Modulo 4

Integration Tier

ORM

Presented by

Paolo Locorotondo

IBM Client Innovation Center - Italy

IBM Client Innovation Center
Italy

Chi sono

Paolo Locorotondo

Application Developer – Red Hat Cloud and Microservices

In IBM Client Innovation Center da 12/2018

Email: paolo.locorotondo1-cic-it@ibm.com
Slack: paolo.locorotondo

IBM Client Innovation Center
Italy

Agenda della settimana

1. ORM - JPA
2. Hibernate
3. Junit – Testing and Mocking
4. Riepilogo – Test Finale

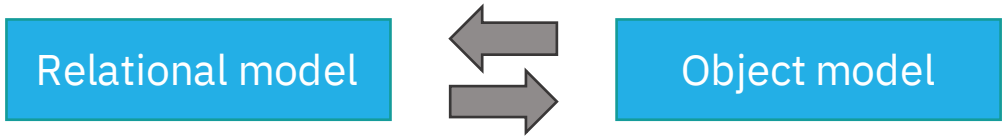
Prima di iniziare

JDK	https://www.oracle.com/java/technologies/downloads
Eclipse IDE for Java Developers	https://www.eclipse.org/downloads/packages/
MySQL	https://dev.mysql.com/downloads/mysql/
Maven	https://maven.apache.org/download.cgi
Git	https://git-scm.com/downloads

Repository del corso:
https://github.com/academy-ibm-cic-italy/2023-Q4-integration-tier-module

Object-Relational Impedance Mismatch

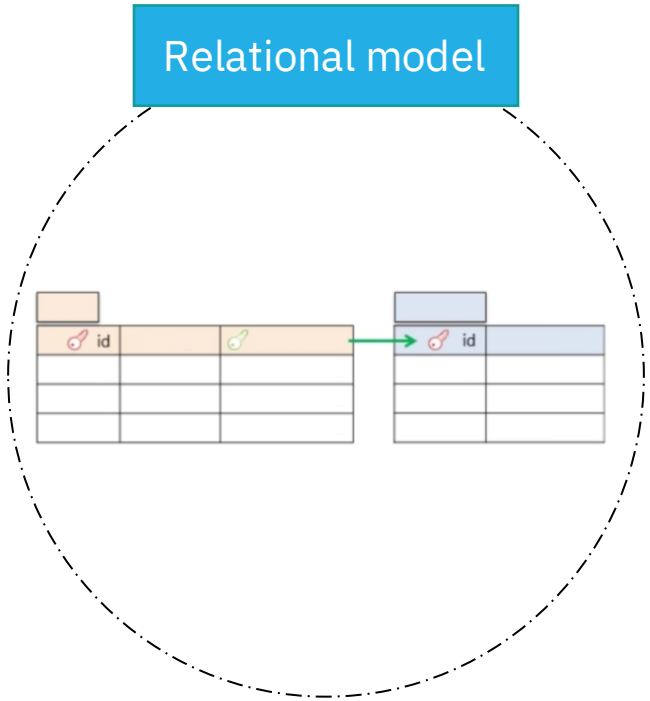
Come possiamo rappresentare i dati di un modello relazionale in un modello orientato agli oggetti?



Con il termine Object-Relation Impedance Mismatch ci si riferisce ad un insieme di problematiche, concettuali e tecniche, che si incontrano quando si vuole far dialogare un RDBMS, basato su un modello relazionale, con un programma scritto con un linguaggio di programmazione orientato agli oggetti, per esempio java.

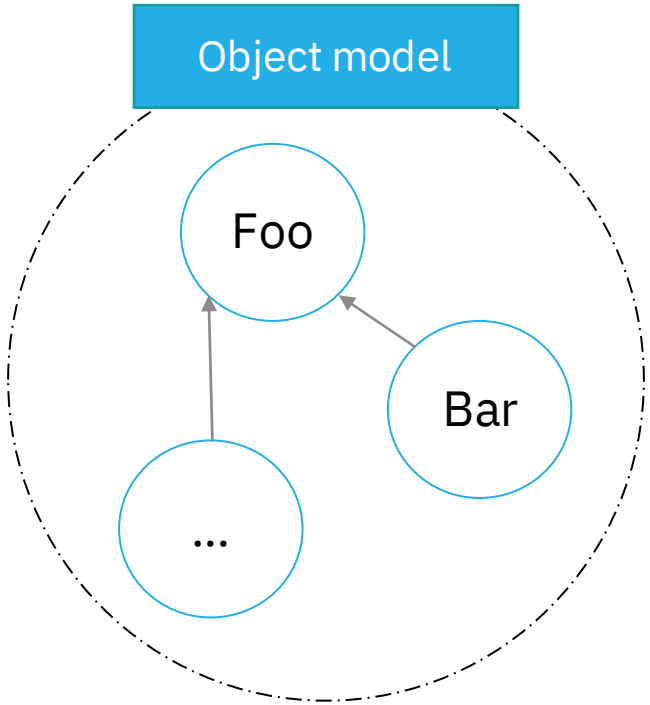
Object-Relational Impedance Mismatch

Relational Database Management System (RDBMS) rappresentano i dati in un formato tabellare (colonne, righe e relazioni tra tabelle).



Object-Relational Impedance Mismatch

Linguaggi di programmazione object-oriented rappresentano i dati come “grafi di oggetti” interconnessi tra di loro.

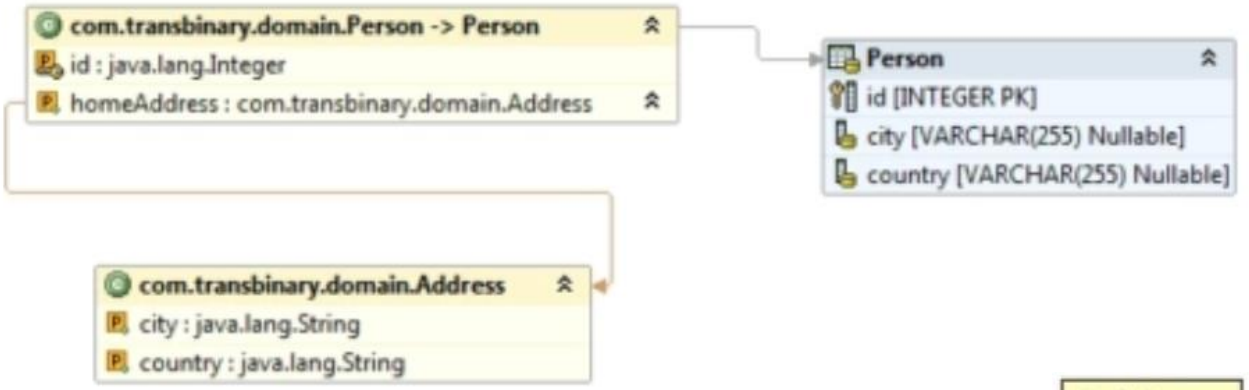


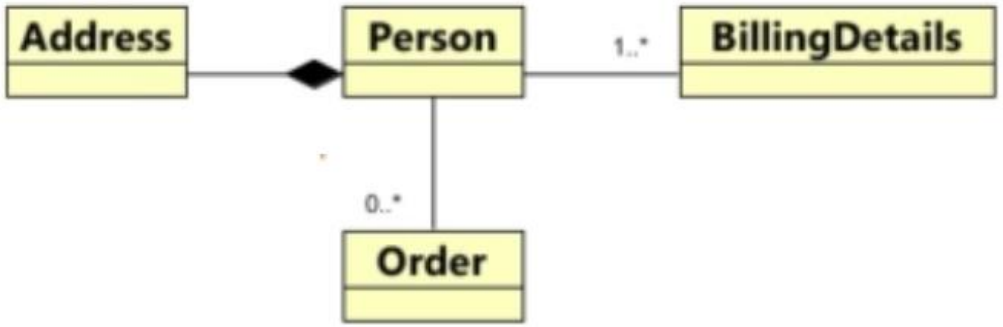
Object-Relational Impedance Mismatch

Le differenze tra i due modelli ci espongono ad almeno 5 problemi di “disallineamento”

1. Granularity
2. Inheritance
3. Identity
4. Associations
5. Data navigation

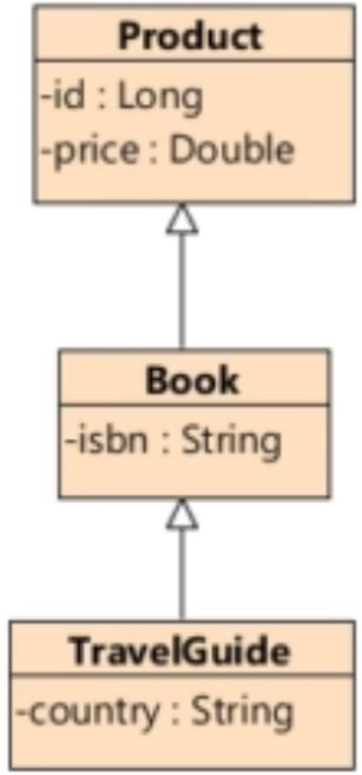
Granularity





Inheritance

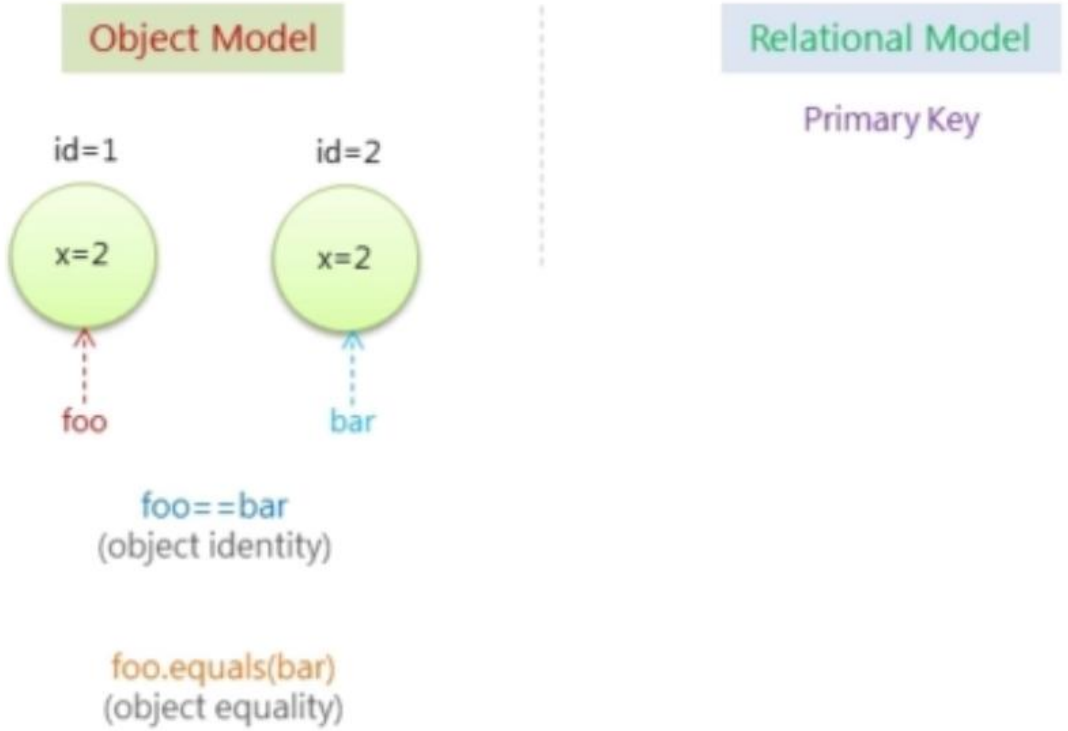
Object Model



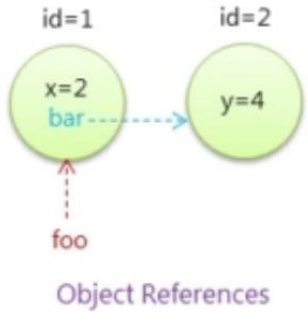
Relational Model

No Inheritance

Identity

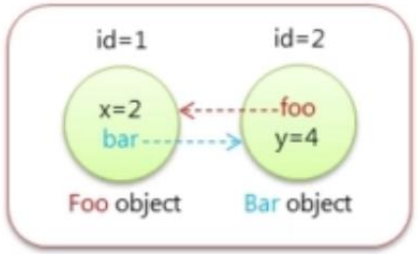

Object Model
Relational Model
Primary Key
id=1
id=2
x=2
x=2
foo
bar
foo==bar
(object identity)
foo.equals(bar)
(object equality)

Associations



Object Model

```
public class Foo {
    private Bar bar;
    private Integer x;
    ...
}

public class Bar {
    private Foo foo;
    private Integer y;
    ...
}
```

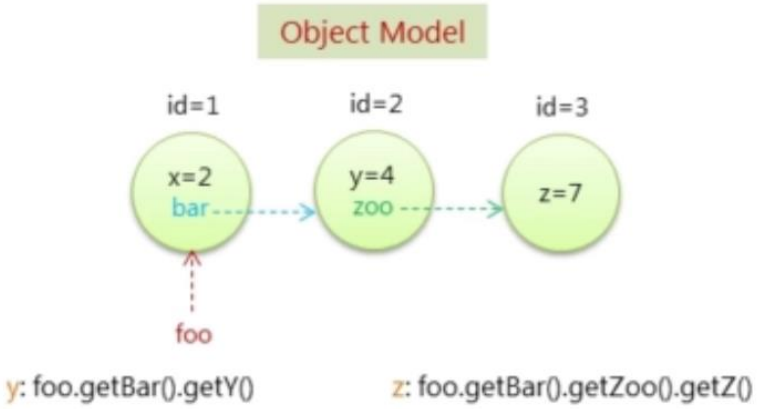


Bi-directional

Relational Model

Foreign Key

Data Navigation



Relational Model

order

id	order_no	customer_id
1	112233	1
2	445566	2
3	778899	2

customer

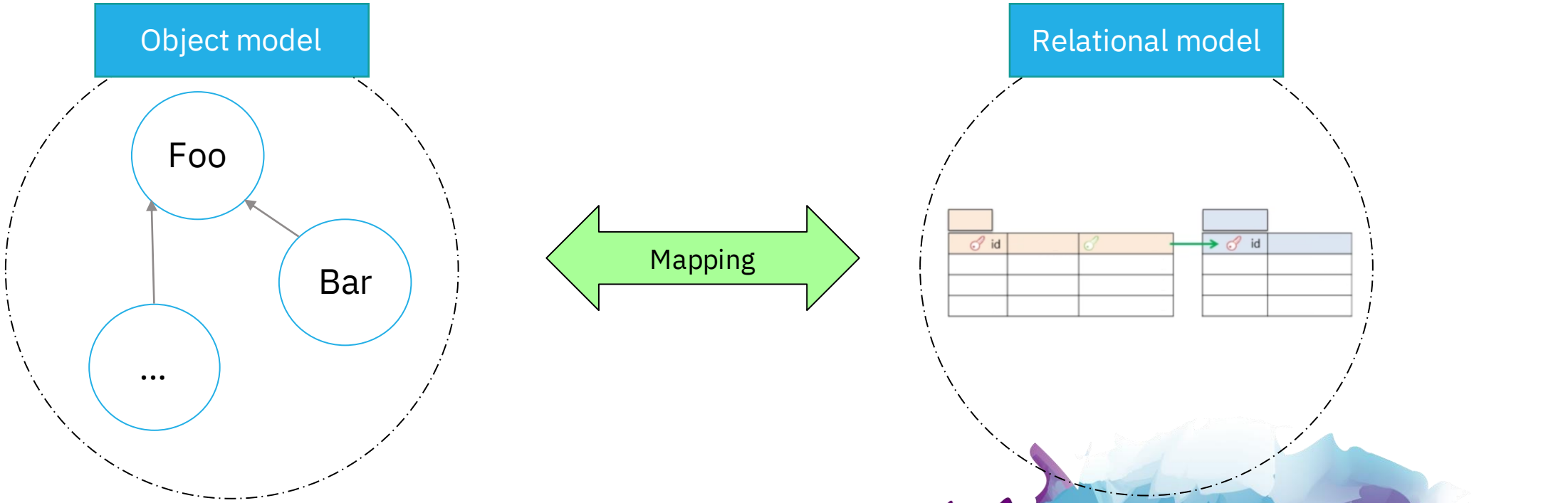
id	name
1	Adam
2	Dave
3	Neha

name	order_no
Adam	112233
Dave	445566
Dave	778899

SQL JOIN Query

Introduzione

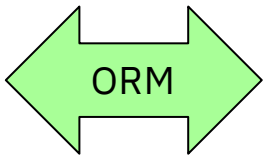
Object-Relational Mapping (ORM) è una tecnica di programmazione tramite cui gestire l'integrazione di sistemi software basati sul paradigma della programmazione orientata agli oggetti con sistemi RDBMS



Introduzione

Orm: insieme di tecniche che consente di mappare un oggetto java con una corrispettiva tabella sql.

```
@Entity
@Table(name = "PERSON")
public class Person {

   public Person () {}

   @Id
   @Column(name = "ID")
   private int id;

   @Column(name = “FIRST_NAME")
   private String firstName;

   @Column(name = "LAST_NAME")
   private String lastName;

   @Column
   private Integer age;

...
}
```




```
create table PERSON (
   ID INT NOT NULL,
   FIRST_NAME VARCHAR(64),
   LAST_NAME VARCHAR(64),
   AGE INT,
   PRIMARY KEY (id)
)
```

Introduzione

Rispetto all'utilizzo di JDBC visto nei moduli precedenti...

Vantaggi	Svantaggi
Semplifica la manipolazione di dati ed oggetti	Riduzione del codice SQL: può essere un problema per query complesse
Elevata portabilità rispetto al DBMS utilizzato	
Genera codice standard per le operazioni CRUD	
Aumenta la velocità di sviluppo	

Introduzione

- Apache Cayenne, open-source for Java
- Apache OpenJPA, open-source for Java
- Ebean, open-source ORM framework
- EclipseLink, Eclipse persistence platform
- Enterprise JavaBeans (EJB)
- MyBatis, iBATIS
- **Hibernate, open-source ORM framework, più usato**
- Java Data Objects (JDO)
- JOOQ Object Oriented Querying (jOOQ)
- TopLink by Oracle

Modulo 4

Integration Tier

JPA

Presented by

Paolo Locorotondo

IBM Client Innovation Center - Italy

IBM Client Innovation Center
Italy

Introduzione

Java Persistence API (JPA) è una specifica per la gestione delle operazioni di persistenza dei dati su un database relazionale

- Annotazioni per mappare le classi Java alle tabelle del database
- Linguaggio di interrogazione JPQL (Java Persistence Query Language), indipendente dal database
- Varie API per gestire e manipolare le Entity Java

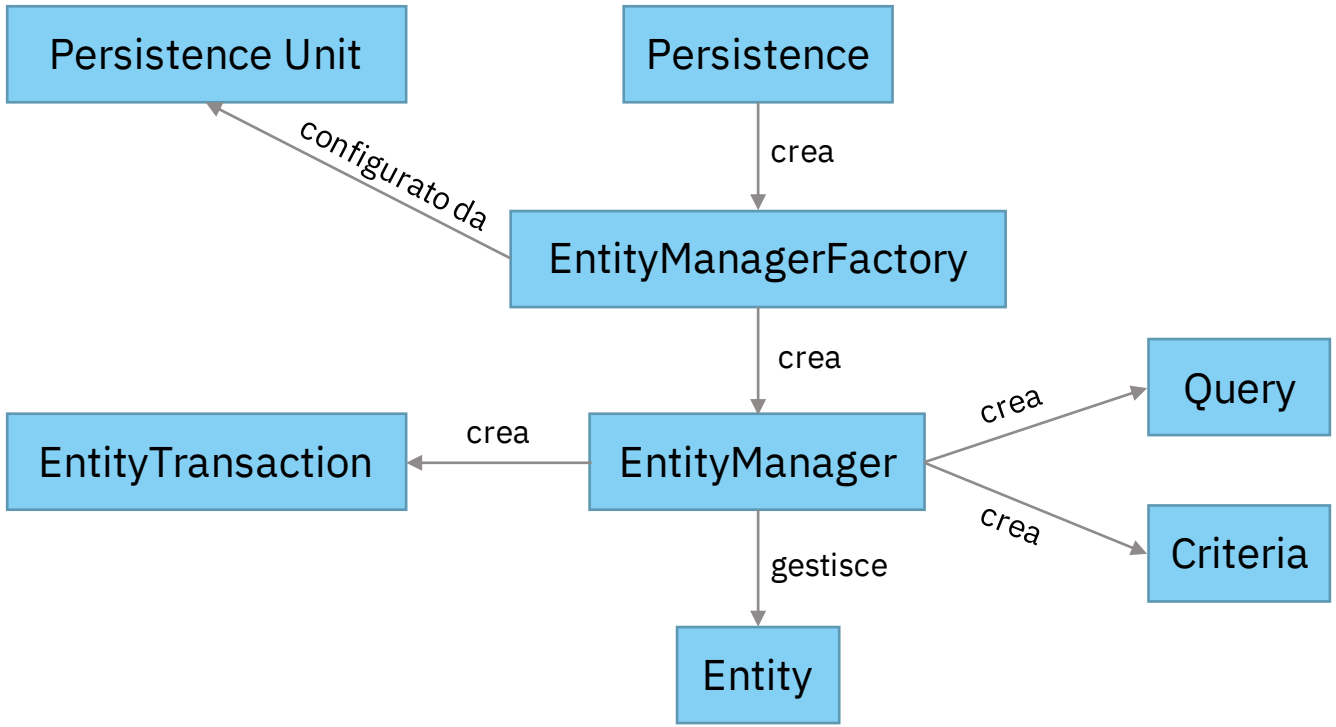
Introduzione

JPA è solo una specifica, per eseguire qualsiasi operazione richiede un'implementazione

Strumenti ORM come **Hibernate**, TopLink e iBatis implementano le specifiche JPA per la persistenza dei dati

Architettura

- **Persistence**: classe contenente metodi statici da utilizzare per creare EntityManagerFactory
- **EntityManagerFactory**: crea e gestisce istanze di EntityManager
- **Persistence Unit**: fornisce le informazioni necessarie per connettersi al database e definisce l'insieme di tutte le Entity
- **EntityManager**: interfaccia tramite cui interagire con il database
- **EntityTransaction**: unità di lavoro
- **Entity**: oggetto del dominio di persistenza
- **Query**: interfaccia utilizzata per l'esecuzione di query su database
- **Criteria API**: supporta la creazione di query SQL utilizzando oggetti java



i criteria consentono di effettuare query più complesse

Persistence Unit

Le unità di persistenza vengono definite nel file **persistence.xml**

JPA usa questo file per creare la connessione al database e configurare l'ambiente richiesto per l'applicazione

Per impostazione di default, l'unità di persistenza include tutte le classi annotate come Entity presenti nella sua root

È possibile utilizzare i tag:

- *<class>* per includere classi presenti altrove
- *<jar-file>* per includere tutte le classi presenti in un jar

Persistence Unit

persistence.xml

```
<persistence
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
version="2.0">
    <persistence-unit name="jpa_esempio">
        <description>Persistence unit for the JPA example</description>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/persons" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="root" />
        </properties>
    </persistence-unit>
</persistence>
```

Vediamo un esempio

(jpa_esempio)

Java Persistence API

Entity

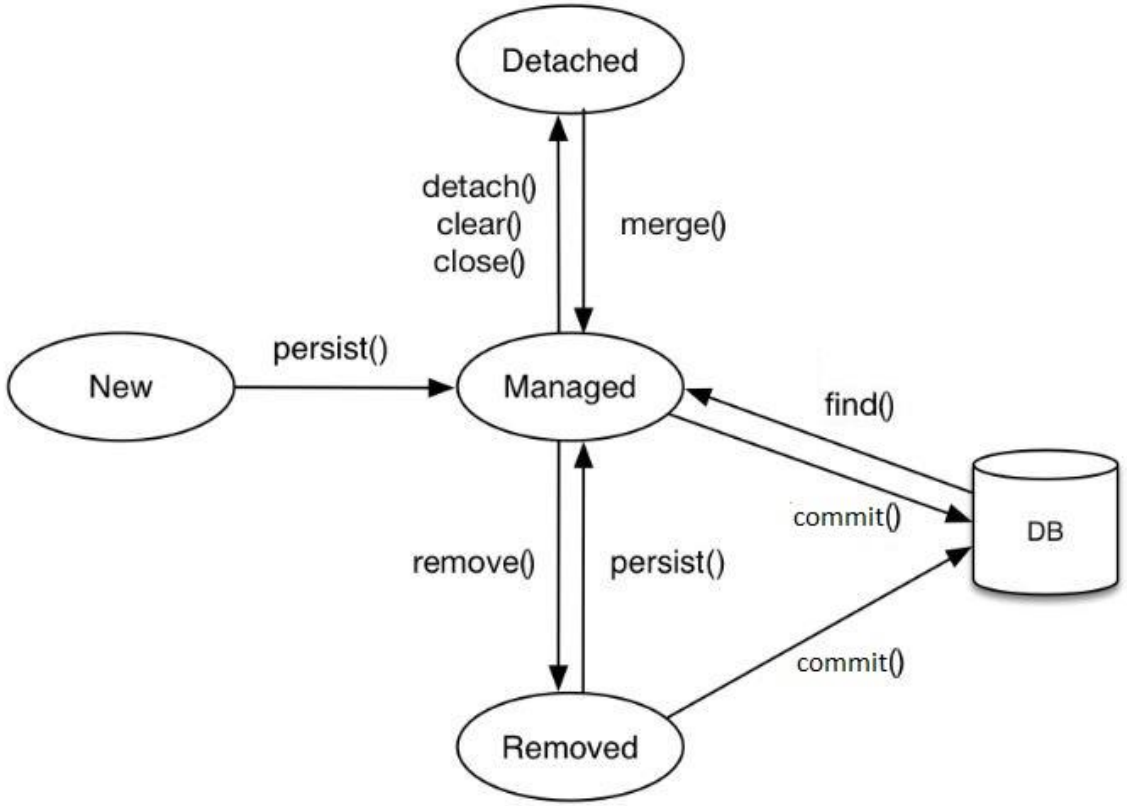
una classe java consentirà a questa classe di rappresentare una tabella del database, capiremo quindi come collegare una classe con una tabella. ogni istanza di una classe rappresenterà una riga della tabella ad essa associata. Ad esempio ogni istanza della classe book rappresenterà una riga della tabella book

Un’entità è un oggetto del dominio di persistenza

Ogni entità rappresenta una tabella del database

Ogni istanza di entità rappresenta una riga nella tabella

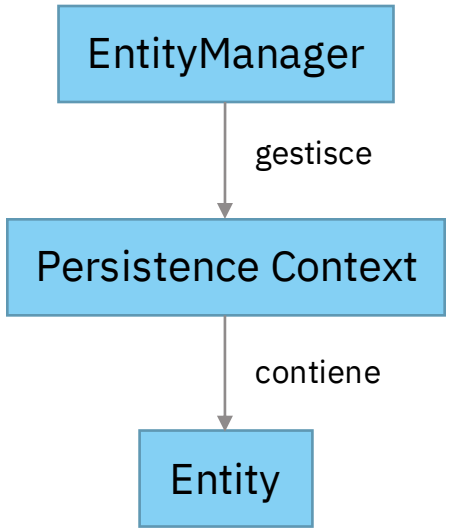
Entity Lifecycle

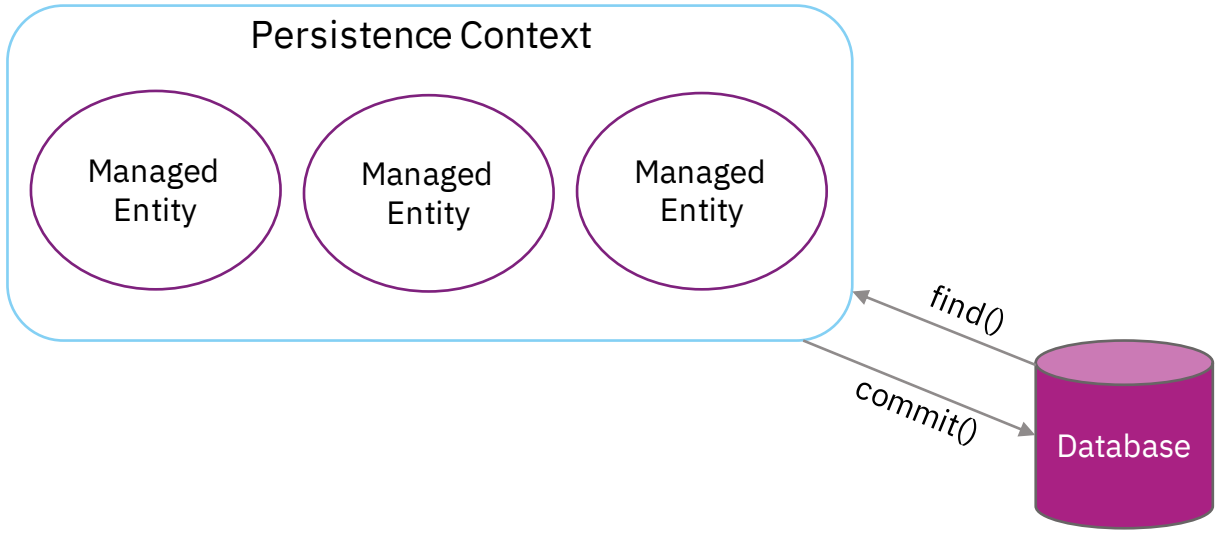


Persistence Context

Insieme di istanze di entità

Un’istanza di EntityManager è associata ad un Persistence Context





Annotazioni

Un’annotazione Java è una forma di metadata che può essere aggiunta al codice sorgente

Può essere specificata su classi, attributi, metodi, parametri

Le annotazioni JPA vengono utilizzate per mappare le classi Java alle tabelle del database

Tutte le annotazioni JPA sono presenti nel package *javax.persistence.**

Java Persistence API

Mapping tabella

con entity si indica che la classe java deve essere trattata come un entità che in futuro potrà essere gestita da un entityManager.
Table ci indica come mappare, a name si associerà il nome della tabella. Rispettare il case sensitive. Di solito si indicano le tabelle col loro nome in maiuscolo.

- **Entity**: indica una classe che dovrà essere gestita dal persistence provider
- **Table**: specifica la tabella del database con cui verrà mappata l'entità

```
@Entity
@Table(name = "PERSON")
public class Person {


}
```

L'annotazione Table è opzionale: se non specificata, il nome della tabella che verrà considerato sarà il nome della entity

Mapping colonne

- **Column**: indica la colonna con cui verrà mappato l'attributo

Oltre a *name*, sono presenti altre proprietà che consentono di fornire informazioni sulla colonna:

- unique: specifica se sulla colonna è definito un vincolo di unique key
- *nullable*: specifica se la colonna ammette valori null
- *length*: lunghezza colonna string
- *precision*: numero digit colonna decimal
- *scale*: numero cifre decimali colonna decimal

se non viene specificato niente, si mappa l'attributo della tabella che ha lo stesso nome dell'attributo della classe

```
@Entity
@Table(name = "PERSON")
public class Person {

  @Column(name = "first_name")
  private String firstName;

  @Column
  private int age;

}
```

Primary Key

- **Id**: indica la colonna impostata come chiave primaria
- **GeneratedValue**: indica la strategia di generazione dei valori della chiave primaria
  4 strategie disponibili:
  - AUTO (default)
  - IDENTITY
  - SEQUENCE
  - TABLE

```
@Entity
@Table(name = "PERSON")
public class Person {

  @Id
  @GeneratedValue
  @Column
  private int id;

}
```

GeneratedValue

AUTO Generation

In base al database utilizzato, il provider di persistenza stabilisce il tipo di generatore da utilizzare

IDENTITY Generation

Il database è responsabile di determinare ed assegnare il valore della chiave primaria

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

GeneratedValue

SEQUENCE Generation

Una sequence rappresenta un oggetto del database che può essere utilizzato come origine di valori della chiave primaria

```
@GeneratedValue(strategy = GenerationType.SEQUENCE)
```

```
@GeneratedValue(generator = "sequence_generator")
@GenericGenerator(name = "sequence_generator",
    strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
    parameters = {
     @Parameter(name = "sequence_name", value = "user_sequence"),
     @Parameter(name = "initial_value", value = "1"),
     @Parameter(name = "increment_size", value = "5")
     }
  )
```

GeneratedValue

TABLE Generation

Utilizza una apposita tabella per determinare i valori della chiave primaria

```
@GeneratedValue(strategy = GenerationType.TABLE, generator = "table_generator")
@TableGenerator(
    name = "table_generator",
    table = "id_gen",
    pkColumnName = "gen_id",
    pkColumnValue = "person_id",
    valueColumnName = "gen_value")
```

ID_GEN	
GEN_ID	**GEN_VALUE**
person_id	1

Composite Primary Key

La chiave primaria di una tabella può essere composta da due o più colonne

Per specificare composite key JPA fornisce le annotazioni: **IdClass** e **EmbeddedId**

La classe che conterrà i campi della chiave deve seguire alcune regole:

- Essere dichiarata come public
- Avere un costruttore senza argomenti
- Implementare i metodi *equals* e *hashCode*
- Essere *Serializable*

Composite Primary Key: IdClass

```
public class AccountId implements Serializable {
   private String accountNumber;
   private String accountType;

   // default constructor

   // equals() and hashCode()
}
```

```
@Entity
@IdClass(AccountId.class)
public class Account {
   @Id
   private String accountNumber;

   @Id
   private String accountType;

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

Composite Primary Key: EmbeddedId

```
@Embeddable
public class BookId implements Serializable {
   private String title;
   private String language;

   // default constructor

   // getters, equals() and hashCode() methods
}
```

```
@Entity
public class Book {
   @EmbeddedId
   private BookId bookId;

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

IdClass vs EmbeddedId

Utilizzando IdClass è necessario specificare due volte le colonne della chiave primaria, con EmbeddedId non è necessario

Query con IdClass	SELECT a.accountNumber FROM Account a
Query con EmbeddedId	SELECT b.bookId.title FROM Book b

Se si accede individualmente a parti della composite key è preferibile usare IdClass, altrimenti EmbeddedId

Mapping relazioni

4 annotazioni disponibili per il mapping delle relazioni:

- *OneToOne*
- *OneToMany*
- *ManyToOne*
- *ManyToMany*

Ogni relazione può essere definita come unidirezionale o bidirezionale

OneToOne

Ogni istanza di una entità è legata ad una singola istanza di un'altra entità

PERSON(id, name, surname, age, address_id)

ADDRESS(id, street, city, state, zipCode)

Esempio: relazione tra persona e indirizzo di residenza

mettere vincolo di unique su address_id: unique = true

```
@Entity
public class Person{
   @OneToOne
   @JoinColumn(name= "address_id")
   private Address address;
}
```

```
@Entity
public class Address{
   @OneToOne(mappedBy= "address")
   private Person person;
}
```

JoinColumn specificata sulla entity proprietaria della relazione

La tabella mappata alla entity avrà una foreign-key che si riferisce all'altra tabella della relazione

OneToMany

Un’istanza di una entità può essere correlata a più istanze di altre entità

Esempio: relazione tra persona e documenti

PERSON(id, name, surname, age, address_id)

DOCUMENT(id, code, person_id)

```
@Entity
public class Person{
   @OneToMany(mappedBy = "person")
   private List<Document> documents;
}
```

ManyToOne

Più istanze di una entità possono essere correlate ad una singola istanza dell'altra entità

Esempio: relazione tra documenti e persona

PERSON(id, name, surname, age, address_id)

DOCUMENT(id, code, person_id)

```
@Entity
public class Document {
   @ManyToOne
   @JoinColumn(name="person_id")
   private Person person;
}
```

Java Persistence API

ManyToMany

la tabella che comprende la tabella di manytomany è detta tabella owner. In realtà anche le due tabelle possono essere entrambe tabelle owner, per farlo si dovranno aggiungere istruzioni in WorkShift (joinColumns e inverseJoinColumns)

Più istanze di una entità possono essere correlate a più istanze di un'altra entità

Esempio: relazione tra persone e turni lavorativi

```
@Entity
public class Person {
   @ManyToMany
   @JoinTable (
      name="PERSON_WORKSHIFT",
      joinColumns=@JoinColumn(name="person_id",
          referencedColumnName="id"),
      inverseJoinColumns=@JoinColumn(name="workshift_id",
          referencedColumnName="id"))
   private List<WorkShift> workShifts;
}
```

PERSON(id, name, surname, age, address_id)

PERSON_WORKSHIFT(person_id, workshift_id)

WORKSHIFT(id, start_time, end_time)

```
@Entity
public class WorkShift {
   @ManyToMany(mappedBy="workShifts")
   private List<Person> persons;
}
```

nb: la classe owner deve implementare cascade per la generazione di istanze delle altre classi, come in sql

Fetch type

lazy migliora le prestazioni: si accedono informazioni solo quando necessario. la relazione viene recuperata solo quando si accedono i documenti della classe Document, ad esempio person.getDocument(), in questo caso la query di join verrà effettuata alla tabella Document

L'operazione di fetch indica il recupero di dati dal database

2 strategie possibili per il recupero di entità in relazione tra di loro:

- *EAGER*: relazione recuperata immediatamente insieme alla entity che la utilizza
- *LAZY*: relazione recuperata solo se acceduta

```
@Entity
public class Person {
   @OneToMany(mappedBy = "person", fetch = FetchType.LAZY)
   private List<Document> documents;
}
```

Cascade

Le relazioni tra entità possono dipendere dall'esistenza di un'altra entità

Esempio: relazione tra Persona e Documenti -> eliminando la persona, devono essere eliminati anche i documenti

Il cascade consente di applicare un'azione eseguita su un'entità target, anche su un'entità associata

```
@Entity
public class Person {
   @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
   private List<Document> documents;
}
```

in cascata verrà effettuato anche il persist dei documenti

Cascade

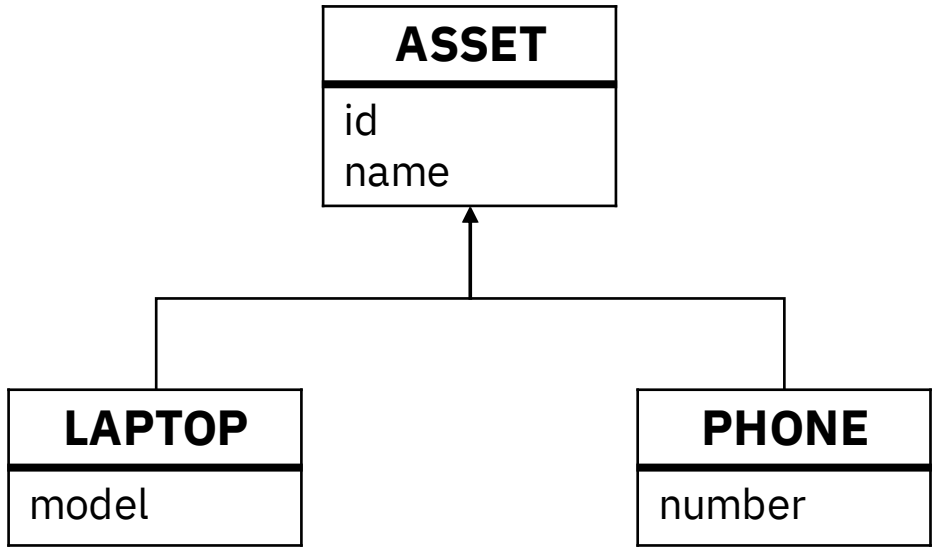
Diverse tipologie di propagazione da entità parent a entità child:

- *ALL*: per tutte le operazioni
- *PERSIST*: solo operazioni di persist
- *MERGE*: solo operazioni di merge
- *REMOVE*: solo operazioni di rimozione
- *REFRESH*: solo operazioni di re-load dell'entity
- *DETACH*: solo operazioni di rimozione dal persistence context

Mapping gerarchie

4 strategie disponibili:

- *MappedSuperclass*
- *Single Table*
- *Join Table*
- *Table per Class*



MappedSuperclass

Gerarchia visibile nel modello delle classi, ma non in quello delle entità

Non a tutte le classi della gerarchia corrisponde una entità nel database

Tabelle:

LAPTOP(id, name, model)

PHONE(id, name, number)

```
@MappedSuperclass
public class Asset {
 @Id
 @GeneratedValue
  private int id;

   @Column
   private String name;
}

@Entity
public class Laptop extends Asset {
   private String model;
}

@Entity
public class Phone extends Asset {
   private String number;
}
```

laptop avrà come attributi model e gli attributi ereditati dalla super classe, ovvero id e name di Asset

Single Table

Unica tabella con tutte le proprietà delle entità della gerarchia

Tabelle:

ASSET(id, name, model, number, DTYPE)

Colonna DTYPE aggiunta automaticamente ed utilizzata per discriminare l'appartenenza della riga ad una entità

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Asset {
 @Id
 @GeneratedValue
  private int id;

   @Column
   private String name;
}

@Entity
public class Laptop extends Asset {
   private String model;
}

@Entity
public class Phone extends Asset {
   private String number;
}
```

Join Table

Ogni entità della gerarchia è mappata ad una tabella, le proprietà comuni sono in una tabella differente

Necessario un join

Tabelle:

ASSET(id, name)

LAPTOP(id, model)

PHONE(id, number)

```
@Entity
@Inheritance(strategy = InheritanceType. JOINED)
public class Asset {
 @Id
 @GeneratedValue
  private int id;

   @Column
   private String name;
}

@Entity
public class Laptop extends Asset {
   private String model;
}

@Entity
public class Phone extends Asset {
   private String number;
}
```

Table per Class

Ogni entità della gerarchia è mappata ad una tabella, che contiene anche le proprietà ereditate

Tabelle:

ASSET(id, name)

LAPTOP(id, name, model)

PHONE(id, name, number)

in questo caso non si effettua la join, dunque questa strategia è più performante

```
@Entity
@Inheritance(strategy =InheritanceType.TABLE_PER_CLASS)
public class Asset {
 @Id
 @GeneratedValue
  private int id;

   @Column
   private String name;
}

@Entity
public class Laptop extends Asset {
   private String model;
}

@Entity
public class Phone extends Asset {
   private String number;
}
```

Entity Lifecycle Events

JPA fornisce delle annotazioni per intercettare alcuni eventi del ciclo di vita di una entità:

- *PrePersist*: prima che il metodo persist venga chiamato
- *PostPersist*: dopo l'operazione di persist
- *PreRemove*: prima della rimozione
- *PostRemove*: dopo l'eliminazione
- *PreUpdate*: prima dell'aggiornamento
- *PostUpdate*: dopo l'aggiornamento
- *PostLoad*: dopo il caricamento dell'entità

Entity Lifecycle Events

```
@Entity
public class Person {
   ...

   @PrePersist
   public void logNewUserAttempt() {
      logger.info("Attempting to add new user with name: " + name);
   }

   @PostPersist
   public void logNewUserAdded() {
      logger.info("Added user '" + name + " with id: " + id);
   }
}
```

Query

jpql: come sql ma consente di utilizzare una sintassi orientata a java, inoltre si possono utilizzare nome di tabelle uguali a quelli deifniti in java, lo stesso vale per gli attributi

3 tipologie di query definite dalla specifica JPA:

- **Query**: scritta utilizzando la sintassi di Java Persistence Query Language
  - **TypedQuery**: da utilizzare se si conosce in anticipo il tipo di risultato della query, consente di evitare il cast
  - **NamedQuery**: può essere definita sulla entity ed avrà un nome che consente di riferirsi alla query
- **NativeQuery**: scritta in SQL
- **Criteria API Query**: costruita programmaticamente tramite diversi metodi

JPQL

Java Persistence Query Language (JPQL) è il linguaggio di interrogazione definito dalla specifica JPA

Lavora su entità e proprietà

SELECT p FROM Person p WHERE p.firstName = :name and p.lastName = :surname

Vediamo un esempio

(jpa_esempio)