**Spring Data JPA**

É mais um projeto do ecossistema Spring utilizado para trabalhar com base de dados em nossas aplicações Spring de forma mais simples.

Mas antes disso, vamos entender alguns termos/camadas quando se trata da relação entre a base de dados e nossa aplicação:

- **Spring Data JPA**: é uma camada de abstração adicional que inclui todos os recursos da especificação JPA, além dos recursos próprio, como a implementação do padrão de repositories, criação de consultas na base de dados a partir de nomes de atributos. Ele engloba todo o JPA.

- **JPA (Java Persistence API)**: é uma especificação para o mapeamento objeto-relacional em Java, ou seja, é a forma de lidarmos com todo esse mapeamento das entidades que vamos criando em forma de código para transformar tudo isso em tabelas, colunas na base de dados, podendo fazer tudo isso incluindo Notations, consultas JPQL, API’s para realizar toda essa interação com a base de dados. Ele é uma abstração, logo ele precisa de alguém que implemente-o, tendo com o Hibernate pra fazer essa função.

- **Hibernate**: É uma das implementações mais comuns e utilizadas na especificação JPA. Usando assim o JDBC por de baixo dos panos para fazer todas as transições de uma base de dados, para que seja possível iniciar as conexões, executas todas as transações/query’s.

**JPA Notations:**

Auxilia todo o mapeamento do objeto-relacional para a base de dados.

**@Entity**: Indica que a classe é uma entidade que será mapeada para uma tabela no banco de dados.

**@Table**: Especifica detalhes adicionais sobre a tabela correspondente à entidade, como o nome da tabela no banco de dados.

**@Id**: Define o atributo como chave primária da tabela.

**@GeneratedValue**: Configura como o valor da chave primária é gerado automaticamente, por exemplo, usando uma estratégia de autoincremento.

**@Column**: Mapeia um atributo da classe para uma coluna específica na tabela do banco de dados. Permite configurar detalhes como nome da coluna, nullable, tamanho máximo, entre outros.

**@Transactional**: Indica que o método deve ser executado dentro de uma transação. Isso é usado para garantir atomicidade das operações no banco de dados.

**@JoinTable**: Utilizado em relacionamentos muitos-para-muitos para especificar a tabela intermediária que será usada para unir as duas entidades.

**@JoinColumn**: Define a coluna na tabela atual que é a chave estrangeira para a entidade relacionada.

Anotações relacionadas ao tipo de relacionamento entre entidades:

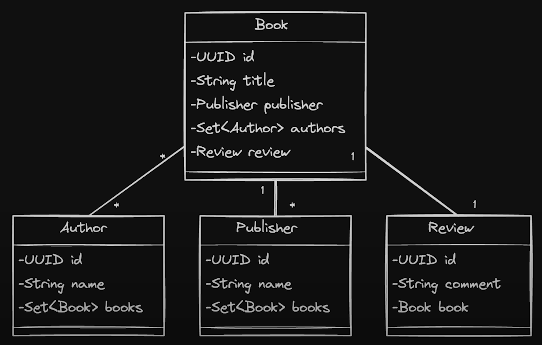
**@ManyToMany**: Define um relacionamento muitos-para-muitos entre duas entidades. Isso requer uma tabela intermediária.

**@ManyToOne**: Define um relacionamento muitos-para-um entre duas entidades. O lado "muitos" é a entidade atual.

**@OneToMany**: Define um relacionamento um-para-muitos entre duas entidades. O lado "muitos" é a entidade relacionada.

**@OneToOne**: Define um relacionamento um-para-um entre duas entidades. Cada entidade pode estar relacionada a no máximo uma instância da outra entidade.

**Projeto que vamos implementar o Spring Data JPA.**

****

**Conexão com o banco de dados:**

Toda a conexão com o BD vai ser feita no application.properties.

spring.application.name=jpa  
  
spring.datasource.url= jdbc:postgresql://localhost:5432/bookstore-jpa  
spring.datasource.username=postgres  
spring.datasource.password=victormacedo  
spring.jpa.hibernate.ddl-auto=update  
  
spring.jpa.properties.hibernate.jdbc.lob.non\_contextual\_creation=true  
spring.jpa.show-sql=true

- Definimos a URL de conexão (qual banco de dados a gente vai usar, endereço local que está instalado essa base de dados na minha máquina e o nome dela).

- Passamos algumas credenciais utilizadas na instalação do postgres (username, password)

- Sempre que iniciamos a aplicação Spring boot, automaticamente refletimos tudo que estamos representando em código Java para a base de dados. Ou seja, se criamos uma entidade em Java, criamos uma tabela na base de dados, a mesma coisa quando criamos atributos na entidade, criamos colunas na base de dados.

- Definimos que o lob do jdbc seja true, pois algumas vezes o hibernate ele vai buscar metadados do postgres e caso os metadados não esteja disponível naquele momento, ele vai gerar log de vários erros em nosso console.

- Vermos todo o SQL que está sendo gerado no nosso console todas as vezes que ele for criado, modificado e etc.

**Mapeando as entidades JPA**

Dentro do pacote principal (com.bookstore.jpa), nós criamos um outro pacote chamado “models” onde vamos inserir nossos models que vamos implementar.

Para isso, utilizamos algumas notations, sendo elas:

**@Entity** – Declaramos que essa classe é uma entidade JPA.

**@Table(name = “TB\_BOOK”)** – declaramos qual vai ser o nome dessa entidade JPA na base de dados.

**implements Serializable** – Interface de marcação para mostrarmos para a JVM que essa classe pode ser serializada e definimos também o serialVersionUID para essas serializações que forem feitas.

**@Id e @GeneratedValue(strategy = GenerationType.AUTO)** – Declaramos que aquele atributo vai ser o identificador da entidade e que o tipo do valor vai ser gerado de forma automática.

**@Column(nullable = false, unique = true)** – Declaramos que aquele atributo vai ser uma coluna com suas determinadas regras: não pode estar vazio e que tem que ser único.

**Implementação dos relacionamentos:**

**OneToMany e ManyToOne:**

Vamos seguir as seguintes lógicas:

Um livro ele pertence a somente uma editora/Publisher, já uma editora/publicher pode ter mais de um livro, logo a relação da editora com os livros é OneToMany e dos livros com a editora é ManyToOne.

**Configurando o BookModel:**

**Vai ter somente uma editora/publisher.**

@JsonProperty(access = JsonProperty.Access.*WRITE\_ONLY*)  
@ManyToOne(fetch = FetchType.*LAZY*)  
@JoinColumn(name = "publisher\_id")  
private PublisherModel publisher;

- Adicionamos a representação do relacionamento (@ManyToOne).

- Com o **@JoinColumn**, nós adicionamos uma coluna e especificamos qual vai ser a chave estrangeira da entidade “TB\_BOOK”, que nesse caso vai ser o id da entidade “TB\_PUBLISHER”, que em resumo vai ser o ID da editora do livro em questão.

Ou seja, nossa entidade “TB\_BOOK” vai ter mais uma coluna que vai ser a chave primária (ID da editora) da entidade “TB\_PUBLISHER”, que vai passar a ser uma coluna com as chaves estrangeiras.

- Iniciamos uma variável Publisher do tipo PublisherModel, que vai ser responsável por fazer essa conexão entre BookModel e PublisherModel, pois ela vai acabar sendo mapeada pela representação do relacionamento de PublisherModel. Ela vai ser tipo o One do OneToMany.

- Geramos os Getters e Setters.

**Configurando o PublisherModel:**

**Vai ter uma coleção de livros.**

@JsonProperty(access = JsonProperty.Access.*WRITE\_ONLY*)  
@OneToMany(mappedBy = "publisher", fetch = FetchType.*LAZY*)  
private Set<BookModel> books = new HashSet<>();

- Para fazermos a ligação e mapearmos a entidade BookModel junto com PublisherModel, nós passamos a variável “publisher” que iniciamos no BookModel para que ela faça todo esse mapeamento ligação das entidades.

- Definimos que uma editora pode ter vários livros, sendo que essa coleção é do tipo BookModel, por isso criamos uma “lista” books com esse tipo.  
Quando vamos trabalhar com vários relacionamentos dentro de uma mesma entidade, se usarmos List podemos ter problemas ao carregar todas essas listas/todos esses relacionamentos, pois o Hibernate ele não consegue trazer todos os relacionamentos quando definimos como lista, por isso utilizamos Set.

Ele vai ser o Many do OneToMany.

- mappedBy = “publisher” (variável iniciada em BookModel) – Quem que é o responsável pelo relacionamento.

- fetch = FetchType.LAZY – Quando nos formos buscar nossa entidade PublisherModel na base de dados, somente vamos incluir a subconsulta pra trazer quais os livros fazem parte daquela editora quando necessário, já que estamos utilizando um carregamento lento.

Porém se ele for do tipo EAGER, ele sempre vai buscar a editora na base de dados e automaticamente ele vai carregar as subconsultas pra trazer cada um dos livros que fazem parte dessa editora.

- JsonProperties – Configurar a propriedade de acesso de escrita como somente escrita, o que significa que o campo associado não será lido durante a desserialização de JSON para objetos Java, apenas durante a serialização de objetos Java para JSON.

- Geramos os Getters e Setters.

**ManyToMany:**

Vamos seguir a seguinte lógica:

Um Autor pode ter escrito vários livros e um livro pode ter mais de um autor, logo, o relacionamento vai ser muitos-para-muitos.

**Configurando o BookModel:**

@ManyToMany  
@JoinTable(  
 name = "tb\_book\_author",  
 joinColumns = @JoinColumn(name = "book\_id"),  
 inverseJoinColumns = @JoinColumn(name = "author\_id"))  
private Set<AuthorModel> authors = new HashSet<>();

Como que esse relacionamento é feito?

- Definimos uma tabela auxiliar, pois já que temos uma coleção em ambos os lados, a gente não vai conseguir criar uma coluna com uma chave estrangeira que esteja dentro de outro, como fizemos anteriormente.

Vamos ter uma tabela auxiliar que vai unir os dois ID’s dessas duas entidades.

- Para relacionar quem vai ser a chave primária e estrangeira do outro e vice versa, implementamos da seguinte forma:  
Ex: BookModel

A chave primária é “book\_id”, já a chave estrangeira é “author\_id” e vice versa.

- Sendo assim, o nosso BookModel tem uma coleção de autores, tem um Set de AuthorModel.

- Implementamos o Getter e o Setter que faltam.

**Configurando o AuthorModel:**

@JsonProperty(access = JsonProperty.Access.*WRITE\_ONLY*)  
@ManyToMany(mappedBy = "authors", fetch = FetchType.*LAZY*)  
private Set<BookModel> books = new HashSet<>();

Inserimos o JsonProperty para evitarmos alguns problemas de serialização e definimos somente permissão de escrita.

Passamos o dono desse relacionamento que é o atributos “authors” da entidade BookModel e deixamos o fetchTime como Lazy, para que sempre que a gente for buscar esses autores ele vai retornar somente a coleção de livros desse autor.

**OneToOne:**

Vamos seguir a seguinte lógica:  
Cada livro vai ter uma resenha/resumo e um resumo vai pertencer a somente um livro.

**Configurando o BookModel:**

@OneToOne(mappedBy = "book", cascade = CascadeType.*ALL*)  
private ReviewModel review;

- Mapeado por “book” que está no ReviewModel.

- **cascade = CascadeType.ALL**: Quando a gente for salvar um livro, ao mesmo tempo a gente vai poder passar internamente, sem usar necessariamente o método save, o review, pois na hora que eu crio um review e eu atribui isso a um livro, ele vai se associar automaticamente, vai ser salvo como uma cascata de associações. A mesma coisa vai ser para quando deletarmos, por exemplo.

Porém, nem sempre use isso, sempre use com cuidado, pois sem ele nós temos um controle e um cuidado muito maior.

Quando for fazer um busca na base de dados um livro/BookModel, ele vem como padrão o fetchtime EAGER.

**Configurando o ReviewModel:**

@JsonProperty(access = JsonProperty.Access.*WRITE\_ONLY*)  
@OneToOne  
@JoinColumn(name = "book\_id")  
private BookModel book;

- Adicionamos uma coluna com a chave estrangeira relacionado ao ID do livro que está sendo comentado.

- Implementamos os Getters e Setters.

**Implementação dos Repositories:**

Criamos o package Repositories e cada repository, que é uma interface, de cada entidade.

Esses repositories vão ser essenciais para criarmos pontos de injeção para a nossa base de dados e extender o nosso JpaRepository, que é ele que torna tudo mais simples em relação a transações e métodos.

public interface BookRepository extends JpaRepository<BookModel, UUID> {}

Quando estendemos o JpaRepository, nós passamos qual que vai ser a entidade que vai se referenciar com esse repository e o tipo do identificador.

Então utilizando o BookRepository, sempre que precisar, nós podemos criar métodos .save, .delete, etc.

Porém se precisarmos buscar um livro passando seu título, nós podemos fazer da seguinte forma:

BookModel findBookModelByTitle(String title);

- Passamos o tipo do retorno, digitamos o nome do método de acordo com o que queremos saber (que vai funcionar como um script SQL) e passamos uma String, que seria o título do livro, como um parâmetro.

Utilizamos quando temos uma consulta mais simples/padrão.

Ou podemos fazer utilizando o **@Query**, podendo realizar consultas nativas:

@Query(value = "select \* from tb\_book where publisher\_id = :id", nativeQuery = true)

List<BookModel> findBookModelByPublisherId(@Param("id") UUID id);

- Estamos buscando os livros de uma determinada editora, aonde passamos um determinado ID. Passamos o script SQL e native query como true.

Ele é utilizado quando precisamos fazer uma consulta mais detalhada/complexa.

Podemos utilizar de queries nativas, quanto também de consultas JPQL.

Como fizemos com o BookRepository, fazemos para as demais entidades.

Após termos feito nossos Models e Repositories, vamos interligar e testar tudo isso.

Para isso nós vamos criar 3 partes/packages: Record (DTO), Service e Controller.

**DTOs**

- Criamos o nosso DTO para dizermos o que queremos que seja passado no JSON e que futuramente irá virar um objeto Java.

public record BookRecordDto(String title,  
 UUID publisherId,  
 Set<UUID> authorIds,  
 String reviewComment) {  
}

**Services**

- Vai ser aonde a gente vai implementar o método para salvar os livros, delete e encontrar a listagem deles.

Ele vai ser um Bean do tipo **@Service**

- Vamos criar os pontos de injeção do BookRepository, AuthorRepository e PublisherRepository. Eles vão ser criados via construtor.

private final BookRepository bookRepository;  
private final AuthorRepository authorRepository;  
private final PublisherRepository publisherRepository;  
  
public BookService(BookRepository bookRepository, AuthorRepository authorRepository, PublisherRepository publisherRepository) {  
 this.bookRepository = bookRepository;  
 this.authorRepository = authorRepository;  
 this.publisherRepository = publisherRepository;  
}

Logo após isso, vamos criar os métodos citados acima.

**Método saveBook:**

- Ele vai receber o bookRecordDto (que é justamente o JSON que vamos enviar) e ele vai retornar um BookModel para o cliente pra ver que foi salvo com todos os seus atributos.

BookModel book = new BookModel();  
book.setTitle(bookRecordDto.title());

- Para isso, vai ser inicializado um objeto Java “book” do tipo BookModel e vamos setar os valores para salvar um o determinado livro, que vai ser enviado pelo bookRecordDto.

book.setPublisher(publisherRepository.findById(bookRecordDto.publisherId()).get());  
book.setAuthors(authorRepository.findAllById(bookRecordDto.authorIds()).stream().collect(Collectors.*toSet*()));

- Para setarmos a editora e os autores dos livros, eles já tem que existir na base de dados, pois quando seus ID’s são passados no JSON, nós verificamos por meio dos pontos de injeção das respectivas entidades e os métodos prontos do JPA se aqueles ID’s existem no banco de dados.

Após setarmos o título, a editora e os autores, nós vamos setar o comentário/resenha.

ReviewModel reviewModel = new ReviewModel();  
reviewModel.setComment(bookRecordDto.reviewComment());  
reviewModel.setBook(book);  
book.setReview(reviewModel);  
  
return bookRepository.save(book);

- No ReviewModel, nós iniciamos um objeto Java “reviewModel” do tipo ReviewModel, pois como está no modo CascadeType.ALL, ele também vai ser salvo assim junto com a transação de salvamento do nosso BookModel.

Com isso, setamos os atributos do ReviewModel, passando o reviewComment do bookRecordDto como setComment do objeto “reviewModel” e passamos o book do BookModel como setBook do “reviewModel”.

Com isso, nós passamos a reviewModel como o setReview do objeto “book” do BookModel.

E no final, nós retornamos o bookRepository.save(book), fazendo o salvamento tanto do nosso livro quanto do review/comentário, seguindo o cascadeType.ALL, na base de dados.

Ele também vai relacionar o livro ao publisher e aos autores que escreveram esse livro.

**Notation @Transactional:**

Ele garante um call-back/rollback para todas as transações que são feitas para a base de dados caso haja algum problema ou interrupção.

Por exemplo, não vamos salvar uma review sem existir um book para aquela review.

Para utilizarmos o método saveBook, nós vamos precisar do nosso Controller (BookController)

**- Package Controller:**

@RestController  
@RequestMapping("/bookstore/books")

-Utilizamos a notation para indicar que é um Controller e indicamos uma URI para onde vai acontecer todas as nossas atividades. E criamos também um ponto de injeção para o BookService.

E esse método de salvar, nós vamos receber no nosso @RequestBody um bookRecordDto e se tudo estiver correto, nós vamos retornar um status CREATED e no body o saveBook com recordDto que acabamos de criar no bookService.

@PostMapping  
public ResponseEntity<BookModel> saveBook(@RequestBody BookRecordDto bookRecordDto) {  
 return ResponseEntity.*status*(HttpStatus.*CREATED*).body(bookService.saveBook(bookRecordDto));  
}

Como não criamos métodos para inserir autores e editoras na base de dados, vamos inserir manualmente via query tool do PostgresQL.

insert into tb\_author values(gen\_random\_uuid(), 'Eric Evans');

insert into tb\_author values(gen\_random\_uuid(), 'Paul Deitel');

insert into tb\_author values(gen\_random\_uuid(), 'Harvey Deitel');

insert into tb\_publisher values(gen\_random\_uuid(), 'Alfa Books');

insert into tb\_publisher values(gen\_random\_uuid(), 'Person');

**Método getAllBooks:**

É um método simples que utilizamos o método findAll() no bookRepository para listar todos os livros.

public List<BookModel> getAllBooks() {  
 return bookRepository.findAll();  
}

No Controller:

@GetMapping  
public ResponseEntity<List<BookModel>> getAllBooks() {  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(bookService.getAllBooks());  
}

Quando definimos o FetchType dos nossos relacionamentos entre as entidades como LAZY, os detalhes tanto de autores quanto de editoras eles acabam sendo minimizados, ou seja, acaba só aparecendo o ID do livro, o título do livro e o review, com seu ID e o conteúdo do comentário (que por default ele já tem o carregamento EAGER/ansioso).

**Método deleteBook (considerando o cascadeType como ALL):**

@Transactional  
public void deleteBook(UUID id){  
 bookRepository.deleteById(id);  
}

- Vamos criar mais um método utilizando um dos métodos prontos do Spring Data JPA (deleteById(id)).

Utilizamos o **@Transactional** por conta do método em cascata, porque caso haja algum erro, ele já volta atrás sem deleta “pela metade”.

No Controller:

@DeleteMapping("/{id}")  
public ResponseEntity<String> deleteBook(@PathVariable UUID id) {  
 bookService.deleteBook(id);  
 return ResponseEntity.*status*(HttpStatus.*OK*).body("Book deleted successfully.");  
}

- Quando passamos o ID do livro após a URI, ele vai ser deletado juntamente com todos os seus relacionamentos que ele conseguiu formar, por exemplo no review e o autor.