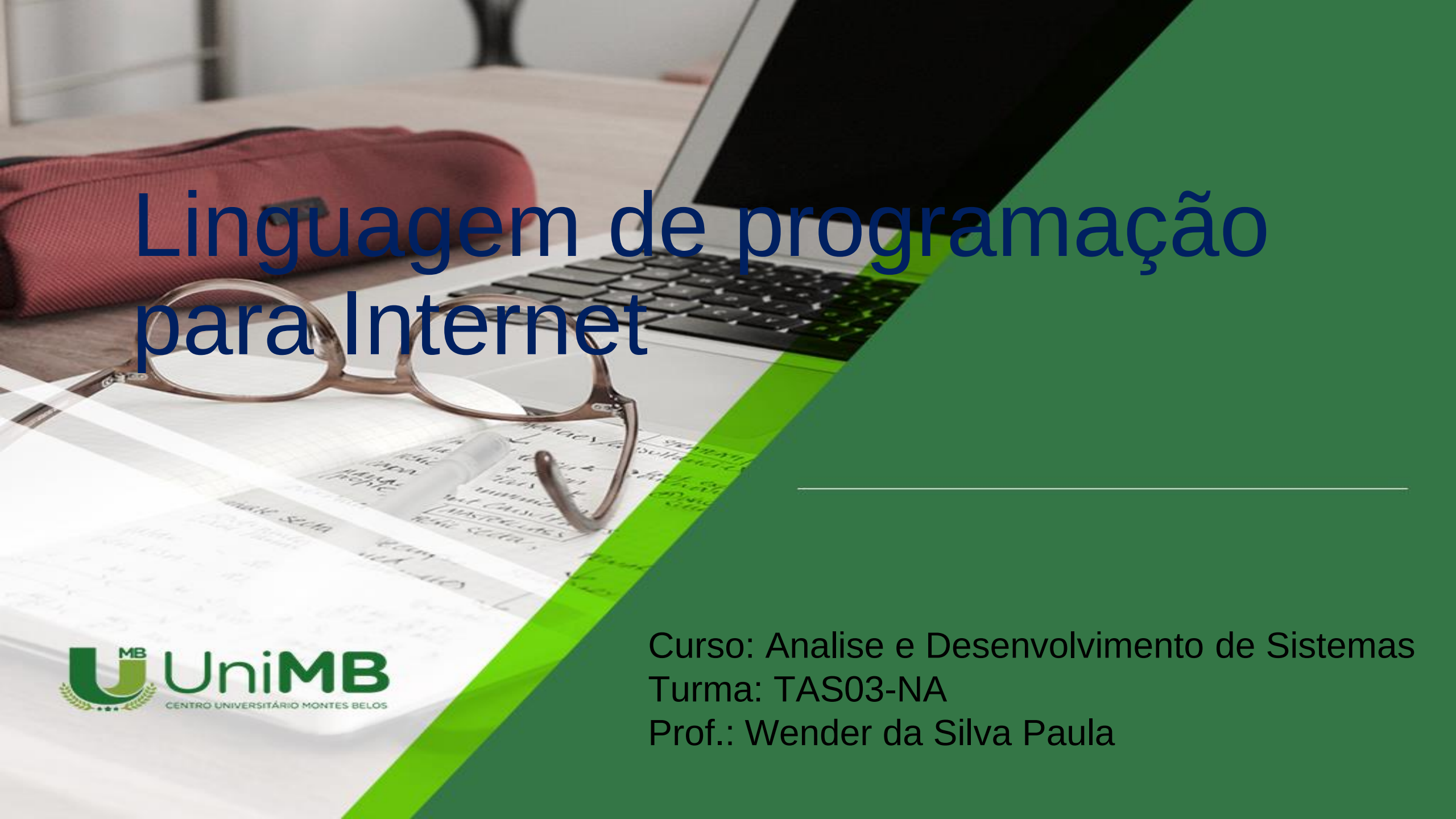


Linguagem de programação para Internet

UniMB
CENTRO UNIVERSITÁRIO MONTES BELOS

Curso: Analise e Desenvolvimento de Sistemas
Turma: TAS03-NA
Prof.: Wender da Silva Paula

Agenda

- **Introdução à JPA - Java Persistence API**

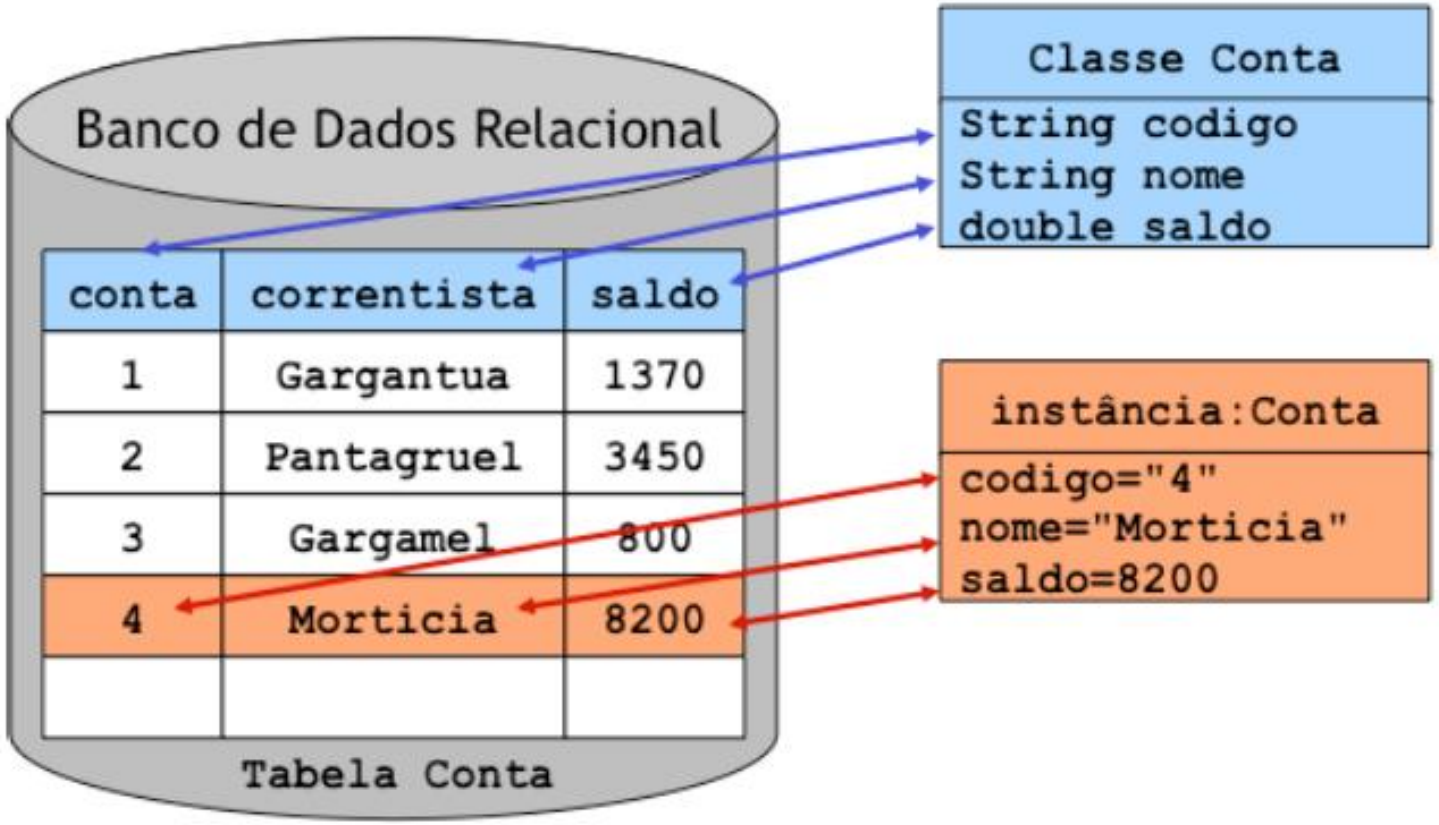
JPA - Java Persistence API

- *JPA – Java Persistence API* é uma especificação do Java EE que define um mapeamento entre uma estrutura relacional e um modelo de objetos em Java. JPA pode ser também usado de forma *standalone*, em aplicações Java SE, utilizando um provedor de persistência independente, mas é um componente obrigatório e nativo de qualquer servidor Java EE.

- **JPA** permite a criação de objetos persistentes, que retém seu estado além do tempo em que estão na memória. Oferece uma camada de persistência que permite pesquisar, inserir, remover e atualizar objetos, além de um mecanismo de mapeamento objeto-relacional (ORM) declarativo.

- **Mapeamento objeto-relacional** em aplicações JPA consiste na declaração de mapeamentos entre classes e tabelas, e atributos e colunas em tempo de desenvolvimento, e da sincronização de instancias e registros durante o tempo de execução.

JPA - Java Persistence API



JPA - Java Persistence API

Unidade de Persistência e persistence.xml

Para configurar JPA em uma aplicação é necessário que essa aplicação tenha acesso, através do seu Classpath, aos seguintes componentes (geralmente empacotados em JARS):

- Módulo contendo a API do JPA (as classes, interfaces, métodos, anotações do pacote javax.persistence)
- Um provedor de persistência JPA que implementa a especificação (Ex: Hibernate, EclipseLink)
- Drivers necessários para configuração do acesso aos datasources usados (e pools de conexão, se houver)
- Além disso, a aplicação deverá incluir um arquivo persistence.xml contendo a configuração da camada de persistência.

JPA - Java Persistence API

- O conjunto de **entity managers** e as entidades que eles gerenciam configura uma **unidade de persistência** (Persistence Unit). Essa configuração é declarada no arquivo ***persistence.xml*** e referenciada nas classes que usam os objetos persistentes, como **DAOs**, **facades**, **session beans, servlets**, etc.
- O arquivo ***persistence.xml*** deve estar no Classpath acessível pelas classes Java usadas na aplicação em **/META-INF/persistence.xml**, e possui a seguinte configuração mínima:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

  <persistence-unit name="MyPU" transaction-type="JTA">   </persistence-unit>

</persistence>
```

JPA - Java Persistence API

- Este exemplo é utilizável em um servidor Java EE que tenha sido configurado para oferecer um datasource default (java:comp/DefaultDataSource). Normalmente aplicações usam datasources selecionadas de forma explícita. Este segundo exemplo é típico para uma aplicação JPA simples em um servidor Java EE 7:

```
<persistence version="2.1" …>
    <persistence-unit name="com.empresa.biblioteca.PU" transaction-type="JTA">

        <jta-data-source>jdbc/Biblioteca</jta-data-source>
        <class>br.empresa.biblioteca.entity.Livro</class>
        <properties>
            <property name="eclipselink.deploy-on-startup" value="true" />
        </properties>

    </persistence-unit>
</persistence>
```

JPA - Java Persistence API

- Ele declara o tipo de transações utilizada (JTA – gerenciada pelo container), o nome JNDI de um datasource específico (que precisa ser previamente configurado no servidor de aplicações), uma entidade mapeada, e uma propriedade de configuração do provedor usado (tipicamente, e principalmente quando usa-se um provedor que não é nativo do servidor, há várias outras propriedades específicas do provedor de persistência que precisam ser configuradas).

JPA - Java Persistence API

- **Entidade e mapeamento**

A arquitetura do JPA baseia-se no mapeamento de tabelas a classes, colunas e atributos. O objetivo do mapeamento é a construção de uma hierarquia de entidades.

Entidade (Entity) é o nome dado a um objeto persistente que em JPA é representado por um JavaBean (ou POJO) mapeado a uma tabela (um JavaBean/POJO é basicamente uma classe Java com atributos privativos acessíveis via métodos get/set e que contém um construtor default sem argumentos.) Uma entidade JPA pode ser mapeada a uma tabela de duas formas:

- Através de elementos XML no arquivo orm.xml

- Através de anotações antes de declarações de classes, métodos, atributos, construtores.

JPA - Java Persistence API

- **Entidade e mapeamento**

Os mapeamentos podem ser declarados através de anotações nas próprias classes Java que representam os objetos persistentes.

Para declarar uma entidade com anotações JPA é necessário no mínimo anotar a classe com *@Entity* e anotar pelo menos um atributo (ou método) com *@id*, indicando sua chave primária.

```
@Entity
public class Estado implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    private Long id;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

JPA - Java Persistence API

- **Entidade e mapeamento**

Por default, todos os atributos da classe, expostos via métodos get/set serão mapeados a colunas de mesmo nome. O nome da classe será usado por default como nome da tabela.

O valor e o tipo usado no ID é importante para a operação do mecanismo de persistência transitiva no JPA, portanto, é comum que seja um valor sequencial gerado automaticamente.

Se houver necessidade adicional de configuração, é possível informar nomes de colunas, tabelas, tipos, limites, relacionamentos, estratégias e diversos outros mecanismo para viabilizar o mapeamento usando anotações como ***@Table, @Column, @CollectionTable, @JoinTable, @JoinColumn***, etc.

JPA - Java Persistence API

- **Exemplo de mapeamento**

```java
@Entity
@Table(name = "estado")
public class Estado implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    private Long id;
    @Column(name = "nome")
    private String nome;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

}
```

JPA - Java Persistence API

- **Classe EntityManager**

As classes que irão instanciar o EntityManager podem obtê-lo via consulta JNDI global a um servidor, se houver um container cliente configurado para tal, ou instacia-lo diretamente. No exemplo abaixo, a configuração foi feita localmente, instanciando diretamente uma fábrica local para obter o EntityManager da unidade de persistência especifica.

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;


public class TesteJPA {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("ExemploJPAPU");
        EntityManager em = emf.createEntityManager();

    }

}
```

JPA - Java Persistence API

- **Classe EntityManager**

Observe que o parâmetro passado ao método de fabrica do *EnitityManagerFactory* corresponde ao nome da unidade de persistência exatamente como foi declarada no *persistence.xml*.

A entidade a ser sincronizada com a tabela é instanciada e configurada normalmente em Java puro.

```
public class TesteJPA {

    public static void main(String[] args) {


        Estado estado = new Estado();
        estado.setNome("GOIAS");

    }

}
```

JPA - Java Persistence API

Como é um objeto novo será usado o método **persist()** para sincronizar o objeto com o banco. Nesse tipo de configuração também poderia ser usado o método **merge(),** que distingue objetos novos (inserções) de antigos (atualizações) através do ID. Tanto **merge()** como **persist()** precisam ser chamados dentro de um contexto transacional, e como as transações são locais, é preciso obtê-las do **EntityManager**:

```
public class TesteJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("ExemploJPAPU");
        EntityManager em = emf.createEntityManager();
        try {
            em.getTransaction().begin();
            Estado estado = new Estado();
            estado.setNome("GOIAS");
            em.persist(estado);
            em.getTransaction().commit();
        } catch (Exception e) {
            em.getTransaction().rollback();
        } finally {
            em.close();
        }
    }
}
```

JPA - Java Persistence API

Ambiente Java EE

Em java EE o suporte a JPA é nativo, e a configuração do banco de dados deve ser preferencialmente realizada através de um datasource previamente configurado no servidor, e que possa ser acessado dentro de um contexto JTA

O arquivo *persistence.xml* declara estratégia transacional JTA (gerenciado pelo container) e em vez de informar dados para acesso ao driver do banco, informa-se o caminho JNDI para o datasource que corresponde ao banco usado.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="exemplojavawebPU" transaction-type="JTA">
    <jta-data-source>jdbc/ExemploJavaWeb_JNDI</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

JPA - Java Persistence API

Ambiente Java EE

Em java EE o *EntityManager* poderá ser obtido via JNDI ou via injeção de dependências através de *@PersistenceContext* (ou ainda via CDI, se previamente configurado). Neste exemplo de acesso em servidor Java EE através de um EJB, um Session Bean, cujos métodos são transacionais por default. Neste caso não é necessário (nem permitido) escrever nenhum código de delimitação para transações, já que ele será iniciada antes do método começar, e será cometida após o término do método, provocando um rollback se houver exceção (as políticas transacionais podem posteriormente ser configuradas via anotações do EJB). Se houver apenas uma unidade de persistência, o nome dela não precisa ser informado ao injetar o *@PersistenceContext*.

```
@Stateless
public class CidadeFacade extends AbstractFacade<Cidade> implements CidadeFacadeRemoto {

    @PersistenceContext(unitName = "ExemploJSFPU")
    private EntityManager em;
```

JPA - Java Persistence API

Exemplo EJB

```java
@Stateless
public class EstadoFacade  {

    @PersistenceContext(unitName = "ExemploJSFPU")
    private EntityManager em;

    public void salvar(Estado estado) {
        em.persist(estado);
    }

    public void excluir(Estado estado) {
        em.remove(estado);
    }

    public Estado pesquisar(Long id) {
        return (Estado)em.find(Estado.class, id);
    }

    public List<Estado> pesquisar(String param) {
        List<Estado> lstEstado;
        Query consulta = em.createNamedQuery("Estado.findByNome");
        consulta.setParameter("nome", param + "%");
        lstEstado = consulta.getResultList();
        return lstEstado;
    }

}
```

JPA - Java Persistence API

No exemplo anterior, o JPQL foi declarado na classe que define a entidade usando o *@NamedQuery*. Essa forma permite que os queries sejam referenciados pelo nome:

```
@Entity
@Table(name = "estado")
@NamedQueries({
    @NamedQuery(name = "Estado.findByNome",
            query = "SELECT e FROM Estado e WHERE e.nome LIKE :nome "
                    + "ORDER BY e.nome")
})
public class Estado implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

JPA - Java Persistence API

Persistência - Operações CRUD

Operações CRUD (Create, Retrieve, Update, Delete – Criar, Recuperar, Atualizar, Remover) são as principais tarefas realizadas pela camada de persistência. Podem envolver uma única entidade ou uma rede de entidades interligados via relacionamentos. JPA oferece várias maneiras de realizar operações CRUD:

- Através dos métodos de persistência transitiva: persist, merge, delete, find;
- Através do JavaPersistence Query Language (JPQL);
- Através da API Criteria;
- Através da SQL nativo;

JPA - Java Persistence API

Persistência - Operações CRUD

A recuperação de uma entidade, o “R” de “Retrieve” do acrônimo CRUD, pode ser feita conhecendo-se sua chave primária através do find:

Estado estado = (Estado)em.find(Estado.class, 10);

Normalmente a recuperação envolve uma pesquisa no estado da entidade, então pode-se usar JPQL através de um Query (ou TypedQuery):

```
TypedQuery query =
        em.createQuery(“SELECT e FROM Estado e WHERE e.id=:id”, Estado.class);
Query.setParameter(“id”, 10);
Estado estado = query.getSingleResult();
```

JPA - Java Persistence API

Persistência - Operações CRUD

Criteria é um query que é construído dinamicamente, através da construção de relacionamentos entre objetos. É ideal para pesquisas que são construídas em tempo de execução:

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Estado> criteria = builder.createQuery(Estado.class);
Root<Estado> queryRoot = criteria.form(Estado.class);
Criteria.where(builder.equal(queryRoot.get(Estado_.id), 10));
TypedQuery(Estado> q = em.createQuery(criteria);
Estado estado = q.getSingleResult();
```

JPA - Java Persistence API

Persistência - Operações CRUD

Para usar a API desta forma, uma classe Estado_class foi gerada (veja: Criteria Metamodel API) automaticamente pelas ferramentas da IDE.

Operações que envolvem atualização, “Create”, “Update” e “Delete”, geralmente são executadas através dos métodos de persistência transitiva persist, merge e delete. Por exemplo, para alterar o nome do estado dentro de um contexto transacional pode-se usar:

```
Estado estado = em.find(Estado.class, 10);
Estado.setNome(“Novo nome”);
Em.merge(estado);
```

JPA - Java Persistence API

Mapeamento

Mapeamento de relacionamentos entre entidades

Associações no JPA funcionam da mesma maneira que associações de objetos em Java. São naturalmente unidirecionais e não são gerenciados pelo container, o que significa que e preciso escrever e chamar explicitamente o código Java necessário para atualizar os campos correspondentes de cada uma instancias que participam do relacionamento, quando dados são adicionados, removidos ou alterados (o container, porém pode gerenciar a propagação da persistência de forma transitiva através de configuração de operações de cascata).
Os relacionamentos são sempre associações entre ***entidades*** (diretamente, ou indiretamente através de coleções). Pode-se também configurar associações entre objetos que não são relacionamentos. Neste caso, para que o estado desses objetos seja persistente, é preciso que esteja dependentes de uma entidade.

JPA - Java Persistence API

Mapeamento

Mapeamento de relacionamentos entre entidades

Para mapear relacionamentos, JPA utiliza informações da estrutura do código Java e das anotações. As anotações podem ser mínimas. Serão usados parâmetros *default* até onde for possível. Assim como *@Column* pode ser usado para informar um nome de coluna diferente do nome da propriedade escalar, *@JoinColumn* pode ser usado quando a coluna que contém a chave estrangeira da associação tiver um nome ou configuração diferente da propriedade mapeada.

A cardinalidade define o número de entidade que existe em cada lado do relacionamento. Podem ser mapeados dois valores: “muitos” e “um”, resultando em quatro possibilidades: Um para muitos; Muitos para um; Um para um; e Muitos para muitos;

JPA - Java Persistence API

Mapeamento

Mapeamento de relacionamentos entre entidades

As duas primeiras são equivalentes, portanto há três diferentes associações:

- *@ManyToOne / @OneToMany*
- *@OneToOne*
- *@ManyToMany*

Dentro de cada contexto, é importante também levar em conta a direção dos relacionamentos, que pode ser unidirecional ou bidirecional.

JPA - Java Persistence API

Mapeamento

Mapeamento de relacionamentos entre entidades

@ManyToOne é a associação mais comum. O modelo relacional é naturalmente um-para-muitos e uma referência simples de objetos é um-para-muitos. Em associação unidirecionais, apenas um lado possui referência para o outro (que não tem conhecimento da ligação). A anotação é usada apenas em um dos lados. Em associações bidirecionais, do outro lado deve possuir uma anotação *@OneToMany* e é preciso informar o nome da referência que determina a associação.

@OneToOne requer constraints (que são gerados) nas tabelas para garantir a consistência da associação. É também possível estabelecer uma relação um-para-um entre entidades usando *@ManyToOne*, desde que seja unidirecional o lado "*many*" seja limitado a um único elemento.

JPA - Java Persistence API

Mapeamento

Mapeamento de relacionamentos entre entidades

@ManyToMany mapeia três tabelas a dois objetos. Também é possível estabelecer uma relação muitos-para-muitos entre entidades usando *@ManyToOne*, desde que haja três objetos, dois contendo associações *@ManyToOne* para um terceiro. Isto pode ser usado se for necessário representar a associação como uma entidade, mapeada a uma tabela via *@JoinTable*.

Várias anotações e atributos permitem configurar ajustes finos dos mapeamentos, como regras de cascade, estratégias de carga, caches, transações e direção, etc.