Java Persistence API

About JPA

Introduction

Introduzione

- Spesso il primo passo nello sviluppo di applicazioni enterprise consiste nella creazione del **domain model**
- Il **domain model** è l'immagine concettuale del problema che il sistema deve risolvere, esso descrive oggetti e relazioni ma non si occupa di definire come il sistema agisce su tali oggetti.
- In EJB 3 la persistenza è gestita mediante **Java Persistence API (JPA)**.

About JPA

- Cos'è Java Persistence API (JPA)?
  - Tecnologia di persistenza su DB per Java
    - Motore Object-relational mapping (ORM)
    - Lavora con entità POJO (Plain Old Java Object)
    - Simile ad Hibernate e JDO
  - JPA mappa le classi Java alle tabelle di un DB
    - Mappa le relazioni tra tabelle come associazioni tra classi
  - Fornisce funzionalità di CRUD
    - Create, read, update, delete

History of JPA

- Storia di JPA
  - Creata come parte del framework EJB 3.0 all'interno della JSR 220
  - Rilasciato a Maggio 2006 come parte della Java EE 5
  - Può essere usata come libreria standalone
- API Standard con molteplici implementazioni
  - OpenJPA
  - Hibernate
  - TopLink JPA
  - DataNucleus

Entities

Defining Simple Entity Classes

Anatomy of an Entity

- È una classe POJO
- Per marcare un POJO come un oggetto del domain model (entity bean) si fa uso dell'annotazione @Entity
- Tutte le entità non astratte devono avere un costruttore vuoto pubblico o protetto che viene usato per creare una nuova istanza usando l'operatore new
- Una delle caratteristiche più interessanti di JPA è che supporta completamente le caratteristiche di ereditarietà e polimorfismo della programmazione Object Oriented.
  - E' possibile cioè avere delle entità che estendono altre entità o classi non entità.

The Minimal Entity

- Deve essere annotata come un'entità
  - annotazione *@Entity* sulla classe:

```
@Entity
public class Employee { … }
```

- Entity entry in un file di mapping XML

```
<entity class="com.acme.Employee"/>
```

Ereditarietà e Polimorfismo di un Entity

- Una delle caratteristiche più interessanti di JPA è che supporta completamente le caratteristiche di ereditarietà e polimorfismo della programmazione Object Oriented.
  - E' possibile cioè avere delle entità che estendono altre entità o classi non entità.

Ereditarietà e Polimorfismo di un Entity

```
@Entity
public class Utente
{
	...
	String id;
	String nome;
	String cognome;
	...
}

@Entity
public class Giocatore extends Utente
{
    ...
}

@Entity
public class Amministratore extends Utente
{
	...
}
```

ORM Mappings

Annotations or XML

Object/Relational Mapping

- Mappa lo stato di un oggetto persistente ad un DB relazionale
- Mappa le relazioni ad altre entità
- I Metadata possono essere annotazioni o XML (o entrambi)

Identifier

- Deve avere un id univoco (primary key):

```
@Entity
public class Employee {
  @Id int id;

  public int getId() { return id; }
  public void setId(int id) { this.id = id; }
}
```

Persistent Identity (Id)

- Identificatore (id) nell'entity → primary key nel DB
- Identifica in maniera univoca l'entità in memoria e nel DB
- Differenti definizioni di ID
  - ID semplici
  - ID composto
  - ID Embedded

Definizioni Id

- ID semplici – singoli campi/proprietà
- ID Class - ID composto da campi multipli
- ID Embedded – singolo campo o classe PK

Definizioni Id
Esempio @Id

```
@Entity
public class Categoria
{
	...
	protected Long id;
	...

	@Id
	public Long getId()
	{
	    return this.id;
	}

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

	...
}
```

Definizioni Id
Esempio @IdClass (1)

```
public class CategoriaPK implements Serializable {
        String nome;
        Date dataCreazione;

        public CategoriaPK() {}

        public boolean equals(Object oggetto) {
                if (oggetto instanceof CategoriaPK) {
                        CategoriaPK altrachiave = (CategoriaPK)oggetto;
                        return (altrachiave.nome.equals(nome) &&
                           altrachiave.dataCreazione.equals(dataCreazione));
                }
                return false;
        }

        public int hashCode() {
            return super.hashCode();
        }
}
```

Definizioni Id
Esempio @IdClass (2)

```
@Entity
@IdClass(CategoriaPK.class)
public class Categoria {

        public Categoria() {}

        @Id protected String nome;

        @Id protected Date dataCreazione;

        ...
}
```

- CategoriaPK è designata come IdClass per Categoria
- 2 campi marcati con l'annotazione @Id
  - questi due campi sono presenti anche nella classe CategoriaPK.
- Il metodo equals nella classe CategoriaPK confronta i due campi che costituiscono la Primary
  - Il persistence provider a runtime determina se due oggetti Categoria sono uguali copiando i campi marcati con @Id nei corrispondenti campi di CategoriaPK e usando il metodo equals.
- Ogni IdClass deve essere Serializable e fornire una implementazione hashCode valida.
- svantaggio ridondanza di codice per garantire l'utilizzo multiplo dell'annotazione @Id.

Definizioni Id
Esempio @EmbeddedId (1)

```
@Embeddable
public class CategoriaPK {
        String nome;
        Date dataCreazione;

        public CategoriaPK() {}

        public boolean equals(Object oggetto) {
        if (oggetto instanceof CategoriaPK) {
                CategoriaPK altrachiave = (CategoriaPK)oggetto;
                return (altrachiave.nome.equals(nome) &&
                    altrachiave.dataCreazione.equals(dataCreazione));
                }
                return false;
        }

        public int hashCode()
        {
            return super.hashCode();
        }
}
```

Definizioni Id
Esempio @EmbeddedId (2)

```
@Entity
public class Category {
        public Category() {}

        @EmbeddedId
        protected CategoriaPK categoriaPK;
        ...
}
```

- i campi identità name e createDate sono assenti dalla classe Categoria
  - Al loro posto viene usato un oggetto categoriaPK annotato con @EmbeddedId.
- L'unica differenza per quanto riguarda l'oggetto CategoriaPK è che questo non deve essere Serializable.
- L'annotazione @Id è omessa in quanto ridondante.
- L'annotazione @Embeddable è usata per progettare oggetti persistenti che non hanno una propria identità ma che sono identificati dall'entità dell'oggetto all'interno del quale sono innestati.
- Un oggetto Embeddable non può avere un'identità propria e nella maggior parte dei casi viene mappato nello stesso record dell'oggetto che lo incapsula materializzandosi soltanto nel mondo Object Oriented.

Esempio @Embeddable

```
@Embeddable
public class Indirizzo {
	protected String via;
	protected int civico;
	protected String citta;
	protected int cap;
	protected String provincia;
	...
}

@Entity
public class Utente {
	@Id
	protected Long id;
	protected String nome;
	protected String cognome;
	@Embedded
	protected Indirizzo indirizzo;
	protected String email;
...
}
```

Identifier Generation

- ID possono essere generati nel DB
  - `@GeneratedValue` sull'ID

```
@Id @GeneratedValue
 int id;
```

- 3 strategie predefinite di generazione:
  - IDENTITY, SEQUENCE, TABLE
- Possono pre-esistere o essere generati
- La strategia AUTO indica che il provider sceglierà per noi una strategia

Using Identity or Sequence

- Usare una identity (colonna di auto-increment nel DB) per la generazione dell'Id:

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

- Usare una sequence DB per la generazione dell'Id:

```
@Id
@GeneratedValue(generator="UsersSeq")
@SequenceGenerator(name="UsersSeq",
  sequenceName="USERS_SEQ")
private long id;
```

Field-based access e Property-based access

- Field-based access quando l'ORM è definito utilizzando le variabili istanza dell'entità
  - Se si vuole usare il field–based access allora occorre dichiarare pubblici i campi del POJO
- Property-based access quando il mapping avviene facendo riferimento alle proprietà dell'entità.
- Non è possibile usare contemporaneamente i due tipi di accesso.

Field-based access e Property-based access

- È possibile evitare che una proprietà diventi persistente marcandola con l'annotazione @Transient.
- Definire un campo con il modificatore transient è equivalente ad applicare l'annotazione @Transient.
- I tipi di dati che possono essere resi persistenti sono:
  - i tipi primitivi
  - il tipo String
  - le classi che implementano l'interfaccia Serializable
  - gli Array
  - le collezioni di entità
  - le classi annotate con @Embeddable

Simple Column Mappings

- Per mappare una proprietà ad una colonna del DB:

```
@Entity
public class Message {
  private String message;
  public void setMessage(String msg) { message = msg; }
  public String getMessage() { return message; }
}
```

- Può essere esplicitato il nome di una colonna:

```
@Column(name="SAL")
private double salary;
```

Simple Mappings

```
@Entity
public class Customer {
    @Id
    int id;

    String name;
    @Column(name="CREDIT")
    int c_rating;
    @Lob
    Image photo;
}
```

Simple Mappings

```
<entity class="example.Customer">
  <attributes>
    <id name="id"/>
    <basic name="c_rating">
      <column name="CREDIT"/>
    </basic>
    <basic name="photo"><lob/></basic>
  </attributes>
</entity>
```

Relationship Mappings

- Una relazione essenzialmente si traduce nel fatto che un'entità fa riferimento ad un'altra.

- Supportati tutti i più comuni mapping di relazione
  - @ManyToOne, @OneToOne – single entity
  - @OneToMany, @ManyToMany – collection

- Una relazione può essere unidirezionale o bidirezionale a seconda di dove sono piazzati i riferimenti.

Relationship Mappings @OneToOne

- Usata per marcare una relazione unidirezionale o bidirezionale uno ad uno.

```
@Entity
public class Utente {
    @Id
    protected String id;
    @OneToOne
    protected InfoUtente Info;
}

@Entity
public class InfoUtente {
    @Id
    protected Long id;
    protected String info;
    protected String info2;
        ...
}
```

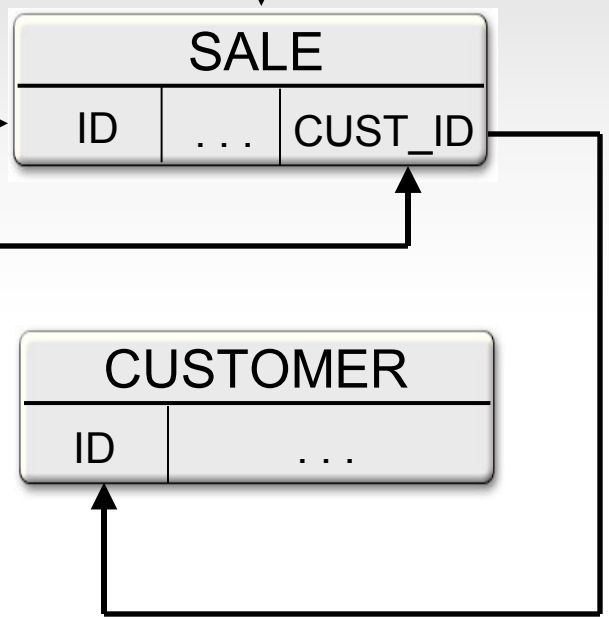
Relationship Mappings @OneToMany e @ManyToOne

- Le relazioni uno a molti e molti a molti sono molto comuni; in queste relazioni un'entità può mantenere uno o più riferimenti ad un'altra.
- In java si fa uso delle classi Set o List.
- Se l'associazione è bidirezionale allora da un lato sarà uno a molti (@OneToMany) e dall'altro molti a uno (@ManyToOne).

ManyToOne Mapping

```
@Entity
public class Sale {
    @Id
    int id;

    ...
    @ManyToOne
    Customer cust;
}
```

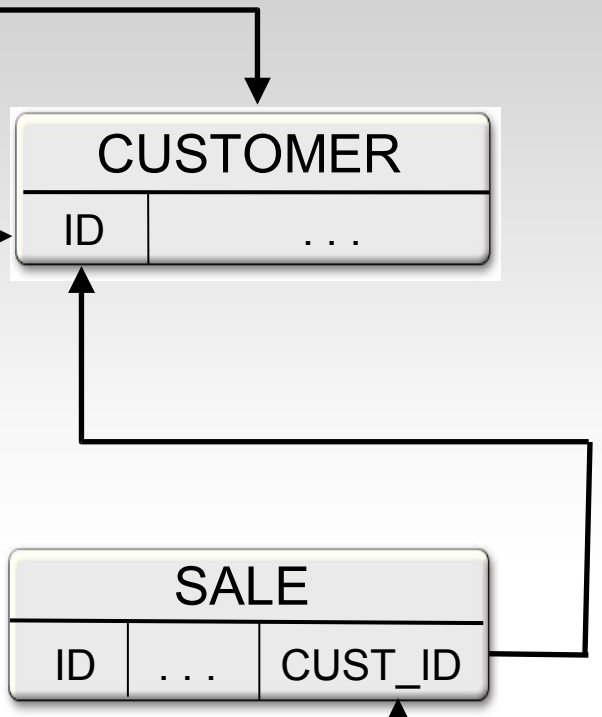


ManyToOne Mapping

```
<entity class="example.Sale">
  <attributes>
    <id name="id" />
    ...
    <many-to-one name="cust" />
  </attributes>
</entity>
```

OneToMany Mapping

```
@Entity
public class Customer {
   @Id
   int id;
   ...
   @OneToMany(mappedBy="cust")
   Set<Sale> sales;
}
@Entity
public class Sale {
   @Id
   int id;
   ...
   @ManyToOne
   Customer cust;
}
```



OneToMany Mapping

```
<entity class="example.Customer">
  <attributes>
    <id name="id" />
      ...
    <one-to-many name="sales" mapped-
     by="cust"/>
  </attributes>
</entity>
```

Relationship Mappings @OneToMany e @ManyToOne

```
@Entity
public class Categoria {
        @Id protected Long id;
        protected String titolo;
        protected String descrizione;
        @OneToMany(mappedBy="categoria") protected Set<Bid> articoli;
}

@Entity
public class Articolo {
        @Id protected Long id;
        protected String titolo;
        protected String autore;
        @ManyToOne protected Categoria categoria;
}
```

Relationship Mappings @ManyToMany

- Le relazioni molti a molti sono quelle nelle quali da entrambi i lati della relazione è possibile fare riferimento a più istanze di uno stesso oggetto.

```
@Entity
public class Associazione {
	@Id protected Long id;
	protected String nome;
	@ManyToMany protected Set<Persona> persone;
}

@Entity
public class Persona {
@Id protected Long id;
protected String nome;
protected String cognome;
 @ManyToMany(mappedBy="persone") protected Set<Associazione> associazioni;
}
```

Persistent Context ed `EntityManager`

Manipolare Entità del DB

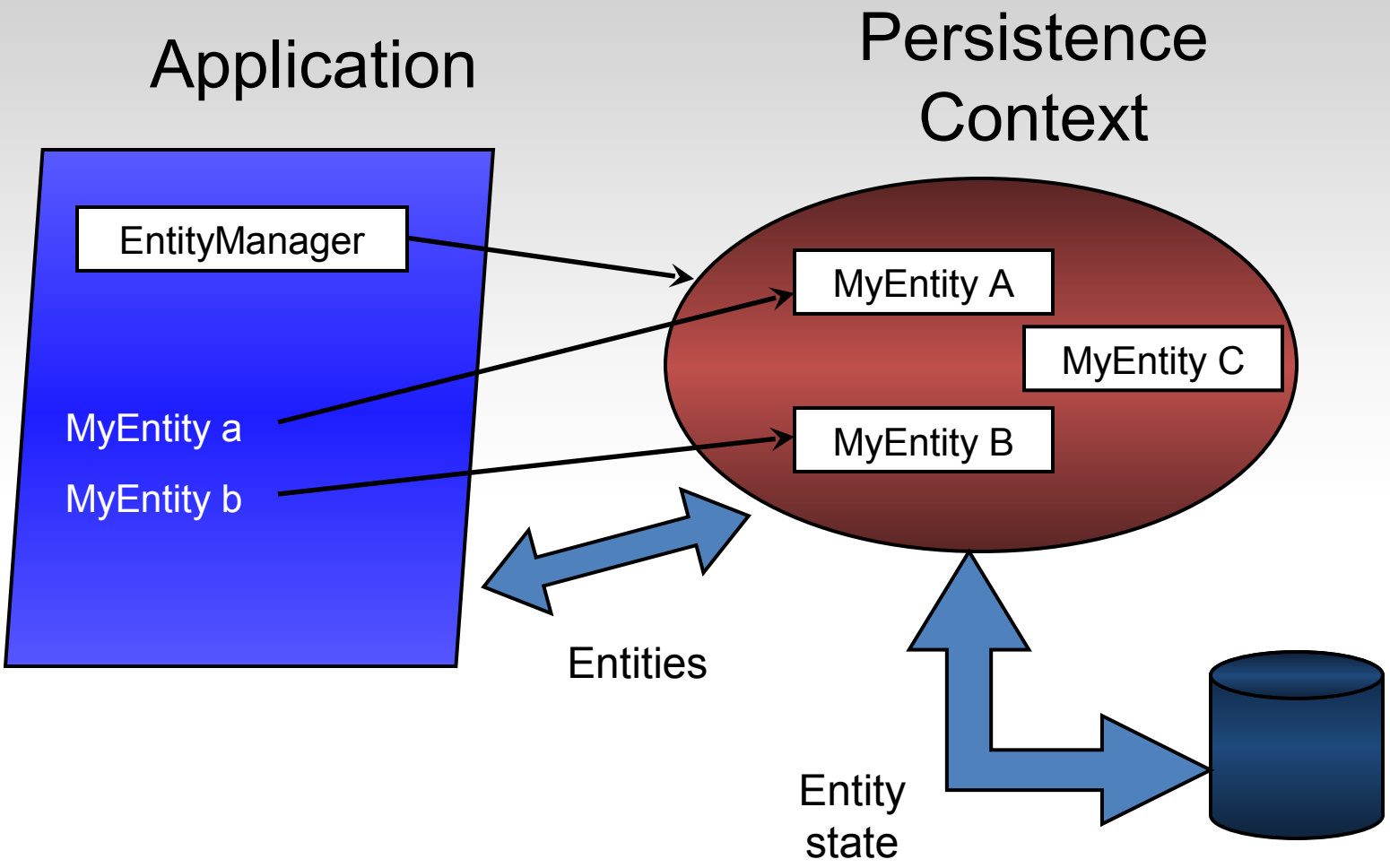
Persistence Context (PC)

- Gioca un ruolo vitale nelle funzionalità interne dell'EntityManager.
- Gestito da un'`EntityManager`
  - Il contenuto di un PC cambia a seguito di operazioni sulle API di un `EntityManager`
- Una collezione di entità gestita dall'EntityManager all'interno di un dato scope.
  - Resi unici da una persistent identity
    - Solo una entità con un persistent ID può esistere nel PC
  - Aggiunta al PC, ma non rimovibile individualmente ("detached")

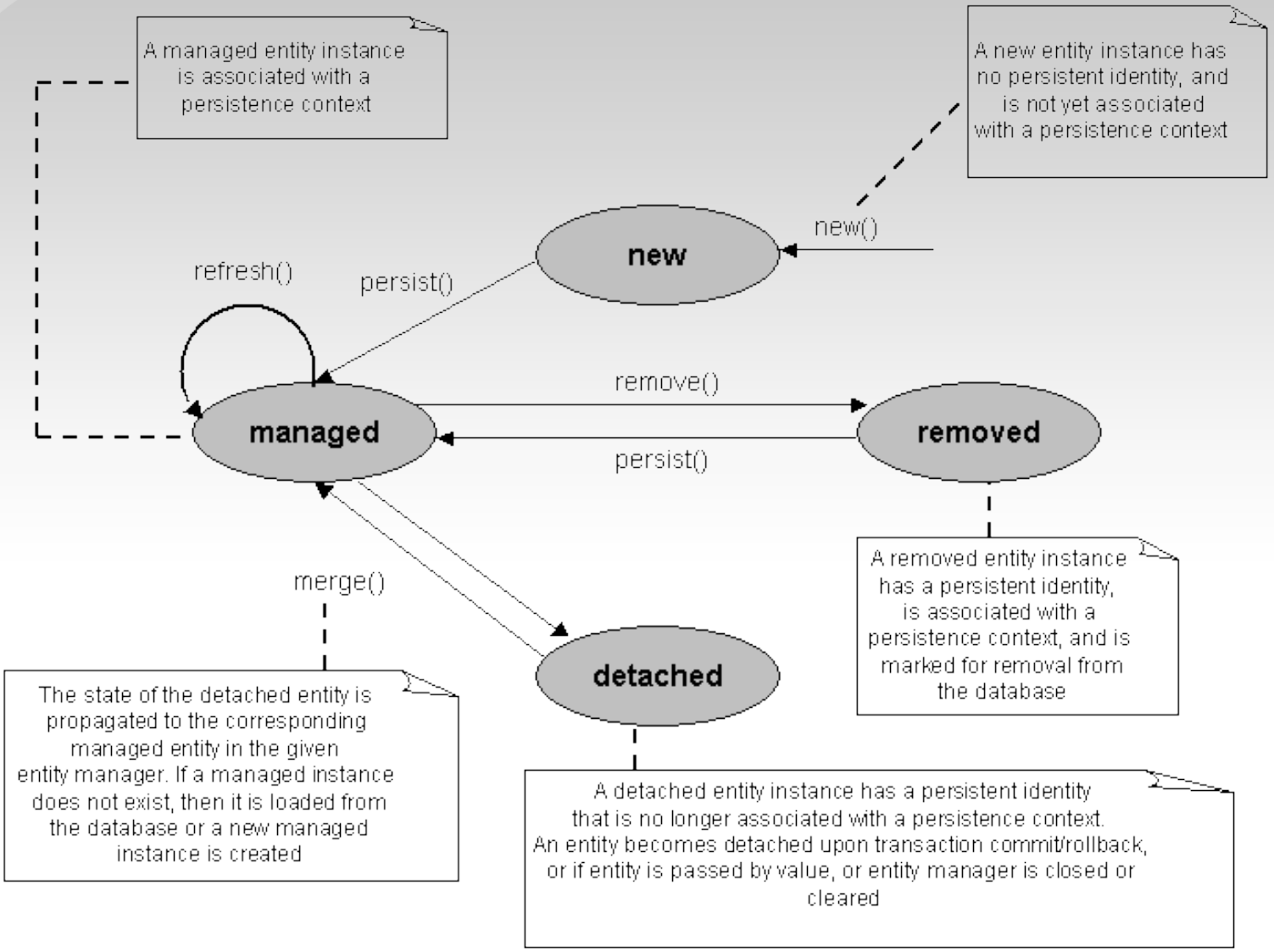
Persistence Context (PC)

- Lo scope è l'intervallo di tempo entro il quale le entità rimangono gestite
- Esistono due tipi differenti di persistence scope:
  - extended:
    - l'EntityManager può essere usato esclusivamente con sessionbean di tipo stateful.
    - Una volta che l'entità viene attaccata, questa viene gestita finché l'istanza dell'EntityManager lo è: un'EntityManager con scope extended manterrà la gestione di tutte le entità attached finché non verrà chiuso o il bean stesso distrutto.
  - transaction:
    - la gestione delle entità avviene esclusivamente entro i confini della transazione.

Persistence Context (PC)



Ciclo di vita di un'Entità



Ciclo di vita di un'Entità

- New: una nuova istanza che non ha ancora un id e non è ancora relazionato ad alcun PC
- Managed: l'entità è associata ad un PC
- Detached: un'entità che non è più gestita dall'EntityManager e il cui stato non è più sincronizzato con il database
- Removed: un'entità associata ad un PC ma segnata per la rimozione dal DB

Entity Manager

- Tale interfaccia è la più importante delle Java Persistence API perchè costituisce un ponte tra il mondo Object-Oriented e quello relazionale.
- Oggetto visibile al Client per lavorare sulle entità
  - API per tutte le funzioni di persistenza di base (CRUD)
  - Gestisce connessioni e transazioni
- Lo si può considerare come un proxy verso un contesto di persistenza

Creazione di un EntityManager

- La prima cosa da fare per poter gestire la persistenza è ottenere un'istanza dell'EntityManager.
- Se siamo all'interno di un container è possibile usare l'annotazione @PersistenceContext, in tal modo il container si prende cura delle operazioni di lookup e dell'apertura e chiusura dell'EntityManager.
- Laddove diversamente specificato lo scope di default dell'EntityManager è TRANSACTION.
- JPA supporta anche gli EntityManager application-managed che vengono esplicitamente creati, usati e rilasciati dall'applicazione per l'utilizzo al di fuori del container.

EntityManager container-managed

- L'utilizzo dell'annotazione @PersistenceContext consente di ottenere l'istanza di un EntityManager container-managed.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface PersistenceContext {
        String name() default "";
        String unitName() default "";
        PersistenceContextType type default TRANSACTION;
        PersistenceProperty[] properties() default {};
}
```

```
@PersistenceContext(unitName="nomeunita")
EntityManager manager;
```

EntityManager container-managed

- name:
  - specifica il nome JNDI del persistence context
  - è usato nel caso in cui si voglia indicare esplicitamente il nome JNDI per una data implementazione.
- unitName:
  - specifica il nome della persistence unit che è essenzialmente un raggruppamento di entità usate dall'applicazione
  - L'idea, quando si hanno applicazioni di una certa dimensione, è quella di separarle in aree logiche delle persistence configurate attraverso il deployment descriptor persistence.xml.
- type:
  - specifica lo scope dell'EntityManager: i valori possibili sono TRANSACTION O EXTENDED.

EntityManager application-managed

- Gli EntityManager application-managed sono appropriati nel caso di ambienti nei quali non è disponibile alcun container
  - i.e. Java SE
  - Un possibile uso dell'EntityManager application-managed in ambiente Java EE potrebbe essere giustificato dalla necessità di mantenere un confrollo fine sul ciclo di vita dello stesso.
- in questo caso occorre scrivere il codice per controllare ogni aspetto del ciclo di vita dell'EntityManager.
- E' possibile ottenere un'istanza di un EntityManager application-managed mediante l'utilizzo dell'interfaccia EntityManagerFactory

Persistence in Java SE

- Nessuna fase di deployment
  - L'applicazione deve usare un "Bootstrap API" per ottenere un'EntityManagerFactory
- resource-local EntityManager
  - L'applicazione usa una EntityTransaction locale ottenuta da un'EntityManager

Entity Transactions

- Usata solo da resource-local EntityManager
- Usa le EntityTransaction API per la gestione delle transazioni
  - begin(), commit(), rollback(), isActive()
- Le risorse sottostanti (JDBC) sono allocate dall'EntityManager così come richiesto

Persistence Class

- javax.persistence.Persistence
- Classe root per l'avvio automatico di un EntityManager
- Localizza un servizio di provider per una determinata persistence unit
- Invoca API sul provider per ottenere un'EntityManagerFactory

EntityManagerFactory Class

- javax.persistence.EntityManagerFactory
- Ottenuta dalla classe Persistence
- Crea un'EntityManager per una determinata persistence unit o configurazione
- In un ambiente Java SE la configurazione della persistence unit è definita nel file `META-INF/persistence.xml`

Sample Configuration (META-INF/persistence.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.0">
  <persistence-unit name="hellojpa">
    <class>hellojpa.Message</class>
    <properties>
      <property name="openjpa.ConnectionURL"
        value="jdbc:derby:openjpa-database;create=true"/>
      <property name="openjpa.ConnectionDriverName"
        value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="openjpa.ConnectionUserName" value=""/>
      <property name="openjpa.ConnectionPassword" value=""/>
    </properties>
  </persistence-unit>
</persistence>
```

JPA Bootstrap – Esempio

```
public class PersistenceExample {
  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("hellojpa");
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    // Perform finds, execute queries,
    // update entities, etc.
    em.getTransaction().commit();
    em.close();
    emf.close();
  }
}
```

Operations on Entities (1)

- EntityManager API
  - persist() – salva un dato oggetto entity nel DB (SQL INSERT)
  - remove() – elimina un dato oggetto entity nel DB (SQL DELETE by PK)
  - refresh() – ricarica un dato oggetto entity dal DB (SQL SELECT by PK)
  - merge() – sincronizza lo stato delle entità detached con il PC
  - find() – esegue una semplice query per PK

Operations on Entities (2)

- createQuery() – crea una query usando JPQL dinamico
- createNamedQuery() – crea un'instanza per una query JPQL predefinita
- createNativeQuery() – crea un'instance per una query SQL
- contains() – determina se una data entità è gestita dal PC
- flush() – forza affinché le modifiche nel PC siano salvate nel DB (invocato automaticamente sulla commit della transazione)

persist()

- Salva una nuova istanza di entità nel DB (SQL INSERT/ UPDATE)
- Salva lo stato persistente dell'entità e di ogni referenza relazionata
- Le nuove entità diventano "managed"

```
public Customer createCustomer(int id, String
name) {
    Customer cust = new Customer(id, name);
    entityManager.persist(cust);
    return cust;
}
```

find() and remove()

- find()
  - Ritorna l'istanza di un entità gestita dal PC (SQL SELECT by PK)
  - Ritorna `null` se non trovata

- remove()
  - Elimina un'istanza di entità per PK

```
public void removeCustomer(Long custId) {
  Customer cust = entityManager.
    find(Customer.class, custId);
  entityManager.remove(cust);
}
```

merge()

- Unisce in una copia managed gli stati di entità detached
- L'entità ritornata ha un id Java differente da quello dell'entità detached

```
public Customer storeUpdatedCustomer(Customer
cust) {
  return entityManager.merge(cust);
}
```

- Un'entità detached è un'entità che non è più gestita dall'EntityManager e il cui stato non è più sincronizzato con il database.
- E' possibile ad esempio che un'entità venga passata al web tier, aggiornata e inviata nuovamente indietro all'EJB tier per effettuarne il merge al persistence context.
- Essenzialmente un'entità diventa detached non appena esce fuori dallo scope dell'EntityManager.

Fetching

- L'EntityManager normalmente carica tutti i dati di un'istanza quando questa viene recuperata dal database, tale modalità è definita eager fetching o eager loading
- Il problema di questo approccio nasce quando si fa uso anche di Large Binary Object (BLOB) il cui caricamento costituisce un'operazione particolarmente onerosa.
  - In questi casi si potrebbe evitare il caricamento del BLOB ed effettuarlo esclusivamente quando è necessario: questa strategia è nota come lazy fetching.
- Lo stato può essere quindi recuperato come EAGER or LAZY
  - LAZY – il container differisce il caricamento fintantoché il campo/proprietà è acceduto
  - EAGER – richiede che il campo o la relazione sia caricata quando l'entità referenziata è caricata

Fetching

- JPA ha diversi meccanismi per supportare il lazy fetching, il più semplice consiste nell'utilizzare l'annotazione @Basic:

```
@Column(name="IMMAGINE")
@Lob
@Basic(fetch=FetchType.LAZY)
public byte[] getImmagine()
{
    return immagine;
}
```

Queries

Using JPQL

Queries

- Definite dinamicamente o staticamente (named queries)
- Criteria che usano JPQL (Java Persistence API Query Language, una specie di SQL)
- Supporto per SQL nativo (quando richiesto)
- Parametri collegati a tempo di esecuzione (nessuna SQL injection)
- Paginazione e capacità di restringere la size del risultato
- Aggiornamenti e cancellazioni di massa

Query API (1)

- Le istanze di Query sono ottenute da factory methods sull'`EntityManager`, e.g.

```
Query query = entityManager.createQuery(
  "SELECT e from Employee e");
```

- Query API:
  - getResultList() – esegue una query ritornando risultati multipli
  - getSingleResult() – esegue query ritornando un singolo risultato
  - executeUpdate() – esegue cancellazioni o aggiornamenti di massa

Query API (2)

- setFirstResult() – imposta il primo risultato da recuperare
- setMaxResults() – imposta il massimo numero di risultati da recuperare
- setParameter() – collega un valore ad un parametro per nome o per posizione

Dynamic Queries

- Usa il factory method createQuery() a runtime e passa una JPQL query string
- Le Query possono essere compilate/verificate a tempo di creazione o mentre eseguite
- Massima flessibilità per la definizione di query e la loro esecuzione

Dynamic Queries – Esempio

- Ritorna tutte le prime 100 istanze di una data entity

```
public List findAll(String entityName){
  return entityManager.createQuery(
    "select e from " + entityName + " e")
    .setMaxResults(100)
    .getResultList();
}
```

- La stringa JPQL è costruita con l'entity type
  - Per esempio, se fosse passato al metodo findAll "Account" avremo una JPQL string: "select a from Account a"

Named Queries

- Usa il factory method createNamedQuery() a runtime e passa il nome della query
- La query deve essere definita staticamente
- I nomi delle query sono visibili globalmente
- Il Provider può precompilare le query e ritornare errori a tempo di deploy
- Si possono includere parametri nella definizione statica della query

Named Queries – Esempio

```
@NamedQuery(name="Sale.findByCustId",
  query="select s from Sale s
         where s.customer.id = :custId
         order by s.salesDate")


public List findSalesByCustomer(Customer cust) {
  return (List<Customer>)entityManager.
    createNamedQuery("Sale.findByCustId")
    .setParameter("custId", cust.getId())
    .getResultList();
}
```

Summary

- JPA è emerso dalle best practices dei migliori prodotti di ORM
- Lightweight persistent POJOs, no extra carico
- API semplici, compatte e potenti
- Metadata ORM standardizzati specificati usando annotazioni o XML
- Linguaggio di query arricchito
- Integrazione con Java EE, ed API addizionali per Java SE