4.6 Implementando um CRUD com Spring Data JPA

É comum que projetos em sua fase inicial comecem com operações básicas de criação, remoção, modificação e seleção de objetos. Para o nosso primeiro exemplo, criaremos um CRUD de *Alunos*.

A seguir temos a descrição de cada anotação da JPA que será utilizada:

- @Table: É utilizada para informar o nome da tabela relacionada a essa entidade no banco de dados;
- @Entity: Indica que a classe será mapeada e passará a ser gerenciada pela JPA;
- @Column: Nos possibilita fazer algumas configurações, como o nome da coluna no banco de dados, tamanho da coluna, obrigatoriedade de campos, etc.;
- @Id: Indica que o atributo da classe será gerenciado pela JPA e terá o seu valor automaticamente incrementado;
- @GeneratedValue: Permite que o banco utilize a estratégia de gerar automaticamente os valores da coluna id, que representa a chave primária da tabela.

```java
@Entity
public class Aluno {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer al_ra_aluno;

    @Column(nullable = false, length = 60)
    private String al_nome_aluno;

    @Column(nullable = true, length = 60)
    private String al_nome_responsavel;

    @Column(nullable = true, length = 100)
    private String al_cidade;

    @Column(nullable = false, length = 50)
    private String al_turma;

}
```

Com a classe Aluno mapeada, podemos criar a interface que será responsável por conter as ações necessárias para um CRUD de alunos.

```java
public interface AlunoRepository extends JpaRepository<Aluno, Integer>{


}
```

Pronto! Essa interface executa as operações crud sem a necessidade de escrever um método. Neste momento você pode estar pensando: Como isso é possível, se não implementamos um método?

Essa é a primeira facilidade que ganhamos quando optamos pelo Spring Data JPA. A interface CrudRepository já oferece 11 métodos diferentes para que seja possível executarmos diversas combinações de operações de CRUD. Neste momento, é necessário criar os métodos da

classe FuncionarioController, com a vantagem de não ter sido necessário criar um método na interface AlunoRepository.

O Spring Data JPA utiliza diversos padrões de projeto internamente e um deles é o próprio Repository, que permite fazer a persistência de dados de forma mais simples e intuitiva. Para que seja possível fazer tal operação de persistência de dados, o Spring Data JPA contém diversas interfaces que veremos ao longo do curso.

Depois da classe AlunoRepository implementada, poderemos utilizá-la. Criamos então o a classe de serviço:

```java
@Service
public class AlunoService {

    @Autowired
    private AlunoRepository alunoRepository;

    public List<Aluno> findAll(){
        return alunoRepository.findAll();
    }

    public Aluno buscarAluno(Integer al_ra_aluno) {
        Optional<Aluno> aluno = alunoRepository.findById(al_ra_aluno);
        return aluno.orElseThrow();
    }

    public Aluno insert(Aluno aluno) {
        return alunoRepository.save(aluno);
    }

    public Aluno update(Aluno aluno) {
        buscarAluno(aluno.getAl_ra_aluno());
        return alunoRepository.save(aluno);
    }

    public void deleteById(Integer al_ra_aluno) {
    alunoRepository.deleteById(al_ra_aluno);

    }
}
```

Com a classe de serviço implementada podemos fazer uso das rotinas do crud, é necessário uma nova classe, AlunoController, onde será utilizada as seguintes anotações:

•@Controller: Indica que a classe será um controller gerenciado pelo Spring e poderá receber chamadas web;
•@RequestMapping: Indica que o método poderá ser invocado a partir de uma URL no browser, por exemplo, por um formulário de cadastro de funcionários;
•@Autowired: Uma das anotações mais conhecidas do Spring, é usada para injetar uma dependência, comumente chamada de bean.

```java
@CrossOrigin
@RestController
@RequestMapping("escola")
public class AlunoController {

    @Autowired
    private AlunoService alunoService;

    @Autowired
    private AlunoRepository alunoRepository;

    @GetMapping("/aluno")
    public List<Aluno> getAluno() {
        List<Aluno> aluno = alunoService.findAll();
        return aluno;
    }

    @GetMapping("/aluno/{al_ra_aluno}")
    public ResponseEntity<?> buscarAlunoId(@PathVariable Integer al_ra_aluno) {
        Aluno aluno = alunoService.buscarAluno(al_ra_aluno);
        return ResponseEntity.ok().body(aluno);

    }

    @PostMapping("/aluno")
    public ResponseEntity<Void> inserirAluno(@RequestBody Aluno aluno) {

        aluno = alunoService.insert(aluno);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(aluno.getAl_ra_aluno()).toUri();

        return ResponseEntity.created(uri).build();

    }

    @PutMapping("/aluno/{al_ra_aluno}")
    public ResponseEntity<Void> atualizaAluno(@RequestBody Aluno aluno, @PathVariable Integer al_ra_aluno) {
        aluno.setAl_ra_aluno(al_ra_aluno);;
        aluno = alunoService.update(aluno);
        return ResponseEntity.noContent().build();

    }

    @DeleteMapping("/aluno/{al_ra_aluno}")
    public ResponseEntity<Void> deletarAluno(@PathVariable Integer al_ra_aluno) {
        alunoService.deleteById(al_ra_aluno);
        return ResponseEntity.noContent().build();

    }
}
```

4.7 Spring Data JPA e Query Methods

O Spring Data JPA chama de Query Methods os métodos que são adicionados na classe que implementa a interface CrudRepository e que executam uma determinada query. Esses métodos devem seguir uma convenção de nomes do próprio Spring Data JPA para que as queries sejam criadas internamente pelo Spring e executadas.

Para ilustrar o primeiro exemplo com Query Methods: retornar todos o aluno cujo nome seja “Andreu”. Essa é uma query bastante simples, pois envolve somente um comando de select na query.

Para esse código, apenas declaramos o método findByNome(). Note que não foi necessária nenhuma implementação para o método. Mas, se não existe nenhuma implementação, como o Spring Data JPA executará uma query para retornar os mesmos resultados do código da Listagem?

Assim como muitos dos projetos Spring, o Spring Data JPA utiliza convenções para montar as consultas ao banco de dados. Nesse método fizemos uso de uma das convenções de nome. Repare que em sua assinatura existe o nome findBy. Esse pequeno trecho é uma convenção do Spring que indica ao Spring Data que gostaríamos de buscar um registro baseado em algum atributo mapeado na classe Aluno.

Em seguida, finalizamos o nome do método com a palavra-chave nome. Isso significa que vamos utilizar o atributo nome da classe Aluno, ou seja, o método findBy precisa ter, também, o nome do atributo que se deseja pesquisar.

Apresentamos mais alguns métodos que retornam Alunos a partir de outros atributos da classe Aluno. Note como seria simples implementar filtros de alunos por nome, responsável e cidade. Para tanto, basta adicionar o atributo desejado para que o Spring Data crie internamente a query com o filtro.

4.7.1 Query Methods com consultas JPQL

Outra opção interessante do Spring Data JPA é que podemos escrever uma consulta personalizada em um método qualquer de um Repository. Como exemplo, criaremos o método findNomeByResponsavel(), que conterá a consulta SQL que gostaríamos de executar quando o método for invocado. Para isso, imagine que precisamos buscar um aluno pelo seu responsavel e retornar somente o seu nome, ao invés do objeto Aluno inteiro.

Explorando os recursos do Spring Data JPA, nesse código foi utilizada a anotação @Query e passamos como valor do atributo value a String que representa a consulta que deverá ser executada quando o método findNomeByResponsavel() for invocado.

Saiba, no entanto, que essa operação só é possível graças à outra anotação que usamos, a @Param, que é responsável por fazer a ligação entre a variável responsavel e o parâmetro responsavel da query.

4.7.2 Tipos possíveis de objetos retornados a partir de um Query Method

O Spring Data disponibiliza três tipos de objetos para isso:

- **Tipo básico:** O objeto retornado poderá ser um tipo básico do Java, como String, Integer, Float, etc. Ou null. Esses objetos também podem ser retornados em um objeto que implementa ou estende a interface Collection, como List, LinkedList, Set, etc.;
- **Entity:** O objeto retornado será do tipo usado no Repository. No nosso caso, a classe Funcionario;
- **Optional:** O objeto retornado poderá ser do tipo Optional, tanto do Java 8 quanto da biblioteca do Google Guava.

Basta indicar qual é o tipo de retorno desejado na assinatura do método e o Spring Data JPA fará a conversão automaticamente.

4.7.3 Tipos possíveis de retorno de um Query Method para uma lista

Quando é necessário ter uma lista de objetos como retorno, o Spring Data disponibiliza dois tipos diferentes de objetos:

- **List:** O resultado é colocado em um objeto do tipo List ou é retornado um objeto do tipo List vazio;
- **Stream:** O resultado é uma stream, um tipo de objeto adicionado ao Java 8 e que permite operações de filtro, redução, conversão em uma lista, entre outras.

A vantagem de o retorno ser uma stream é que podemos fazer operações de filtro, redução e conversão diretamente no objeto e em seguida transformá-lo em um List.

4.8 Query Methods com parâmetros nos métodos do repositório

Até agora os nossos exemplos foram simples, criando queries que só recebiam um parâmetro em seus métodos. Nos próximos casos serão criadas queries contendo duas ou mais colunas, utilizando parâmetros nas consultas. Podemos fazer uma consulta na classe de Alunos, e para buscar todos os alunos de determinada cidade e turma.

Conforme visto nos exemplos, o Spring faz uso intensivo de convenções de nomes para os métodos e parâmetros. Na operação de busca de alunos mostrada, declaramos também a palavra-chave And, entre Cidade e Turma. O Spring fará a leitura dessa palavra e, a partir disso, entenderá que deve criar uma query com a cláusula where usando a Cidade e Turma para realizar o filtro de alunos.

4.9 Parâmetros de métodos para queries nativas

Outra funcionalidade interessante do Spring Data é a possibilidade de executar queries nativas do banco de dados. Para que isso seja possível, devemos utilizar novamente a anotação @Query e escolher um dos dois tipos de passagem de parâmetro para o método: o tipo por posição ou o tipo por nome.

Para isso, o parâmetro nativeQuery é adicionado à anotação @Query para indicar que o Spring deverá executar a query como nativa. Note que o atributo value contém uma string, representando a nossa query desejada, e utiliza o símbolo de interrogação para sinalizar a posição na qual os parâmetros cidade e turma devem ser inseridos, respectivamente. Assim, a posição 1, representada pelo símbolo ?1, informa que será utilizado o parâmetro cidade, e a posição 2, representada pelo símbolo ?2, informa que será utilizado o parâmetro turma.

É válido ressaltar que os parâmetros posicionais são fáceis de utilizar e igualmente fáceis de proporcionarem erros. Imagine que em uma mudança de código, você decida trocar a ordem dos parâmetros cidade e turma no método. Como os dois são do tipo String, o código continuará compilando, mas sabemos que a funcionalidade estará errada, dado que a turma será pesquisada como cidade e vice-versa. Por esse motivo o seu uso é desencorajado, sendo preferível trabalhar com parâmetros nomeados.

Os parâmetros nomeados são mais intuitivos e com menor possibilidade de erros, já que normalmente possuem o mesmo nome que os parâmetros do método.

Nesse exemplo usamos dois parâmetros com nomes na string de consulta, :turma e :cidade, que fazem referência, respectivamente, aos parâmetros turma e cidade. Note que se trocarmos a ordem dos parâmetros no método, a query continuará funcionando. Quem assegura esse funcionamento é a anotação @Param, que é usada justamente para fazer a ligação entre o nome do parâmetro no método e na query.

4.10 Method names em Query Methods

Até o momento apresentamos diversas queries, mas em todas elas passamos no máximo dois parâmetros para os métodos. Note, ainda, que na maioria das queries o operador And do Spring Data esteve presente. Contudo, existem muitos outros operadores a serem utilizados em queries, como o Or, Group By, Order By e Like.

Dito isso, nossos próximos exemplos estarão relacionados à criação de métodos que executam queries com as seguintes características:

- Busca por alunos que esteja na cidade de São Paulo ou que esteja na turma de Robotica;
- Busca por aluno que esteja na turma que contenha a palavra “Arduino”;
- Busca pelos três primeiros alunos que morem em São Paulo.

A novidade no primeiro método é o uso da palavra de convenção Or, que será usada para criar a query com a cláusula or no where. No segundo método foi usada a convenção AllIgnoreCase, que indica que a query será case insensitive, ou seja, não levará em consideração a distinção entre letras maiúsculas e minúsculas na cláusula where. E no terceiro método usamos a

convenção First3, que limitará a quantidade de registros retornados pela query. O interessante é que basta trocar o número 3 por outros números para que o limite seja modificado.

4.11 Ordenação de dados com Spring Data JPA

Uma das operações mais comuns no dia a dia é a ordenação de uma determinada lista de objetos. Isso pode ser feito de duas maneiras: de forma ascendente e de forma descendente. Para exemplificar essas opções, vamos resolver os requisitos listados a seguir:

1. Buscar todos os Alunos e ordená-los de forma ascendente pelo seu nome;
2. Buscar todos os alunos da turma de Robotica, ordenando-os de forma ascendente pelo nome e em seguida de forma descendente pela turma.

Note como é simples o código com Spring Data JPA. Não precisamos nos preocupar com a query, pois novamente o Spring Data utiliza a convenção de código. Novas convenções de nome foram adotadas, como a findAll, para buscar todos os registros da tabela de alunos, e a convenção OrderByNome, para ordenar a lista de funcionários pelo atributo nome.

Observe que a string nome precisa ser igual ao nome do atributo nome da classe Aluno. O método contém, ainda, o termo Asc, para indicar que a ordenação será feita de forma ascendente. Se você não indicar que tipo de ordenação deseja para a sua lista, o Spring, por padrão, fará a ordenação de forma ascendente, não necessitando, portanto, que a palavra reservada Asc seja adicionada.

Para que a ordenação seja feita de forma descendente, basta adicionar o termo Desc no nome do método. Por último, utilizamos a convenção de nome findByTurma no segundo método, sinalizando que deve ser feita uma consulta com um filtro; no caso, a turma.

4.12 Mapeando Associações com Anotações

Vários tipos de associações, comuns em banco de dados, são representados pelas anotações do Hibernate. Veremos abaixo as principais associações:

Associação Um-para-Um

É possível associar entidades através de um relacionamento um-para-um usando a anotação **@OneToOne**. Existem três casos para utilização deste tipo de relacionamento: ou as entidades associadas compartilham os mesmos valores da chave primária; ou uma chave estrangeira é armazenada por uma das entidades (note que a coluna de chave estrangeira no banco de dados deve ter uma restrição de valor único para suportar a multiplicidade um-para-um); ou uma tabela

associativa é usada para armazenar o link entre as duas entidades (uma restrição de valor único deve ser definida em cada FK para garantir a multiplicidade um-para-um).

Associação Muitos-para-Um

As associações muitos-para-um são declaradas em nível de propriedades com a anotação **@ManyToOne**.

Associação Um-para-Muitos

Associações um-para-muitos são declaradas em nível de propriedades com a anotação **@OneToMany**. Como esse relacionamento é bidirecional, na classe que representa o lado “muitos”, utilizamos a anotação **@OneToMany(mappedBy=...)**.

Associação Muitos-para-Muitos

Uma associação muitos-para-muitos é definida logicamente usando a anotação @ManyToMany. Você também pode descrever uma tabela associativa e a condição de ligação usando a anotação @JoinTable. Se a associação é bidirecional, um lado poderá atualizar as informações do relacionamento na tabela associativa enquanto o outro lado será ignorado em tal atualização.