DESENVOLVIMENTO WEB – BACK END

AULA 4

Profª Luciane Yanase Hirabara Kanashiro

CONVERSA INICIAL

Iniciamos esta etapa com conceitos fundamentais que moldam a eficiência e a estruturação de aplicações. A Classe Java Revisitada nos conduz a uma compreensão mais profunda da serialização, uma técnica essencial para transformar objetos em fluxos de bytes, possibilitando a persistência e o transporte de dados de maneira eficaz. A Persistência de Dados introduzirá a Java Persistence API, uma especificação que redefine a interação entre objetos Java e o banco de dados, simplificando a camada de persistência. O conceito de Object Relational Mapping (ORM) destaca-se como uma abordagem que mapeia objetos Java para entidades do banco de dados, unificando a lógica de negócios e a camada de dados.

Ainda em persistência de dados, e não menos importante, veremos as classes persistentes e o ciclo de vida na JPA, pois compreender o ciclo de vida das entidades ajudará a gerenciar eficientemente os recursos do banco de dados e evitar comportamentos indesejados. Nesse contexto, o Padrão Data Access Object (DAO) emerge como uma prática organizacional, encapsulando a lógica de acesso a dados e promovendo a modularidade. Na interação entre a apresentação e a lógica de negócios, destacam-se a "Classe de Serviço" e o "Gerenciamento de Transações". Enquanto a primeira encapsula a lógica de negócios da aplicação, a segunda garante a integridade dos dados ao garantir que todas as operações ocorram com sucesso, ou que, em caso de falha, sejam revertidas.

Esses pilares formam a base para a criação de aplicações robustas e eficientes, em que a persistência e a manipulação de dados são realizadas de maneira coesa e estruturada.

TEMA 1 – CLASSE JAVA REVISITADA: SERIALIZABLE

A serialização de objetos Java é o processo de converter um objeto em uma sequência de bytes, geralmente para armazenamento persistente, transmissão de dados pela rede ou para permitir que o objeto seja reconstruído em um ambiente diferente. Após a transmissão ou o armazenamento, essa cadeia de bytes pode ser transformada novamente no objeto Java que a originou.

No Java, a serialização é frequentemente realizada implementando a interface Serializable ou usando anotações específicas.

A interface Serializable é fundamental para a persistência de objetos em Java, permitindo que eles sejam gravados em arquivos, transmitidos pela rede e armazenados em bancos de dados de forma eficiente.

A Serializable é uma interface vazia, o que significa que não tem nenhum método que precise ser implementado. A simples implementação dessa interface é suficiente para sinalizar a capacidade de serialização da classe. A seguir, vemos um exemplo de implementação da interface Serializable.

```
import java.io.Serializable;

public class MinhaClasse implements Serializable {
    // Código da classe
}
```

Ao implementar a interface Serializable, a classe informa à JVM que seus objetos podem ser convertidos em uma forma que pode ser gravada em um fluxo de saída e, posteriormente, reconstruída a partir de um fluxo de entrada.

1.1 Parametrização

Embora parametrização e a serialização sejam conceitos independentes, eles podem estar relacionados quando temos classes parametrizadas que precisam ser serializadas. Por essa razão, a Parametrização será abordada nesta seção.

Em Java, um "parâmetro genérico" refere-se ao uso de tipos genéricos (ou generics). Generics permitem criar classes, interfaces e métodos que podem operar com diferentes tipos de dados, enquanto ainda fornecem segurança de tipo durante a compilação.

A principal vantagem do uso de parâmetros genéricos é a criação de código mais flexível e reutilizável, permitindo escrever componentes que podem funcionar com diferentes tipos de dados sem perder a segurança de tipo.

A seguir, vemos um exemplo simples de uma classe com um parâmetro genérico em Java:

```
public class Caixa<T> {
    private T conteudo;

    public void adicionarConteudo(T novoConteudo) {
        this.conteudo = novoConteudo;
    }
```

```
    public T obterConteudo() {
        return this.conteudo;
    }

    public static void main(String[] args) {
        // Uso da classe Caixa com um tipo específico
(Integer)
        Caixa<Integer> caixaDeInteiro = new Caixa<>();
        caixaDeInteiro.adicionarConteudo(42);
        Integer valorInteiro =
caixaDeInteiro.obterConteudo();
        System.out.println("Conteúdo da Caixa de Inteiro: "
+ valorInteiro);

        // Uso da classe Caixa com um tipo diferente
(String)
        Caixa<String> caixaDeString = new Caixa<>();
        caixaDeString.adicionarConteudo("Olá, mundo!");
        String valorString = caixaDeString.obterConteudo();
        System.out.println("Conteúdo da Caixa de String: "
+ valorString);
    }
}
```

Nesse exemplo, Caixa<T> é uma classe genérica que pode aceitar qualquer tipo de objeto. O tipo específico (como Integer ou String) é determinado quando instanciamos a classe. Isso proporciona flexibilidade e, ao mesmo tempo, mantém a segurança de tipo durante a compilação.

A parametrização permite criar classes, interfaces e métodos que podem trabalhar com tipos de dados de forma flexível. Quando utilizamos um parâmetro genérico, devemos especificar o tipo que será usado quando estivermos instanciando um objeto dessa classe. No caso, se for uma interface, a classe que for utilizar essa interface deve especificar de qual tipo será esse parâmetro.

Nesse outro exemplo, se for para uma classe nomeada ClienteDao, o parâmetro T deve ser substituído por Cliente:

```
public class ClienteDAO implements CRUD<Cliente, Long>
```

TEMA 2 – PERSISTÊNCIA DE DADOS – JPA e ORM

A persistência de dados é um aspecto fundamental no desenvolvimento de aplicações modernas. Ela se refere à capacidade de uma aplicação em armazenar e recuperar informações de forma durável, ou seja, mesmo após o encerramento da aplicação ou reinicialização do sistema. Neste tópico, exploraremos os conceitos essenciais da persistência de dados em aplicações Java Web.

2.1 Persistência de Dados com JPA

A Java Persistence API (JPA) representa uma evolução significativa no desenvolvimento de aplicativos Java ao fornecer uma abordagem robusta e padronizada para a persistência de dados. Sua base é construída sobre o conceito de Plain Old Java Objects (POJOs), trazendo simplicidade e eficiência ao processo de persistência.

A JPA, em essência, é uma API que facilita o gerenciamento de persistência e o mapeamento objeto-relacional (ORM) em aplicações Java. O seu principal foco reside na camada ORM, buscando harmonizar a representação dos dados no código Java com a estrutura relacional de um banco de dados. Essa integração é vital para que os desenvolvedores possam manipular dados de maneira intuitiva e eficaz, utilizando objetos Java em vez de consultas SQL tradicionais.

Uma característica distintiva da JPA é o seu papel no mapeamento objeto-relacional. Esse processo envolve a tradução das classes Java para tabelas no banco de dados relacional e dos objetos dessas classes para linhas nas respectivas tabelas. Esse fenômeno é conhecido como Mapeamento Objeto-Relacional. Veremos mais adiante uma explicação mais clara do que é ORM. Em outras palavras, a JPA fornece uma ponte entre o mundo orientado a objetos do Java e o modelo relacional do banco de dados, permitindo uma interação mais natural e simplificada entre as duas camadas.

É importante mencionar que o Hibernate é uma implementação padrão da especificação JPA. Isso significa que o Hibernate segue as diretrizes estabelecidas pela JPA, proporcionando uma implementação concreta e amplamente utilizada dessa API.

Com o Hibernate, os desenvolvedores podem aproveitar os recursos avançados de ORM oferecidos pela JPA de maneira consistente e eficiente.

2.2 ORM

A definição mais básica que encontramos em qualquer lugar que fale de **ORM** será algo parecido com*:* O Object-Relational Mapping (ORM) é uma técnica de programação que facilita a interação entre sistemas orientados a objetos e bancos de dados relacionais. Essencialmente, ele cria uma camada de abstração entre o código da aplicação, que é orientado a objetos, e o banco de dados, que é relacional, permitindo que os desenvolvedores realizem operações no banco de dados usando objetos em vez de escrever consultas SQL diretamente.

Mas o que isso quer dizer? Por que usar ORM?

Existe um termo conhecido como ***"Incompatibilidade de impedância objeto-relacional"*** ou ***"incompatibilidade de paradigma"***. Essa incompatibilidade é apenas uma maneira elegante de dizer que modelos de objetos e modelos relacionais não funcionam muito bem juntos. Ao adotar a orientação a objetos, surgem conceitos cruciais, como identidade, herança e associações, que nem sempre são expressos de maneira natural em um banco de dados relacional. A identidade de um objeto, por exemplo, pode ser difícil de representar de forma direta em uma tabela de banco de dados. Da mesma forma, hierarquias de herança e associações entre objetos podem ser desafiadoras de traduzir para um modelo relacional sem uma abordagem adequada.

A solução para esse desafio é encontrada no ORM, também conhecido como Object Relational Mapping. Por meio do ORM, os desenvolvedores podem persistir objetos diretamente em um banco de dados relacional, eliminando a necessidade de converter manualmente objetos para um formato relacional. Isso simplifica significativamente o desenvolvimento de aplicações, permitindo que os desenvolvedores se concentrem na lógica do aplicativo em vez de se concentrarem nos detalhes de como os dados são armazenados.

2.3 Entidades e Anotações

As entidades no JPA nada mais são do que POJOs. Os POJOs representam dados que podem ser persistidos no banco de dados. Sendo assim, uma entidade representa uma tabela armazenada em um banco de dados, e cada instância de uma entidade representa uma linha na tabela.

O exemplo a seguir é um POJO que representa os dados de uma pessoa.

```
public class Funcionario{

    private Long id;
    private String name;
    private Integer idade;


    //getters and setters

}
```

Anotação @Entity

Se quiséssemos que esses dados fossem armazenados no Banco de Dados, deveríamos definir uma entidade para que o JPA tenha conhecimento dela. Para isso, utilizamos a anotação **@Entity**.

O nome da entidade será, por padrão, o nome da classe.

```
@Entity
public class Funcionario{

    // fields, getters and setters

}
```

Podemos mudar o nome da entidade usando o elemento name.

```
@Entity (name = "Empregado")
public class Funcionario{

    // fields, getters and setters

}
```

Anotação @Id

Cada entidade no contexto da Java Persistence API (JPA) deve ter uma chave primária única que a identifique de forma exclusiva. A anotação **@Id** é utilizada para definir a chave primária.

A geração dos identificadores pode ser realizada de diversas maneiras, sendo especificadas pela anotação **@GeneratedValue**.

Existem quatro estratégias de geração de identificadores disponíveis, as quais são especificadas pelo elemento strategy. As opções incluem AUTO, TABLE, SEQUENCE ou IDENTITY. O código a seguir ilustra a utilização da anotação @Id e @GeneratedValue.

```
@Entity
public class Funcionario {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;
    private String name;

    // getters and setters
}
```

Anotação @Table

Muitas vezes, não queremos que o nome da tabela no BD seja o mesmo da nossa entidade. Podemos especificar, nesses casos, o nome da tabela usando a anotação @Table, como mostrado no código a seguir.

```
@Entity
@Table(name="FUNCIONARIO")
public class Funcionario {

    // fields, getters and setters

}
```

Anotação @Column

Podemos usar de modo análogo a anotação Column para nomear e detalhar uma coluna da tabela. O código a seguir mostra a utilização da anotação @Column

```
@Entity
@Table(name="Funcionario)
public class Funcionario {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="NOME_FUNCIONARIO", length=50,
nullable=false, unique=false)
    private String nome;
```

```
    // other fields, getters and setters
}
```

O elemento name especifica o nome da coluna na tabela. O elemento length especifica seu comprimento. O elemento nullable especifica se a coluna é anulável ou não, e o elemento unique especifica se a coluna é exclusiva.

Anotação @Transient

Algumas vezes, não queremos que um campo seja persistido na tabela. Para isso, podemos usar a anotação Transient. O código a seguir mostra a utilização do @Transient.

```
@Entity
@Table(name="FUNCIONARIO")
public class Funcionario {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name=" NOME_FUNCIONARIO ", length=50,
nullable=false)
    private String nome;

    @Transient
    private Integer idade;

    // other fields, getters and setters
}
```

2.4 Relacionamento e Anotações

É necessário relembrar, ainda, que uma classe pode ser mapeada como uma tabela do BD. Tal mapeamento é realizado de acordo com a direção do relacionamento (unidirecional ou bidirecional) e a cardinalidade do relacionamento que está no Banco de dados. A tabela a seguir resume os relacionamentos e as respectivas anotações.

Tabela 1 – Tabela do BD

Relacionamento	Anotação
Um para um	@OneToOne
Muitos para um	@ManyToOne
Um para muitos	@OneToMany
Muitos para muitos	@ManyToMany

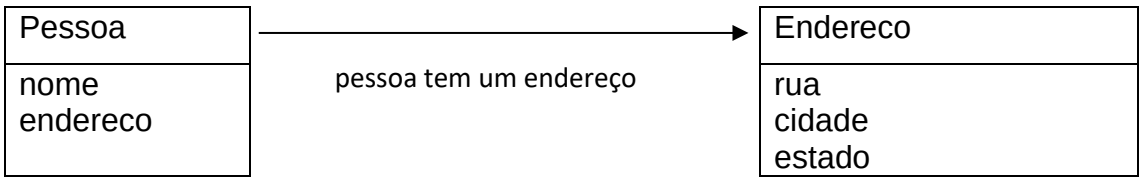
A seguir, veremos exemplos de como utilizamos essas anotações no código para representar os relacionamentos unidirecional e bidirecional.

No relacionamento unidirecional, apenas uma das entidades envolvidas é mapeada para o banco de dados. Isso significa que, se tivermos uma relação entre as entidades A e B, apenas uma delas, digamos A, será mapeada, enquanto a outra (B) permanecerá ignorada pelo ORM.

No contexto do relacionamento unidirecional, a entidade mapeada é conhecida como "entidade fonte". A entidade fonte contém as informações do relacionamento e tem conhecimento da existência da outra entidade (alvo). A entidade alvo, por outro lado, não está diretamente mapeada e não tem conhecimento da entidade fonte.

A seguir, veremos um **exemplo de mapeamento unidirecional**. Para isso, consideraremos o diagrama 1, que possui duas classes: pessoa e endereço.

Diagrama 1 – Mapeamento unidirecional



A entidade fonte, nesse caso, é a classe Pessoa. O mapeamento será feito nessa classe . O código a seguir ilustra a classe Pessoa já mapeada com a anotação

`@OneToOne.`

A anotação `@JoinColumn(name="endereco_id_fk")` adiciona uma coluna estrangeira ao lado forte (classe Pessoa). O atributo `cascade={cascadeType.ALL}` implica que quando for inserir (deletar/atualizar) uma pessoa, será inserido também por cascata o endereço.

```
@Entity
public class Pessoa {
private String nome;
private Endereco endereco;
    @OneToOne(cascade={cascadeType.ALL})
    @JoinColumn(name="ENDERECO_ID_fk")
    public Endereco getEndereco(){
        return endereco;
```

```
    }
}
```

Na classe Endereço não haverá qualquer mapeamento, como pode ser observado na codificação a seguir.

```
@Entity
public class Endereco{
 private String rua;
 private String cidade;
 private String estado;
}
```

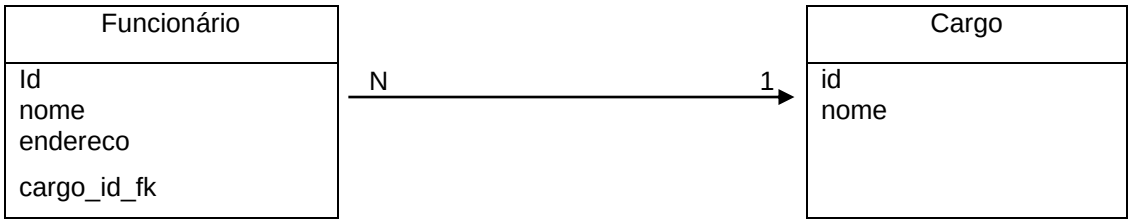
No Relacionamento Bidirecional, a entidade possuidora em um relacionamento bidirecional é a entidade que tem a chave estrangeira no banco de dados. Ela é responsável por definir a associação e influencia a estrutura da tabela no banco de dados.

A tabela correspondente a essa entidade será a possuidora da chave estrangeira. Essa entidade gerencia a persistência da relação e tem conhecimento tanto da entidade inversa quanto da própria relação.

A entidade inversa, por sua vez, é a outra entidade na relação que não tem a chave estrangeira em sua tabela. Essa entidade não gerencia a persistência da relação, mas tem conhecimento dela. Em um relacionamento bidirecional, a entidade inversa é anotada com mappedBy para indicar qual atributo na entidade possuidora gerencia o mapeamento bidirecional. O atributo anotado com mappedBy na entidade inversa deve ser configurado para indicar o relacionamento inverso.

Para o **exemplo de mapeamento bidirecional**, vamos considerar o diagrama 2.

Diagrama 2 – Mapeamento bidirecional



Cargo está relacionado a vários funcionários,
Mas um funcionário pode ter um único cargo

Temos nesse diagrama duas classes: Cargo e Funcionário. Vamos analisar o relacionamento entre Cargo e Funcionário. A primeira coisa que devemos fazer é identificar quem é o dono da relação. Nesse caso, a tabela Funcionário é a proprietária, já que a coluna da *foreign key* (chave estrangeira) está nela.

Devemos, então, adicionar um novo atributo à classe Funcionário e mapear essa classe com a anotação @ManyToOne. A anotação @ManyToOne indica que temos um relacionamento muitos-para-um entre Funcionário e Cargo.

A anotação `@JoinColumn(name="cargo_id_fk")` adiciona uma coluna estrangeira ao lado forte (classe Funcionario).

```
  @ManyToOne
  @JoinColumn(name = "cargo_id_fk")
    private Cargo cargo;
```

A seguir, é mostrado o código completo:

```
@Entity
@Table(name = "FUNCIONARIO")
public class Funcionario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String nome;
    @Column(nullable = false, unique = true)
    private String endereco;

@ManyToOne
@JoinColumn(name = "cargo_id_fk")
      private Cargo cargo;
}
```

Como é um relacionamento bidirecional, na classe Cargo também deveremos colocar uma anotação apropriada. Devemos criar um atributo, e o mapeamos com a anotação `@OneToMany`.O trecho de código a seguir ilustra esse mapeamento.

```
    @OneToMany(mappedBy = "cargo")
    //@JsonIgnore
    private List<Funcionario> funcionarios;
```

A anotação @JsonIgnore pode ser usada para ignorar um campo ou método durante a serialização e deserialização. Pode ser utilizada para testes utilizando postman.

A seguir, é mostrado o código completo da classe Cargo:

```
@Entity
public class Cargo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "nome", nullable = false, unique = true,
length = 60)
    private String nome;


    @OneToMany(mappedBy = "cargo")
    //@JsonIgnore
    private List<Funcionario> funcionarios;
}
```

TEMA 3 – PERSISTÊNCIA DE DADOS – CICLO DE VIDA DE CLASSES PERSISTENTES E PADRÃO DAO

3.1 Classes persistentes e o ciclo de vida

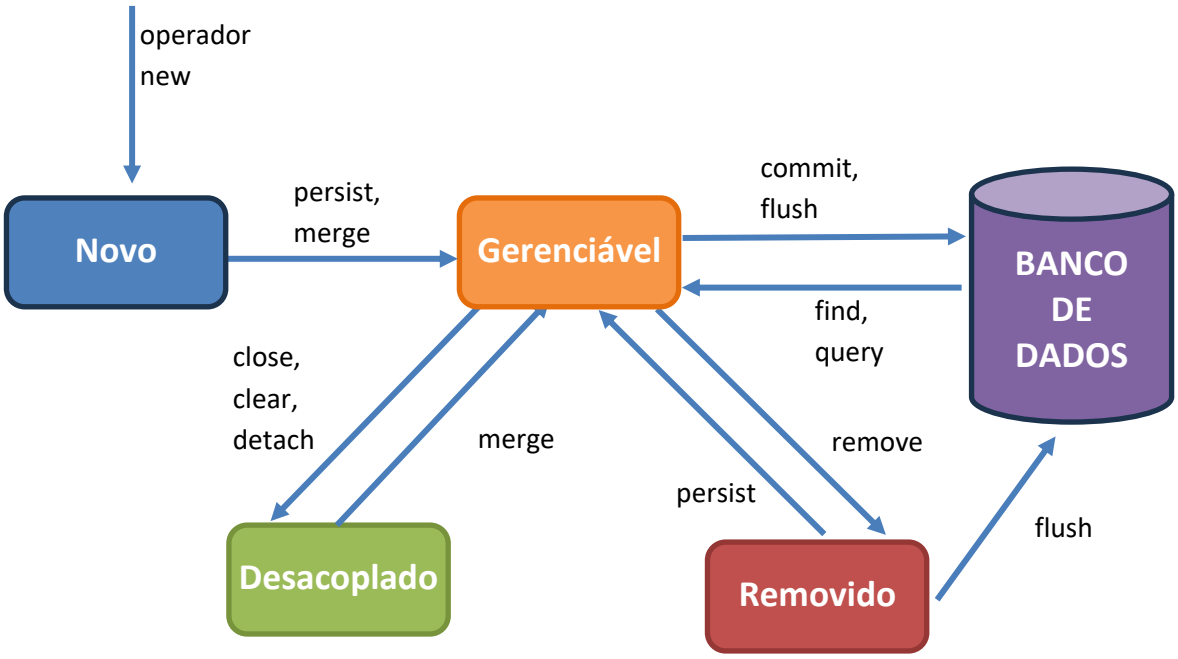
Agora que já aprendemos sobre persistência com JPA e suas anotações, veremos sobre as classes persistentes e seu ciclo de vida.

As Classes persistentes são classes em Java que estão associadas a uma camada de persistência de dados, geralmente usando tecnologias como Java Persistence API (JPA) ou Hibernate. Essas classes representam entidades de negócios que precisam ser armazenadas e recuperadas de um banco de dados de maneira transparente. A persistência de dados permite que objetos Java sejam mapeados para registros de banco de dados, facilitando a interação entre o código Java e o armazenamento de dados. Simplificando, foi o que abordamos no começo desta etapa quando vimos que algumas classes podem ser mapeadas para tabelas em um banco dados. Cada instância de uma classe representa, assim, uma linha na tabela. As instâncias têm, ainda, ciclo de vida que é gerenciado pelo JPA.

O diagrama 3 representa o **ciclo de vida de um objeto na JPA** desde o momento em que é instanciado com *new* até o momento em que é salvo no banco de dados e, se necessário, removido. Os retângulos representam os estados possíveis (Novo, Gerenciável, Desacoplado, Removido), e as palavras que acompanham as setas representam métodos da Interface EntityManager.

A **Entity Manager** é responsável por gerenciar as entidades (objetos Java que representam dados no banco de dados) em um contexto de persistência. Ela fornece métodos para persistir novas entidades, recuperar entidades existentes, remover entidades, realizar consultas no banco de dados e controlar transações.

Diagrama 3 – Ciclo de vida de um objeto na JPA



Fonte: elaborado por Kanashiro, 2024, com base em Cordeiro, 2012.

Vamos reforçar, a seguir, algumas das características das classes persistentes, pois isso é fundamental para a compreensão do ciclo de vida.

Como visto anteriormente, as classes persistentes são frequentemente anotadas com metadados de persistência que indicam como elas devem ser mapeadas para o banco de dados. Por exemplo, em JPA, as anotações como @Entity, @Table, @Id, entre outras, são comumente usadas, como mostradas no exemplo a seguir.

```java
@Entity
@Table(name = "clientes")
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Outros atributos, construtores, getters, setters
}
```

As Operações de persistência estão relacionadas ao estado do ciclo de vida de objetos JPA. As operações de persistência incluem persistir (salvar), recuperar, atualizar e excluir, sendo realizadas por meio de APIs de persistência, como o EntityManager em JPA. Essas operações garantem que as mudanças nas classes persistentes sejam refletidas no banco de dados. O exemplo a seguir mostra a criação de um objeto cliente, desde o momento em que é instanciado com o operador new até o momento em que é persistido no banco de dados.

```java
EntityManager entityManager = ...;
Cliente cliente = new Cliente();
entityManager.persist(cliente); // Persistindo uma nova entidade
```

Ao revisar as características das classes persistentes, é importante ressaltar, ainda, que elas desempenham um papel crucial em ambientes de desenvolvimento orientados a objetos, permitindo que as aplicações Java interajam de maneira eficaz com sistemas de gerenciamento de banco de dados relacionais e proporcionando uma abstração de alto nível para o acesso a dados.

No desenvolvimento de software, se o código de um aplicativo estiver intimamente ligado aos detalhes específicos dos recursos de dados que ele utiliza, isso dificultará a substituição ou a modificação dos recursos de dados de um aplicativo. Ou seja, se o desenvolvedor quiser trocar ou modificar os recursos de dados (por exemplo, mudar de um banco de dados para outro, ou atualizar uma API), será difícil fazer isso sem afetar a lógica de negócios do aplicativo. Isso cria uma situação em que a manutenção, atualização ou substituição de componentes individuais do sistema se torna complicada e propensa a erros.

Em termos práticos, a boa prática de design de software geralmente sugere manter a lógica de negócios separada da lógica de acesso a dados, para isso, podemos utilizar o padrão de projeto Data Access Object (DAO).

3.2 Padrão DAO

O **padrão Data Access Object (DAO)** é um padrão de projeto que fornece uma interface para algum tipo de mecanismo de armazenamento ou recuperação de dados. Ele separa a lógica de acesso e armazenamento de dados da lógica de negócios do restante do código. Isso permite que a lógica de negócios seja mais flexível em relação às fontes de dados, pois ela não precisa conhecer os detalhes específicos do armazenamento ou do banco de dados. O padrão DAO é amplamente utilizado em projetos JAVA EE (JAKARTA EE).

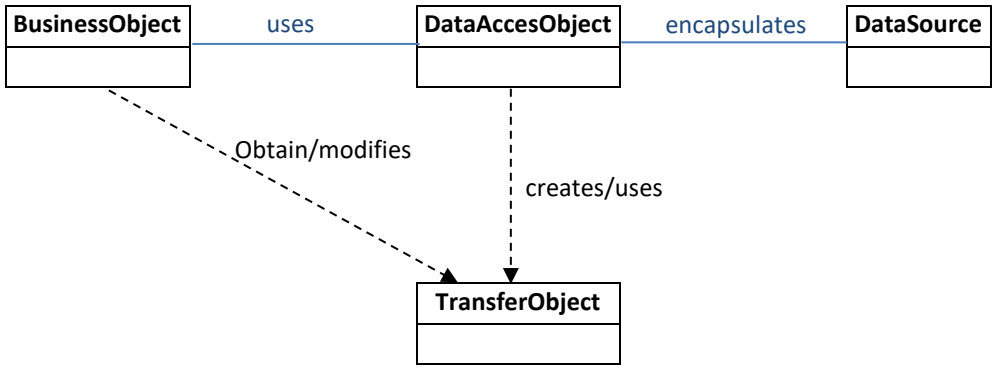
Entre as principais características do padrão DAO, podemos citar:

- Separa a interface do cliente de um recurso de dados de seus mecanismos de acesso a dados;
- Fornece uma interface abstrata para acessar os dados, definindo métodos genéricos como create, read, update e delete.

O padrão DAO permite que os mecanismos de acesso a dados mudem independentemente do código que utiliza os dados.

A seguir, temos um diagrama de classes que representa os relacionamentos para o padrão DAO. Na representação do diagrama de classes, temos quatro classes: BusinessObject, DataAccessObject, DataSource e TransferObject.

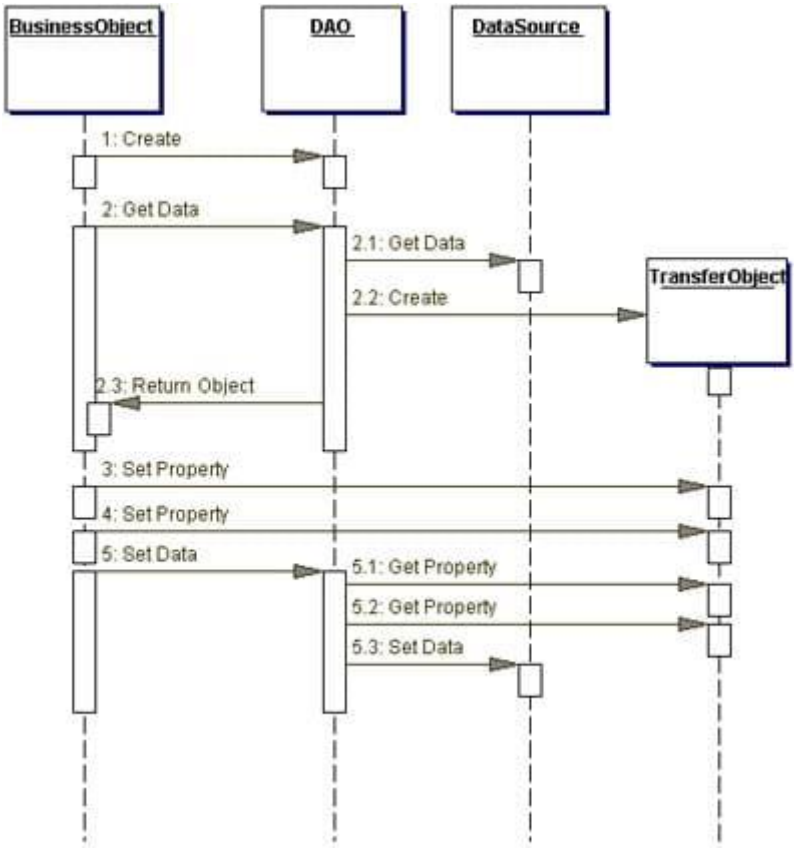
Diagrama 4 – Relacionamentos padrão DAO



Fonte: Oracle, 2001.

O Diagrama de sequência a seguir ilustra a interação entre os vários participantes no Padrão DAO.

Diagrama 5 – Sequência



Fonte: Oracle, 2001.

A seguir estão descritas as características de cada classe (Oracle, 2001):

- **BusinessObject**: o BusinessObject representa o cliente de dados. É o objeto que requer acesso à fonte de dados para obter e armazenar dados. Um BusinessObject pode ser implementado como um bean de sessão, um bean de entidade ou algum outro objeto Java, além de um servlet ou bean auxiliar que acessa a fonte de dados;
- **DataAccessObject**: DataAccessObject é o objeto principal desse padrão. O DataAccessObject abstrai a implementação de acesso a dados subjacente do BusinessObject para permitir acesso transparente à fonte de dados. O BusinessObject também delega operações de carregamento e armazenamento de dados ao DataAccessObject;
- **DataSource**: representa uma implementação de fonte de dados. Uma fonte de dados pode ser um banco de dados como um Relational Database Management System (RDBMS), Object-Oriented Database Management System (OODBMS), repositório XML, sistema de arquivos simples e assim por diante. Uma fonte de dados também pode ser outro

sistema (legado/mainframe), serviço (serviço B2B ou agência de cartão de crédito) ou algum tipo de repositório (LDAP);

- **TransferObject**: representa um Objeto de Transferência usado como portador de dados. O DataAccessObject pode usar um Transfer Object para retornar dados ao cliente. O DataAccessObject também pode receber os dados do cliente em um Transfer Object para atualizar os dados na fonte de dados.

Em termos práticos, o Padrão DAO é utilizado quando se deseja encapsular o acesso e a manipulação de dados em uma camada separada. A interface a seguir ilustra um exemplo de utilização do padrão DAO.

```
public interface FuncionarioDao {

     void save(Funcionario funcionario);
     void update(Funcionario funcionario);
     void delete(Long id);
     Funcionario findById(Long id);
     List<Funcionario> findAll();

}
```

TEMA 4 – CAMADA DE MODELO - CLASSE DE SERVIÇO

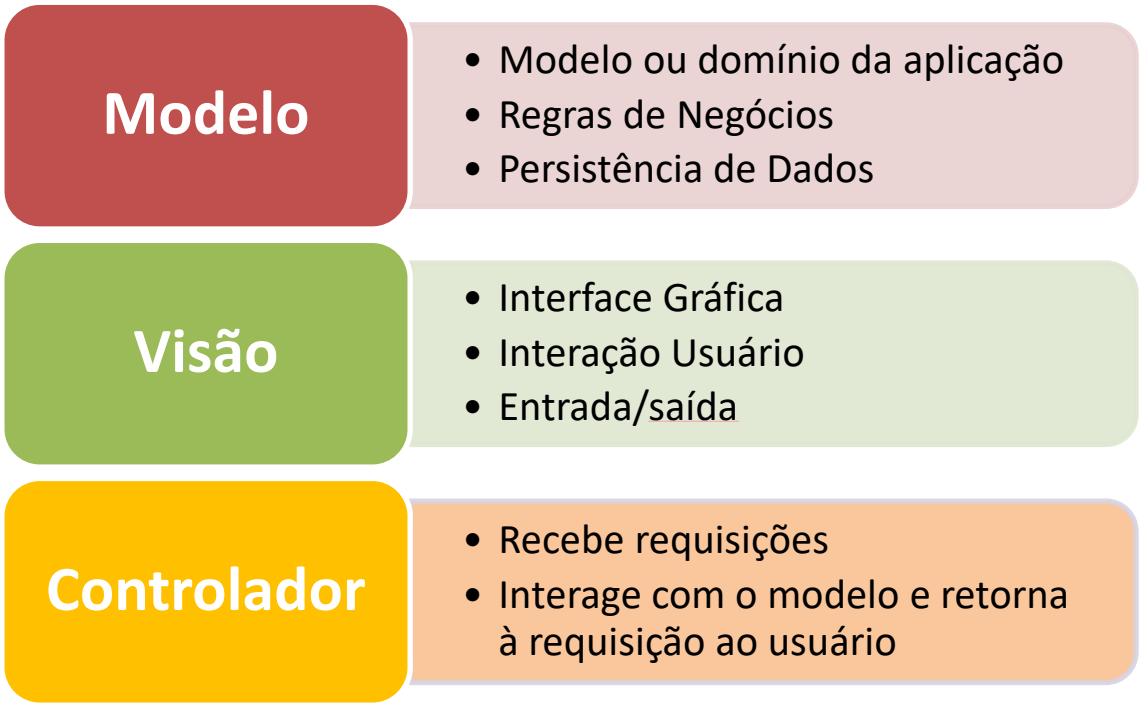
No contexto do Spring MVC (Model-View-Controller), uma "Classe de Serviço" geralmente refere-se a uma classe que contém a lógica de negócios da aplicação. Essas classes são responsáveis por realizar operações específicas que não pertencem à camada de controle (Controller), nem à camada de visualização (View), mas, sim, à manipulação e ao processamento de dados no modelo (Model) da aplicação.

O esquema a seguir resume as tarefas de cada camada do modelo MVC. Resumidamente, a camada de visão é responsável pela interface gráfica e interação com o usuário. A camada de Controle recebe as requisições, interage com o modelo e retorna à requisição ao usuário.

Observe que a camada de modelo está responsável pelo domínio da aplicação, pelas regras de negócio e pelo processamento de dados da aplicação. Ao se utilizar uma classe de serviço, ela operará na camada de modelo, ficando responsável pela lógica de negócios da aplicação.

A camada de Controle passa então a enxergar a classe de Serviço e o repositório fica então encapsulado nela.

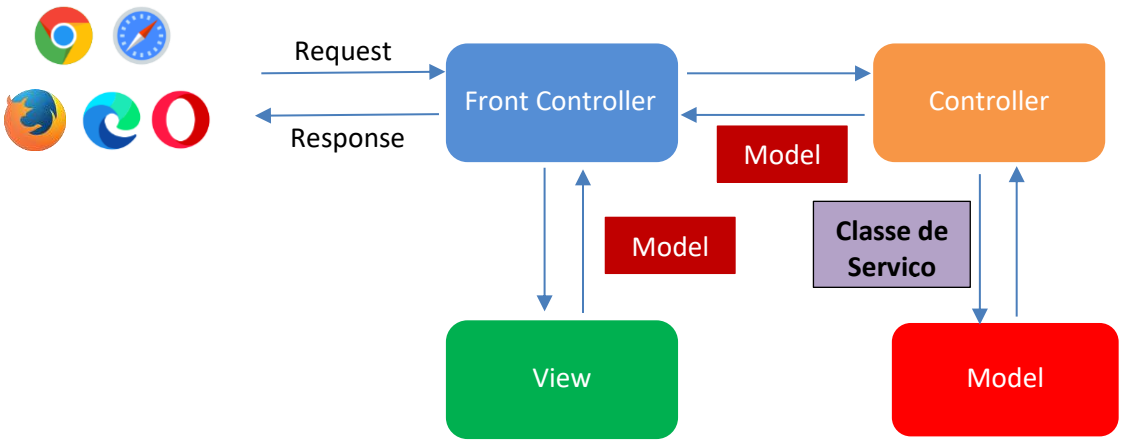
Figura 1 – Cama de controle



A utilização de uma classe de Serviço colabora para que a separação das regras (negócio, aplicação e apresentação) fiquem bem mais definidas. Isso colabora também para que elas possam ser facilmente testadas e reutilizadas em outras partes do programa.

A figura a seguir ilustra a classe de serviço no contexto do Spring MVC.

Figura 2 – Classe de serviço



Entre as principais características de uma classe de serviço no Spring MVC estão o encapsulamento da lógica de Negócios, a reutilização de lógica e a Injeção de dependência. Essas características da classe de serviço no Spring MVC estão explicadas a seguir:

- **Lógica de Negócios**: as classes de serviço encapsulam a lógica de negócios da aplicação. Elas podem realizar operações como validação de dados, cálculos, chamadas a serviços externos e outras tarefas relacionadas à regra de negócios;
- **Reutilização de Lógica**: uma boa prática é encapsular a lógica de negócios em classes de serviço para promover a reutilização. Isso facilita a manutenção e a evolução do código, pois a lógica central da aplicação está centralizada em um local;
- **Injeção de Dependência**: o Spring MVC, sendo parte do ecossistema Spring, utiliza o princípio de Injeção de Dependência. Isso significa que você pode injetar (por exemplo, usando anotações como @Autowired) uma instância de uma classe de serviço em outras partes da aplicação, como controladores ou outros serviços.

O código a seguir implementa um exemplo de uma classe de serviço no contexto do Spring MVC.

Classe ProductService:

```
@Service
public class ProductService {
  @Autowired
  private ProductRepository productRepository;

  public List<Product> getAllProducts() {
  return productRepository.findAll();
  }

  public Product getProductById(Long productId){
return productRepository.findById(productId).orElse(null);
}

 public void saveProduct(Product product) {
  productRepository.save(product);
}
 public void deleteProduct(Long productId) {
  productRepository.deleteById(productId);
```

```
}
}
```

Na codificação anterior, ProductService é uma classe de serviço que encapsula operações relacionadas a produtos. Ela interage com um repositório (ProductRepository), que geralmente lida com a persistência de dados, enquanto a classe de serviço se concentra na lógica de negócios associada aos produtos.

Utilizar classes de serviço no Spring MVC ajuda a manter a organização do código, seguindo o princípio de separação de responsabilidades e facilitando a manutenção e a evolução do sistema.

TEMA 5 – CAMADA DE MODELO – GERENCIAMENTO DE TRANSAÇÕES

O gerenciamento de transações é uma parte crucial do desenvolvimento de aplicações que interagem com bancos de dados. As transações garantem a consistência e a integridade dos dados ao realizar operações que podem afetar várias partes do sistema. Dois princípios fundamentais no gerenciamento de transações são a Garantia do Sucesso das Operações e o Isolamento entre Transações e Efeitos no Banco de Dados.

Na Garantia do Sucesso das Operações, o gerenciamento de transações visa assegurar que todas as operações realizadas dentro de uma transação ocorram com sucesso ou que, em caso de falha, o sistema seja revertido para um estado consistente e seguro. Isso é essencial para manter a integridade dos dados e evitar situações em que apenas algumas operações são realizadas com sucesso, deixando o sistema em um estado inconsistente.

Em um contexto de banco de dados, imagine uma transação que envolve a transferência de fundos entre duas contas bancárias. Se, por algum motivo, uma parte da transação falhar (por exemplo, a retirada de fundos de uma conta), é imperativo que a transação seja revertida para garantir que o sistema permaneça em um estado consistente. O gerenciamento de transações lida com esse tipo de cenário, garantindo que todas as operações ocorram ou que nenhuma ocorra se algo der errado.

Outro princípio fundamental é o isolamento entre transações e efeitos no Banco de Dados, o que significa que o processamento de uma transação não deve afetar outras transações em execução simultaneamente. Cada transação deve ser executada de maneira independente, sem interferir nas operações de outras transações concorrentes.

Além disso, uma transação bem-sucedida não deve ter efeitos permanentes no banco de dados ou em qualquer outra transação até que seja confirmada explicitamente. Isso é conhecido como o conceito de **"commit"**. Somente quando todas as operações dentro de uma transação foram concluídas com sucesso e a transação é confirmada é que as alterações são permanentemente aplicadas ao banco de dados. Caso contrário, se ocorrer um erro ou falha, a transação é revertida, descartando as mudanças temporárias que foram feitas.

Esses princípios garantem a consistência e a integridade dos dados em sistemas que envolvem operações complexas e concorrentes. O gerenciamento de transações é uma parte essencial do desenvolvimento de aplicações robustas, especialmente em ambientes em que várias operações concorrentes podem ocorrer simultaneamente. Isso promove a confiabilidade e a estabilidade do sistema, mesmo diante de situações adversas.

Podemos implementar o gerenciamento de transações na classe de serviço ou em uma classe Data Access Object (DAO). As duas estratégias são válidas, porém, a implementação na classe de serviço garante mais segurança nas transações, pois evita problemas relacionados a rollback. O **rollback** é uma medida de segurança para garantir que, em caso de falha durante uma transação, o banco de dados seja retornado a um estado consistente e seguro, evitando qualquer efeito colateral indesejado das operações que foram parcialmente concluídas.

Para uma classe gerenciar o processo de transações, ela tem que ter a anotação **@Transactional**. A anotação @Transactional pode ser, ainda, seguida do parâmetro ReadOnly. Se esse parâmetro estiver omitido, significa que o ReadOnly está setado com false. O ReadOnly = false implica que vai ter que abrir uma transação de escrita. Se tiver setado com true, não vai precisar que uma transação de escrita seja aberta, pois não será feita nenhuma modificação no banco de dados.

Nas subseções a seguir serão mostrados exemplos de implementação de gerenciamento de transações em Classes DAO e na classe de serviço.

5.1 Gerenciamento de transações na classe DAO

A **anotação @Transactional** é a forma declarativa do gerenciamento de transação fornecida pelo Spring, enquanto a anotação **@Autowired** é uma anotação do Spring Framework utilizada para realizar a injeção de dependência automática em classes gerenciadas pelo contêiner de inversão de controle (IoC) do Spring.

No exemplo utilizado, consideraremos parte da implementação para um dicionário de expressões da área de Tecnologia da Informação (TI). No código a seguir, podemos observar que as duas classes (*PalavraDaoImpl e ExpressaoDaoImpl*) foram anotadas com a anotação @Transactional. Isso implica que as transações referentes ao Banco de Dados serão realizadas nessas classes.

Seguem as implementações das classes.

Classe PalavraDaoImpl:

```
@Transactional
public class PalavraDaoImpl implements PalavraDao {
void insert(Palavra palavra);

}
```

Classe ExpressaoDaoImpl:

```
@Transactional
public class ExpressaoDaoImpl implements ExpressaoDao {
void insert(Expressao expressao);

}
```

Por sua vez, quanto à classe de Serviço nomeada *ExpressaoServiceImpl*, note que ela trabalha com um método interno (método insert) que chama duas operações internas referente aos dados. A primeira operação insere a palavra no Banco de Dados, ou seja, abre a transação, insere e fecha a transação com o Banco de Dados. Na linha a seguir, a operação representada pela linha de código expressao.setPalavra(palavra) está recebendo aquela palavra (dado) que foi inserida no comando anterior e, a seguir, será executado um insert da expressão.

Classe ExpressaoServiceImpl:

```
@Service
public class ExpressaoServiceImpl implements
```

```
ExpressaoService {
    @Autowired
    private PalavraDao palavraDao;
    @Autowired
    private ExpressaoDao expressaoDao;
    void insert(PalavraDao palavraDao,Expressao expressao){
        palavraDao.insert(palavra);
        expressao.setPalavra(palavra);
        expressaoDao.insert(expressao);

}
```

O problema acontece nesse ponto: se ocorrer algum problema na inserção da expressão, a operação sofrerá um **rollback**. O commit então não será executado, e todas as operações serão revertidas. O "grande porém" é que esse rollback não afetará a inserção da palavra, pois ela já sofreu o commit.

5.2 Gerenciamento de transações na classe de Serviço

A classe de serviço é uma Camada que fica acima da camada de persistência. Uma classe Service usa **anotação @Service**. Podemos, ainda, dizer que as classes de serviço funcionam como se fossem facilitadoras no acesso dos Models por meio do framework de persistência.

Utilizaremos o mesmo exemplo exposto anteriormente, o Dicionário de expressões de TI (Tecnologia da Informação). A seguir, temos as classes PalavraDaoImpl e ExpressaoDaoImpl. Observe que, nessa codificação, não colocamos a anotação @Transactional nas classes DAOs.

Classe PalavraDaoImpl:

```
public class PalavraDaoImpl implements PalavraDao {
void insert(Palavra palavra);

}
```

Classe ExpressaoDaoImpl:

```
public class ExpressaoDaoImpl implements ExpressaoDao {
void insert(Expressao expressao);

}
```

Quem recebe a anotação @Transactional, por sua vez, é a nossa classe de serviço (*ExpressaoServiceImpl)*. Nossa classe de serviço tem um método que chama as duas operações referentes aos dados.

Diferentemente da outra implementação, se houver falha no insert da expressão, acontecerá um rollback, que afetará também o insert da palavra, pois a transação com o banco de dados agora acontece na classe de serviço.

Segue a implementação da classe de Serviço. Especial atenção para a **anotação @Service e @Transactional**.

Classe ExpressaoServiceImpl:

```
@Service
@Transactional(readOnly=false)
public class ExpressaoServiceImpl implements
ExpressaoService {
@Autowired
private PalavraDao palavraDao;
@Autowired
private ExpressaoDao expressaoDao;
void insert(PalavraDao palavraDao ,Expressao expressao){
palavraDao.insert(palavra);
expressao.setPalavra(palavra);
expressaoDao.insert(expressao);

}
```

Obviamente, o gerenciamento de transações tem suas vantagens e desvantagens tanto na classe DAO quanto na classe de serviço, e a decisão de onde vai acontecer o gerenciamento depende do contexto e dos requisitos da aplicação.

FINALIZANDO

Nesta etapa, exploramos conceitos fundamentais que são a espinha dorsal do desenvolvimento de software moderno. Aprofundamos nossa compreensão acerca da Classe Java Revisitada, explorando a serialização como uma ferramenta valiosa para a persistência de objetos, e resgatamos o conceito de parametrização. A Persistência de Dados (JPA) introduziu-nos à Java Persistence API, uma tecnologia que simplifica e padroniza o acesso a dados em aplicações Java. Ao abordarmos o Padrão DAO, compreendemos a importância de encapsular a lógica de acesso a dados, promovendo a separação de preocupações e facilitando a manutenção do código.

O Object-Relational Mapping (ORM) emergiu como uma peça-chave, conectando o mundo orientado a objetos ao universo relacional do banco de dados de maneira transparente.

As Anotações proporcionaram uma abordagem declarativa, simplificando a configuração e acelerando o desenvolvimento. Vimos, ainda, que compreender o escopo transacional (com base no entendimento das classes persistentes e do ciclo de vida dos objetos) é essencial para garantir a consistência dos dados, evitando problemas comuns relacionados à manipulação de dados em aplicações Java. E, por fim, a integração harmoniosa com a "Classe de Serviço" e o "Gerenciamento de Transações" assegura a consistência e a confiabilidade das operações, enquanto a "Camada de Controle" atua como orquestradora, garantindo uma interação fluida entre a lógica de negócios e a apresentação.

REFERÊNCIAS

ALUR et al. **Core J2EE Patterns**: best practices and design Strategies. São Paulo: Prentice Hall PTR, 2003.

CORDEIRO, G. **Aplicações Java para web com JSF e JPA**. São Paulo: Casa do Código, 2012.

DEITEL, P. J.; DEITEL, H. M. **Java: como programar**. 10. ed. São Paulo: Pearson, 2017.

HIBERNATE. Hibernate ORM. Disponível em: <https://hibernate.org/orm/>. Acesso em: 16 abr. 2024.

OPENJPA. OpenJPA user's guide. Disponível em: <https://openjpa.apache.org/docs/openjpa-0.9.0-incubating/manual/manual.html>. Acesso em: 16 abr. 2024.

ORACLE. Core J2EE Patterns – data access object. Disponível em: <https://www.oracle.com/java/technologies/data-access-object.html>. Acesso em: 16 abr. 2024.

ORACLE. Core J2EE Patterns – data access object. 2001-2002, Sun Microsystem. Disponível em: <https://www.oracle.com/java/technologies/dataaccessobject.html>. Acesso em: 16 abr. 2024.

ORACLE. What is a JPA Entity? Disponível em: <https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/undejbs003.htm#BABIAAGE>. Acesso em: 16 abr. 2024.