Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte
Campus Natal-Central
Diretoria Acadêmica de Gestão e Tecnologia da Informação

Tecnologia em Análise e Desenvolvimento de Software

Java Persistence API

Prof. Fellipe Aleixo
fellipe.aleixo@ifrn.edu.br

O Princípio: Os *Entity Beans*

- No JEE 5, a persistência deu origem a uma especificação própria – Java Persistence API
  - Abstração de um nível maior que o JDBC
  - Mecanismo de ORM (*object/relational mapping*)
  - Mapeia automaticamente objetos Java para e a partir de um banco de dados relacional
  - Oferece linguagem de consulta semelhante à SQL para trabalhar com objetos
  - As "classes" de entidades são POJOs

Definindo as Classes de "Entidade"

Realizando o mapeamento objeto-relacional

Classes de Entidade

- As classes de entidade representam conceitos – do modelo de domínio do sistema
  - Aqueles para os quais há a necessidade de guardar informações entre uma execução e outra do sistema (informações persistentes)
  - Os conceitos para os quais não necessitamos guardar informações são ditos transientes
- Uma classe de entidade encapsula os dados e regras de negócio associadas a um conceito

Classes de Entidade: Requisitos

1. Anotada com javax.persistence.Entity
2. Deve possuir um construtor, sem argumentos, público ou protegido
   - Pode ter outros construtores
3. Não pode ser declarada como final
4. Se uma instância for passada como argumento de um objeto remoto, a classe de entidade deve implementar a interface Serializable
5. Podem estender outras classes (entidades ou não)
6. Atributos persistentes devem ser declarados como privados ou protegidos – acessados por métodos

Classes de Entidade: Exemplo

- Em um sistema de biblioteca, um conceito importante a ser persistido é o Livro

```
package exemplo.biblioteca;
import javax.persistence.*;
@Entity
public class Livro implements java.io.Serializable {
        private long id;
        private String titulo;
        private String isbn;

        @Id
        public long getId() { return id; }
        public void setId(long novo) { id = novo; }
        ...
}
```

Classes de Entidade

- Java Persistence exige apenas dois metadados na definição de um bean e entidade:
  - @javax.persistence.Entity
    - Denota que a classe em questão precisa ser mapeada para o banco de dados
  - @javax.persistence.Id
    - Marca a propriedade a ser utilizada como chave primária
- O provedor de persistência assume:
  - Que o nome da classe será o nome da tabela
  - Cada um dos atributos será mapeado em coluna

Classes de Entidade

- A anotação @Entity possui um atributo name que é utilizado para referenciar a entidade dentro de uma expressão JPQL
  - Por padrão, esse nome é o do da classe do bean
- Alternativa: usar o arquivo orm.xml

```
<entity-mappings>
        <entity class="exemplo.biblioteca.Livro" access="PROPERTY">
                <attributes>
                        <id name="id"/>
                </attributes>
        </entity>
</entity-mappings>
```

Classes de Entidade: Atributos

- Tipos primitivos Java
- java.lang.String
- Outros tipos "serializáveis", incluindo:
  - Wrappers para tipos primitivos Java
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.TimeStamp
  - User-defined serializable types
  - byte[]
  - Byte[]
  - char[]
  - Character[]
- Enumerações Java
- Outras entidades e/ou coleções de entidades
- Classes embutidas (*embeddable*)

Mapeamento Relacional Básico

- É sugerido que o desenvolvimento da aplicação inicie pelo (1º) modelo de objetos, do qual derivará o (2º) esquema do banco de dados
  - Embora JPA possibilite o caminho inverso
  - Seguindo a sugestão, não precisará utilizar anotações como @Table e @Column
  - Parte dos fornecedores oferece ferramentas para auto-gerar código Java correspondente às entidades

Mapeamentos Elementares de Esquema

- Dado que desejamos chegar na definição da seguinte tabela do banco de dados:

```
create table TABELA_LIVRO
(
        LIVRO_ID integer primary key not null,
        TITULO varchar(255) not null,
        ISBN varchar(20) not null
);
```

- Podemos modificar a definição da classe Livro que fizemos para se adequar a essa tabela do banco de dados

Mapeamentos Elementares de Esquema

```
@Entity
@Table(name="TABELA_LIVRO")
public class Livro implements java.io.Serializable {
        private long id;
        private String titulo;
        private String isbn;
        @Id
        @Column(name="LIVRO_ID", nullable=false, columnDefinition="integer")
        public long getId() { return id; }
        public void setId(long novo) { id = novo; }
        @Column(name="TITULO", nullable=false)
        public String getTitulo() { return titulo; }
        public void setTitulo(String novo) { titulo = novo; }
        @Column(name="ISBN", length=20, nullable=false)
        public String getIsbn() { return isbn; }
        public void setIsbn(String novo) { isbn = novo; }
}
```

Mapeamentos Elementares de Esquema

- @Table – informa ao EntityManager a tabela relacional para a qual a classe de entidade será mapeada
- @Column – informa o nome da coluna da tabela correspondente a um dado atributo da entidade
  - Descreve também a maneira específica como uma propriedade em particular é mapeada para uma coluna no banco de dados

Mapeamentos Elementares de Esquema

```
public @interface Column
{
        String name( ) default “”;
        boolean unique( ) default false;
        boolean nullable( ) default true;
        boolean insertable( ) default true;
        boolean updatable( ) default true;
        String columnDefinition( ) default “”;
        String table( ) default “”;
        int legnth( ) deafult 255;
        int precision( ) default 0;
        int scale( ) default 0;
}
```

- Informações utilizadas na geração automática do esquema do banco de dados correspondente
- É possível definir as mesmas restrições no XML

Mapeamentos Elementares de Esquema

```
<entity-mappings>
    <entity class="exemplo.biblioteca.Livro" access="PROPERTY">
        <table name="TABELA_LIVRO"/>
        <attributes>
            <id name="id">
                <column name="LIVRO_ID" nullable="false"
                    column-definition="integer"/>
            </id>
            <basic name="titulo">
                <column name="TITULO" nullable="false"/>
            </basic>
            <basic name="isbn">
                <column name="ISBN" nullable="false" length="20"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

Validação de Atributos Persistentes

- O mecanismo de validação das classes de entidade permite a definição de restrições (através de anotações) para cada atributo
- Exemplos de validação:
  - Não poder ter valor nulo
  - Obedecer certos padrões (expressões regulares)
  - Representar uma data passada

Validação de Atributos Persistentes

```
@Entity
public class Contact implements Serializable {
    ...
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp="[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
       +"[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
       +"(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?",
           message="{invalid.email}")
    protected String email;
    @Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
           message="{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
           message="{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(javax.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

Chaves Primárias

- Uma chave primária é a identificação de um determinado *bean* de entidade
  - Cada *bean* de entidade deve ter uma
- A chave primária pode ser mapeada para uma ou mais propriedades
- O atributo "chave" deverá ser do tipo:
  - Tipos primitivos Java (incluindo as wrapper classes)
  - java.lang.String
  - Ou uma classe de chave primária, composta por tipos primitivos e Strings

Chaves Primárias

- Uma chave primária em uma propriedade simples é identificada pela anotação @javax.persistence.Id
- A chave primária pode ser automaticamente gerada pelo provedor de persistência
  - @javax.persistence.GeneratedValue
  - Pode ser definida a estratégia utilizada para a geração automática
  - Por padrão os provedores de persistência precisam fornecer a geração de chave primária primitiva

Chaves Primárias

```
package exemplo.biblioteca;
import javax.persistence.*;
@Entity
public class Atendente implements java.io.Serializable {
        private long id;
        private String nome;
        private String sobrenome;

        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        public long getId() { return id; }

        public void setId(long novo) { id = novo; }
        public String getNome() { return nome; }
        public void setNome(String novo) { nome = novo; }
        public String getSobrenome() { return sobrenome; }
        public void setSobrenome(String novo) { sobrenome = novo; }
}
```

Chaves Primárias

- Equivalente em XML:

```
<entity-mappings>
        <entity class="exemplo.biblioteca.Atendente"
                                access="PROPERTY">
                <attributes>
                        <id name="id">
                                <generated-value strategy="IDENTITY"/>
                        </id>
                </attributes>
        </entity>
</entity-mappings>
```

Classes de Chave Primária e Chaves Compostas

- A Java Persistence possui várias formas de especificar chaves compostas, dentre elas:
  - @javax.persistence.IdClass
    - Indica o nome da classe que será utilizada como chave primária nas interações com o EntityManager
    - Na classe do *bean* são indicadas as propriedades que fazem parte da chave composta com a anotação @Id
    - As propriedades marcadas devem mapear exatamente as propriedades da classe que foi identificada
  - @javax.persistence.EmbeddedId
    - Indica que a classe de chave primária está embutida na própria classe do *bean*

Classes de Chave Primária e Chaves Compostas

```
package exemplo.biblioteca;
public class AtendentePK implements java.io.Serializable {
        private String sobrenome;
        private long rg;

        public AtendentePK() { }

        public AtendentePK( String sobrenome, long rg) {
                this.sobrenome = sobrenome;
                this.rg = rg;
        }

        public String getSobrenome() { return sobrenome; }
        public void setSobrenome(String novo) { sobrenome = novo; }

        public long getRg() { return rg; }
        public void setRg(long novo) { rg = novo; }
        ...
```

Classes de Chave Primária e Chaves Compostas

```
        public boolean equals(object obj) {
                if (obj == this) return true;
                if (!(object instanceof AtendentePK)) return false;
                AtendentePK pk = (AtendetePK) obj;
                if (!sobrenome.equals(pk.sobrenome)) return false;
                if (rg != pk.rg) return false;
                return true;
        }

        public int hashCode() {
                return sobrenome.hasCode() + rg;
        }
}
```

- Após criar a classe que será utilizada, a classe do *bean* de entidade pode ser alterada

Classes de Chave Primária e Chaves Compostas

```
package exemplo.biblioteca;
import javax.persistence.*;
@Entity
@IdClass(AtendentePK.class)
public class Atendente implements java.io.Serializable {
        private String nome;
        private String sobrenome;
        private long rg;

        public String getNome() { return nome; }
        public void setNome(String novo) { nome = novo; }
        @Id
        public String getSobrenome() { return sobrenome; }
        public void setSobrenome(String novo) { sobrenome = novo; }
        @Id
        public long getRg() { return rg; }
        public void setRg(long novo) { rg = novo; }
}
```

Classes de Chave Primária e Chaves Compostas

- Equivalente em XML:

```
<entity-mappings>
        <entity class="exemplo.biblioteca.Atendente"
                                access="PROPERTY">
                <id-class>exemplo.biblioteca.AtendentePK</id-class>
                <attributes>
                        <id name="sobrenome"/>
                        <id name="rg"/>
                </attributes>
        </entity>
</entity-mappings>
```

Mapeamento de Propriedades

- JPA oferece um conjunto adicional de anotações que permitem um melhor mapeamento de propriedades com características específicas:
  - Transientes
  - Tipo de recuperação
  - BLOB
  - Versões (concorrência)
  - etc.

@Transient

- Propriedades marcadas como transientes são ignoradas pelo EntityManager

```
@Entity
public class Atendente implements java.io.Serializable {
        private long id;
        private String nome;
        private long cpf;
        @id
        public long getId() { return id; }
        public void setId(long novo) { id = novo; }
        @Transient
        public long getDigitoVerificadorCpf() { return ...; }
        ...
}
```

@Basic e FetchType

- @Basic
  - Forma mais simples para o mapeamento para uma propriedade persistente (de um tipo wrapper)
    - java.lang.String, byte[], Byte[], char[], Character[], java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp
  - Normalmente essa especificação não é necessária pois é descoberta pelo provedor de persistência
  - É útil nos casos em que precisamos definir o atributo ***fetch*** – tipo de carregamento

@Basic e FetchType

- Tipos de carregamento (FetchType):
  - Sob demanda – LAZY
  - Imediato – EAGER
- Carregamento LAZY permite que provedor de persistência otimize o acesso à banco de dados
  - Uma determinada propriedade pode não ser carregada até que você a acesse especificamente
  - Pode minimizar a quantidade de dados carregada em uma consulta
  - O provedor pode optar pelo carregamento imediato

@Basic e FetchType

```
package exemplo.biblioteca;
import javax.persistence.*;
@Entity
public class Atendente implements java.io.Serializable {
        private long id;
        private String nome;
        private String sobrenome;
        @Id
        @GeneratedValue
        public long getId() { return id; }
        public void setId(long novo) { id = novo; }
        @Basic(fetch=FetchType.LAZY, optional=false)
        public String getNome() { return nome; }
        public void setNome(String novo) { nome = novo; }
        @Basic(fetch=FetchType.LAZY, optional=false)
        public String getSobrenome() { return sobrenome; }
        public void setSobrenome(String novo) { sobrenome = novo; }
}
```

@Temporal

- Fornece informações adicionais ao provedor de persistência sobre o mapeamento de
  - java.util.Date ou java.util.Calendar
  - Tipos temporais: DATE, TIME e TIMESTAMP

```
@Entity
public class Atendente implements java.io.Serializable {
        private long id;
        private String nome;
        private java.util.Date horaCriacao;
        @Id
        public long getId() { return id; }
        @Temporal(TemporalType.TIME)
        public Date getHoraCriacao() { return horaCriacao; } ...
}
```

@Lob

- Utilizada para o mapear objetos grandes
  - java.sql.Blob – byte[], Byte[] ou Serializable
  - java.sql.Clob – char[], Character[] ou String

```
import com.acme.imaging.JPEG;
@Entity
public class Atendente implements java.io.Serializable {
        private long id;
        private String nome;
        private JPEG foto;
        @Id
        public long getId() { return id; }
        @Lob
        @Basic(fetch=FetchType.LAZY)
        public JPEG getFoto() { return foto; }
}
```

@Enumerated

- Mapeia enumerações (ORDINAL ou STRING)

```
public enum TipoAtendente {BIBLIOTECARIA, NORMAL, ESTAGIARIA}
@Entity
public class Atendente implements java.io.Serializable {
        private long id;
        private String nome;
        private TipoAtendente tipo;
        @Id
        @GeneratedValue
        public long getId() { return id; }
        public void setId(long novo) { id = novo; }
        @Enumerated(EnumType.STRING)
        public TipoAtendente getTipo() { return tipo; }
        ...
}
```

Mapeamentos Avançados

- @SecondaryTable
  - Permite que uma entidade lógica seja mapeada para várias tabelas no banco de dados
  - São definidos:
    - Nome da tabela secundária e atributo de ligação (JOIN)
    - De qual tabela uma coluna faz parte
- @Embeddable
  - Permite embutir (@Embedded) objetos (@Embeddable) dentro de um *bean* de entidade
  - Os atributos da classe embutida podem aparecer como propriedades (@AttributeOverrides)

Java Persistence API: EntityManager

Interagindo com o "gerente de entidades"

Persistência

- As ações que envolvem a persistência são realizadas por meio do serviço javax.persistence.EntityManager
  - Responsável pela persistência das entidades
  - Gerencia o mapeamento O/R
  - Fornece API para: criar consultas, localizar objetos, sincronizar objetos e inserir objetos no banco de dados
  - Fornece armazenamento em *cache*
  - Co-gerencia as transações que envolvem entidades

Entidades Gerenciadas e não Gerenciadas

- Uma entidade pode estar gerenciada (acoplada) ou não gerenciada (desacoplada) pelo EM
- Quando uma entidade esta acoplada ao EM:
  - O EM monitora as alterações no estado da entidade
  - Sincroniza essas alterações com o banco de dados
- Um contexto de persistência é um conjunto de instâncias gerenciadas de uma dada entidade

Contexto de Persistência

- O EM monitora as alterações feitas nos objetos de entidade dentro do contexto de persistência
- As alterações são sincronizadas com o banco de dados seguindo as regras do modo *flush*
- Quando um contexto de persistência é fechado, as instâncias dos objetos de entidade gerenciadas são desacopladas
- Há dois contextos de persistência:
  - Contexto de persistência com escopo de transação
  - Contexto de persistência estendido

Contexto de Persistência com Escopo de Transação

- Duram pelo tempo de uma transação e são fechados quando uma transação se completa
  - Abaixo temos um método invocado no contexto de transação: a entidade permanece gerenciada durante a transação – e sincronizadas ao acabar

```
@PersistenceContext(unitName="biblioteca")
EntityManager entityManager;

@TransactionAttribute(REQUIRED)
public Livro metodoQualquer() {
        Livro livro = entityManager.find(Livro.class, 1);
        livro.setTitulo("Novo Título");
        return livro;
}
```

Contexto de Persistência Estendido

- Perpassam o tempo de uma transação
- Instâncias de um objeto de entidade acopladas a um contexto de persistência permanecem gerenciadas mesmo com a transação concluída
- Podem ser criados e gerenciados pelo código da aplicação – *beans* de sessão com informação de estado

Entidades Desacopladas

- As instâncias de entidade tornam-se desacopladas quando o contexto de persistência é fechado
  - Essas instâncias podem ser serializadas e envidadas a um cliente remoto
  - Este cliente pode realizar alterações nessas instâncias
  - Este cliente pode serializá-las novamente e enviá-las de volta ao servidor a fim de serem mescladas e sincronizadas com o banco de dados

Empacotando uma Unidade de Persistência

- Um EntityManager mapeia um conjunto fixo de classes para um banco de dados específico
  - Esse conjunto denomina-se Unidade de Persistência
  - Uma unidade de persistência é definida em um arquivo
  - Esse arquivo persistence.xmlé um descritor de implantação específico para Java Persistence
  - Em um arquivo persistence.xml podem ser definidos uma ou mais unidades de persistência
  - O arquivo persistence.xml está localizado no diretório META-INF do EJB-JAR

Empacotando uma Unidade de Persistência

- No persistence.xml são definidas as entidades e as propriedades de configuração de cada unidade de persistência
- O conjunto de entidades de uma unidade de persistência pode ser especificado no descritor ou identificado pelo provedor de persistência
  - A identificação ocorre automaticamente através da pesquisa no JAR da aplicação, procurando @Entity
- Cada unidade de persistência é associada a apenas uma origem de dados

Empacotando uma Unidade de Persistência

- A raiz do esquema XML do persistence.xml é o elemento <persistence>, que contém um ou mais elementos <persistence-unit>
- Cada unidade de persistência possui
  - Dois atributos: name e transaction-type (opcional)
  - Subelementos:
    - <description> - descrição (opcional)
    - <provider> - provedor de persistência (opcional)
    - <jta-data-source> - fonte de dados, suporte JTA (opcional)
    - <non-jta-data-source> - fonte de dados (opcional)

Empacotando uma Unidade de Persistência

- Sub-elementos (continuação):
  - <mapping-file> - arquivo de mapeamento O/R (opcional)
  - <jar-file> - arquivo JAR adicional com entidades (opcional)
  - <class> - identificação de uma entidade (opcional)
  - <properties> - propriedades (opcional)
  - <exclude-unlisted-classes> - exclui classes de entidade não listadas (opcional)

Empacotando uma Unidade de Persistência

```
<persistence>
        <persistence-unit name="biblioteca">
                <jta-data-source>java:/PostgreDS</jta-data-source>
                <properties>
                        <property name="org.hibernate.hbm2ddl">
                                update
                        </property>
                </properties>
        </persistence-unit>
</persistence>
```

Empacotando uma Unidade de Persistência

- Exemplo de persistence.xml:

```
<persistence>
        <persistence-unit name="biblioteca">
                <jta-data-source>java:/PostgreDS</jta-data-source>
                <class>exemplo.biblioteca.Livro</class>
                <class>exemplo.biblioteca.Atendente</class>
                <class>exemplo.biblioteca.Emprestimo</class>
                <exclude-unlisted-classes/>
                <jar-file>../lib/usuario.jar</jar-file>
                <properties>
                        <property name="org.hibernate.hbm2ddl">
                                update
                        </property>
                </properties>
        </persistence-unit>
</persistence>
```

Empacotando uma Unidade de Persistência

- O conjunto final das classes de entidade é dada pela união dos seguintes elementos:
  1. Classes anotadas com @Entity no JAR do arquivo persistence.xml
  2. Classes anotadas com @Entity em quaisquer JARs listados com o elemento <jar-file>
  3. Classes mapeadas no arquivo META-INF/orm.xml
  4. Classes mapeadas em quaisquer arquivos XML referenciados pelo elemento <mapping-file>
  5. Classes listadas com elementos <class>

EntityManagerFactory

- O EntinyManager pode ser criado ou obtido de uma EntityManagerFactory
  - Aplicação JSE precisa usar a EntityManagerFactory

```
Package javax.persistence;

public interface EntityManagerFactory {
        EntityManager createEntityManager();
        EntityManager createEntityManager(java.util.Map map);
        void close();
        boolean isOpen();
}
```

  - Um Map é passado para sobrescrever ou estender as propriedades específicas do provedor de persistência não informadas no persistence.xml

EntityManagerFactory

- No JSE, a classe javax.persistence.Persistence é responsável pela inicialização de uma EntityManagerFactory
  - Método createEntityManagerFactory()
    - Informando o nome da unidade de persistência
    - E opcionalmente um Map contendo propriedades
  - No ato da criação é procurado persistence.xml correspondente
  - No JSE é recomendável usar o método close() para fechar a EntityManagerFactory para liberar recursos

EntityManagerFactory

- A referência ao EntityManagerFactory pode ser injetada diretamente em um atributo ou em um método *setter* - @PersistenceUnit

```
import javax.persistence.*;
import javax.ejb.*;
@Stateless
public MeuBean implements MinhaInterfaceDeNegocio {
        @PersistenceUnit(unitName="negocioPU")
        private EntityManagerFactory factory;
        private EntityManagerFactory factory2;
        @PersistenceUnit(unitName="auxiliarPU")
        public void setFactory2(EntityManagerFactory f) {
                this.factory2 = f;
        }
}
```

Obtendo um EntityManager

- Um contexto de persistência pode ser criado chamando o método createEntityManager()
  - A instância retornada representa um contexto de persistência estendido
  - No caso de componentes que gerenciam transações com JTA – o EntityManager precisa ser instruído a participar da transação
    - EnityManager.joinTransaction()
- Para facilitar um EntityManager pode ser injetado diretamente - @PersistenceContext

Obtendo um EntityManager

```
@Stateless
public MeuBean implements MinhaInterfaceDeNegocio {
        @PersistenceContext(unitName="negocioPU")
        private EntityManager entityManager;
        ...
}
```

- A anotação @PersistenceContext funciona semelhantemente à @PersistenceUnit
  - Com a exceção que esta injeta diretamente uma instância do EntityManager
  - Por padrão, o EntityManager injetado é de um contexto de persistência com escopo de transação

Obtendo um EntityManager

- Um gerenciador de entidades com contexto de persistência estendido só pode ser injetado em um *bean* de sessão com informação de estado

```
@Stateful
public MeuBeanStateful implements MeuStatefulRemote {
        @PersistenceContext(unitName="negocioPU",
                            type=PersistenceContextType.EXTENDED)
        private EntityManager entityManager;
        ...
}
```

  - Nesse caso o contexto de persistência tem o mesmo ciclo de vida do *bean*

Interagindo com um EntityManager

```
package javax.persistence;
public interface EntityManager {
        public void persist(Object entity);
        public <T>T find(Class <T>entityClass,obj primKey);
        public <T>T getReference(Class <T>entityClass,obj primKey);
        public <T>T merge(T entity);
        public void remove(Object entity);
        public void lock(Object entity,LockModeType lockMode);

        public void refresh(Object entity);
        public boolean contains(Object entity);
        public void clear();

        public void joinTransaction();
        public void flush();
        public FlushModeType getFlushMode();
        public void setFlushMode(FlushModeType type);
        ...
```

Interagindo com um EntityManager

```
        public Query createQuery(String queryString);
        public Query createNamedQuery(String name);
        public Query createNativeQuery(String sqlString);
        public Query createNativeQuery(String sqlquery, String resultSetMapping);
        public Query createNativeQuery(String sqlQuery, Class resultClass);

        public Object getDelegate();

        public void close();
        public boolean isOpen();
}
```

- Com uma referência a um EntityManager é possível inserir, remover e mesclar entidades com o banco de dados, além de criar consultas específicas

Persistindo Entidades

- É o ato de inseri-las em um banco de dados
  1. alocar uma instância da entidade
  2. definir as propriedades da entidade
  3. definir também os possíveis relacionamentos
  4. solicitar ao EntityManager para "persistir" à entidade – que passa a ser gerenciada

```
Aluno aluno = new Aluno();
aluno.setNome("João");
aluno.setMatricula("12345");
aluno.setCurso(cursoTADS);  // cursoTADS instanciado antes

entityManager.persist(aluno);
```

Persistindo Entidades

- O momento em que a real inserção no banco depende de algumas variáveis
- Com o persist chamado em uma transação, a inserção pode acontecer imediatamente ou enfileirada até o fim da transação
  - Depende do modo de *flush* configurado
  - A inserção imediata pode ser forçada invocando-se o método flush()
- O persist só pode ser invocado de fora de uma transação se o contexto desta for estendido

Persistindo Entidades

- Em caso de relacionamentos, essas entidades também poderão ser criadas no banco
  - Se forem configuradas as diretivas para cascata
- Na execução do método persist podem ser lançadas as seguintes exceções:
  - IllegalArgumentException – se o parâmetro não for um tipo de entidade
  - TransactionRequiredException – se for invocado fora do escopo de uma transação (exceto no caso se for um contexto de persistência estendido)

Localizando Entidades

- Dois mecanismos para localizar entidades:
  - Métodos próprios do gerenciador de entidades
  - Criando e executando consultas específicas
- Métodos do EntityManager para a localização de entidades: find() e getReference()

```
public interface EntityManager {
        <T>T find(Class<T> entityClass, Object primaryKey);
        <T>T getReference(Class<T> entityClass, Object primaryKey);
}
```

  - O método find retorna null se a entidade não for encontrada e usa por padrão a diretiva de carregamento sob demanda – lazy-loading

Localizando Entidades

- Exemplo:

```
Aluno aluno;
aluno = entityManager.find(Aluno.class, 2);
```

  - Localização de um aluno com chave primária de 2
  - Conversão: tipo primitivo em Object (autoboxing)
- Com o método getReference() se a entidade não for encontrada é lançada uma exceção: javax.persistence.EntityNotFoundException

```
Aluno aluno = null;
try {
        aluno = entityManager.getReference(Aluno.class, 2);
} catch (EntityNotFoundExcepion naoEncontrada) { ... }
```

Localizando Entidades

- Tanto find() quanto getReference() lançam IllegalArgumentException se seus parâmetros não forem um tipo de entidade
- Ambos podem ser invocados fora do escopo de transação:
  - Os objetos retornados são desacoplados quando o contexto de persistência tiver escopo de transação
  - O objetos retornados permanecem gerenciados quando o contexto de persistência for estendido

Localizando Entidades

- Os objetos persistentes pode ser localizados utilizando-se JPA-QL
  - Para tal é necessário criar uma consulta – Query

```
public interface EntityManager {
        Query createQuery(String queryString);
        Query createNamedQuery(String name);
        Query createNativeQuery(String sql);
        Query createNativeQuery(String sql,String resultSetMap);
        Query createNativeQuery(Striog sqlQuery,Class resultClass);
}
```

  - Exemplo:

```
Query query = entityManager.createQuery
                                ("SELECT a FROM Aluno a WHERE id=2");
Aluno aluno = (Aluno) query.getSingleResult();
```

Atualizando Entidades

- As entidades criada ou localizadas permanecem gerenciadas até o contexto de persistência ser fechado
  - Alterações no estado da classe de entidade são sincronizadas automaticamente com o banco

```
@PersistenceContext (unitName="negocioPU")
EntityManager entityManager;

@TransactionAttribute(REQUIRED)
public void atualizaMatricula(int id, String novaMatricula) {
        Aluno aluno = entityManager.find(Aluno.class, id);
        aluno.setMatricula(novaMatricula);
}
```

Mesclando Entidades

- A especificação Java *Persistence* permite que alterações feitas em entidades desacopladas possam ser **mescladas** com o banco de dados
  - Utilizado o método **merge()**
- Exemplo – código de um cliente remoto

```
// acervo é um bean de sessão
Livro livro = acervo.recuperaLivro(1);
livro.setAutor("Novo Autor");
acervo.atualizaLivro(livro);
```

Mesclando Entidades

```
@TransactionType(REQUIRED)
public void atualizaLivro(Livro atualizado) {
        Livro copia = entityManager.merge(atualizado);
}
```

- Se o *EntityManager* ainda não estiver gerenciando nenhuma entidade Livro com a mesma chave primária (id), é criada e retornada uma cópia – que permanece acoplada
- Se o *EntityManager* já estiver gerenciando uma instância com a mesma chave primária, o conteúdo é copiado para essa instância e esta é retornada

Removendo Entidades

- Uma entidade é removida do banco através do método remove() do *EntityManager*

```
@TransactionType(REQUIRED)
public void removeLivro(int id) {
        Livro livro = entityManager.find(Livro.class, id);
        if (livro != null) {
                entityManager.remove(livro);
        }
}
```

  - Depois do remove() a instância é desacoplada
  - As entidades relacionadas poderão ser removidas
  - Só pode ser desfeita recriando a entidade (persist)

refresh()

- O *EntityManager* permite também que o estado de uma entidade possa ser atualizado com as informações do banco de dados – refresh()
  - Sobrescreve quaisquer alterações realizadas na entidade que não tiverem sido sincronizadas com o banco de dados
  - Se a entidade tiver outras relacionadas, estas podem ser atualizadas em cascata (se configurado)

flush() e FlushModeType

- Ao chamar os métodos persist(), merge() ou remove(), essas alterações não são simultaneamente sincronizadas com o banco
- Para forçar tal sincronização podemos chamar a qualquer momento o método flush()
- O *EntityManager* também permite que seja configurado o mecanismo de *flush*
  - Método setFlushMode() - parâmetro segundo a enumeração javax.persistente.FlushModeType (AUTO ou COMMIT)

Bloqueios e Delegate

- *EntityManager* suporta tanto bloqueios de leitura como de gravação
  - Utilizado o método lock()
  - Uso associado ao conceito de transação
- O método getDelegate() do EntityManager permite obter uma referência ao objeto provedor de persistência
  - A maioria dos fornecedores tem extensões de API para a interface *EntityManager* – é possível o acesso através de um *casting* nesse objeto delegado

contains() e clear()

- contains()
  - Recebe uma instância de entidade como parâmetro
  - Retornará true se tal instância estiver sendo atualmente gerenciada
- clear()
  - Permite desacoplar todas as instância de entidade gerenciadas de um contexto de persistência
  - Todas as alterações feitas nas entidades gerenciadas são perdidas – é prudente chamar flush() antes de clear()

Recurso de Transações Locais

- Num contexto não-JEE é possível a gerência de uma transação específica para a Java *Persistence*
  - Método getTransaction() do *EntityManager*
  - Retorna uma referência EntityTransaction

```
public interface EntityTransaction {
        public void begin();
        public void commit();
        public void rollback();
        public boolean isActive();
}
```