Criando uma aplicação *back-end* com Spring 3

1 Mapeando relacionamentos entre tabelas

Nós sabemos que, como o próprio nome deixa claro, bancos de dados modelo relacional possuem vários relacionamentos entre tabelas. Partiremos agora para o mapeamento em classes de relacionamentos dos tipos 1-N e N-N.

1.1 Relacionamento 1-N (@OneToMany e @ManyToOne)

Vamos começar pelo mapeamento da tabela de anotações, "ant_anotacao", visualizado em Código 1 (sem getters e setters, que precisam ser gerados). Essa tabela, ao contrário da anterior possui uma coluna com restrição de chave estrangeira, "ant_usr_id", que referencia o usuário que cadastrou a anotação. Reparem que, apesar da coluna ser do tipo Long (BigInt), a mapeamos com um atributo do tipo "Usuario". Isso acontece porque estamos trabalhando com orientação a objeto e o correto é incluir a classe completa em uma referência. Para mapear essa referência usaremos duas novas anotações:

- @JoinColumn – Indica o nome da coluna que faz referência à tabela associada à classe "Usuario" (nesse caso "ant_usr_id");
- @ManyToOne – Indica que o relacionamento entre as classes é muitos (N) para um (1). Ou seja, cada anotação está associada a somente um usuário, mas cada usuário pode ter várias anotações. O parâmetro "fetch" indica a forma como os dados do usuário serão recuperados. Com valor "FetchType.EAGER" sempre buscaremos os dados do usuário associado ao buscar uma anotação. Por outro lado, o valor "FetchType.LAZY" faz com que os dados do usuário somente sejam buscados se algum código tentar acessar o atributo anotado ("usuario"). O valor padrão é "FetchType.LAZY", visto que a outra opção pode causar sérios problemas de desempenho. Nesse caso, optamos por definir como "FetchType.EAGER", pois sempre precisaremos dos dados do usuário ao exibir uma anotação.

Código 1 – Mapeamento da tabela de anotação

```
package br.gov.sp.fatec.springboot3app2025.entity;

import java.time.LocalDateTime;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "ant_anotacao")
public class Anotacao {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ant_id")
    private Long id;

    @Column(name = "ant_texto")
    private String texto;

    @Column(name = "ant_data_hora")
    private LocalDateTime dataHora;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "ant_usr_id")
    private Usuario usuario;
```

```
    // Coloque aqui Getters e Setters
    .
    .
    .

}
```

Nós também podemos mapear as anotações na classe "Usuario", conforme Código 2. Reparem que aqui a única anotação necessária é a @OneToMany (o par correspondente a @ManyToOne, que usamos em "Anotacao"). Examente por essa anotação, o atributo aqui consiste em uma lista de "Anotacao", visto que um usuário pode ter várias anotações associadas. Entretanto, com relação ao mapeamento, não temos na tabela "usr_usuario" uma coluna que permita descobrir as anotações associadas. Por esse motivo, precisamos utilizar o parâmetro "mappedBy", para indicar onde se encontra mapeado o relacionamento entre essas tabelas. Como valor desse parâmetro, precisamos indicar o nome do atributo, na classe "Anotacao", que contém o mapeamento. Nesse caso específico, o nome desse atributo é "usuario", conforme mostrado em Código 2. Como utilizamos o valor "FetchType.EAGER" para o parâmetro "fetch" no outro lado da relação, aqui somos obrigados a utilizar o padrão, "FetchType.LAZY". Usar "**FetchType.EAGER**" em ambos os lados não é recomendado, pois pode acarretar em sérios problemas de desempenho, incluindo **"loop" infinito** (neste exemplo, um usuário tem anotações que, por sua vez, têm usuários, que têm anotações, etc). Da mesma forma, quando temos mapeamento do relacionamento em ambos os lados, precisamos impedir que um dos lados apareça no JSON, ou também teremos um erro de **"loop" infinito**. Para isso, usaremos **@JsonProperty** em um dos lados (existem outras formas, mas essa é a mais simples). Escolhemos usá-lo no lado do usuário, pois geralmente não nos interessa saber as anotações associadas ao buscar os dados de um usuário.

Código 2 – Mapeamento atualizado da tabela de usuário

```
package br.gov.sp.fatec.springboot3app2025.entity;

import java.util.List;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToMany;
import jakarta.persistence.Table;
import com.fasterxml.jackson.annotation.JsonProperty;

@Entity
@Table(name = "usr_usuario")
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "usr_id")
    private Long id;

    @Column(name = "usr_nome")
    private String nome;

    @Column(name = "usr_senha")
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private String senha;

    @OneToMany(mappedBy = "usuario", fetch = FetchType.LAZY)
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private List<Anotacao> anotacoes;

    // Coloque aqui Construtores, Getters e Setters
    .
    .
    .

}
```

1.2 Relacionamento N-N (@ManyToMany)

As tabelas "usr_usuario" e "aut_autorizacao" são relacionadas por meio de uma tabela de ligação, "uau_usuario_autorizacao", em um relacionamento N-N. Primeiramente, vamos mapear a tabela "aut_autorizacao" na classe "Autorizacao", conforme Código 3.

Código 3 – Mapeamento da tabela de autorização

```
package br.gov.sp.fatec.springboot3app2025.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "aut_autorizacao")
public class Autorizacao {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "aut_id")
    private Long id;

    @Column(name = "aut_nome")
    private String nome;

    // Coloque aqui Getters e Setters
    .
    .
    .

}
```

Ao contrário do que se possa esperar, não é necessário mapear a tabela de ligação, "uau_usuario_autorizacao", em uma classe, pois ela não possui nenhum atributo próprio. O mapeamento dessa tabela ocorrerá junto da definição do relacionamento que, no caso N-N, pode ocorrer em qualquer uma das classes envolvidas. Como sempre precisaremos saber as autorizações de um usuário e dificilmente teremos interesse em saber quais usuários estão associados a uma data autorização, iremos realizar o mapeamento na classe "Usuario", conforme Código 4. Reparem que o atributo que mapeia o relacionamento, "autorizacoes", consiste em uma lista de "Anotacao", pois um usuário pode estar associado a várias anotações.

Código 4 – Mapeamento mais atualizado da tabela de usuário

```
package br.gov.sp.fatec.springboot3app2025.entity;

import java.util.List;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.OneToMany;
import jakarta.persistence.Table;

import com.fasterxml.jackson.annotation.JsonProperty;

@Entity
@Table(name = "usr_usuario")
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "usr_id")
```

```
    private Long id;

    @Column(name = "usr_nome")
    private String nome;

    @Column(name = "usr_senha")
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private String senha;

    @OneToMany(mappedBy = "usuario", fetch = FetchType.LAZY)
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private List<Anotacao> anotacoes;

    @ManyToMany
    @JoinTable(name = "uau_usuario_autorizacao",
        joinColumns = {@JoinColumn(name = "usr_id")},
        inverseJoinColumns = {@JoinColumn(name = "aut_id")})
    private List<Autorizacao> autorizacoes;

    // Coloque aqui Getters e Setters
    .
    .
    .

}
```

Aqui usamos duas novas anotações:

- @ManyToMany – Essa anotação indica que o relacionamento é do tipo N-N, onde cada classe pode estar associada a várias diferentes instâncias da outra classe. Aqui decidimos por não informar o parâmetro "fetch", o que indica que seu valor é, por padrão, "FetchType.LAZY";
- @JoinTable – Essa anotação nos ajuda a descrever a tabela que proporciona o relacionamento entre "Usuario" e "Autorizacao". No parâmetro "name" informamos o nome da tabela de ligação. Em "joinColumns" informamos o nome da coluna na tabela de ligação que possui uma chave estrangeira referenciando a classe atual ("Usuario"), "usr_id". Por último, em "inverseJoinColumns", informamos o nome da coluna na tabela de ligação que possui uma chave estrangeira referenciando a classe com a qual nos associaremos ("Autorizacao"), "aut_id".

Da mesma forma que fizemos com o relacionamento 1-N, também podemos fazer o mapeamento do relacionamento na classe "Autorizacao", conforme Código 5. Reparem que no parâmetro "mappedBy" informamos o atributo "autorizacoes" da classe "Usuario" como o responsável pelo mapeamento.

Código 5 – Mapeamento atualizado da tabela de autorização

```
package br.gov.sp.fatec.springboot3app2025.entity;

import java.util.List;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.Table;
import com.fasterxml.jackson.annotation.JsonProperty;

@Entity
@Table(name = "aut_autorizacao")
public class Autorizacao {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "aut_id")
    private Long id;

    @Column(name = "aut_nome")
    private String nome;

    @ManyToMany(mappedBy = "autorizacoes")
```

```
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private List<Usuario> usuarios;

    // Coloque aqui Getters e Setters
    .
    .
    .

}
```

Também criaremos um repositório para essa entidade, apresentado em Código 6, que será utilizado na próxima Seção. O método *findByNome* é uma consulta que realiza uma busca com base no atributo "nome" (verifiquem a Seção 1.4 para mais detalhes sobre essa estrutura de consulta).

Código 6 – Repositório para a entidade autorização

```
package br.gov.sp.fatec.springboot3app2025.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import br.gov.sp.fatec.springboot3app2025.entity.Autorizacao;

public interface AutorizacaoRepository extends JpaRepository<Autorizacao, Long>{

    public Optional<Autorizacao> findByNome(String nome);

}
```

1.3 Como cadastrar um relacionamento no Banco de Dados

Uma dúvida comum ao se lidar com mapeamento objeto-relacional (de tabelas modelo relacional para orientação a objetos) é "se eu não mapeio diretamente as colunas que possuem chave estrangeira, como vou preenchê-las ao inserir um novo registro?". Esse processo é realizado automaticamente, mas exige alguns cuidados. O Código 6 apresenta um exemplo de método, que associa um novo usuário a uma autorização. Reparem que eu preciso buscar a autorização no banco (ou criar uma nova se não existir), incluir no atributo "autorizacoes" e depois salvar. Ao buscar no Banco de Dados ou criar uma nova autorização eu garanto que o objeto possui o atributo "id" preenchido e está sincronizado com o registro correspondente. Isso faz como que, ao salvar meu usuário, a tabela de ligação ou chave estrangeira (no caso de @ManyToOne) sejam preenchidas automaticamente.

Importante: O preenchimento automatico somente ocorre se você incluir um objeto que você recuperou do Banco de Dados em um atributo anotado com @JoinColumn ou @JoinTable e depois salvar.

Código 7 – Método para cadastrar novos usuários

```
package br.gov.sp.fatec.springboot3app2025.service;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.springframework.http.HttpStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.server.ResponseStatusException;
import org.springframework.transaction.annotation.Transactional;

import br.gov.sp.fatec.springboot3app2025.entity.Autorizacao;
import br.gov.sp.fatec.springboot3app2025.entity.Usuario;
import br.gov.sp.fatec.springboot3app2025.repository.AutorizacaoRepository;
import br.gov.sp.fatec.springboot3app2025.repository.UsuarioRepository;

@Service
public class UsuarioService implements IUsuarioService{

    @Autowired
    private UsuarioRepository usuarioRepo;
```

```
    @Autowired
    private AutorizacaoRepository autRepo;

    @Transactional
    public Usuario novoUsuarioAutorizacao(String nome,
            String senha, String nomeAutorizacao) {
        Usuario usuario = new Usuario(nome, senha);
        Optional<Autorizacao> autOp = autRepo.findByNome(nomeAutorizacao);
        Autorizacao autorizacao;
        if(autOp.isEmpty()) {
            autorizacao = new Autorizacao();
            autorizacao.setNome(nomeAutorizacao);
            autRepo.save(autorizacao);
        }
        else {
            autorizacao = autOp.get();
        }
        usuario.setAutorizacoes(new ArrayList<Autorizacao>());
        usuario.getAutorizacoes().add(autorizacao);
        return usuarioRepo.save(usuario);
    }

    // Demais métodos
    .
    .
    .
}
```

Nesse exemplo também incluímos uma anotação nova: @Transactional. Essa anotação cria uma transação de Banco de Dados que garante que qualquer alteração somente será gravada fisicamente em disco se o método finalizar sem erros. Reparem que temos duas instruções "save" no método: uma para a autorização e outra para o usuário. Sem a anotação @Transactional, se ocorresse um erro ao salvar o usuário, a anotação previamente salva ficaria no Banco de Dados, gerando uma inconsistência de dados (um cadastro pela metade). **Sempre que executarem mais de uma alteração no Banco de Dados em um método, usem a anotação @Transactional.**

1.4 Consultas

Repositórios criados com Spring Data JPA são apenas interfaces, portanto não há como programar consultas neles. A criação de consultas é realizada de duas formas: query methods ou por meio da anotação @Query.

Vamos começar com Query Methods. De forma bem resumida, o Spring Boot JPA usa o nome do método declarado para gerar uma consulta. Para tanto, o nome precisa seguir um padrão no formato **<prefixo><modificador>By<condições>**, onde:

- O **prefixo** indica a finalidade da consulta. Prefixos "find", "get", "query", "read" e "search" são usados para retornar os resultados da consulta (todos tem a mesma função). O prefixo "count" realiza a consulta e retorna um número inteiro, corresponente à quantidade de registros. O prefixo "exists" retorna um booleano indicando se a consulta retorna resultados ou não. Os prefixos "delete" e "remove" excluem do Banco de Dados todos os registros retornados pela consulta;
- O **modificador** é um elemento opcional e serve para limitar os resultados. Podemos, por exemplo, usar "TopN" ou "FirstN" para trazer apenas os N primeiros registros da consulta (substitua N pelo valor desejado, como em "Top5");
- As **condições** seguem o formato **<nome do atributo><comparador>**. Exemplo: Para a classe "Usuario", seriam condições válidas "SenhaEquals" ou "SenhaContains". É importante ressaltar que o comparador "Equals" é opcional, o que permite utilizar apenas o nome do atributo como condição (no exemplo, apenas "Senha"). Podemos usar os operadores "And" e "Or" para incluir mais condições. Exemplo: "NomeAndSenha" (equivalente a "NomeEqualsAndSenhaEquals").

Com relação ao retorno para os prefixos "find", "get", "query", "read" e "search" podemos ter 3 tipos:

- Uma instância da classe – Deve ser usado quando a consulta somente pode retornar um registro. Isso ocorre quando pesquisamos por uma chave primária ou única ou quanto usamos o modificador "Top1". Quando o registro não é encontrado, o método retorna "null". Considerando o repositório para a classe "Usuario", um exemplo seria: "public Usuario findById(Long id);";
- Um Optional da classe – Usado nos mesmos casos do item anterior, mas retorna um Optional.empty() se não encontrou nenhum registro. Utilizando o mesmo exemplo, o query method seria: "public Optional<Usuario> findById(Long id);";
- Um List com elementos da classe – Deve ser usado quando a consulta pode retornar mais de um registro. Considerando novamente o repositório para a classe "Usuario", um exemplo seria: "public List<Usuario> findByNomeContains(String nome);".

Também podemos utilizar o conceito de "JOIN" em nossas consultas, graças aos mapeamentos que realizamos. Para o repositório da classe "Usuario" eu poderia buscar usuários com permissões de um determinado nome com o query method "public List<Usuario> findByAutorizacoesNome(String nome);". Reparem que aqui eu uso o atributo "autorizacoes", que liga as classes "Usuario" a "Autorizacao", e comparo o atributo interno "nome" (da classe "Autorizacao" e não da classe "Usuario"). O SQL equivalente ao query method apresentado seria: "select u.* from usr_usuario u join uau_usuario_autorizacao ua on u.usr_id = ua.usr_id join aut_autorizacao a on ua.aut_id = a.aut_id where a.aut_nome = ?".
Mais detalhes sobre query methods podem ser encontrados em: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repository-query-keywords.
A anotação @Query, por outro lado, não exige que o método possua um nome específico. Basta incluir uma instrução JPQL como parâmetro. Mais informações em: https://www.baeldung.com/spring-data-jpa-query.
Código 7 apresenta o repositório da classe "Usuario", "UsuarioRepository", com algumas consultas. Cada consulta possui uma versão em query method e uma versão correspondente com a anotação @Query. Reparem que, na anotação @Query, nós utilizamos "?" para indicar um parâmetro, onde "?1" corresponde ao primeiro parâmetro, "?2" corresponde ao segundo parâmetro e assim por diante. Na última consulta com a anotação @Query utilizamos uma forma diferente de referenciar o parâmetro, ":nome". Quando nomeamos um parâmetro, se faz necessário utilizar a anotação @Param para indicar qual o parâmetro correspondente (neste caso a ordem não importa).

Código 8 – UsuarioRepository

```
package br.gov.sp.fatec.springboot3app2025.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import br.gov.sp.fatec.springboot3app2025.entity.Usuario;

public interface UsuarioRepository extends JpaRepository<Usuario, Long>{

    public Usuario findByNome(String nome);

    @Query("select u from Usuario u where u.nome = ?1")
    public Usuario buscarPeloNome(String nome);

    public List<Usuario> findByNomeContains(String nome);

    @Query("select u from Usuario u where u.nome like %?1%")
    public List<Usuario> buscarPeloNomeContendo(String nome);

    public Usuario findByNomeAndSenha(String nome, String senha);

    @Query("select u from Usuario u where u.nome = ?1 and u.senha = ?2")
    public Usuario buscarPeloNomeESenha(String nome, String senha);

    public List<Usuario> findByAutorizacoesNome(String nomeAutorizacao);
```

```
    @Query("select u from Usuario u join u.autorizacoes a where a.nome = :nome")
    public List<Usuario>
        buscarPeloNomeAutorizacao(@Param("nome") String nomeAutorizacao);

}
```