

## GIORNO 02

### JPA CRUD

Obiettivo del giorno è utilizzare Java con i DB attraverso apposite librerie.

Java offre strumenti che permettono di adattare lo stesso codice a diversi DB senza la necessità di dover cambiare codice se cambiamo DB. Tali strumenti mappano le classi create in Java in tabelle per essere elaborate dai DB relazionali che vogliamo sperimentare, fungendo da “traduttori” che traducono il codice che abbiamo in Java per essere interpretato dai vari DB.

L’approccio usato ieri è definito JDBC che crea stringhe contenenti SQL senza la possibilità di correzione durante la compilation time.

Java Persistence API è una specifica relativa alla **persistenza**, che in senso generale indica la capacità di oggetti Java di sopravvivere oltre il ciclo di vita dell'applicazione che li ha generati.

Non tutti gli oggetti Java devono essere resi persistenti, ma la maggior parte delle applicazioni enterprise necessitano che i loro oggetti principali vengano salvati.

La specifica JPA consente di definire quali oggetti debbano essere resi persistenti e **come** la loro persistenza debba essere gestita all'interno delle applicazioni, tali oggetti vengono chiamati Entities.

Rispetto alla persistenza implementata integrando manualmente i costrutti SQL all'interno del codice Java vista fino ad ora, JPA offre un approccio molto più avanzato ed efficace.

JPA= approccio ai DB con Java. P=Persistence (persistenza dei dati). Si tratta di una serie di concetti utili per interagire con i DB. Si tratta di un approccio più mirato alla logica e alla sintassi Java.

La libreria che utilizziamo è HIBERNATE che si basa sull’approccio JPA.

JPA non è uno strumento o un framework che possa essere utilizzato direttamente, ma è piuttosto un insieme di concetti che possono essere implementati da tools o frameworks. Il framework più diffuso che implementa le specifiche JPA è Hibernate (<https://hibernate.org/>).

Hibernate è stato rilasciato per la prima volta nel 2002 e si è evoluto nel tempo per supportare sempre maggiori funzioni, seguendo lo sviluppo delle specifiche JPA. La versione stabile più recente di Hibernate, che supporta le specifiche JPA 2.2, è la 5.4.

Hibernate, così come tutte le altre implementazioni di JPA, definisce uno strato ORM (Object Relational Mapping).

Gli ORM danno la possibilità di mappare le nostre classi Java alle tabelle dei DB: definisco una classe con attributi e tipi che viene convertita in tabella e quando creo di istanziare quella classe posso decidere di persisterli nella tabella corrispondente alla classe creata.

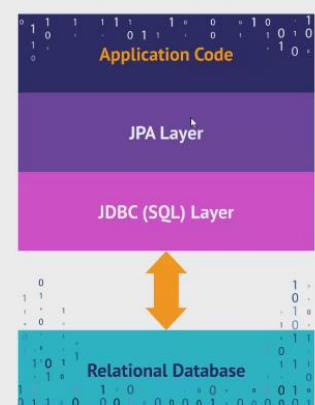
Il JPA Layer “traduce” le classi Java in tabelle DB.

Inoltre JPA lavora con più DB relazionali adattando il codice Java al “dialetto” del DB scelto.

L'Object Relational Mapping si concentra sulla necessità di trovare una corrispondenza tra la struttura delle classi Java, costruite secondo il paradigma Object Oriented, e la struttura dei database relazionali, realizzati per mezzo di tabelle e relazioni.

Uno strato ORM (JPA layer) è responsabile della conversione degli oggetti Java in tabelle e colonne del database e viceversa.

Poiché tale conversione è molto importante per la realizzazione di applicazioni enterprise con dati persistenti, l'impiego di uno strumento che possa effettuarla in modo automatico ed efficiente è utilissimo per gli sviluppatori.



Per poter utilizzare la tecnologia JPA è necessario effettuare le seguenti operazioni:

- Configurare il progetto per utilizzare JPA
- Configurare le impostazioni JPA (indirizzo del server, username, password...)
- Effettuare il mapping delle entities utilizzando le apposite annotations
- Effettuare le operazioni di interazione con il db impiegando gli strumenti JPA come l'EntityManager

L'impiego del JPA porta i seguenti vantaggi:

- Lo sviluppatore, una volta configurato il sistema, non deve più curarsi della logica SQL
- E' possibile effettuare in modo immediato il passaggio ad altri DBMS senza dover modificare il codice SQL per renderlo compatibile con il nuovo sistema (es. da PostgreSQL a MySQL)
- Nel caso di applicazioni sviluppate ex-novo si può far generare automaticamente lo schema SQL necessario a garantire la persistenza

Configurare il progetto:  
passare password, port  
ecc...

L'EntityManager possiede  
tutte le funzionalità per  
salvare, recuperare ecc  
oggetti dal DB.

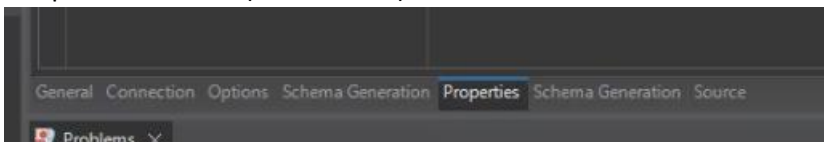
Dopo aver abilitato JPA tra i facets del progetto, occorre aggiungere nel pom.xml la dipendenza dalle librerie Hibernate, oltre alle librerie già previste, tra cui i driver del DBMS che si intende usare::

```
<dependencies>
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-entitymanager</artifactId>
<version>5.4.30.Final</version>
</dependency>
</dependencies>
```

Una volta effettuate queste attività si può procedere alla configurazione dell'interconnessione tra applicazione e database.

1. Creazione progetto JPA (New->Other->JPA Project. CI chiede NOME del Progetto -> Next -> Disabilitare Configurazione Libreria -> spuntiamo DISCOVER ANNOTATED CLASS in PERSISTENCE CLASS MANAGMENT... e in questo modo le classi creano le tabelle per il DB relazionale che useremo)
2. Creato il progetto troveremo il persistence.xml con il quale configureremo la connessione al DB
3. Conversione in Maven (tasto dx ->Configures-> ...in Maven project) e creazione del pom.xml dove metteremo tutte le dipendenze esterne:
  - HIBERNATE Entitymanager Relocation
  - POSTGRESQL JDBC DRIVER
  - LOGBACK CLASSIC (eventualmente)NB le singole dipendenze andranno messe all'interno del tag dependencies nel pom.xml e controllare i .jar in Maven Dependencies

4. Nel persistence.xml (in META-INF):



1. Andiamo in PROPERTIES, Clicchiamo su Add e aggiungiamo le proprietà e i valori che andranno a configurare la connessione di Java al DB

This table lists all properties that are defined for this persistence unit.

Name	Value
javax.persistence.jdbc.driver	org.postgresql.Driver
javax.persistence.jdbc.url	jdbc:postgresql://localhost:5432/d12
javax.persistence.jdbc.user	postgres
javax.persistence.jdbc.password	1234
hibernate.hbm2ddl.auto	update

## 5. Possiamo creare la nostra Class main (eventualmente col Logger)

(NB LE SLIDES PARTONO DAL PROGETTO MAVEN E CONVERTE IN JPA)

Una volta configurati i parametri dell'applicazione è necessario effettuare la mappatura delle classi del modello che devono essere rese persistenti e diventare quindi delle Entities.

Per effettuare il mapping è possibile utilizzare apposite annotations da inserire nel codice sorgente delle classi, che appartengono al package javax.persistence.

Le annotazioni principali sono:

- @Entity – indica che la classe deve essere gestita come entity
- @Table – indica il nome della tabella DB corrispondente alla classe
- @Column – indica la colonna della tabella a cui far corrispondere un attributo
- @Id – indica l'attributo che deve essere utilizzato come chiave primaria dell'elemento
- @Enumerated – Indica come deve essere mappato un campo corrispondente ad una proprietà di tipo Enum

@Entity permette la gestione della Classe dove viene applicate, come Tabella nel DB relazionale che scelgo

```
@Entity
@Table(name = "automobile")
public class Automobile {
    @Id
    @Column(name = "id")
    private Long id;

    @Column(name = "targa")
    private String targa;

    @Column(name = "colore")
    private String colore;

    @Enumerated(EnumType.STRING)
    @Column(name = "tipo")
    private TipoAutomobile tipo;

    // Costruttori
    // Setters e Getters
}
```



id [PK] bigint	colore character varying (255)	targa character varying (255)	tipo character varying (255)
1	Bianco	AB 245 TR	COUPE
2	Nero	RE 546 WQ	BERLINA
3	Rosso	GT 679 LG	BERLINA

La conversione di Classe in Tabella comporta anche la conversione dei tipi (String -> character varying, ecc...)

## 6. Creazione classe standard (esempio Studente) in package = utilities

```

1 package u4d12.entities;
2
3 import javax.persistence.Entity;
4 import javax.persistence.Id;
5 import javax.persistence.Table;
6
7 @Entity
8 @Table(name = "students")
9 public class Student {
10     @Id
11     private long id;
12     private String firstName;
13     private String lastName;
14
15     public Student() {}
16
17     public Student(String firstName, String lastName) {
18         this.firstName = firstName;
19         this.lastName = lastName;
20     }
21 }
22

```

Non passo l'id nel costruttore perché viene gestito dal DB. Eventualmente creo setters e getters (ovviamente tranne dell'id)

NB @Entity ha bisogno di un costruttore vuoto.

L'annotation @Id dice che l'id (di tipo long) sarà la chiave primaria della tabella (creata a partire dalla classe Student)

@Table serve per definire un nome custom per la nostra tabella

Quando si implementa una applicazione con entità persistenti possono presentarsi due scenari:

**Le tabelle relative alle entities sono già presenti nel database** - In questo caso occorre utilizzare le annotation in modo da far corrispondere classi e tabelle usando l'annotation @Table e attributi con colonne usando @Column. Nel caso in cui non si specifichino tali attributi, JPA cerca una corrispondenza esatta tra nome della classe persistente e tabella e attributo e colonna

**Le tabelle relative alle entities non sono ancora state create** - Questo può avvenire sia per nuove applicazioni che per applicazioni pre-esistenti a cui si aggiungono classi e funzionalità. In questo caso, per velocizzare lo sviluppo, si può impiegare la funzione di autogenerazione dei DDL

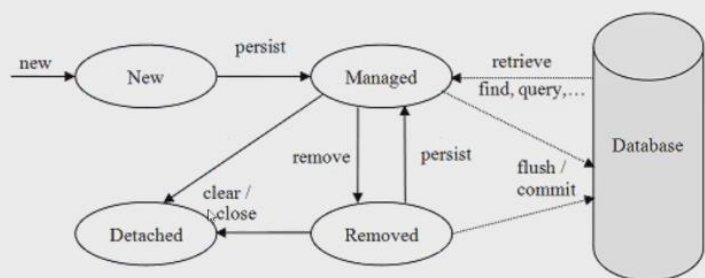
La funzione si attiva inserendo nel persistence.xml la seguente istruzione:

```
<property name="hibernate.hbm2ddl.auto" value="update"/>
```

essa indica al sistema di effettuare un controllo sulle tabelle del db e creare o aggiornare le tabelle corrispondenti alle entities in caso di necessità

Quando si lavora con le entities in JPA, si stanno manipolando oggetti che possono rappresentare elementi già presenti in un db, oppure elementi che, se salvati, verranno inseriti nel db, passando da uno stato di persistenza all'altro. Tali passaggi rappresentano il **ciclo di vita (lifecycle)** delle entities.

L'insieme di tutti gli oggetti in stato Managed rappresenta il **Persistence Context**. Se si cerca di caricare dal db una entity che risulta già presente nel Persistence Context, esso la restituirà senza accedere al db, garantendo che non ci sia in memoria più di una istanza della stessa entity collegata ad un persistence context.



Gli oggetti che manipolo in Java rimangono sempre collegati agli oggetti nel DB (Persistence Context): leggo l'oggetto dal DB, lo modifico con un metodo (clear, remove ecc...) verrà modificato anche nel DB. Creo un nuovo oggetto che diventa Managed e posso decidere di persisterlo o di cambiarlo. Tutte le modifiche avvengono anche nel DB dopo aver fatto flush/commit. Per modificare gli oggetti, li leggo dal DB, li modifico con Java e questi verranno modificati anche nel DB.



```

http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd (xsi:schemaLocation with catalog)
1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
3   <persistence-unit name="u4d12">
4     <properties>
5       <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>
6       <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/d12"/>
7       <property name="javax.persistence.jdbc.user" value="postgres"/>
8       <property name="javax.persistence.jdbc.password" value="1234"/>
9       <property name="hibernate.hbm2ddl.auto" value="update"/>
10      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
11      <property name="hibernate.default_schema" value="public"/>
12    </properties>
13  </persistence-unit>
14 </persistence>
15

```

La persistence-unit serve per raggruppare tutte le informazioni relative al nostro DB. Molto importante il nome che viene fornito!

EntityManager = Oggetto che ha tutti i metodi utili per collegarsi al DB (leggerli, modificarli, renderli persistente ecc...)

Consiglio: crearlo in package separato (util)

```

1 package u4d12.util;
2
3 import javax.persistence.EntityManagerFactory;
4 import javax.persistence.Persistence;
5
6 public class JpaUtil {
7     private static final EntityManagerFactory emf = Persistence.createEntityManagerFactory("u4d12");
8     // occhio a u4d12 che deve essere il nome della persistence-unit settato in
9     // persistence.xml
10
11     public static EntityManagerFactory getEntityManagerFactory() {
12         return emf;
13     }
14 }
15
16

```

Nel main:

```

1 package u4d12.app;
2
3 import javax.persistence.EntityManager;
4 import javax.persistence.EntityManagerFactory;
5
6 import u4d12.util.JpaUtil;
7
8 public class Application {
9     private static EntityManagerFactory emf = JpaUtil.getEntityManagerFactory();
10
11     public static void main(String[] args) {
12         EntityManager em = emf.createEntityManager();
13     }
14 }
15
16

```

L'oggetto em contiene tutti i metodi utili per collegarsi al DB.

NB SE LA CONSOLE ESPLODE DI ROSSO....VA TUTTO BENE!

Nel DB è stata creata la tabella con le proprietà (id, firstname e lastname) che abbiamo definito come attributi della classe Student grazie all'annotation @Entity sulla classe Student.

Ora che siamo collegati al DB, possiamo crearci i metodi per interagire con esso e modificare le tabelle:

- Creo una classe (StudentDAO) per salvare, cercare, modificare gli oggetti istanziati da Student e che interagiranno col DB. In questo modo rendo scalabili e riutilizzabili tali metodi.

I nomi ovviamente sono, come sempre, "parlanti".

Il DAO ha bisogno di un riferimento all'EntityManager all'interno di un costruttore appositamente creato.

```
1 package u4d12.app;
2
3 import javax.persistence.EntityManager;
4 import javax.persistence.EntityManagerFactory;
5
6 import u4d12.entities.StudentsDAO;
7 import u4d12.util.JpaUtil;
8
9 public class Application {
10     private static EntityManagerFactory emf = JpaUtil
11
12
13     public static void main(String[] args) {
14         EntityManager em = emf.createEntityManager();
15         System.out.println("CIAO");
16
17         StudentsDAO sd = new StudentsDAO(em);
18     }
19 }
20
21
22 package u4d12.entities;
23
24 import javax.persistence.EntityManager;
25
26 public class StudentsDAO {
27     private final EntityManager em;
28
29     public StudentsDAO(EntityManager em) {
30         this.em = em;
31     }
32
33     public void save(Student s) {
34     }
35
36     public Student findById(long id) {
37     }
38
39     public void findByIdAndDelete(long id) {
40     }
41 }
```

Consiglio: avere tanti DAO quante sono le tabelle/classi. DAO = DATA ACCESS OBJECT

Salvare un oggetto:

```
1 package u4d12.entities;
2
3 import javax.persistence.EntityManager;
4 import javax.persistence.EntityTransaction;
5
6 public class StudentsDAO {
7     private final EntityManager em;
8
9     public StudentsDAO(EntityManager em) {
10         this.em = em;
11     }
12
13     public void save(Student s) {
14         EntityTransaction t = em.getTransaction();
15         t.begin();
16         em.persist(s);
17         t.commit();
18     }
19
20     // public Student findById(long id) {
21     // }
22
23     public void findByIdAndDelete(long id) {
24     }
25 }
```

```
1 package u4d12.app;
2
3 import javax.persistence.EntityManager;
4 import javax.persistence.EntityManagerFactory;
5
6 import u4d12.entities.StudentsDAO;
7 import u4d12.util.JpaUtil;
8
9 public class Application {
10     private static EntityManagerFactory emf = JpaUtil
11
12
13     public static void main(String[] args) {
14         EntityManager em = emf.createEntityManager();
15         System.out.println("CIAO");
16
17         Student aldo = new Student("Aldo", "Baglio");
18         StudentsDAO sd = new StudentsDAO(em);
19         sd.save(aldo);
20
21         //
22         em.close();
23         emf.close();
24     }
25 }
```

JPA per salvare gli oggetti utilizza le Transazioni. Tra .begin() e .commit() posso cambiare l'oggetto come voglio e tale modifica viene resa persistente, cioè l'oggetto viene aggiunto al persistence context e non viene ancora salvato al DB. Viene salvato alla fine della transizione, cioè nel commit().

```
1 package u4d12.entities;
2
3 import javax.persistence.EntityManager;
4 import javax.persistence.EntityTransaction;
5
6 public class StudentsDAO {
7     private final EntityManager em;
8
9     public StudentsDAO(EntityManager em) {
10         this.em = em;
11     }
12
13     public void save(Student s) {
14         EntityTransaction t = em.getTransaction();
15         t.begin(); // inizia la transazione
16         em.persist(s);
17         // qua l'oggetto viene aggiunto al persistence context, non viene ancora salvato al DB
18         t.commit(); // termina la transazione, qua l'oggetto viene salvato
19         System.out.println("Studente salvato correttamente");
20     }
21
22     // public Student findById(long id) {
23     // }
24 }
```

Sd è di tipo StudentDAO e il metodo save accetta come parametro uno Student (aldo nell'esempio che è stato istanziato con nome e cognome).

```

@Entity
@Table(name = "students") // Serve per definire un nome custom per la nostra tab
public class Student {
    @Id // Serve per definire chi sarà la chiave primaria
    @GeneratedValue // Obbligatorio usarlo se si vuol fare gestire gli id al db
    private UUID id;
    private String firstName;
    private String lastName;

    public Student() {
    }

    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public UUID getId() {
        return id;
    }

    public String getFirstName() {
        return firstName;
    }
}

```

Entity manager e factory  
vanno sempre chiusi con  
.close()

NB runnando la prima volta  
tutto ok, ma la seconda volta ci  
ritorna un errore perché lo  
Student aldo viene duplicato.

Per evitare usiamo  
l'annotazione

@GeneratedValue col la quale  
l'id viene generato dal DB.

NB l'id può essere usato col  
tipo UUID per avere come id  
una stringa univoca.

L'annotation @Column(unique=true)  
rende la proprietà della classe unica.

	id [PK] uuid	firstname character varying (255)	lastname character varying (255)
1	5700cd5b-c447-4051-bb28-6d42da895433	Aldo	Baglio
2	cf8fc293-e9c4-4ee0-9f62-77852es3574	Aldo	Baglio
3	ffccbd82-cedb-48ac-b724-3653c7d67ea7	Aldo	Baglio

Creazione di un metodo che dato un id mi cerca lo studente nel DB:

Nella classe DAO:

```

public Student findById(UUID id) {
    Student found = em.find(Student.class, id);
    return found;
}

public void findByIdAndDelete(long id) {
}

```

.find() vuole due parametri: la classe dove agire e  
l'id. .find() è come se facesse:  
select \* from students where id=id

Nel main:

```

Student aldoFromDB = sd.findById(UUID.fromString("5700cd5b-c447-4051-bb28-6d42da895433"));
System.out.println(aldoFromDB);

```

Ovviamente bisogna fare Override di toString nella Classe Student per poterlo printare. UUID.fromString si usa se abbiamo utilizzato il tipo UUID per l'id. Consiglio: utilizzare long per gli id.

```

Student [id=ffccbd82-cedb-48ac-b724-3653c7d67ea7, firstName=Giacomo, lastName=Baglio]

```

```

public void findByIdAndDelete(UUID id) {
    // 1. faccio una find prima per ottenere lo studente
    Student found = em.find(Student.class, id);
    if (found != null) {
        // 2. Poi elimino quello studente
        // 2.1 Ottengo la transazione
        EntityTransaction t = em.getTransaction();
        // 2.2 faccio partire la transazione
        t.begin();
        // 2.3 Rimuovo l'oggetto
        em.remove(found);
        // 2.4 faccio il commit della transazione
        t.commit();
        System.out.println("Studente eliminato correttamente");
    } else {
        System.out.println("Studente non trovato");
    }
}

```

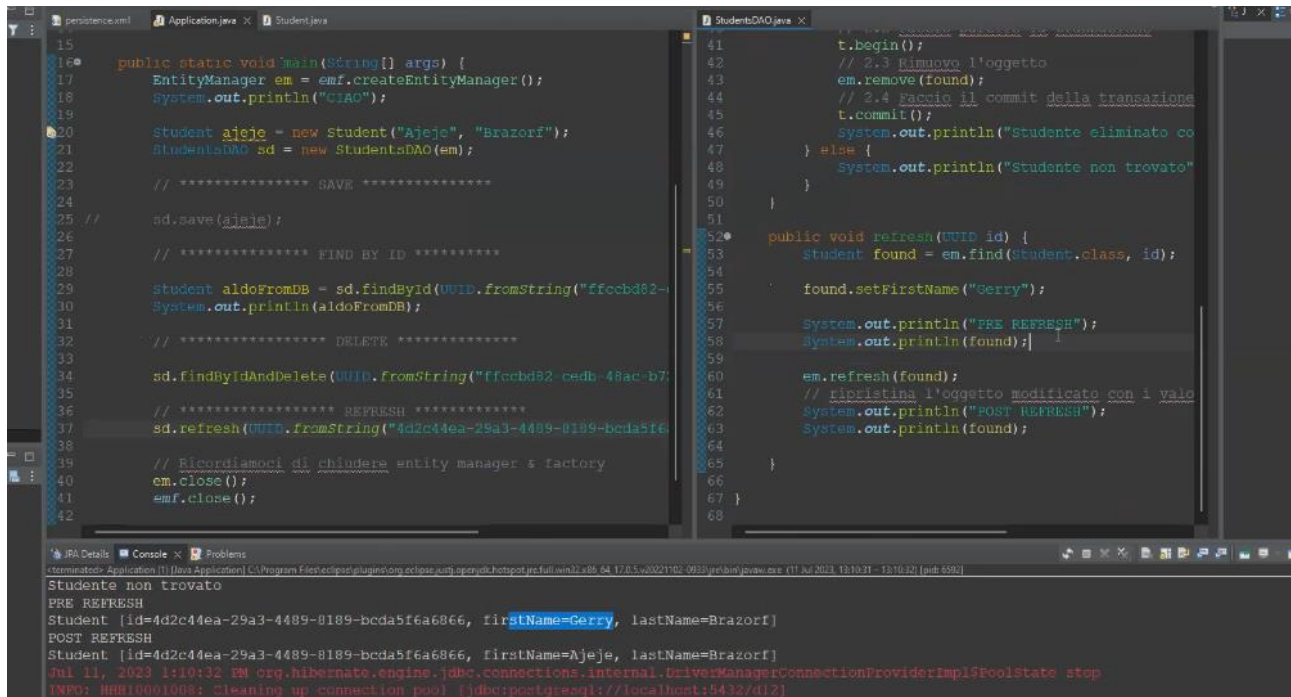
Eliminare un record:

Creo nel DAO il metodo  
findByIdAndRemove che dovrà cercare  
(find per ottenere uno studente) e  
rimuoverlo.

Quando faccio `.commit()` chiamo automaticamente il metodo `.flush()` che aggiorna il persistent context col DB. Il refresh serve per fare il contrario del flush(): se il DB è più aggiornato al Persistence Context, allora uso `refresh()` per allinearli.

Nel DAO:

Cerco un oggetto, lo modifico (`setFirstName`) all'interno del persistence context, e uso `refresh()` per ripristinare l'oggetto modificato con i valori provenienti dal DB.



```
15 public static void main(String[] args) {
16     EntityManager em = emf.createEntityManager();
17     System.out.println("CIAO");
18
19     Student ajeje = new Student("Ajeje", "Brazorf");
20     StudentsDAO sd = new StudentsDAO(em);
21
22     // ***** SAVE *****
23
24     //
25     sd.save(ajeje);
26
27     // ***** FIND BY ID *****
28
29     Student aldoFromDB = sd.findById(UUID.fromString("ffecb382-"));
30     System.out.println(aldoFromDB);
31
32     // ***** DELETE *****
33
34     sd.findByIdAndDelete(UUID.fromString("ffecb382-cedb-48ac-b7"));
35
36     // ***** REFRESH *****
37     sd.refresh(UUID.fromString("4d2c44ea-29a3-4489-8189-bcda5f6"));
38
39     // Ricordiamoci di chiudere entity manager & factory
40     em.close();
41     emf.close();
42 }
```

```
41 t.begin();
42 // 2.3 rimuovo l'oggetto
43 em.remove(found);
44 // 2.4 faccio il commit della transazione
45 t.commit();
46 System.out.println("Studente eliminato co");
47 } else {
48     System.out.println("Studente non trovato");
49 }
50 }
51
52 public void refresh(UUID id) {
53     Student found = em.find(Student.class, id);
54
55     found.setFirstName("Gerry");
56
57     System.out.println("PRE REFRESH");
58     System.out.println(found);
59
60     em.refresh(found);
61     // ripristina l'oggetto modificato con i valo
62     System.out.println("POST REFRESH");
63     System.out.println(found);
64 }
65 }
66
67 }
```

JBRA Details Console Problems

<terminated> Application [1] (Java Application) C:\Program Files\chocolatey\org.eclipse.sisu.launcher\org.eclipse.sisu.launcher.exe (11 Jul 2023 13:10:31 - 13:16:32) [pid: 6562]

Studente non trovato

PRE REFRESH

Student [id=4d2c44ea-29a3-4489-8189-bcda5f6a6866, firstName=Gerry, lastName=Brazorf]

POST REFRESH

Student [id=4d2c44ea-29a3-4489-8189-bcda5f6a6866, firstName=Ajeje, lastName=Brazorf]

Jul 11, 2023 1:10:32 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl\$PoolState stop

INFO: HH00000008: Cleaning up connection pool [jdbc:postgresql://localhost:5432/d12]