType.LAZY) private
List<Assunto> assuntos;
```

3. **Eager (Padrão para relacionamentos @ManyToOne e @OneToOne):**

Relacionamentos são carregados imediatamente com a entidade principal, mesmo que não sejam acessados.

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "disciplina_id")
private Disciplina disciplina;
```

Cuidado:

- Use FetchType.EAGER com moderação para evitar problemas de desempenho, especialmente em associações que carregam grandes volumes de dados.

20. Customização de Consultas com @Query

O Spring Data JPA permite criar consultas personalizadas com a anotação @Query.

Exemplo com JPQL:

```
@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

   @Query("SELECT u FROM Usuario u WHERE u.email = :email")
   Optional<Usuario> buscarPorEmail(@Param("email") String email);
}
```

Exemplo com SQL nativo:

```
@Query(value = "SELECT * FROM usuarios WHERE email = :email", nativeQuery = true)
Optional<Usuario> buscarPorEmail(@Param("email") String email);
```

Dica: Sempre prefira JPQL para aproveitar o suporte ao modelo orientado a objetos do JPA.

21. Projeções no Spring Data JPA

Projeções permitem retornar dados específicos em vez de toda a entidade, economizando recursos e melhorando o desempenho.

1. **Interface-Based Projections**: Crie uma interface com os campos desejados.

```
public interface UsuarioProjecao {
   String getNome();
   String getEmail(); }
```

Uso no repositório:

```
List<UsuarioProjecao> findByAtivoTrue();
```

2. **DTO-Based Projections**:

Use construtores de classes DTO para retornar dados específicos.

```
@Query("SELECT new com.exemplo.dto.UsuarioDTO(u.nome, u.email) FROM Usuario u
WHERE u.ativo = true")
List<UsuarioDTO> buscarUsuariosAtivos();
```

22. Considerações sobre Banco de Dados

1. **Indexes para Performance:**
   a. Adicione índices em colunas frequentemente usadas em filtros ou ordenações com @Index (Spring JPA 2.2+).

```
@Entity
@Table(indexes = @Index(columnList = "email"))
public class Usuario {     private String email; }
```

2. **Unique Constraints:**

Garanta valores únicos com @UniqueConstraint.

```
@Table(     uniqueConstraints = @UniqueConstraint(columnNames =
"email")
)
```

3. **Verifique as configurações de schema.sql e data.sql:**
   a. Se precisar de carregamento condicional para evitar sobrescrever dados existentes, considere spring.sql.init.mode=always ou spring.sql.init.mode=never.