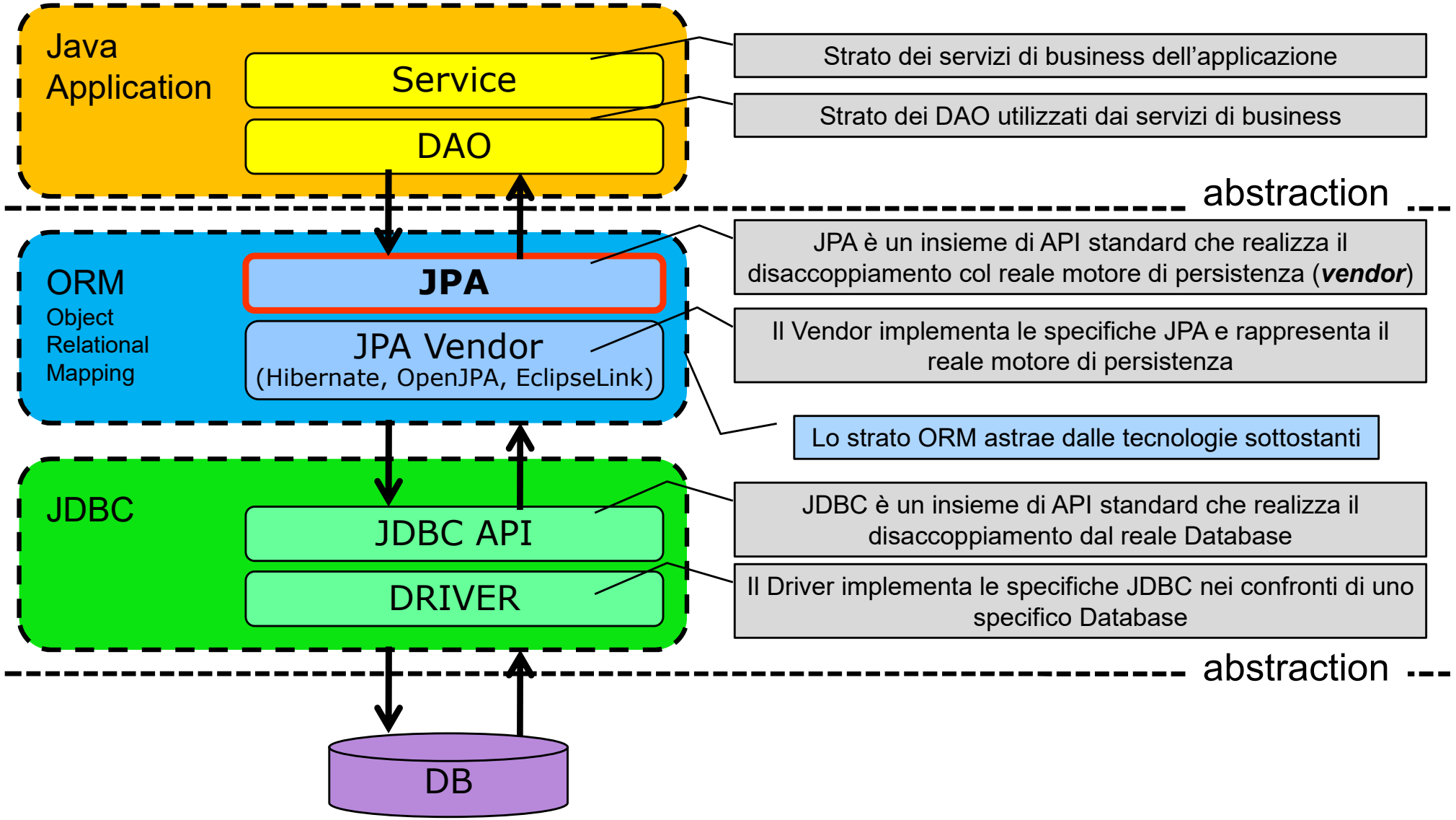
Java Persistence API

Concetti base

Dott. Doria Mauro
doriamauro@gmail.com

L’architettura generale



Il ruolo di JPA

- JPA è una specifica della Oracle e non è un motore di persistenza
  - Oracle definisce le specifiche della tecnologia, l'implementazione è di terze parti
  - I veri motori di persistenza NON sono sviluppati da Oracle e sono liberi di aderire alle specifiche JPA. Se lo fanno diventano **Vendor JPA**
- Scopo di JPA è rendere l'applicazione Java indipendente dal reale motore di persistenza sottostante

JPA sta ai framework ORM come JDBC sta ai DBMS

- FAQ:
  - E' possibile utilizzare JPA senza un vendor? →NO! JPA è un interfaccia e NON basta! Serve disporre ANCHE del reale motore di persistenza!
  - E' possibile utilizzare un motore di persistenza senza passare da JPA? → Si, ma in tal caso, l'applicazione utilizzerebbe direttamente le classi e le interfacce di questo framework ed i suoi servizi specifici (legandosi ad esso!)
  - Se utilizzo JPA, posso sempre accedere a tutti i servizi del vendor? → NO! Essendo una specifica generale non può contemplare le peculiarità specifiche di un singolo framework ORM

Le fasi per operare con JPA

- Per lavorare con JPA è necessario riconoscere 2 fasi distinte
  - **Fase 1: mapping tra entity e tabelle** → in questa fase, si mappano tutte le corrispondenze tra classi Java e tabelle.

    → è una fase preliminare e l'applicazione non è stata ancora sviluppata.

    → ci vuole esperienza del modello relazionale e del modello ad oggetti

  - **Fase 2: sviluppo dell'applicazione** → in questa fase si scrive la logica di business che usa le API di JPA per realizzare la persistenza.

    →JPA astrae completamente dal DBMS ed il mondo dei DB scompare completamente.

    → ci vuole esperienza del modello ad oggetti, delle specifiche di JPA e del suo funzionamento
    - Potremmo *non conoscere il modello relazionale* ma per comprendere il comportamento di JPA (e le sue regole) bisogna affrontare la problematica del **model mismatch**

Configurazione minima di un entity

```
@Entity
public class Employee {
    @Id
    private int id;
    private String name;
    private long salary;

    public Employee() {}
    public Employee(int id) { this.id = id; }

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary (long salary) { this.salary = salary; }
}
```

La configurazione di default prevede:
- nome della tabella = nome della classe
- nome delle colonne= nome dei campi della classe.

Se non c'è questa corrispondenza, si possono aggiungere le annotazioni `@Table` e `@Column`

Tutte le annotazioni indicate sono nel package **`jakarta.persistence`**

Il concetto di persistenza

- JPA offre un nuovo concetto di persistenza: scompaiono la tabella e le righe, i dati persistiti sono direttamente gli oggetti java →**entity**
- JPA obbliga - in fase di mapping- a definire la **persistence identity**, cioè l'**id** che corrisponderà alla Primary Key sulla tabella
  - Non è obbligatorio il mapping delle altre colonne
  - Questo è sufficiente per JPA per identificare l'oggetto con la riga nella tabella

- **L'applicazione crea, modifica e distrugge oggetti mentre JPA si occupa di allineare tali modifiche sul DB**

JPA INSTANT REPOSITORY

JPA Repository

- Inserendo nel progetto la dipendenza **Spring Data JPA** si dispone di un DAO universale (istantaneo) che offre i metodi per dialogare con il meccanismo di persistenza e interagire quindi con il database.

Cosa bisogna implementare?

- Dobbiamo creare una nuova interfaccia DAO che estenda quella di Spring → `JpaRepository<EntityName, IdType>`
  - La nuova interfaccia sarà dedicata ad uno specifico `Entity`, dunque offre i metodi per dialogare con una specifica tabella (e dovrà specificare il tipo della Primary Key, `IdType`)
  - Vengono già ereditati i metodi standard (CRUD)
  - Si possono aggiungere nuovi metodi per query specifiche

- NON serve creare la classe DAO concreta, viene creata da Spring!

Metodi del DAO istantaneo

I principale metodi del DAO istantaneo sono:

- **`save(Entity) : Entity`** → si usa per inserimenti e modifiche
  - per eseguire l'inserimento, bisogna verificare preventivamente la presenza dell'elemento, viceversa tenta di eseguire l'update

- **`findById(IdType) : Optional<Entity>`** → ricerca per PK (id)
  - L'oggetto **`Optional`** contiene **`l'Entity`** (corrispondente alla chiave, se lo trova) o contiene **`null`** (viceversa)
  - Per ottenere l'entity bisogna invocare **`Optional.get()`**
  - Per verificare la presenza dell'oggetto **`Optional.isPresent()`**

- **`findAll(): List<Entity>`** → ricerca tutti gli elementi

- **`deleteById(IdType): void`** → per le cancellazioni

Esempio

- Supposto di avere l'entity **`Employee`**, per gestire la persistenza devo creare solo la nuova interfaccia **`DaoEmployee`**
  - NON implemento il DAO concreto

```
public interface DaoEmployee extends JpaRepository<Employee, Integer>{

}
```

- Questa interfaccia eredita i metodi CRUD.
- La classe Service dichiarerà una proprietà del tipo di questa interfaccia e se la farà iniettare via Autowired

- NB: affinché le operazioni di modifica sul DB siano efficaci, la classe Service dovrà aggiungere l'annotazione **`@Transactional`**

Configurazione del progetto

- Il progetto deve indicare le dipendenze per gestire la persistenza:
  1. quella specifica per il DBMS scelto (nel nostro caso **MySQL**)
  2. quella relativa a JPA (nel nostro caso **spring data jpa**)

- Bisogna quindi aggiornare il pom.XML con :

```
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Configurazione del properties

- Il file `application.properties` è il file di configurazione di Spring.
- Dobbiamo aggiungere le chiavi per il driver DBMS usato (nel nostro caso MySQL) e le impostazioni per JPA:

```
# chiavi per configurare il driver per MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/nomeSchema?createDatabaseIf
NotExist=true&autoReconnect=true&allowPublicKeyRetrieval=true&useSSL=false
spring.datasource.username=root
spring.datasource.password=password_root

# chiavi per jpa : attivano il tool di autogenerazione delle tabelle
spring.jpa.hibernate.ddl-auto=update

# mostra sul server le istruzioni sql eseguite da Hibernate
spring.jpa.show-sql=true
```

Configurazione per postgres

- Configurazioni per postgres in application.properties:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/nomeDb
spring.datasource.username=postgres
spring.datasource.password=password_postgres

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

- Dipendenze per postgres nel pom.XML:

```
<dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
</dependency>
```

Configurazione per derby db

- Configurazioni per derby db in application.properties:

```
spring.datasource.url=jdbc:derby:SA;create=true
spring.datasource.username=derbyuser
spring.datasource.password=password

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

- Dipendenze per derby db nel pom.XML:

```
<dependency>
        <groupId>org.apache.derby</groupId>
        <artifactId>derby</artifactId>
        <scope>runtime</scope>
</dependency>
<dependency>
        <groupId>org.apache.derby</groupId>
        <artifactId>derbytools</artifactId>
        <scope>runtime</scope>
</dependency>
```

QUERY NATIVE

Le query native

- Quando estendiamo l'interfaccia **`JpaRepository`** abbiamo la possibilità di aggiungere metodi custom per eseguire letture (query) specifiche

- Come?

1. Aggiungere un metodo alla nuova interfaccia (con nome e parametri arbitrari)

2. Aggiungere l'annotation **`@Query`**, settando le 2 proprietà:
    2. boolean **`nativeQuery`** → **`true`** indica che il **`value`** specificherà la query in formato SQL
    3. String **`value`** → **`è la query in formato SQL`**

Esempio

- Voglio eseguire le seguenti SELECT sulla tabella Employee:
  - Mostra tutti gli impiegati con salario superiore ad un salario specificato
  - Mostra tutti gli impiegati ordinati per salario
  - Mostra tutti i nomi (distinti) degli impiegati ordinati in ordine alfabetico

- Allora aggiungo i seguenti metodi all'interfaccia **`DAOEmployee`**

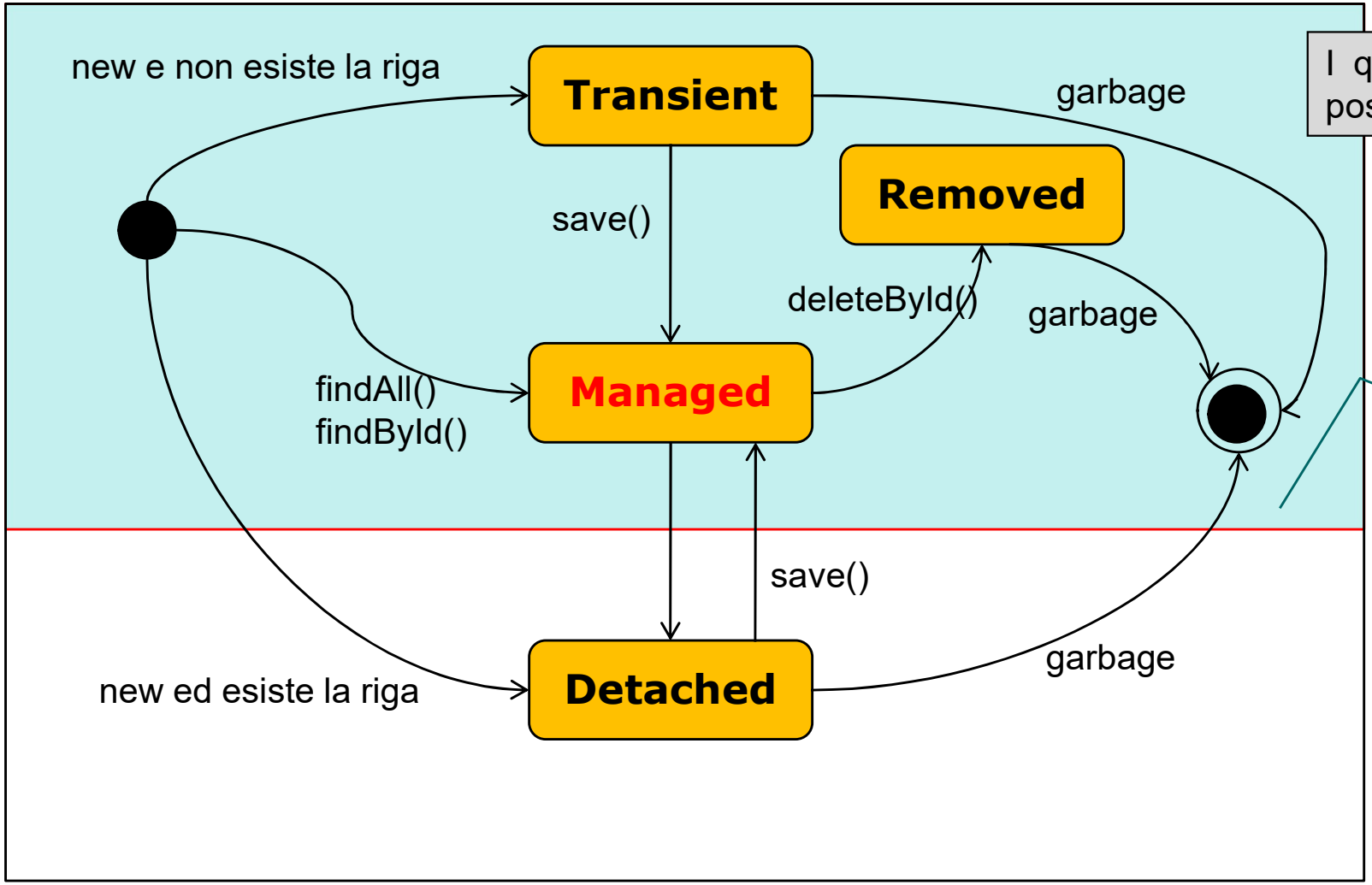
```
// query native
@Query(nativeQuery = true, value = "select * from employee where salario > :salario")
public List<ImpiegatoDTO> getRicchi(double salario);

@Query(nativeQuery = true, value = "select * from employee order by salario")
public List<ImpiegatoDTO> ordinaPerSalario();

@Query(nativeQuery = true, value = "select distinct nome from employee order by nome")
public List<String> ordinaNomi();
```

- I metodi sono astratti per default e non serve aggiungere l'implementazione

STATO DELL'OGGETTO

Persistence lifecycle: stato degli oggetti

- Un entity object si può trovare in uno dei seguenti stati:



I quattro stati e le possibili transizioni.

I sistemi REST non fanno uso dello stato Detached: essendo tutti servizi stateless, non è possibile che un oggetto finisca in questo stato

MAPPING DELLE RELAZIONI

Relazioni tra tabelle e relazioni tra entity

- Le tabelle del database sono collegate tra loro tramite chiavi esterne.

Le relazioni tra le tabelle possono essere:

- 1 → 1
- 1 → N
- N → N

Per ognuno di questi scenari esistono le equivalenti annotation che verranno opportunamente posizionate nelle classi entity gestite dall'ORM:

- `@OnetoOne`
- `@OneToMany`
- `@ManyToOne`
- `@ManyToMany`

} Stessa relazione nelle 2 direzioni

Mapping degli entity

- Una buona progettazione delle classi entity dovrebbe escludere dall'analisi la struttura reale delle tabelle.

- Il modello ad oggetti guida l'analisi, poi si utilizzano le annotation per riallacciarsi al modello relazionale.

- Esempio: supponiamo di dover progettare il seguente dominio.

Gli articoli hanno delle categorie di appartenenza e vengono venduti dagli utenti. Un articolo può essere venduto da un solo utente.

Inoltre una categoria può appartenere a più articoli.

Infine l'utente possiede un indirizzo di residenza, ma tale indirizzo appartiene ad un solo utente.

Come realizziamo gli entity? Come eseguiamo il mapping?

Diagramma tabelle

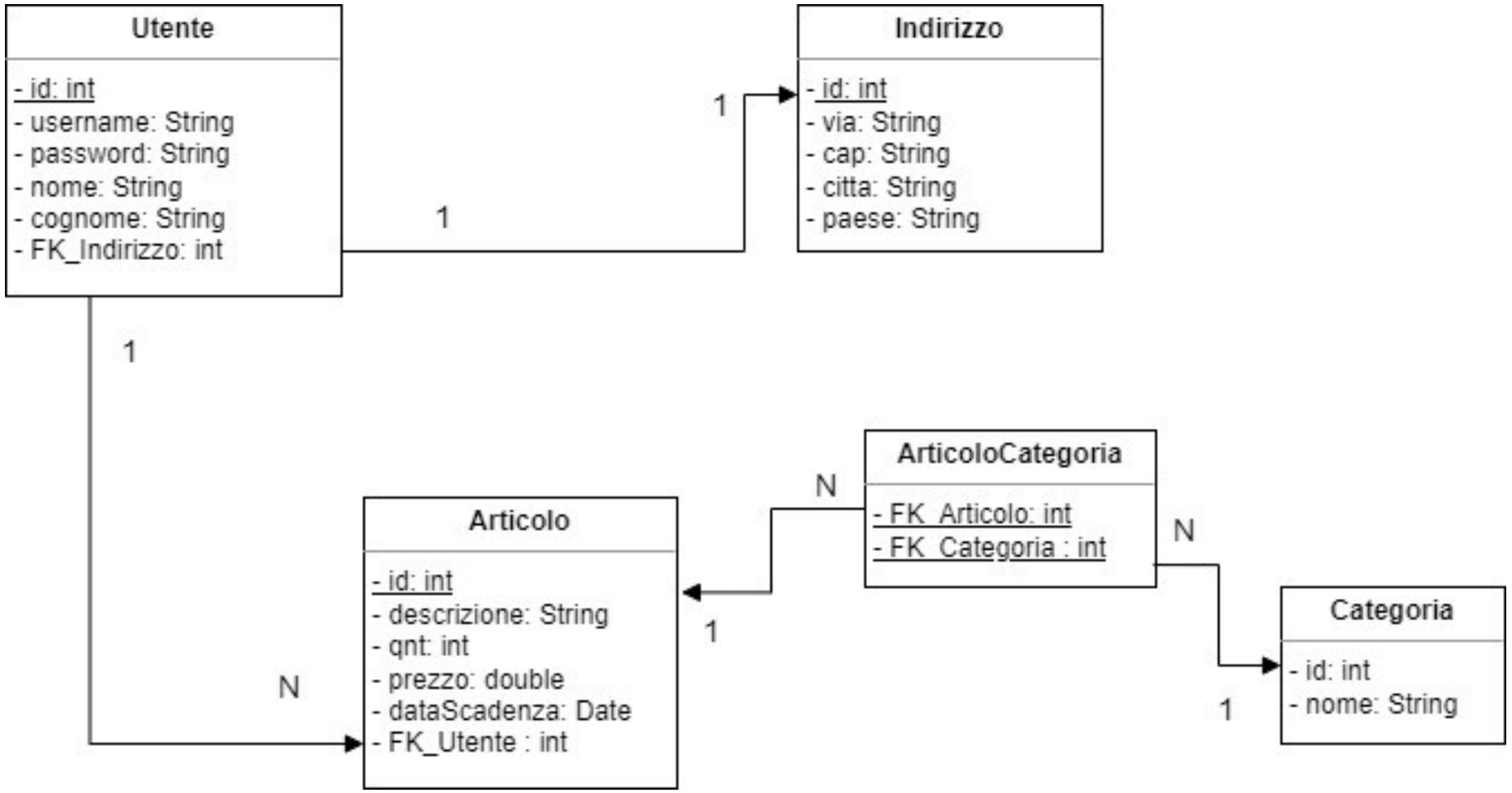
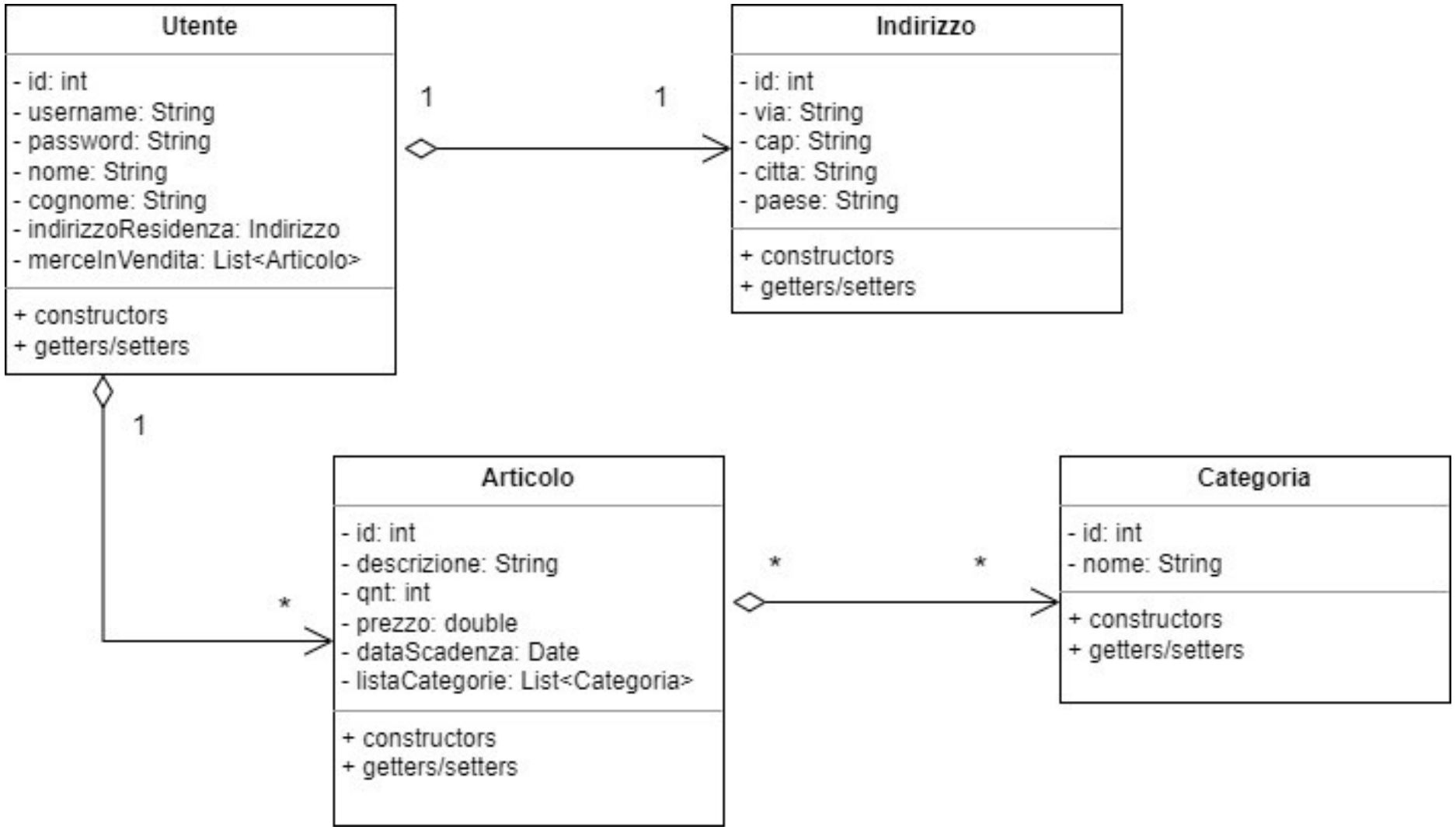


Diagramma delle classi



L’indirizzo

```
@Entity
public class Address {

@Id
@GeneratedValue(strategy = GenerationType.AUTO) // autoincrement
private int id;
private String via;
private String cap;
private String citta;
private String paese;

// costruttori
// getters/setters
}
```

La categoria

```
@Entity
public class Category {

@Id
@GeneratedValue(strategy = GenerationType.AUTO) // autoincrement
private int id;
private String nome;

// costruttori
// getters/setters


}
```

L’articolo

```
@Entity
public class Item {

@Id
private int id;
private String descrizione;
private int qnt;
private double prezzo;

@Temporal(TemporalType.DATE) // solo data, no orario
private Date dataScadenza;

@ManyToMany
@JoinTable(name = "ItemCategory",
joinColumns = @JoinColumn(name= "FK_Item"),
inverseJoinColumns = @JoinColumn(name= "FK_Category") )
// nome della tabella intermedia per realizzare la relazione N a N

private List<Category> listaCategorie = new ArrayList<>();

// costruttori
// getters/setters
}
```

L’utente


```
@Entity
public class MyUser {


@Id
private int id;
private String username, password;
private String nome, cognome;

@OneToOne
@JoinColumn(name = "FK_Address")
private Address indirizzoResidenza;

@OneToMany
@JoinColumn(name = "FK_MyUser")
private List<Item> merceInVendita = new ArrayList<>();

// costruttori
// getters/setters
}
```

LAZY LOADING E CASCADE

Impostazione cascade

- Nelle annotation che mappano le relazioni è possibile impostare il comportamento che l'ORM deve assumere rispetto alle azioni da compiere **sulle entità correlate**.
- Per default le operazioni NON avvengono in cascata → cascate NON attive

- Esempio: supponiamo che le rubriche hanno contatti e che essi esistono solo se collegati alla rubrica a cui appartengono.
  - Il contatto avrà un FK_Rubrica sempre not null

- Come aggiungo un contatto ad una rubrica?

- Come cancello un contatto da una rubrica?

In entrambi i casi uso solo il DAORubrica e recupero la rubrica per PK.

MA DEVO ATTIVARE LE CASCATE

Esempio cascade

- Per aggiungere il contatto è sufficiente aggiungerlo alla lista dei contatti della rubrica ma devo impostare la cascata **PERSIST** nell'annotazione **OneToMany** relativa alla lista contatti

```
@OneToMany(cascade = CascadeType.PERSIST)
```

- In questo modo il contatto viene aggiunto alla tabella contatti e viene impostata la FK_Rubrica con la PK della rubrica relativa

- Per cancellare il contatto dalla rubrica, cerco il contatto nella lista dei contatti e se lo trovo lo rimuovo, ma devo impostare la cascata **REMOVE**

```
@OneToMany(cascade = CascadeType.REMOVE)
```

- In questo modo, il contatto viene sganciato dalla sua rubrica, cioè viene rimossa la FK (ma continua ad esistere la riga del contatto).

- NB: Posso specificare il **cascade** su singole azioni, oppure su tutte con

```
cascade = CascadeType.ALL
```

Gestione degli ‘orfani’

- Supposto che siano attivate le cascate sulla REMOVE, si può attivare un particolare tipo di cancellazione a cascata degli eventuali oggetti ‘orfani’.
- Un ‘orfano’ è un oggetto correlato ad un altro (per esempio il contatto rispetto alla rubrica cui appartiene) che potrebbe continuare ad esistere ma sganciato dall’oggetto a cui è correlato.

- Supponiamo di aver recuperato una rubrica per PK e di aver rimosso un contatto, cancellandolo dalla lista dei contatti.
  - L’ORM sgancerebbe solo la chiave esterna del contatto, il quale risulterebbe NON più collegato ad alcuna rubrica (errore, un contatto esiste solo se collegato alla sua rubrica)
- L’impostazione che gestisce il problema si ottiene con la proprietà `orphanRemoval` relativa sempre all’annotation di relazione

```
@OneToMany(cascade = CascadeType.REMOVE, orphanRemoval = true)
```

NB: l’impostazione è false per default, funziona solo con la cascata REMOVE attiva

Problema del recupero dati

- Il modello ad oggetti è un modello navigabile, è necessario quindi preoccuparsi di come navigare i dati che arrivano dal database.

- Esempio: le rubriche contengono contatti e i contatti hanno un indirizzo, alla richiesta di visualizzazione di una o tutte le rubriche dovremmo recuperare tutti i dati delle rubriche e tutti quelli correlati (a qualunque profondità)

  → problema di **eccessiva (e forse inutile) occupazione di RAM** con una conseguente **perdita di performance**.

Gestire il mismatch

- Per mitigare il mismatch tra il modello ad oggetti e quello relazionale, l'ORM prevede una impostazione per definire la possibilità di recuperare i dati correlati on demand.

- Nell'esempio precedente, per ottenere i contatti dovrò fare richiesta esplicita col metodo getter della rubrica → solo allora l'ORM esegue la seconda select
  - Più precisamente l'ORM crea un proxy e tramite questa classe esegue la query quando chiamiamo il getter method.

- In questo modo si ottengono solo i dati che realmente servono e si migliorano, in parte, le prestazioni

Lazy o Eager?

- Sulle relazioni `OneToMany` `ManyToOne` e `ManyToMany` l'impostazione di fetch è **`LAZY`** → pigra!

- Sulla relazione `OneToOne` l'impostazione è **`EAGER`** → avido, urgente!

- E' possibile impostare un diverso meccanismo di fetch in questo modo:

```
@OneToOne(fetch = FetchType.LAZY)
```

```
@OneToMany(fetch = FetchType.EAGER)
```