



Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Curso de Bacharelado em Ciência da Computação

Técnicas de Programação Web

Spring Boot com JPA – Guia de Referência

Atualizado em 27/01/2022

Sumário

Spring Data	4
Mapeamento Objeto-Relacional	4
Nomes de tabelas e campos x nomes de classes e atributos	5
Tipos de campos x tipos de atributos	6
Incluindo Spring Data JPA num projeto Spring Boot	7
Criação de projeto Spring Boot com JPA no Initializr	7
Inclusão de dependência do Spring Data JPA num projeto Spring Boot já existente	8
Configuração de conexão com o SGBD	9
Configurando a exibição das instruções SQL	9
Configuração de atualização/validação da estrutura das tabelas	10
Anotações JPA e Hibernate	11
@Entity (JPA)	13
@Table (JPA)	13
@Id (JPA)	14
@GeneratedValue (JPA)	14
@SequenceGenerator (JPA)	16
@TableGenerator (JPA)	16
@Column (JPA)	17
@Temporal (JPA)	18
@Enumerated(JPA)	19
@CreationTimestamp (Hibernate)	19
@UpdateTimestamp(Hibernate)	20
@Formula (Hibernate)	20
Interfaces Repository	21
Usando uma Repository numa Service ou Controller	22
Prática de Repository com Controller	25
Criando métodos no Repository	25
Criando métodos padronizados na Repository	26
Nome da operação	27

findBy	27
findAllBy	27
countBy	27
existsBy	28
deleteBy	28
Nome de atributo	29
Condição de atributo	29
Validação automática dos métodos de Repository	31

Spring Data

O acesso a bancos de dados, sejam relacionais, sejam NoSQL é comum na maioria dos sistemas corporativos. Para ter acesso a esses tipos de sistemas, exista o **Spring Data**, que é um conjunto de bibliotecas Spring que tem como objetivo abstrair o uso da **JDBC** e facilitar o acesso a bases de dados de vários fabricantes e naturezas. A Figura 1 ilustra a composição do Spring Data.

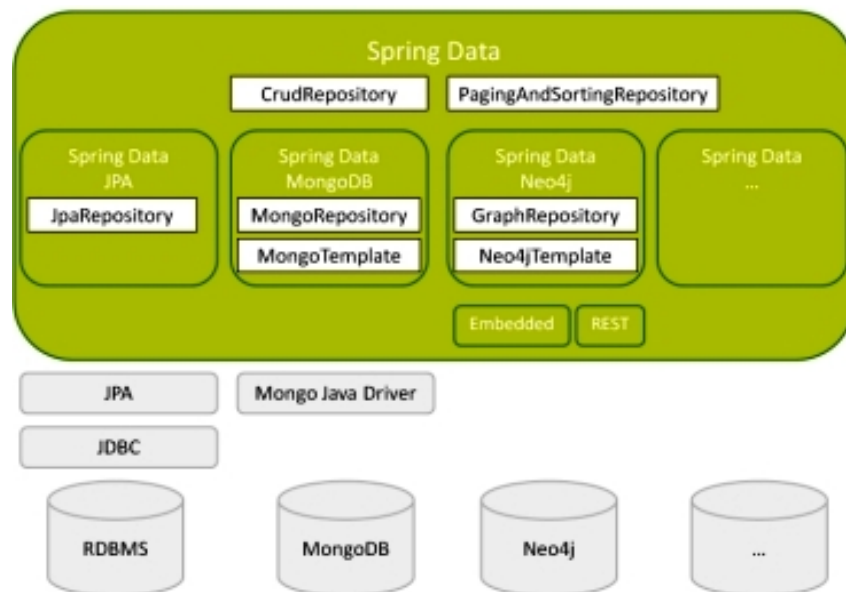


Figura 1: Componentes do Spring Data

Aqui trataremos apenas do **Spring Data JPA**, que promove o uso da técnica do **Mapeamento Objeto-Relacional** para facilitar o trabalho com sistemas de bancos de dados relacionais. A seguir, veremos do que se trata essa técnica.

Mapeamento Objeto-Relacional

O **“Mapeamento Objeto-Relacional”** (do original em inglês “Object-Relational Mapping”) é uma técnica que permite “espelhar” as tabelas de um banco de dados relacional em classes de uma linguagem de programação que seja orientada a objetos, como é o caso de Java. Esse processo pode ser feito a partir de tabelas novas, criadas junto com o projeto Java ou a partir de tabelas legadas, ou seja, já existentes e usadas até por sistemas antigos.

Essa técnica facilita muito o trabalho em projetos com muitas tabelas, pois torna mais fácil as manutenções evolutiva e corretiva. Isso se deve ao fato de ficarem explícitas a quantidade de tabelas, seus campos, suas chaves primárias e os relacionamentos. Por fim, essa abordagem facilita a criação de testes unitários automatizados.

Nomes de tabelas e campos x nomes de classes e atributos

Durante o processo de ORM, é importante respeitar as diferenças de convenções de nomes entre o mundo dos bancos de dados relacionais e o mundo das linguagens de programação orientadas a objeto, como é o caso do Java.

Bancos de dados: **snake_case**

Nos bancos de dados relacionais é muito comum os nomes das tabelas e de seus campos seguirem o padrão **snake_case**, ou seja, todas as letras minúsculas e palavras separadas por *underline* (“_”) ao invés de espaço em branco. Exemplos: tabela **“tipo_estabelecimento”**. Campo **“id_tipo_estabelecimento”**.

Java: **camelCase** ou **PascalCase**

Já na linguagem Java, os padrões são o **camelCase** e o **PascalCase**. No **camelCase** a primeira letra é minúscula e no **PascalCase**, maiúscula. Não há nenhum caractere separador entre as palavras, mas cada palavra nova inicia com letra maiúscula. No caso de **classes**, **interfaces** e **enums**, usamos o **PascalCase** e para os demais identificadores, usamos **camelCase**. Exemplos: Classe **TipoEstabelecimento** (PascalCase). Atributo **idTipoEstabelecimento** (camelCase). Método **public void salvar()** (camelCase).

Outro ponto é que o mapeamento objeto relacional não deve ser uma mera tradução **snake_case** para **camelCase**. É comum algumas tabelas terem algum prefixo no nome (“tb” ou “tab” ou “tbl”, etc.) para deixar claro que são tabelas. Algo parecido ocorre com campos: muitas vezes possuem um prefixo que indica o tipo de dado (“fl” para flag, “dt” para data, “ds” para descrição, etc.), além de possuírem um sufixo, que normalmente é o próprio nome da tabela. Assim, um campo de “data de criação de um tipo de tarifa” numa tabela “tipo_tarifa”, poderia ser algo como **“dt_criacao_estabelecimento”**.

Esses prefixos e sufixos ocorre principalmente quando os nomes das tabelas e campos foram definidos por profissionais chamados **DA** (*Data Administrador*) ou **DBA** (*Database Administrador*). Eles preferem usar esses prefixos e sufixos pois facilitam suas tarefas de administração e segurança dos bancos de dados. **Importante:** costumamos ignorar esses prefixos e sufixos nas classes Java e seus atributos quando fazemos o ORM, pois os fatores que motivam seu uso nos bancos de dados não existem em projetos Java.

Existe ainda a possibilidade de tabelas e campos terem nomes totalmente diferentes de suas finalidades, aparentemente até aleatórios. Isso costuma ser feito em sistemas cujos dados possuem alto grau de confidencialidade, como dados de investidores milionários em um sistema de custódia, por exemplo.

No Quadro 1 temos um exemplo de como os nomes de uma tabela e seus campos poderiam ficar mapeados numa classe Java.

Tabela	Classe
tipo_estabelecimento	TipoEstabelecimento
id_tipo_estabelecimento	id
nome_estabelecimento	nome
dt_inauguracao	dataInauguracao

Quadro 1: Nomes de tabela campos e seus campos numa classe Java

Tipos de campos x tipos de atributos

Outro ponto muito importante no processo de ORM, é saber que há uma relação entre os tipos de campos usados pelos bancos de dados relacionais com os tipos em Java. Às vezes um mesmo tipo de campo no banco pode ser representados por diferentes classes ou tipos primitivos. O Quadro 2 mostra a relação dos principais tipos de campos com seus possíveis tipos em Java.

Tipo de campo	Classes ou tipos primitivos Java
VARCHAR	String
INT	Integer, int, Long ou lng
NUMBER	Integer, int, Long, long, Double, double ou BigDecimal
DATE	LocalDate, LocalDateTime,, Date ou Calendar
TIMESTAMP	LocalDate, LocalDateTime, Date ou Calendar
BYTEA	Byte[] ou byte[]
BLOB	Byte[] ou byte[]
CLOB	String

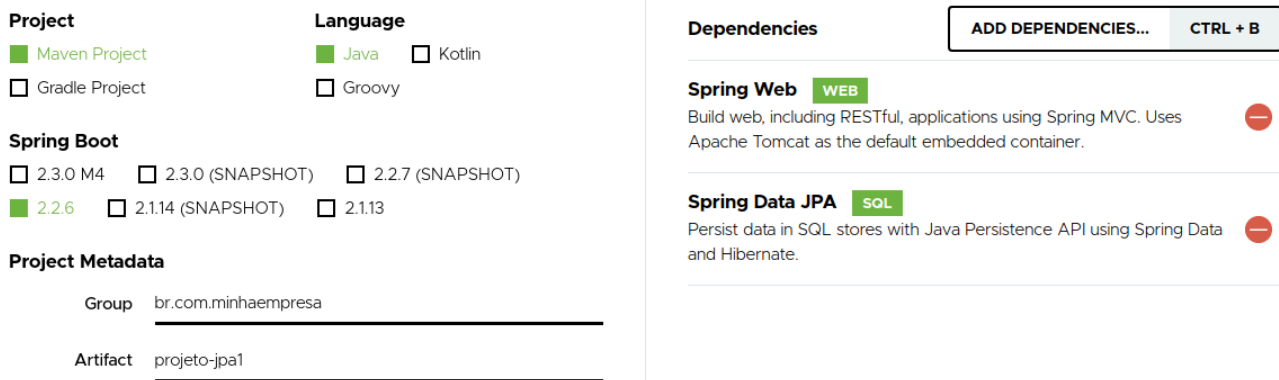
Quadro 2: Tipos de campos x Classes Java

Incluindo Spring Data JPA num projeto Spring Boot

É possível definir, já na criação, que um projeto vai usar Spring Data com JPA ou incluir uma dependência no pom.xml no caso de um projeto já existente. A seguir, vejamos como proceder em ambos os casos.

Criação de projeto Spring Boot com JPA no Initializr

Ao usar o Spring Initializr para criar um projeto Spring Boot com Data JPA, basta marcar, além da opção **Web**, a opção **JPA** no item **Dependencies**. Vide a Figura 2.



The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven Project' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', version '2.2.6' is selected. The 'Project Metadata' section shows 'Group' as 'br.com.minhaempresa' and 'Artifact' as 'projeto-jpa1'. On the right, the 'Dependencies' section shows 'Spring Web' (WEB) and 'Spring Data JPA' (SQL) selected. A button 'ADD DEPENDENCIES...' and a keyboard shortcut 'CTRL + B' are visible at the top right of the dependencies section.

Figura 2: Criação de projeto Spring Boot com JPA no Spring Initializr

Inclusão de dependência do Spring Data JPA num projeto Spring Boot já existente

Para incluir o Spring Data JPA num projeto Spring Boot já existente, basta incluir a seguinte dependência no **pom.xml** do projeto.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Além da dependência do Data JPA, normalmente incluímos uma dependência do driver de algum banco de dados também ser incluída. O código fonte de um **pom.xml** a seguir tem alguns exemplos de dependências de drivers de alguns dos principais bancos do mercado.

```
<!-- H2 -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>

<!-- MySQL e MariaDB -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
```



```
</dependency>

<!-- SQL Server -->
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
</dependency>

<!-- PostgreSQL -->
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
```

Configuração de conexão com o SGBD

Quando chegar o momento de acessar um SGBD “real” como MySQL, PostgreSQL, Oracle etc, além da dependência do respectivo driver, será necessário incluir as seguintes configurações no **application.properties** do projeto:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=root
spring.datasource.password=admin

spring.datasource.url=jdbc:mysql://localhost:3306/meuBanco
```

A configuração **spring.datasource.driver-class-name** é a classe do Driver JDBC. A **spring.datasource.username** é o login de acesso ao SGBD e a **spring.datasource.password** é a senha. Por fim, a **spring.datasource.url** é a URL de acesso ao SGBD.

Configurando a exibição das instruções SQL

Pode ser muito prático visualizar as instruções SQL que o JPA cria e envia ao SGBD. Para liberar essa funcionalidade, basta incluir a seguinte configuração no **application.properties**:

```
spring.jpa.show-sql=true
```

Com essa configuração, as instruções serão exibidas no log de execução do Spring Boot sempre que forem enviadas ao SGBD, como no log de exemplo a seguir.

Hibernate:

```
select      estabelecimento_id_estabelecimento      as      id_estab1_0_,
estabelecimento_cnpj_estabelecimento  as  cnpj_est2_0_,  estabelecimento_dh_criacao  as
dh_criac3_0_,  estabelecimento_nome_estabelecimento  as  nome_est4_0_  from
tbl_estabelecimento estabelecimento_ where estabelecimento_nome_estabelecimento=?
```

Essa configuração, como você pode ver no log de exemplo, mostra as instruções em linhas um tanto grandes. É possível solicitar ao JPA que “formate” as instruções antes de enviá-las. Para tal, temos que usar a seguinte configuração no **application.properties**:

```
spring.jpa.properties.hibernate.format_sql=true
```

Essa configuração faz com que as instruções SQL sejam exibidas como no log de exemplo a seguir.

Hibernate:

```
select
    estabelecimento_id_estabelecimento as id_estab1_0_,
    estabelecimento_cnpj_estabelecimento as cnpj_est2_0_,
    estabelecimento_dh_criacao as dh_criac3_0_,
    estabelecimento_nome_estabelecimento as nome_est4_0_
from
    tbl_estabelecimento estabelecimento_
where
    estabelecimento_nome_estabelecimento=?
```

Configuração de atualização/validação da estrutura das tabelas

Também é possível configurar o Hibernate para atuar na atualização ou validação da estrutura das tabelas envolvidas no mapeamento objeto-relacional do projeto. Para isso, podemos usar a seguinte opção no **application.properties**.

```
spring.jpa.hibernate.ddl-auto=update
```

Os valores possíveis para a configuração **spring.jpa.hibernate.ddl-auto** são **create**, **update**, **create-drop**, **validate** e **none**. A descrição delas está a seguir.

create. O Hibernate primeiro exclui todas as tabelas existentes e depois cria novas tabelas.

update. O mapeamento objeto-relacional configurado é comparado com o esquema do banco de dados e, em seguida, o Hibernate atualiza o esquema de acordo com as diferenças encontradas. **Atenção!** Tabelas ou colunas existentes e não mapeados nunca são excluídos.

create-drop. Atua quase como o create. A diferença é que o esquema é excluído do banco de dados após o encerramento da aplicação. É muito comum seu uso em testes de integração automatizados. Se **nenhuma informação de conexão** com banco for feita essa será a **opção padrão**, pois, nesse caso, o Spring Boot tentará usar um banco de dados em memória.

validate. Com essa opção, o Hibernate apenas compara o esquema do banco de dados com o mapeamento objeto-relacional do projeto. Em havendo divergências (ex: um campo está como varchar na tabela mas Double na entidade), será lançada uma exceção durante a inicialização do projeto. É muito comum seu uso em ambientes de homologação e produção.

none. Essa opção simplesmente indica que nada deve ser feito. Nem validações, nem atualizações, nem exclusões. Caso existam informações de conexão com banco de dados configuradas, essa será a **opção padrão**.

Anotações JPA e Hibernate

Em Java, as tecnologias mais usadas para esse mapeamento são o **JPA** (Java Persistence API), que é apenas uma especificação, e o **Hibernate**, que é um *framework* que implementa essa especificação. A seguir, veremos os primeiros detalhes do funcionamento deles.

Para implementar o ORM com JPA e Hibernate, a técnica mais usada atualmente é incluir **Anotações** (*Annotations*) nas classes que vão “espelhar” as tabelas do banco de dados. Algumas anotações ficam sobre a classe e outras sobre os atributos. É possível fazer o mapeamento com arquivos XML, porém é uma técnica muito pouco utilizada atualmente e não será abordada aqui.

Quando uma classe Java é mapeada para uma tabela, seja com o uso de anotações, seja com XML, podemos chamar essa classe de **Entidade**. Os códigos fonte a seguir contém exemplos de entidades na quais foram usadas anotações JPA e Hibernate.

```
@Entity
public class Estabelecimento {

    @Id
    @GeneratedValue
    private Integer id;

    private String nome;

    private String cnpj;

    // construtores, getters e setters

}
```

```
@Entity
@Table(name = "tbl_estabelecimento")
public class Estabelecimento {

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

    @Column(name = "nome_estabelecimento", length = 50)
    private String nome;
```

```

@Column(name = "cnpj_estabelecimento", length = 14, unique=true)
private String cnpj;

@CreationTimestamp
@Temporal(TemporalType.TIMESTAMP)
@Column(name = "dh_criacao")
private LocalDate dataCriacao;

@Formula("(select avg(a.nota)+1 from tlb_avaliacao a where
a.id_estabelecimento = id_estabelecimento)")
private Double mediaAvaliacoes;

// construtores, getters e setters
}

```

A seguir, as principais anotações para ORM do JPA e Hibernate serão explicadas.

@Entity (JPA)

Nome completo: *javax.persistence.Entity*

Objetivo: Indicar que a classe será usada para mapear uma tabela do banco de dados.

Onde deve ser incluída: Sobre a classe somente. **Obrigatória.**

Atributos: Não possui.

Como atuou no exemplo: Indicou para o JPA que a classe é uma Entidade ORM, ou seja, que serve para mapear uma determinada tabela do banco de dados.

@Table (JPA)

Nome completo: *javax.persistence.Table*

Objetivo: Configura as informações da tabela que está sendo “espelhada” na classe. Se essa anotações for omitida, o JPA vai procurar uma tabela com o EXATO nome da classe no banco. O interessante desse atributo é que a tabela pode ter um

nome diferente de sua classe mapeada. Isso ajuda a resolver a questão de convenções de nomes já abordada neste material.

Onde deve ser incluída: Sobre a classe somente. **Opcional.**

Atributos:

- **name** (obrigatório): nome da tabela no banco de dados.
- **catalog** (opcional): catálogo da tabela no banco de dados.
- **schema** (opcional): esquema da tabela no banco de dados.
- **indexes** (opcional): vetor de objetos que mapeiam índices no banco de dados.
- **uniqueConstraints** (opcional): vetor de objetos que mapeiam as restrições de valor único no banco de dados.

Como atuou no exemplo: Indicou o nome da tabela no banco de dados como sendo "tbl_estabelecimento".

@Id (JPA)

Nome completo: *javax.persistence.Id*

Objetivo: Indica qual atributo da classe será mapeado para a **chave primária** da tabela.

Onde deve ser incluída: Sobre um atributo*. **Obrigatório** em pelo menos 1 (um) atributo.

Atributos: Não possui.

Como atuou no exemplo: Indicou que o atributo **id** está mapeado para o campo de **chave primária** na tabela.

@GeneratedValue (JPA)

Nome completo: *javax.persistence.GeneratedValue*

Objetivo: Configura a forma de preenchimento automático do valor do campo da chave primária. Se não for usada, o programa deve configurar “manualmente” o valor do atributo da chave primária.

Onde deve ser incluída: Sobre um atributo*. **Opcional.**

Atributos:

- **strategy** (opcional): Indica a estratégia para a geração do valor do atributo. Os tipos de estratégias são os valores da *enum* **javax.persistence.GenerationType**:

- **AUTO**: Indica que a estratégia padrão de preenchimento automático do banco de dados configurado será utilizada. Em alguns bancos, a chave primária “cresce sozinha”, ou seja, possui um valor com **auto incremento**. Para outros, o JPA pegará o maior valor atualmente na tabela e usará ele mais 1. Se o atributo **strategy** for omitido, esta estratégia é a que será usada.

- **IDENTITY**: Em alguns bancos, a chave primária “cresce sozinha”, ou seja, possui um valor com **auto incremento**. O IDENTITY indica que o JPA irá gerar uma instrução de *insert* apropriada para o banco de dados configurado para que ele use esse recurso no momento da criação de um novo registro.

- **SEQUENCE**: Alguns bancos de dados não possuem o recurso de **auto incremento** de valor. Assim, uma forma de fazer o valor “crescer” de forma consistente é consultar o novo valor de uma **sequence** no banco de dados. Essa opção indica e configura o uso desse recurso para a obtenção do valor que será usado na chave primária.

- **TABLE**: Opção muito parecida com a **SEQUENCE**. A diferença é que com ela se indica uma **tabela** e não uma **sequence** de onde se pega o novo valor que será usado na chave primária.

- **generator** (opcional, porém obrigatório para **SEQUENCE**): Caso tenha usado o **SEQUENCE** no **strategy**, nesse atributo deve indicar o mesmo valor que usou no **name** da anotação **@SequenceGenerator** (descrita a seguir). Caso tenha usado o **TABLE** no **strategy**, nesse atributo deve indicar o mesmo valor que usou no **name** da anotação **@TableGenerator** (descrita a seguir).

Como atuou no exemplo: Indicou que o campo **id** terá seu valor preenchido automaticamente com uso da estratégia **IDENTITY**.

@SequenceGenerator (JPA)

Nome completo: *javax.persistence.SequenceGenerator*

Objetivo: Configura o acesso a uma **sequence** do banco para ser usada na chave primária.

Onde deve ser incluída: Sobre um atributo*. **Opcional.** Obrigatória apenas se for usado **SEQUENCE** no atributo **strategyType** na anotação **@GeneratedValue**.

Atributos:

- **name** (obrigatório): Indica o nome da **sequence** na classe mapeada. O valor desse atributo é que deve ser usado no atributo **generator** da anotação **@GeneratedValue**.

- **sequenceName** (obrigatório): Indica o nome da **sequence** no banco de dados.

- **schema** (opcional): nome do **esquema** do banco onde está a **sequence**. Se omitido, o JPA irá considerar que está no mesmo que a tabela.

- **catalog** (opcional): nome do **catálogo** do banco onde está a **sequence**. Se omitido, o JPA irá considerar que está no mesmo que a tabela.

@TableGenerator (JPA)

Nome completo: *javax.persistence.TableGenerator*

Objetivo: Configura o acesso a uma **tabela** do banco para ser usada na chave primária.

Onde deve ser incluída: Sobre um atributo*. **Opcional.** Obrigatória apenas se for usado **TABLE** no atributo **strategyType** na anotação **@GeneratedValue**.

Atributos:

- **name** (obrigatório): Indica o nome da **tabela** na classe mapeada. O valor desse atributo é que deve ser usado no atributo **generator** da anotação **@GeneratedValue**.

- **table** (obrigatório): Indica o nome da **tabela** no banco de dados.

- **valueColumnName** (obrigatório): Indica o nome do **campo** da tabela no banco de dados.

- **schema** (opcional): nome do **esquema** do banco onde está a **tabela**. Se omitido, o JPA irá considerar que está no mesmo que a tabela.

- **catalog** (opcional): nome do **catálogo** do banco onde está a **tabela**. Se omitido, o JPA irá considerar que está no mesmo que a tabela.

@Column (JPA)

Nome completo: *javax.persistence.Column*

Objetivo: Mapeia uma coluna da tabela junto a um atributo na classe.

Onde deve ser incluída: Sobre um atributo*. **Opcional.**

Atributos:

- **name** (opcional): Indica qual o nome da campo na tabela. Se omitido, o JPA entenderá que o campo possui exatamente o mesmo nome do atributo. O interessante desse atributo é que um campo pode ter um nome na tabela diferente de seu atributo mapeado na classe. Isso ajuda a resolver a questão de convenções de nomes já abordada neste material.

- **length** (opcional): Indica o tamanho do campo na tabela. Por exemplo, para um campo **varchar** esse campo indicaria a quantidade de caracteres que ele comporta.

- **precision** (opcional): Indica a precisão do campo. Esse atributo só se aplica a campos numéricos.

- **scale** (opcional): Indica a escala do campo. Esse atributo só se aplica a campos numéricos.

- **nullabe** (opcional): Indica se o campo é obrigatório (**false**) ou se é possível criar/atualizar um valor de um registro deixando-o em vazio (**true**) .

- **unique** (opcional): Indica se o campo deve possuir valor único na tabela, ou seja, se é um campo com a restrição **unique** na tabela.

- **insertable** (opcional): Indica se o campo pode ter valor no momento da criação de um registro. Se esse atributo for **false** um eventual valor do atributo da classe será ignorado no momento da criação de um registro.

- **updatable** (opcional): Indica se o campo pode ter valor no momento da atualização de um registro. Se esse atributo for **false** um eventual valor do atributo da classe será ignorado no momento da atualização de um registro.

Como atuou no exemplo: Indicou os vários atributos que estão mapeados para campos da tabela. Sobre o atributo **id** indicou que seu respectivo campo na tabela era **id_estabelecimento**; sobre **nome** indicou que seu respectivo campo na tabela era **nome_estabelecimento**; sobre **dataCriacao** indicou que seu respectivo campo na tabela era **dh_criacao**.

@Temporal (JPA)

Nome completo: *javax.persistence.Temporal*

Objetivo: Usado para indicar o tipo de dado **temporal** que será guardado no campo do atributo mapeado.

Onde deve ser incluída: Sobre um atributo*. **Opcional.** Só pode ser usada em atributos dos tipos **LocalDate**, **LocalDateTime**, **Calendar** ou **Date**.

Atributos:

- **value** (obrigatório): Indica o tipo de dado temporal do campo. Os tipos de estratégias são os valores da *enum* **javax.persistence.TemporalType**:

- o **TIMESTAMP**: Indica que o campo irá receber a data e a hora. Muito usado em campos dos tipos **datetime** e **timestamp**.

- o **DATE**: Indica que o campo irá receber somente a data. Muito usado em campos do tipo **date**. Quando recuperado do banco, o atributo do tipo **LocalDate**, **Calendar** ou **Date** estará sempre com a hora “zerada” (ex: “00:00:00”).

- o **TIME**: Indica que o campo irá receber somente a hora. Muito usado em campos do tipo **time**. Quando recuperado do banco, o atributo do tipo **LocalDate**, **Calendar** ou **Date** estará com o dia em 1 de janeiro de 1970, sendo relevante apenas a hora, minuto, segundo e milissegundos. Ocorre que esses dois tipos em Java não conseguem representar somente uma “hora”.

Como atuou no exemplo: Indicou que o atributo **dataCriacao** receberia valores com **data e hora**.

@Enumerated(JPA)

Nome completo: `javax.persistence.Enumerated`

Objetivo: Usado em campos mapeados em atributos de tipos de **enums**. Indica se no banco será armazenado o valor literal do **enum** (valor alfanumérico) ou sua ordem na **classe enum** (número inteiro, a partir do 0).

Onde deve ser incluída: Sobre um atributo*. **Opcional.** Só pode ser usada em atributos de tipos de **enums**.

Atributos:

- **value** (obrigatório): Indica a estratégia para o preenchimento e recuperação do valor do atributo. Os tipos de estratégias são os valores da `enum javax.persistence.EnumType`:

- o **STRING**: Indica que o valor do **enum** será literalmente convertido em uma String para ser armazenado no campo mapeado. Por exemplo, um **tipo enum** com os valores **AZUL** e **VERMELHO**, teria os valores **"AZUL"** e **"VERMELHO"**, respectivamente, como possibilidades para o campo.

- o **ORDINAL**: Indica que a ordem do enum em sua classe será usada para determinar o valor que será armazenado no campo mapeado. Por exemplo, um **tipo enum** com os valores **AZUL**, **VERDE** e **VERMELHO**, teria os valores **0**, **1** e **2**, respectivamente, como possibilidades para o campo.

@CreationTimestamp (Hibernate)

Nome completo: `org.hibernate.annotations.CreationTimestamp`

Objetivo: Indica que o atributo receberá automaticamente a data e hora do sistema no momento da **criação** de um registro.

Onde deve ser incluída: Sobre um atributo*. **Opcional.** Só pode ser usada em atributos do tipo **LocalDate**, **LocalDateTime**, **Calendar** ou **Date**.

Atributos: Não possui.

Como atuou no exemplo: Indicou que o **dataCriacao** teria seu valor preenchido automaticamente com a data e hora atuais do sistema quando um novo registro fosse criado.

@UpdateTimestamp(Hibernate)

Nome completo: *org.hibernate.annotations.UpdateTimestamp*

Objetivo: Indica que o atributo receberá automaticamente a data e hora do sistema no momento da **criação** E **atualização** de um registro.

Onde deve ser incluída: Sobre um atributo*. **Opcional.** Só pode ser usada em atributos do tipo **LocalDate**, **LocalDateTime**, **Calendar** ou **Date**.

Atributos: Não possui.

@Formula (Hibernate)

Nome completo: *org.hibernate.annotations.Formula*

Objetivo: Usada para indicar que um determinado atributo não está mapeado para um campo da tabela, mas que seu valor, sempre que solicitado, será uma **sub select** ou uma **função de agregação**. Muito útil para os chamados **campos calculados**.

Onde deve ser incluída: Sobre um atributo*. **Opcional.**

Atributos:

- **value** (obrigatório): atributo no qual indicamos a **instrução SQL** que será usada para determinar o valor do atributo da classe anotado com **@Formula**. Para evitar efeitos colaterais, é recomendado que o **select** dentro da instrução esteja entre parêntesis. Se, ao invés de um **sub select** for apenas usada uma função de agregação (**avg**, **sum**, **count**, **min**, **max**, etc.), os parêntesis não serão necessários.

Como atuou no exemplo: Indicou que o campo `mediaAvaliacoes` não corresponde a nenhuma tabela no banco, mas que, ao ser solicitado, seu valor corresponderia à `sub select select avg(a.nota)+1 from avaliacao a where a.id_estabelecimento = id_estabelecimento`.

Interfaces Repository

Um dos grandes recursos do Spring Data é a possibilidade de criar interfaces **Repository**. São interfaces nas quais definimos coisas muito interessantes, como métodos que, se seguirem determinados padrões de nomes, poupam um grande tempo na escrita de código de operações de baixa complexidade junto ao banco de dados.

O uso de **Repository** do **Spring Data** dispensa o uso do padrão de projetos chamado **DAO**, que figurou por muito tempo como o mais utilizado para a construção de classes de acesso a tabelas de banco de dados. Aliás, esse nome faz referência a um padrão de projeto de mesmo nome.

O Spring Data facilita imensamente a implementação do padrão Repository. Basta criar uma interface que estenda a **JpaRepository** (pacote **org.springframework.data.repository**) e criar métodos seguindo determinados padrões de nomes. A seguir, um exemplo de uma Repository para a entidade **Estabelecimento**, codificada anteriormente.

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

public interface EstabelecimentoRepository extends
    JpaRepository<Estabelecimento, Integer>{

}
```

A interface **JpaRepository** contém os métodos CRUD mais comuns usados na manipulação de tabelas. A mágica é que **o Spring Data implementa todos esses métodos em tempo de execução** com as instruções SQL apropriadas de acordo com o banco de dados configurado no projeto.

Os métodos que a **JpaRepository** possui e que funcionam em tempo de execução sem que precisemos implementá-los manualmente, são os do Quadro 3.

Método	Retorno
count() - Retorna o número de registros da tabela. Retorno: long.	long
delete(T entidade) - Exclui o registro da entidade na tabela.	void
deleteAll() - Exclui todos os registros na tabela mapeada na entidade.	void

deleteAll (Iterable <? extends T > entidades) - Exclui todos os registros na tabela mapeada na entidade de acordo com a coleção de objetos da entidade enviados.	void
deleteById (ID id) - Exclui um registro na tabela mapeada na entidade de acordo com um id (chave primária).	void
existsById (ID id) - Retorna a informação se existe algum registro com um determinado id (chave primária).	boolean
findAll () - Retorna uma coleção com todos os registros da tabela mapeada (o famoso “ <i>select * from...</i> ”)	Iterable < T >
findAllById (Iterable < ID > ids) - Retorna uma coleção de registros de acordo com uma lista de ids (chaves primárias)	Iterable < T >
findById (ID id) - Retorna um registro de acordo com um id (chave primária)	Optional < T >
save (T entity) - Salva (cria ou atualiza) uma entidade no banco de dados. Se o atributo de chave primária estiver nulo, o JPA tenta fazer um insert na tabela. Caso contrário, tenta um update.	T
saveAll (Iterable < T > entities) - Salva (cria ou atualiza) vários registros numa só operação, de acordo com uma coleção de entidades recebida.	Iterable < T >

Quadro 3: Métodos da interface JpaRepository do Spring Data

O tipo **T** é o tipo da entidade e o tipo **ID** é o tipo do atributo mapeado para a chave primária. O tipo **Iterable** é uma das interfaces que estão acima de coleções como **List** e **Set** na hierarquia de coleções do Java.

Assim, uma instância de **EstabelecimentoRepository**, ao invocar, por exemplo, o método **count()**, obtém a quantidade de registros na tabela **tbl_estabelecimento**. Outro exemplo: ao invocar o **findAll()**, obtém uma coleção com todos os registros de **tbl_estabelecimento**. Ratificando: não precisamos implementar esses métodos manualmente. O Spring Data faz isso em tempo de execução.

Usando uma Repository numa Service ou Controller

Supondo que tenhamos uma classe Service chamada **EstabelecimentoService** no mesmo projeto Spring. Para que ela use a **EstabelecimentoRepository**, basta fazer como no código de exemplo a seguir.

```

@Service
public class EstabelecimentoService {

    @Autowired
    private EstabelecimentoRepository repositorio;

    public Estabelecimento salvar(Estabelecimento estabelecimento) {
        // regras antes de tentar salvar
        return this.repositorio.save(estabelecimento);
    }

    // demais atributos e métodos da Service
}

```

Nessa classe Service de exemplo, a **EstabelecimentoRepository** é injetada por meio da anotação **@Autowired** do Spring já estudada anteriormente. A injeção ocorre porque quando uma interface estende a **JpaRepository**, o Spring já a configura como um componente no contexto da aplicação.

Supondo que tenhamos uma classe Controller chamada **EstabelecimentoController** no mesmo projeto Spring. Para que ela use a **EstabelecimentoRepository**, basta fazer como no código de exemplo a seguir.

```

public class EstabelecimentoController {

    @Autowired
    private EstabelecimentoRepository repository;

    @GetMapping
    public ResponseEntity listarTodos() {
        List<Estabelecimento> lista = repository.findAll();
        return lista.isEmpty() ? noContent().build() : ok(lista);
    }

    @GetMapping("/{id}")
    public ResponseEntity recuperar(@PathVariable("id") int id) {
        Optional<Estabelecimento> registro = repository.findById(id);
        return registro.isPresent() ? ok(registro.get()) : notFound().build();
    }
}

```

```

@DeleteMapping("/{id}")
public ResponseEntity excluir(@PathVariable("id") int id) {
    if (repository.existsById(id)) {
        repository.deleteById(id);
        return ok().build();
    } else {
        return notFound().build();
    }
}

@PostMapping
public ResponseEntity criar(@RequestBody Estabelecimento estabelecimento) {
    this.repository.save(estabelecimento);
    return status(HttpStatus.CREATED).build();
}

@PutMapping("/{id}")
public ResponseEntity atualizar(
    @PathVariable("id") int id, @RequestBody Estabelecimento estabelecimentoAlterado) {
    if (repository.existsById(id)) {
        estabelecimentoAlterado.setId(id);
        repository.save(estabelecimentoAlterado);
        return ok().build();
    } else {
        return notFound().build();
    }
}
}

```

Na Controller de exemplo, usamos os conceitos de status de resposta HTTP para demonstrar como usar vários dos métodos da **JpaRepository** do Spring Data.

Nos métodos de recuperação unitária e de todos usamos 2 dos diferentes métodos que permitem a consulta por meio do que já disponibiliza por padrão a **JpaRepository**.

Nos métodos de exclusão e atualização, primeiro verificamos se o **id** é de um registro de **Estabelecimento** antes de realizar as respectivas suas operações, tudo isso com métodos disponibilizados pela **JpaRepository**.

Atenção! Segundo os princípios do DDD não seria uma boa prática injetar Repository direto numa Controller. O exemplo anterior foi apenas para fins didáticos e deve ser evitado sempre que possível. Num bom design, as interfaces do tipo Repository devem ser injetadas em Services e, estas, injetadas nas Controllers.

Prática de Repository com Controller

Crie o **EstabelecimentoController** em seu projeto Spring Boot e teste ele em execução. Para fins de testes rápidos, vamos adicionar a dependência do banco de dados **H2**. Esse banco ficará embarcado na aplicação, é super leve e, por padrão, cria um banco de dados em memória junto do início da aplicação. A dependência desse banco pode ser feita como no trecho de código a seguir.

```
<!-- H2 Database -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

Criando métodos no Repository

Além dos métodos que “ganhamos” da JpaRepository, podemos criar outros apenas criando suas assinaturas em nossa interface Repository. Contanto que sigamos alguns padrões de nomes de método e tipos de retorno e argumento, o Spring Data cria suas implementações pra nós.

No código de exemplo a seguir criamos alguns métodos seguindo esses padrões na **EstabelecimentoRepository**.

```
public interface EstabelecimentoRepository extends
    JpaRepository<Estabelecimento, Integer>{

    Estabelecimento findByCnpj(String cnpj);

    List<Estabelecimento> findByNome(String nome);
```

```
List<Estabelecimento> findByNameLike(String nomeContendo);

long countByCnpjIsNull();

boolean existsByDataCriacaoBefore(LocalDate criadoAte);

long deleteByCnpjIsNull();

}
```

O método **findByCnpj(String cnpj)** retorna **null** ou uma instância de **Estabelecimento**, conforme encontrar ou não um registro com o **cnpj** informado.

O método **findByName(String nome)** retorna uma **List vazia** ou uma **List** com algumas instâncias de **Estabelecimento**, conforme encontrar ou não registros com o **nome** informado.

O método **findByNameLike(String nomeContendo)** retorna uma **List vazia** ou uma **List** com algumas instâncias de **Estabelecimento**, conforme encontrar ou não registros cujo atributo **nome** contenha o valor **nomeContendo** informado.

O método **countByCnpjIsNull()** retorna um **inteiro longo** que é a quantidade de registros encontrados com o atributo **cnpj** nulo.

O método **existsByDataCriacaoBefore(LocalDate criadoAte)** retorna um **boolean** com valor **true** caso exista algum registro cujo atributo **dataCriacao** seja anterior à data informada em **criadoAte** ou **false** caso não existe nenhum.

O método **deleteByCnpjIsNull()** tenta excluir todos os registro com o atributo **cnpj** nulo e retorna um **inteiro longo** que é a quantidade de registros excluídos.

A seguir, os padrões que, se seguidos, fazem com que o Spring Data implemente e execute as instruções SQL em tempo de execução.

Criando métodos padronizados na Repository

De forma geral, os métodos devem seguir o seguinte padrão de nome:

nome da operação + atributo #1 em PascalCase + condição do atributo #1 (opcional)
+ nome do atributo #2 em PascalCase (opcional) + condição do atributo #2 (opcional)
+ nome do atributo #N em PascalCase (opcional) + condição do atributo #N (opcional)

Nome da operação

É a identificação da consulta ou atualização que deseja fazer. Os nomes de operação do Spring Data estão descritos a seguir.

findBy

Define realização de uma consulta a partir de 1 ou mais atributos e/ou critérios. Esse consulta pode retornar 1 ou vários registros.

Retornos: **List<TipoDaEntidade>**, **Set<TipoDaEntidade>**, **Iterable<TipoDaEntidade>**, **TipoDaEntidade**, **Optional<TipoDaEntidade>**.

Argumentos: **Um ou mais, de acordo com os tipos dos atributos informados no nome do método.**

Se usar algum retorno de tipo de **coleção (List, Set, Iterable)**, o Spring vai tentar por os registros encontrados numa coleção do tipo solicitado.

Se usar como retorno a **classe da entidade**, o Spring vai tentar executar uma instrução SQL que recupera 1(um) registro a partir do critério informado. Se não encontrar nenhum registro retorna **null**.

Se usar como retorno um **Optional** da entidade, o Spring vai tentar executar uma instrução SQL que recupera 1(um) registro a partir do critério informado. Se não encontrar nenhum registro retorna um Optional sem valor.

Exemplos:

- Estabelecimento **findByCnpj**(String cnpj);
- Optional<Estabelecimento> **findByCnpj**(String cnpj);
- List<Estabelecimento> **findByNome**(String nome);

findAllBy

Funciona de forma semelhante ao **findBy**, porém usa-se apenas para consultas que retornam uma **coleção (List, Set, Iterable)**. É apenas uma opção de nome a mais para que não achar muito claro que um **findByXXX()** retorna uma coleção e não apenas 1 objeto.

countBy

Define a realização de uma consulta que só faz uma contagem de registros encontrados a partir de 1 ou mais atributos e/ou critérios.

Retornos: **Long, long, Integer** ou **int**.

Argumentos: **Um ou mais, de acordo com os tipos dos atributos informados no nome do método.**

Exemplos:

- `long countByNome(String nome);`
- `int countByNome(String nome);`

existsBy

Define a realização de uma consulta que só verifica se uma contagem de registros encontrados a partir de 1 ou mais atributos e/ou critérios é maior ou não que 0 (zero). Ou seja, ela retorna **true** se houver pelo menos 1 (um) registro que atenda ao critério programado ou false se não houver nenhum.

Retornos: **Boolean** ou **boolean**.

Argumentos: **Um ou mais, de acordo com os tipos dos atributos informados no nome do método.**

Exemplos:

- `Boolean existsByNome(String nome);`
- `boolean existsByNome(String nome);`

deleteBy

Define a realização de uma consulta que só verifica se uma contagem de registros encontrados a partir de 1 ou mais atributos e/ou critérios é maior ou não que 0 (zero).

Retornos: **void, Long, long, Integer** ou **int**.

Argumentos: **Um ou mais, de acordo com os tipos dos atributos informados no nome do método.**

Obs: Se usado um dos tipos numéricos permitidos como retorno, o método vai retornar a quantidade de registros excluídos. Se for usado **void**, o método simplesmente tenta realizar as exclusões, sem informar nada.

Exemplos:

- long **deleteBy**Nome(String nome);
- void **deleteBy**Nome(String nome);

Nome de atributo

O nome de um atributo usado como critério na operação desejada deve estar em PascalCase. Exs: **NomeDaPessoa**; **CnpjResponsavel**.

Importante: Usamos o nome do **atributo** mapeado na **entidade**, não o nome do campo da tabela do banco de dados.

Condição de atributo

O Spring Data possui padrões para vários tipos de condições que podem ser usadas logo após os nomes dos atributos. Essas condições são configuradas a partir de palavras-chave. Vide o Quadro 4.

Palavra-chave	Exemplo
And Usado para adicionar outro atributo/condição obrigatórios na operação	findByNome And Cnpj(String nomeX, String cnpjY)
Or Usado para adicionar atributo/condição opcionais na operação	findByNome Or Cnpj(String nomeX, String cnpjY)
Is,Equals (<i>opcional</i>) Apenas para deixar explícito o termo “igual”	findByCnpj Is (String cnpjX), findByCnpj Equals (String cnpjX)
Between “entre”. Funciona com números e datas (LocalDate, LocalDateTime, Calendar e Date)	findByDataCriacao Between (LocalDate d1, LocalDate d2) findBySalario Between (Double s1, Double s2)
LessThan “menor que”	findByDataCriacao LessThan (LocalDate dataQq)

LessThanEqual “menor ou igual a”	findBySalario LessThanEqual (Double salarioX)
GreaterThan “maior que”	findByDataCriacao GreaterThan (LocalDate dataQq)
GreaterThanOrEqualTo “maior ou igual a”	findBySalario GreaterThanOrEqualTo (Double salarioY)
After “após” (apenas para campos de “tempo”)	findByDataCriacaoDate After (LocalDate dataQq)
Before “após” (apenas para campos de “tempo”)	findByDataCriacaoDate Before (LocalDate dataQq)
IsNull “é null”	findByDataCriacao IsNull ()
IsNotNull, NotNull “não é null”	findByDataCriacao NotNull (), findByDataCriacao IsNotNull ()
Like “like do SQL”	findByCnpj Like (String cnpjX)
NotLike “not like do SQL”	findByCnpj NotLike (String cnpjX)
StartingWith “like ‘?’ do SQL”. Dispensa o uso de ‘%’ no argumento	findByNome StartingWith (String nomeC)
EndingWith “like ‘%?’ do SQL”. Dispensa o uso de ‘%’ no argumento	findByNome EndingWith (String nomeC)
Contains, Containing “like ‘%?’ do SQL”. Dispensa o uso do “%” no argumento	findByNome Contains (String nomeC) findByNome Containing (String nomeC)
OrderBy Define a ordenação (Desc ou Asc) de um atributo	findBy DataCriacaoOrderByNomeDesc (LocalDate dataC)
Not “<> do SQL”	findBy NomeNot (String nomeA)
In	findByCnpj In (Collection<String> cnpjs)

"in do SQL". Aceita qualquer coleções ou vetores como argumento	
NotIn "not in do SQL". Aceita qualquer coleções ou vetores como argumento	findByCnpj NotIn (Collection<String> cnpps)
TRUE "= true" ou equivalente, dependendo do SGBD	findByAtivo True ()
FALSE "= true" ou equivalente, dependendo do SGBD	findByAtivo False ()
IgnoreCase Tenta usar função equivalente a "ignorecase" do SGBD configurado	findByCnpj IgnoreCase (String cnpjX)

Quadro 4: Condições para atributos em uma Repository do Spring Data

Usando **And** e **Or** podemos criar vários critérios diferentes, embora posso ficar confuso um nome com mais de três atributos envolvidos.

Validação automática dos métodos de Repository

Outra vantagem muito útil do uso de Repository, é que caso digitemos um nome de atributo que não existe (erro de digitação, por exemplo) ou erramos num dos critérios, a aplicação Spring Boot não inicia e em seu log de execução fica claro em qual método da Repository está o erro.

Bibliografia

GUTIERREZ, Felipe. Pro Spring Boot. New York (EUA): Apress, 2016.

JBOSS.ORG. Hibernate ORM 5.2.12.Final User Guide. Disponível em: <https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html>. Acesso em: 24/03/2019.

JENDROCK, Eric. Persistence - The Java EE5 Tutorial. Disponível em: <<https://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html>>. Acesso em: 20/03/2019.

PANDA, Debu; RAHMAN, Reza; CUPRAK, Ryan; REMIJAN, Michael. EJB 3 in Action, Second Edition. Shelter Island, NY, EUA: Manning Publications, 2014.