SENAI

*Serviço Nacional de Aprendizagem Industrial*

**PELO FUTURO DO TRABALHO**

# Entidades e Banco de Dados

## Desenvolvimento de Sistemas

**Prof. Me. Reneilson Santos**

**Março/2024**

SENAI
PELO FUTURO DO TRABALHO

# Agenda

- ORM
- Entidades
- Lombok
- Enumeradores
- Bibliografia

# ORM

# Modelos

O Spring trabalha sobre uma arquitetura MVC - Model View Controller.

Na camada de modelos nós temos as nossas **entidades** que serão armazenadas em um banco de dados.

O Spring permite que criemos classes de modelo com a annotation ***@Entity***, isso facilitará a criação de nossas entidades dentro do banco de dados, pois, usando um banco de dados relacional, juntamente com o módulo JPA (Java Persistence Application) podemos facilmente criar nosso banco sem grandes complicações.

Isso é feito utilizando um **mapeamento objeto-relacional.**

# ORM

Em muitos sistemas, os objetos da vida real são modelados como objetos em sistemas para facilitar a representação e manipulação de seus atributos.

ORM (*Object-Relational Mapping*) é uma técnica de mapeamento de tais objetos e seus atributos no banco de dados por meio de Mapeadores Objeto-Relacionais.

# ORM

Um dos benefícios de usar um ORM inclui **acelerar o processo de desenvolvimento**, uma vez que os desenvolvedores **não precisam escrever o código de acesso ao banco de dados** e repeti-lo toda vez que quiserem acessar um banco de dados.

Depois que um modelo é projetado e o código de manipulação é escrito, isso não precisa ser feito novamente, tornando o código fácil de atualizar, manter e reutilizar.

**Hibernate** é um exemplo de framework que implementa um ORM Java.

# Hibernate

O Hibernate é um framework que permite aos desenvolvedores armazenar facilmente os dados de uma aplicação em bancos de dados relacionais usando JDBC, trata-se de **uma implementação da Java Persistence API (JPA)**.

Além disso, é uma ferramenta *leve e de código aberto* que **simplifica** a criação, manipulação e acesso de dados de um banco de dados em aplicativos baseados em Java.

Funciona mapeando um objeto criado a partir de uma classe Java e seus atributos para dados armazenados no banco de dados.

# Iniciando Projeto

# Inicializando Projeto com JPA

- **Spring Web** para criar uma aplicação baseada em web;
- **Spring Data JPA** para fornecer a API de acesso a dados que o Hibernate utilizará;
- **Spring Boot DevTools** para permitir reload automático e algumas configurações para auxiliar no desenvolvimento.

# Inserindo Banco de Dados Reais

# Inicializando Projeto com JPA e MySQL

Adiciona o driver do MySQL na inicialização:

- **MySQL Driver**

Cria-se o banco pelo MySQL Workbench ou coloca o ***create*** ou ***create-drop*** na configuração do spring.jpa.hibernate.ddl-auto e ***true*** na configuração do spring.jpa.generate-ddl.

# Inicializando Projeto com JPA e MySQL

Com o banco criado, podemos configurar o acesso ao mesmo no arquivo **application.properties** do nosso projeto Spring.

```
# Database Properties
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/pets
spring.datasource.username=root
spring.datasource.password=senai

# Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto=update
```

# Inicializando Projeto com JPA e MySQL

**spring.datasource.url**: Especifica a URL do banco de dados MySQL. Certifique-se de substituir localhost:3306/nomedobanco pelo endereço do seu servidor MySQL e o nome do banco de dados.

**spring.datasource.username** e **spring.datasource.password**: São as credenciais de acesso ao banco de dados.

**spring.jpa.properties.hibernate.dialect**: Define o dialeto do Hibernate para MySQL.

**spring.jpa.hibernate.ddl-auto**: Define como o Hibernate deve atualizar o esquema do banco de dados. Neste exemplo, está configurado para update, o que significa que o Hibernate tentará automaticamente atualizar o esquema do banco de dados com base nas entidades JPA definidas em seu aplicativo.

# DDL Auto

**none:** Esta configuração desabilita a geração automática do esquema pelo Hibernate. Você será responsável por criar e manter o esquema do banco de dados manualmente. Esta opção é útil em ambientes de produção onde você deseja controle total sobre o esquema do banco de dados.

**validate:** O Hibernate valida o esquema do banco de dados existente, mas não faz alterações nele. Se houver inconsistências entre as entidades JPA e o esquema do banco de dados, uma exceção será lançada durante a inicialização do aplicativo.

# DDL Auto

**update:** O Hibernate verifica o esquema do banco de dados existente e faz as alterações necessárias para refletir as entidades JPA. Isso inclui a criação de novas tabelas, adição de colunas, remoção de colunas, etc. No entanto, o Hibernate tentará preservar os dados existentes, evitando a perda de dados sempre que possível.

**create:** O Hibernate cria o esquema do banco de dados do zero toda vez que o aplicativo é iniciado. Isso implica que os dados existentes serão perdidos, já que as tabelas serão descartadas e recriadas.

**create-drop:** Similar a create, mas o esquema do banco de dados é descartado quando o aplicativo é encerrado. Isso é útil durante o desenvolvimento, mas não é adequado para ambientes de produção, pois implica em perda de dados a cada reinicialização do aplicativo.

# Entidades

# Classe de Entidade

Criado o banco de dados, podemos então passar às entidades e seus relacionamentos. Comecemos criando as classes com seus atributos e métodos getters, setters e construtores, para depois adicionarmos as Annotations que as transformarão em tabelas.

# Classes de Entidade

Com as classes criadas, e os métodos get, sets e construtores criados, podemos então adicionar os *Annotations* que nos permitirá transformar, através do ORM, as classes em tabelas no nosso banco de dados.

A primeira *annotation*, e principal, é **@Entity** que transformará a classe em uma classe de entidade, e, consequentemente, em uma tabela no banco de dados (relacional).

Podemos ainda adicionar a *annotation* **@Table** passando algumas configurações para o banco de dados (como o nome), *por padrão o nome da tabela será o valor em minúsculo do nome da classe*.

# Atributos -> Colunas

Automaticamente, todos os atributos de uma classe de entidade será transformado em uma coluna no banco de dados a partir do ORM, usando como nome o próprio nome do atributo. Caso queiramos modificar algo, podemos usar a annotation **@Column** passando o nome (*name*) da coluna além de possíveis restrições:

- **nullable** (se pode ser nula) (booleano)
- **updatable** (se pode ser editada) (booleano)
- **length** (tamanho máximo) (inteiro)
- **unique** (se o campo é único) (booleano)

# Classes de Entidade

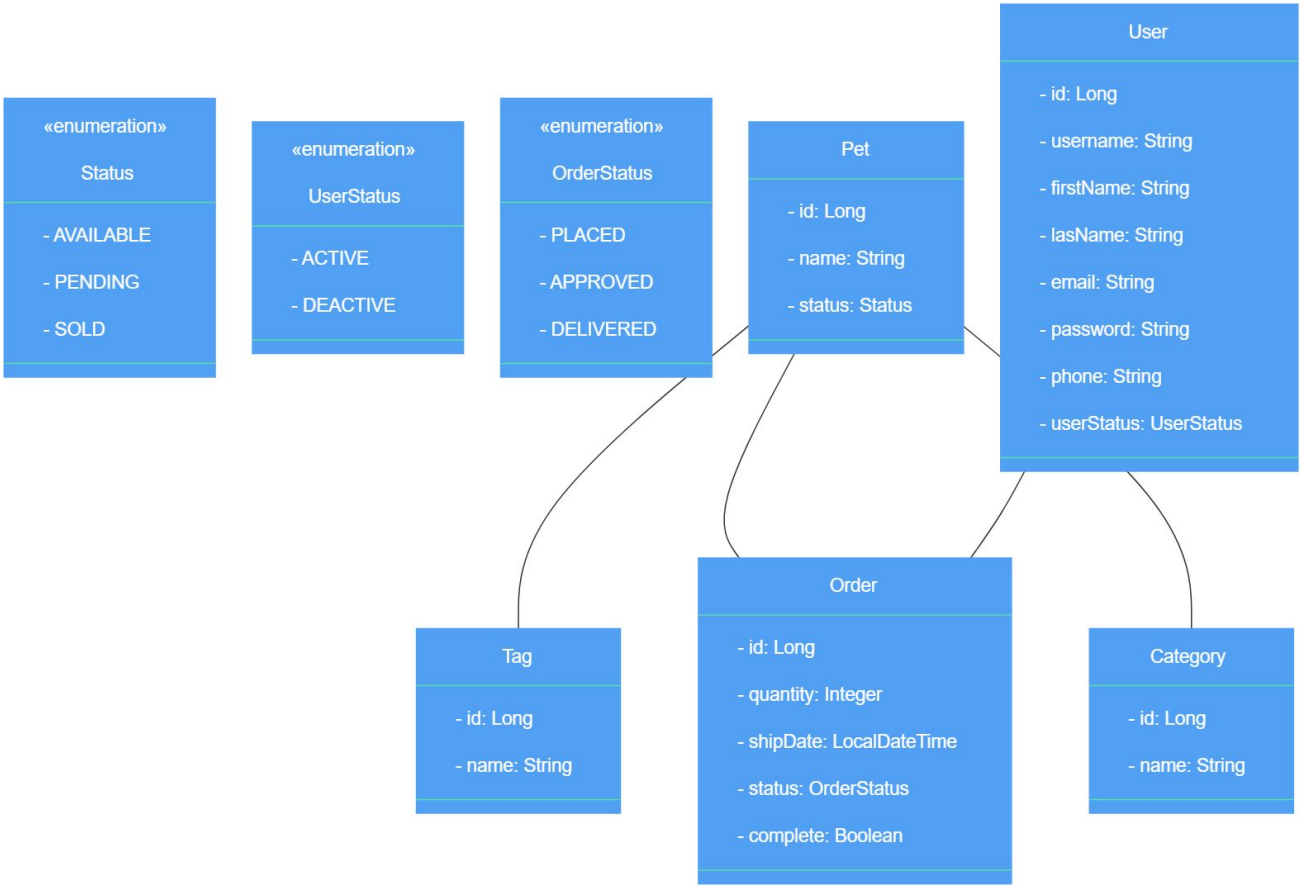
No caso dos ids das classes, adicionamos a annotation **@Id** além do **@GeneratedValue** passando a estratégia de geração (strategy) e o gerador (generator).

O nome do gerador de chave primária a ser usado conforme especificado na annotation quando utilizado uma estratégia do tipo SEQUENCY ou TABLE irá especificar o nome do SequenceGenerator ou TableGenerator.

# Estratégias

A estratégia utilizada indica qual tipo de persistência será usada para gerar a primary key. Pode ser:

- **AUTO**: Indica que o provedor de persistência deve escolher uma estratégia apropriada para o banco de dados específico.
- **IDENTITY:** Indica que o provedor de persistência deve atribuir chaves primárias para a entidade usando uma coluna de identidade do banco de dados.
- **SEQUENCY**: Indica que o provedor de persistência deve atribuir chaves primárias para a entidade usando uma sequência de banco de dados.
- **TABLE**: Indica que o provedor de persistência deve atribuir chaves primárias para a entidade usando uma tabela de banco de dados subjacente para garantir exclusividade.


«enumeration»
Status
- AVAILABLE
- PENDING
- SOLD
«enumeration»
UserStatus
- ACTIVE
- DEACTIVE
«enumeration»
OrderStatus
- PLACED
- APPROVED
- DELIVERED
Pet
- id: Long
- name: String
- status: Status
User
- id: Long
- username: String
- firstName: String
- lasName: String
- email: String
- password: String
- phone: String
- userStatus: UserStatus
Tag
- id: Long
- name: String
Order
- id: Long
- quantity: Integer
- shipDate: LocalDateTime
- status: OrderStatus
- complete: Boolean
Category
- id: Long
- name: String

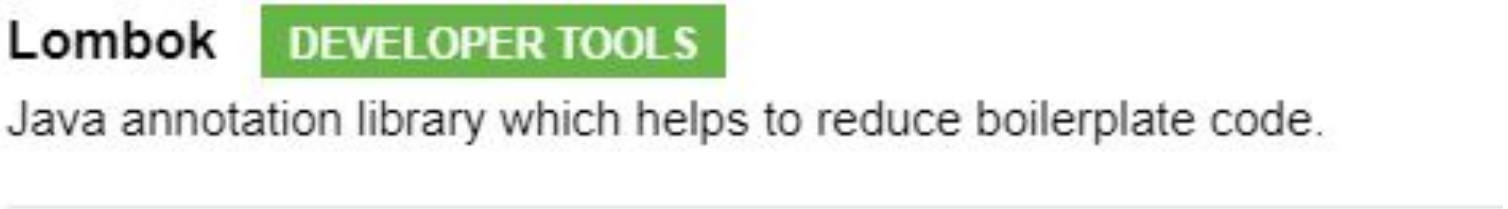
```java
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Pet {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private Status status;
}
```

# Lombok

# Lombok

Para não precisar estar adicionando os métodos getters e setters nas nossas classes de modelo (entidades), podemos então adicionar a biblioteca **Lombok** ao nosso projeto.



# Lombok

Caso já tenha sido inicializado, basta adicionar no arquivo de configuração do maven a dependência do Lombok (https://projectlombok.org/setup/maven):

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

# Lombok

Para usar o lombok nas nossas classes de entidade, basta usar a annotation **@Getter** e **@Setter** no nome da classe que, automaticamente, os métodos gets e sets serão gerados em tempo de execução e não ficarão poluindo o nosso código.

O Lombok ainda disponibiliza a anotação **@Data** que é a junção das anotações **@Getter , @Setter , @EqualsAndHashCode e @ToString** .

Deve-se ressaltar que, ao usar o Lombok, o código dos métodos equals, toString e hashCode serão utilizados nos mesmos todos os atributos da classe a não ser que seja configurado de forma diferente.

# Lombok - Construtores

**@NoArgsConstructor**

- Construtor vazio

**@RequiredArgsConstructor**

- Apenas os atributos marcados com a *annotation @NonNull*

**@AllArgsConstructor**

- Construtor com todos os atributos

```
…
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class Pet {
    …
}
```

# Enumeradores

# Enumeradores

Para criar os enumeradores no nosso banco de dados podemos utilizar a annotation **@Enumerated** (não na classe, na variável que usa o enum).

Essa annotation recebe como parâmetro o tipo, se é:

- STRING - para enumerações que serão armazenadas como string; (EnumType.STRING)
- ORDINAL - para enumerações que serão armazenadas como inteiro. (EnumType.ORDINAL)

# Exemplo

- Projeto em turma
  - Enumerador Status em Pet

```
public enum Status {
    AVAILABLE, PENDING, SOLD;
}
```

```
…
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
…

@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class Pet {
    …
    @Enumerated(EnumType.STRING)
    private Status status;
}
```

# Atividade

- Continuação Individual do Projeto de Pets.
  - Criar demais entidades.
  - Criar demais enumeradores.

# Bibliografia

# Bibliografia


- Annotations Hibernate (https://www.objectdb.com/api/java/jpa/annotations)
- Implementing Hibernate with Spring Boot and PostgreSQL (https://stackabuse.com/implementing-hibernate-with-spring-boot-and-postgresql/)
- Swagger PetStore (https://petstore.swagger.io/)