

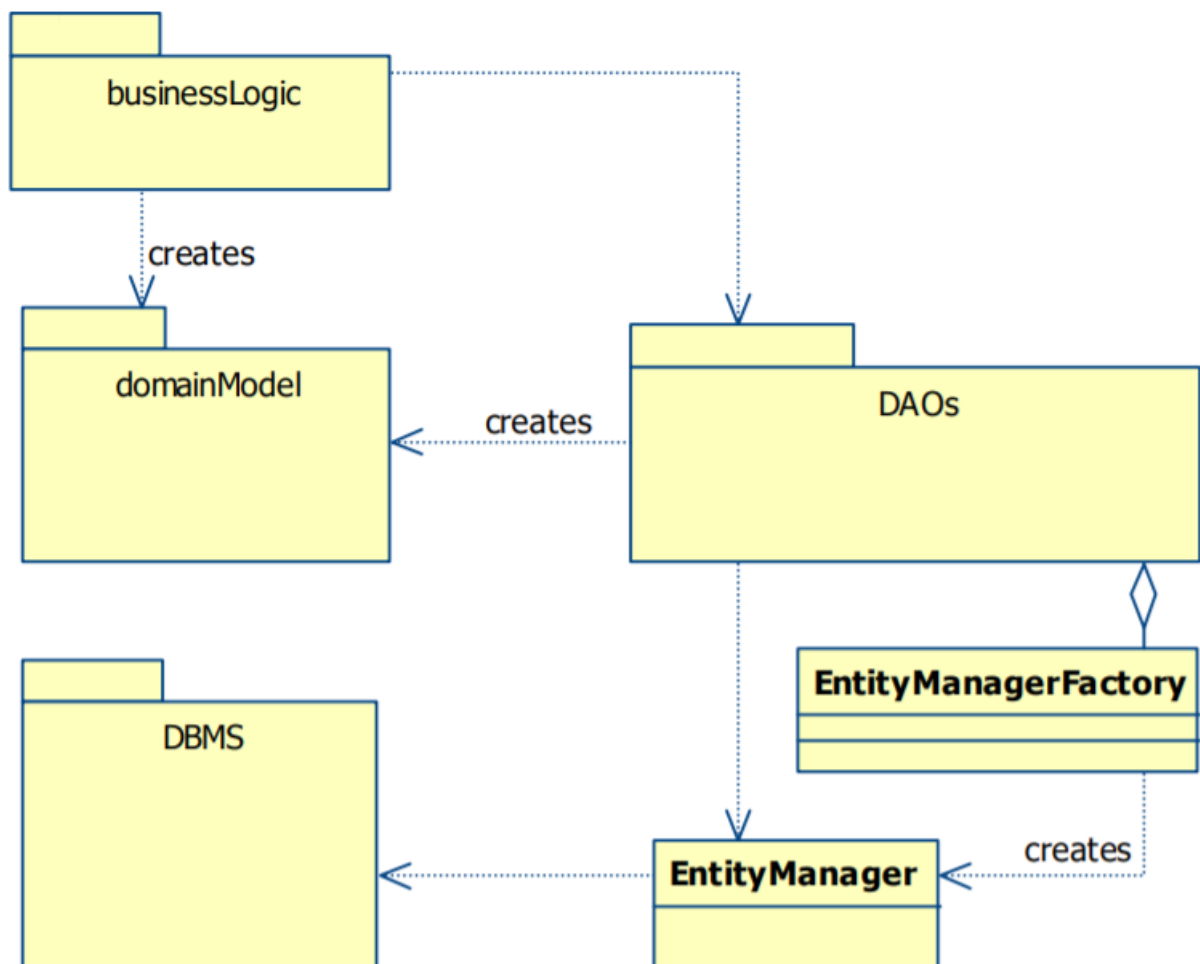
Progetto JPA (Hibernate) con database relazioni (MySQL e PostgreSQL) e non relazionale (Infinispan)

JPA, acronimo di Java Persistence API è una specifica Java per la gestione dei dati tra oggetti Java e database relazionali.

Hibernate è un framework che implementa questa specifica e offre delle API che permettono di accedere, persistere e gestire i dati in un database relazionale. In questo progetto, verrà usato il database MySQL come database.

Esiste anche Hibernate OGM che permette di gestire i database non relazionali e in questo tutorial verrà usato il database non relazionale in-memory Infinispan.

L'architettura utilizzata sarà la seguente:



Nel domainModel ci saranno le entità (Entity) che rispecchieranno gli oggetti che verranno persistiti nel Database.

I DAOs (Data Access Object) sono classi Java che rappresentano un'interfaccia per la gestione degli oggetti persistiti nel database e permettono di isolare l'accesso ad una tabella tramite query.

La Business logic conterrà la logica dell'applicazione, ovvero tutte quelle operazioni che l'utente intende compiere.

L'EntityManager è l'oggetto Java che viene messo a disposizione dalle API JPA per comunicare a basso libello con il DBMS, ovvero il database.

L'EntityManagerFactory è l'oggetto Java che mette in vita l' EntityManager.

Database Relazionale (MySql)

- Configurazione Maven

Innanzitutto è necessario creare un progetto Maven ed inserire tra le dipendenze hibernate-core e mysql-connector-java e hibernate-jpa-2.1-api

```
<!-- Definition of JPA APIs intended for use in developing Hibernate JPA
      implementation -->
<dependency>
  <groupId>org.hibernate.javax.persistence</groupId>
  <artifactId>hibernate-jpa-2.1-api</artifactId>
  <version>1.0.0.Final</version>
</dependency>

<!-- Hibernate's core ORM functionality -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.3.1.Final</version>
</dependency>

<!-- JDBC (Java DataBase Connectivity) for mySql -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.46</version>
</dependency>
```

- Container Docker

Per far funzionare tutto è necessario utilizzare un container Docker per poter creare il database MySql e poter effettuare tutti i test necessari.

Creiamo quindi un file docker-compose.yml con il seguente contenuto:

```
version: "2"

services:
  # Use root as user and a blank password
  mysql:
    image: mysql:5.7
    networks:
      - mysqlnet
    restart: always
    ports:
      - 3306:3306
    environment:
      - MYSQL_DATABASE=timekeep
      - MYSQL_ALLOW_EMPTY_PASSWORD=yes
```

```
networks:
  mysqlnet:
    driver: bridge
```

- Prima Entity

Creiamo nel package la nostra prima Entity, ovvero una classe Java che rappresenterà i dati che saranno permanentemente memorizzati in un database relazionale.

```
public class Project {

    private Long id;
    private String title;
    private String description;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}
```

Aggiungiamo adesso le annotazioni `@Entity` e `@Id` e `@GeneratedValue` per farla diventare un'entità e fare in modo che venga generato una chiave primaria, Id in questo caso, che possa identificare ogni singolo Progetto.

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
```

`@Entity`

```
public class Project {

    private Long id;
    private String title;
    private String description;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getDescription() {
        return description;
    }
}
```

```

    public void setDescription(String description) {
        this.description = description;
    }
}

```

Facciamo anche l'override del metodo equals e hashCode che ci permetterà di controllare se due Progetti sono uguali.

Utilizziamo quella che viene chiamata Business key equality, ovvero controlliamo ogni singolo campo che ha una semantica nel modello, quindi non la chiave primaria Id.

```

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((title == null) ? 0 : title.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Project other = (Project) obj;
    if (title == null) {
        if (other.title != null)
            return false;
    } else if (!title.equals(other.title)) {
        return false;
    }
    return true;
}

```

- Configurazione Persistence.xml

Adesso è necessario creare un file persistence.xml nella cartella src/main/resources/META-INF.

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

    <persistence-unit name="mysql-pu" transaction-type="RESOURCE_LOCAL">
        <!-- The persistence-unit name must to be UNIQUE -->
        <!-- The value RESOURCE_LOCAL to replace the default value (JTA) allows
            you to manage connection with database manually -->

        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <!-- The provider specifies the JPA implementation that you want to use
            (Hibernate) -->

        <class>com.garritano.keepchronos.model.sql.Project</class>
        <!-- Specify names of managed persistable classes -->

        <properties>

```

```

<property name="hibernate.dialect"
    value="org.hibernate.dialect.MySQL5InnoDBDialect" />
<!-- The dialect specifies the language of your DB (MySQL 5 InnoDB)
-->

<property name="hibernate.hbm2ddl.auto" value="create" />
<!-- Possible values for hbm2ddl:
    validate: validate the schema, makes no changes to the
    database.
    update: update the schema.
    create: creates the schema, destroying previous data.
    create-drop: drop the schema when the SessionFactory
    is closed explicitly, typically when the application
    is stopped.
-->

<property name="hibernate.connection.url"
    value="jdbc:mysql://localhost:3306/timekeep?
createDatabaseIfNotExist=true" />
<!-- the address with port of your mySql service, followed by the
    name of DB that is created if not exist -->

<property name="hibernate.connection.username" value="root" />
<property name="hibernate.connection.password" value="" />
<!-- username and password of your mySql connection -->

<property name="hibernate.show_sql" value="true" />
<!-- show_sql=true permits to view queries in console -->

<property name="hibernate.format_sql" value="true" />
<!-- format_sql=true permits to view formatted queries in console
-->

</properties>
</persistence-unit>
</persistence>

```

Questo è un file di configurazione che permette a Hibernate di configurare una serie di parametri che sono commentati per capire meglio il funzionamento.

In particolare è necessario dare un nome alla persistence-unit. In questo caso ho inserito “mysql-pu”. È necessario anche fornire una lista delle classi che saranno persistite, inserendo il nome delle classi tra i tag `<class>*`, il dialect che indica con quale database saranno persistiti i dati (altri dialetti sono disponibili qui: <https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/dialect/package-summary.html>), l’url per la connessione al Database e le credenziali di accesso. Nel valore della proprietà `hibernate.connection.url` è stato aggiunta la dicitura `?createDatabaseIfNotExist=true` che, come dice il nome, crea il database se non presente.

Per motivi di testing, ho inserito come proprietà `hibernate.hbm2ddl.auto` il valore `create` che distrugge eventuali dati presenti sul database ogni volta che il programma si avvia. In fase di produzione è necessario utilizzare un differente valore, come ad esempio `update`.

Ovviamente è necessario cambiare i dati dei nomi delle classi da persistere, l’url del database a cui connettersi e le credenziali di accesso, se diverse da quanto indicato.

Adesso è necessario persistere i dati ed useremo dei test per fare ciò. È necessario quindi inserire i plugin `maven-surefire-plugin` e `maven-failsafe-plugin`.

Si crei una copia del file `src/main/resources/META-INF/persistence.xml` in `src/test/resources/META-INF/` e si crei quindi la classe di test `ProjectIntegrationTest.java` per persistere i dati e vedere se essi vengano persistiti in modo corretto.

- Persistenza degli oggetti e test

Per una corretta scrittura del codice, è necessario testare il metodo `equals()` e `hashCode()`, ma che tralascerò al lettore per concentrarci più sulla parte di Jpa.

Per persistere i dati è necessario instaziare un `entityManager`, ma prima di questo è necessario istanziare l'`entityManagerFactory`.

```
public class ProjectIntegrationTest {
    private static final String PERSISTENCE_UNIT_NAME = "mysql-pu";
    private static EntityManagerFactory entityManagerFactory;

    @BeforeClass
    public static void setUpClass() {
        entityManagerFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
    }

    @AfterClass
    public static void tearDownClass() {
        entityManagerFactory.close();
    }
}
```

È importantissimo che `PERSISTENCE_UNIT_NAME` che viene passata al metodo `createEntityManagerFactory` sia inizializzata con lo stesso nome della persistence-unit del file `persistence.xml`.

È buona norma utilizzare la funzione `close()` per liberare le risorse.

Creiamo adesso l'`EntityManager` che ci permetterà di persistere le risorse, effettuiamo una query di cancellazione della tabella `Project` e inizializziamo gli oggetti `Project` che verranno persistiti.

```
private EntityManager entityManager;
private Project project1;
private Project project2;

@Before
public void setUp() throws Exception {
    entityManager = entityManagerFactory.createEntityManager();

    // make sure to drop the Project table for testing
    entityManager.getTransaction().begin();
    entityManager.createNativeQuery("delete from Project").executeUpdate();
    entityManager.getTransaction().commit();

    project1 = new Project();
    project1.setTitle("First project");
    project1.setDescription("This is my first project, hi!");

    project2 = new Project();
    project2.setTitle("Second project");
    project2.setDescription("This is my second project, wow!");
}
```

Come è possibile vedere, per effettuare una query è necessario invocare il metodo `entityManager.getTransaction().begin();` e `entityManager.getTransaction().commit();` rispettivamente prima e dopo l'esecuzione della query.

Come per l'EntityManagerFactory, è buona norma chiudere anche l'EntityManager.

```
@After
public void tearDown() {
    project1 = null;
    project2 = null;
    entityManager.close();
}
```

Testiamo adesso il salvataggio su database di un singolo oggetto, ma prima creiamo una funzione di utility che ci permetterà di aprire una transazione, eseguire la persistenza dell'oggetto e chiudere la transazione. Una transazione è un oggetto messo a disposizione da Jpa per permettere al software di persistere un oggetto in modo atomico.

```
private void transactionPersist(Project p) {
    entityManager.getTransaction().begin();
    entityManager.persist(p);
    entityManager.getTransaction().commit();
}
```

Creiamo quindi un metodo per eseguire avere tutti i Project presenti nel database, ricordandoci che è necessario aprire e chiudere la transazione.

```
private TypedQuery<Project> query;

private List<Project> getAllProjects() {
    //
    entityManager.getTransaction().begin();
    query = entityManager.createQuery("select p from Project p",
Project.class);
    entityManager.getTransaction().commit();
    return query.getResultList();
}
```

Andiamo adesso a testare la persistenza.

```
@Test
public void testBasicPersistence() {
    transactionPersist(project1);
    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();

    List<Project> projectsList = getAllProjects();

    // We should have only one project in the database
    assertTrue(projectsList.size() == 1);

    // We should have the same title
    assertEquals(project1.getTitle(), projectsList.get(0).getTitle());

    // and the same description
    assertEquals(project1.getDescription(), projectsList.get(0).getDescription());
}
```

È importante utilizzare la funzione `entityManager.clear()`; perché senza di essa non andremo a testare la presenza del record sul database, ma solo nella memoria cache. Questa strategia viene adottata da Hibernate per effettuare un numero minore di query sul database e risparmiare risorse.

È possibile aggiungere un secondo metodo per testare la persistenza di più oggetti, in questo modo:

```
@Test
public void testMultiplePersistence() {
    transactionPersist(project1);
    transactionPersist(project2);

    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();

    List<Project> projectsList = getAllProjects();

    // We should have 2 projects in the database
    assertTrue(projectsList.size() == 2);
}
```

- Creazione Dao di Project e testing

Adesso creiamo il Dao per gestire la tabella Project:

Creiamo la classe `ProjectDao` e impostiamo il seguente costruttore:

```
protected EntityManager entityManager;

public ProjectDao(EntityManager entityManager) {
    this.entityManager = entityManager;
}
```

Andiamo adesso a creare una sorta di interfaccia che possa rendere trasparente il layer DBMS, ma che possa farci comunicare con la tabella Project. Creiamo quindi i metodi `save` (per salvare un oggetto Project sulla tabella), `getAll` (per avere tutti gli oggetti di tipo Project della tabella), `findById` (per ricevere in output un oggetto di tipo Project, dato il suo identificativo), `update` (per aggiornare lo stato di un oggetto Project già persistito, o crearne uno nuovo in caso non esista sul database).

Il risultato dello sviluppo sarà il seguente:

```
public class ProjectDao {

    private EntityManager entityManager;

    public ProjectDao(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    public void save(Project project) {
        entityManager.getTransaction().begin();
        entityManager.persist(project);
        entityManager.getTransaction().commit();
    }

    public List<Project> getAll() {
        return entityManager.createQuery("select p from Project p",
Project.class).getResultList();
    }
}
```



```

    }

    public Project findById(Long id) {
        return entityManager.find(Project.class, id);
    }

    public void update(Project project) {
        entityManager.getTransaction().begin();
        entityManager.merge(project);
        entityManager.getTransaction().commit();
    }
}

```

Con l'utilizzo dell'EntityManager, che ci alleggerisce notevolmente il lavoro, è possibile svolgere tutte le seguenti operazioni attraverso l'uso dei metodi persist(), find() e merge(), oltre che effettuare delle query sul database utilizzando quello che viene chiamato JPQL (Java Persistence Query Language).

Il linguaggio JPQL ci permette di effettuare delle query indipendentemente da quale layer DBMS stiamo utilizzando.

Esiste anche la possibilità di effettuare delle query native con il metodo createNativeQuery(), ma in questi casi si potrebbe perdere la flessibilità offerta da JPA di essere indipendente dal tipo di DBMS utilizzato.

Testiamo adesso i vari metodi con vari test di integrazione.

```

public class ProjectDaoIntegrationTest {

    private static final String PERSISTENCE_UNIT_NAME = "mysql-pu";
    private static EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;
    private ProjectDao projectDao;

    private Project project1;
    private Project project2;

    @BeforeClass
    public static void setUpClass() {
        entityManagerFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
    }

    @Before
    public void setUp() throws Exception {
        entityManager = entityManagerFactory.createEntityManager();

        // make sure to drop the Project table for testing
        entityManager.getTransaction().begin();
        entityManager.createNativeQuery("delete from Project").executeUpdate();
        entityManager.getTransaction().commit();

        projectDao = new ProjectDao(entityManager);

        project1 = new Project();
        project1.setTitle("First project");
        project1.setDescription("This is my first project, hi!");

        project2 = new Project();
        project2.setTitle("Second project");
        project2.setDescription("This is my second project, wow!");
    }
}

```

```

@Test
public void testSave() {
    projectDao.save(project1);

    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();

    assertEquals(project1, entityManager.createQuery("from Project where id
=:id", Project.class)
        .setParameter("id", project1.getId()).getSingleResult());
}

@Test
public void testEmptyGetAll() {
    assertTrue(projectDao.getAll().size() == 0);
}

@Test
public void testOneGetAll() {
    projectDao.save(project1);

    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();

    assertEquals(project1, projectDao.getAll().get(0));
    assertTrue(projectDao.getAll().size() == 1);
}

@Test
public void testMultipleGetAll() {
    projectDao.save(project1);
    projectDao.save(project2);

    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();

    assertTrue(projectDao.getAll().size() == 2);
}

@Test
public void testEmptyFindById() {
    assertNull(projectDao.findById((long) 34214342));
}

@Test
public void testNotEmptyFindById() {
    projectDao.save(project1);

    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();

    assertEquals(project1, projectDao.findById(project1.getId()));
}

@Test
public void testUpdate() {
    projectDao.save(project1);
    project1.setDescription("new description!");
    projectDao.update(project1);

    // Clear Hibernate's cache to make sure data is retrieved from the store

```

```

        entityManager.clear();

        assertEquals(project1.getDescription(),
projectDao.findById(project1.getId()).getDescription());
    }

    @After
    public void tearDown() {
        project1 = null;
        project2 = null;
        entityManager.close();
    }

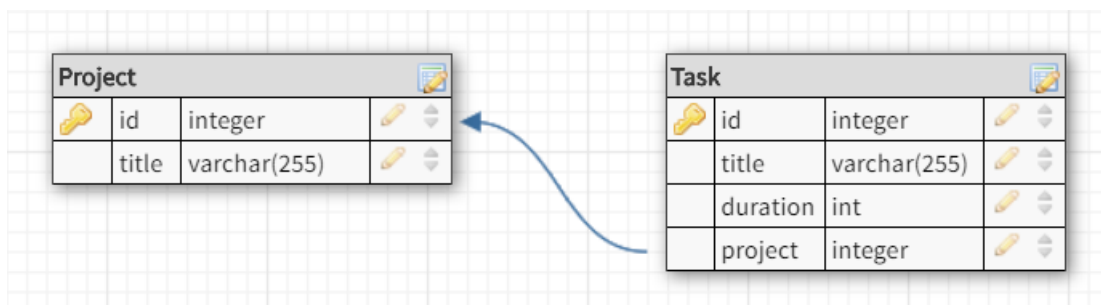
    @AfterClass
    public static void tearDownClass() {
        entityManagerFactory.close();
    }
}

```

Ogni volta che si testa il contenuto del database, è importante cancellare la cache di Hibernate con `entityManager.clear()`;

- Creazione Task Entity

Andiamo adesso a creare una seconda entità in modo che venga mappata sul database nel modo riportato in figura:



La prima cosa da fare è inserire nei file persistence.xml il nome della classe tra i tag `<class></class>`, come è stato fatto per la classe Project.

È necessario quindi che la tabella Task abbia una foreign key. Jpa ci permette di fare questo creando un attributo Project sulla classe Task e andando ad annotare l'attributo Project come `@ManyToOne`. L'annotazione `@ManyToOne` ci permette di specificare a JPA che deve esserci una colonna di tipo foreign key sulla tabella Task.

Creiamo quindi la classe Task:

```

@Entity
public class Task{

    private Long id;
    private String title;
    private Project project;
    private int duration;

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

```

```

    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
}

@ManyToOne
public Project getProject() {
    return project;
}

public void setProject(Project project) {
    this.project = project;
}

public int getDuration() {
    return duration;
}

public void setDuration(int duration) {
    this.duration = duration;
}
}

```

Aggiungiamo tra i tag `<class></class>` nei file persistence.xml la classe Task.

Dato che molte parti del codice di Task sono uguali, in particolare il metodo equals, gli attributi id e title e i relativi getter e setter, è possibile utilizzare una classe astratta che rende il codice più pulito. Per JPA andremo ad utilizzare l'annotazione `@MappedSuperclass` per poter mappare gli attributi ereditati in modo corretto, senza utilizzare una tabella separata.

Per una corretta comparazione degli oggetti di tipo Task è necessario, anche qui, utilizzare la Business key equality e testare il tutto.

Questo è il risultato delle 3 classi: `BasicEntity` entità base che contiene gli attributi e i metodi in comune, `Project` senza alcun metodo o attributo e `Task` con attributi duration e Progetto associato (e i relativi getter/setter)

```

@MappedSuperclass
public abstract class BasicEntity {

    private Long id;
    private String title;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }
}

```

```

    }
    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((title == null) ? 0 : title.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        BasicEntity other = (BasicEntity) obj;
        if (title == null) {
            if (other.title != null)
                return false;
        } else if (!title.equals(other.title)) {
            return false;
        }
        return true;
    }
}

@Entity
public class Project extends BasicEntity {

}

@Entity
public class Task extends BasicEntity{

    private Project project;
    private int duration;

    @ManyToOne
    public Project getProject() {
        return project;
    }

    public void setProject(Project project) {
        this.project = project;
    }

    public int getDuration() {
        return duration;
    }

    public void setDuration(int duration) {
        this.duration = duration;
    }
}

```

Per testare la classe si può usare la stessa filosofia di Project, andando però a testare anche il progetto associato Project. Lasciamo che sia Eclipse a creare per noi i metodi equals e hashCode.

Andiamo quindi a creare la base dei nostri test come per Project.

```
public class TaskIntegrationTest {
    private static final String PERSISTENCE_UNIT_NAME = "mysql-pu";
    private static EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;
    private TypedQuery<Task> query;

    private Project project_another;
    private Task task1;
    private Task task2;

    private Project project_another;
    private Task task1;
    private Task task2;

    @BeforeClass
    public static void setUpClass() {
        entityManagerFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
    }

    @Before
    public void setUp() throws Exception {
        entityManager = entityManagerFactory.createEntityManager();

        // make sure to drop the Task table for testing
        entityManager.getTransaction().begin();
        entityManager.createNativeQuery("delete from Task").executeUpdate();
        entityManager.getTransaction().commit();

        // make sure to drop the Task table for testing
        entityManager.getTransaction().begin();
        entityManager.createNativeQuery("delete from Project").executeUpdate();
        entityManager.getTransaction().commit();

        project_another = new Project();
        project_another.setTitle("Another project");
        project_another.setDescription("Another exciting project!");
        entityManager.persist(project_another);

        task1 = new Task();
        task1.setTitle("First task");
        task1.setDescription("This is my first task, hi!");
        task1.setProject(project_another);
        task1.setDuration(30);

        task2 = new Task();
        task2.setTitle("Second task");
        task2.setDescription("This is my second task, wow!");
        task2.setDuration(20);
    }

    @After
    public void tearDown() {
        task1 = null;
        task2 = null;
    }
}
```

```

        entityManager.close();
    }

    @AfterClass
    public static void tearDownClass() {
        entityManagerFactory.close();
    }
}

```

Aggiungiamo quindi i test veri e propri, andando a creare i metodi utility per evitare di riscrivere codice duplicato:

```

private void transactionPersist(Task t) {
    entityManager.getTransaction().begin();
    entityManager.persist(t);
    entityManager.getTransaction().commit();
}

/**
 * Perform a simple query for all the Project entities
 */
private List<Task> getAllTasks() {
    entityManager.getTransaction().begin();
    query = entityManager.createQuery("select p from Task p", Task.class);
    entityManager.getTransaction().commit();
    return query.getResultList();
}

@Test
public void testBasicPersistence() {
    transactionPersist(task1);

    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();

    List<Task> projectsList = getAllTasks();

    // We should have only one task in the database
    assertTrue(projectsList.size() == 1);

    // We should have the same title
    assertEquals(task1.getTitle(), projectsList.get(0).getTitle());

    // and the same description
    assertEquals(task1.getDescription(),
projectsList.get(0).getDescription());

    // and the same associate project
    assertEquals(project_another, projectsList.get(0).getProject());

    // and the same duration
    assertEquals(task1.getDuration(), projectsList.get(0).getDuration());
}

@Test
public void testMultiplePersistence() {
    transactionPersist(task1);
    transactionPersist(task2);
}

```

```

        // Clear Hibernate's cache to make sure data is retrieved from the store
        entityManager.clear();

        List<Task> projectsList = getAllTasks();

        // We should have 2 tasks in the database
        assertTrue(projectsList.size() == 2);
    }

```

- Creazione Dao di Task e testing

Nello stesso modo, è possibile creare il DAO di Task, inserendo anche un metodo per ritrovare il progetto associato ad un determinato Task.

```

public Project getProjectByTaskId(Long id) {
    Task tempTask = entityManager.find(Task.class, id);
    if (tempTask != null) {
        return tempTask.getProject();
    }
    return null;
}

```

Nuovamente è possibile testare con dei test di integrazione il corretto funzionamento del Dao utilizzando la stessa filosofia di Project, ma andando a controllare anche il progetto associato.

Creiamo quindi un metodo per fare il setUp della classe test:

```

private static final String PERSISTENCE_UNIT_NAME = "mysql-pu";
private static EntityManagerFactory entityManagerFactory;
private EntityManager entityManager;
private TaskDao taskDao;

private Project project_another;
private Task task1;
private Task task2;

@BeforeClass
public static void setUpClass() {
    entityManagerFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
}

@Before
public void setUp() throws Exception {
    entityManager = entityManagerFactory.createEntityManager();

    // make sure to drop the Task table for testing
    entityManager.getTransaction().begin();
    entityManager.createNativeQuery("delete from Task").executeUpdate();
    entityManager.getTransaction().commit();

    project_another = new Project();
    project_another.setTitle("Another project");
    project_another.setDescription("Another exciting project!");
    entityManager.persist(project_another);

    task1 = new Task();
    task1.setTitle("First task");
}

```



```

task1.setDescription("This is my first task, hi!");
task1.setDuration(20);
task1.setProject(project_another);

task2 = new Task();
task2.setTitle("Second task");
task2.setDescription("This is my second task, wow!");
task2.setDuration(30);
task1.setProject(project_another);

taskDao = new TaskDao(entityManager);
}

```

E andiamo a liberare le risorse:

```

@After
public void tearDown() {
    task1 = null;
    task2 = null;
    entityManager.close();
}

@AfterClass
public static void tearDownClass() {
    entityManagerFactory.close();
}

```

Adesso andiamo a svolgere tutti i test necessari:

```

@Test
public void testSave() {
    taskDao.save(task1);

    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();

    assertEquals(task1, entityManager.createQuery("from Task where id =:id",
Task.class).setParameter("id", task1.getId()).getSingleResult());
}

@Test
public void testEmptyGetAll() {
    assertTrue(taskDao.getAll().size() == 0);
}

@Test
public void testOneGetAll() {
    taskDao.save(task1);

    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();

    assertEquals(task1, taskDao.getAll().get(0));
    assertTrue(taskDao.getAll().size() == 1);
}

@Test
public void testMultipleGetAll() {
    taskDao.save(task1);
    taskDao.save(task2);

    // Clear Hibernate's cache to make sure data is retrieved from the store
    entityManager.clear();
}

```

```

        assertTrue(taskDao.getAll().size() == 2);
    }

    @Test
    public void testEmptyFindById() {
        assertNull(taskDao.findById((long) -1));
    }

    @Test
    public void testNotEmptyFindById() {
        taskDao.save(task1);

        // Clear Hibernate's cache to make sure data is retrieved from the store
        entityManager.clear();

        assertEquals(task1, taskDao.findById(task1.getId()));
    }

    @Test
    public void testUpdate() {
        taskDao.save(task1);
        task1.setDescription("new description!");
        taskDao.update(task1);

        // Clear Hibernate's cache to make sure data is retrieved from the store
        entityManager.clear();

        assertEquals(task1.getDescription(), taskDao.findById(task1.getId()).getDescription());
    }

    @Test
    public void testGetProjectByTaskIdWithNonExistingId() {
        assertNull(taskDao.getProjectByTaskId((long) -1));
    }

    @Test
    public void testGetProjectByTaskIdWithExistingId() {
        taskDao.save(task1);

        // Clear Hibernate's cache to make sure data is retrieved from the store
        entityManager.clear();

        assertEquals(task1.getProject(), taskDao.getProjectByTaskId(task1.getId()));
    }
}

```

Ulteriore database Relazionale (PostgreSQL)

Per aggiungere un secondo database relazionale è necessario inserire innanzitutto la dipendenza nel file pom.xml.

```

<!-- JDBC (Java DataBase Connectivity) PostgreSQL -->
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.3-1102-jdbc41</version>
</dependency>

```

Successivamente è necessario creare una seconda unità di persistenza nei file persistence.xml

```
<persistence-unit name="postgresql-pu" transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <!-- The provider specifies the JPA implementation that you want to use
      (Hibernate) -->

  <class>com.garritano.keepchronos.model.Project</class>
  <class>com.garritano.keepchronos.model.Task</class>

  <properties>
    <property name="javax.persistence.jdbc.driver"
      value="org.postgresql.Driver" />
    <property name="javax.persistence.jdbc.url"
      value="jdbc:postgresql://localhost:5432/timekeep?
createDatabaseIfNotExist=true" />
    <!-- the address with port of your postgresql service -->

    <property name="javax.persistence.jdbc.user"
      value="postgres" />
    <property name="javax.persistence.jdbc.password" value="" />
    <!-- username and password of your postgresql connection -->

    <property name="hibernate.dialect"
      value="org.hibernate.dialect.PostgreSQL95Dialect" />
    <!-- The dialect specifies the language of your DB (PostgreSQL) -->
    <property name="hibernate.hbm2ddl.auto" value="create" />
    <!-- Possible values for hbm2ddl: validate: validate the schema, makes
        no changes to the database. update: update the schema.create:
        creates the schema, destroying previous data. create-drop: drop the
        schema when the SessionFactory is closed explicitly, typically when
        the application is stopped. -->

    <property name="hibernate.show_sql" value="false" />
    <!-- show_sql=true permits to view queries in console -->

    <property name="hibernate.format_sql" value="false" />
    <!-- format_sql=true permits to view formatted queries in console -->
  </properties>
</persistence-unit>
```

Anche in questo caso è necessario inserire il quale implementazione di Jpa utilizzare (Hibernate), fare una lista delle classi da persistere, inserire quale driver jdbc usare, inserire l'url di accesso, le credenziali, il dialetto che verrà utilizzato, impostiamo la configurazione di Hibernate in modo che cancelli e ricrei sempre il database ogni volta che si avvia l'applicazione e mettiamo a false la configurazione che ci permette di vedere quali query vengono eseguite da Hibernate sul database.

Adesso aggiungiamo il container PostgreSQL sul file docker-compose.yml.

```
services:
  postgres:
    image: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: ""
      POSTGRES_DB: timekeep
      PGDATA: /data/postgres
    volumes:
      - /data/postgres:/data/postgres
    ports:
      - "5432:5432"
    networks:
      - postgres

networks:
  postgres:
    driver: bridge
```

E possiamo creare una classe di test per testare se l'elemento Project viene persistito.

```
public class ProjectPostgresqlIntegrationTest {

    private static final String PERSISTENCE_UNIT_NAME = "postgresql-pu";
    private static EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;
    private TypedQuery<Project> query;

    private Project project1;
    private Project project2;

    @BeforeClass
    public static void setUpClass() {
        entityManagerFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
    }

    @Before
    public void setUp() throws Exception {
        entityManager = entityManagerFactory.createEntityManager();

        // make sure to drop the Project table for testing
        entityManager.getTransaction().begin();
        entityManager.createNativeQuery("delete from Project").executeUpdate();
        entityManager.getTransaction().commit();

        project1 = new Project();
        project1.setTitle("First project");
        project1.setDescription("This is my first project, hi!");

        project2 = new Project();
        project2.setTitle("Second project");
        project2.setDescription("This is my second project, wow!");
    }

    private void transactionPersist(Project p) {
        entityManager.getTransaction().begin();
```

```

        entityManager.persist(p);
        entityManager.getTransaction().commit();
    }

    /**
     * Perform a simple query for all the Project entities
     */
    private List<Project> getAllProjects() {
        //
        entityManager.getTransaction().begin();
        query = entityManager.createQuery("select p from Project p",
Project.class);
        entityManager.getTransaction().commit();
        return query.getResultList();
    }

    @Test
    public void testBasicPersistence() {
        transactionPersist(project1);

        // Clear Hibernate's cache to make sure data is retrieved from the store
        entityManager.clear();

        List<Project> projectsList = getAllProjects();

        // We should have only one project in the database
        assertTrue(projectsList.size() == 1);

        // We should have the same title
        assertEquals(project1.getTitle(), projectsList.get(0).getTitle());

        // and the same description
        assertEquals(project1.getDescription(),
projectsList.get(0).getDescription());
    }

    @After
    public void tearDown() {
        project1 = null;
        project2 = null;
        entityManager.close();
    }

    @AfterClass
    public static void tearDownClass() {
        entityManagerFactory.close();
    }
}

```

Database Non Relazionale (Infinispan)

- Configurazione Maven

Inserire tra i tag `<dependencyManagement>` e `</dependencyManagement>` le seguenti dipendenze:

```
<dependencies>
  <dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-bom</artifactId>
    <version>5.3.1.Final</version>
    <type>pom</type>
    <scope>import</scope>
  </dependency>
</dependencies>
```

Inserire tra i tag `<dependencies>` e `</dependencies>` le seguenti dipendenze:

```
  <dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-core</artifactId>
    <scope>compile</scope>
  </dependency>
<dependency>
  <groupId>org.hibernate.javax.persistence</groupId>
  <artifactId>hibernate-jpa-2.1-api</artifactId>
</dependency>
  <dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-infinispan-embedded</artifactId>
  </dependency>
  <dependency>
    <groupId>org.jboss.narayana.jta</groupId>
    <artifactId>narayana-jta</artifactId>
    <version>5.5.23.Final</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-orm</artifactId>
  </dependency>
```

Non è necessario inserire le versioni delle dipendenze, dato che hibernate-ogm-bom si occuperà in automatico di utilizzare la giusta versione. Nel caso della dipendenza narayana-jta però, è necessario fare l'override della versione a causa della presenza di un bug.

- Configurazione Persistence.xml

Adesso è necessario modificare il file persistence.xml per creare aggiungere una nuova unità di persistenza in questo modo:

```
<persistence-unit name="infinispan-pu" transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>
  <!-- The provider specifies the JPA implementation that you want to use
       (Hibernate OGM) -->

  <class>com.garritano.keepchronos.model.TaskNoSQL</class>
  <class>com.garritano.keepchronos.model.ProjectNoSQL</class>
  <properties>
```

```

        <property name="hibernate.ogm.datastore.provider"
value="infinispan_embedded"/>
        <!-- datastore provider you want to use -->
    </properties>
</persistence-unit>







```

Come si vede, è necessario inserire il nome dell'unità di persistenza, ad esempio *infinispan-pu* e indicare quali sono le classi da persistere.

- Entity non relazionale

Dato che non è consigliato utilizzare relazioni ManyToOne, ho optato per inserire l'oggetto Project direttamente nella tabella Task.

L'oggetto Project non sarà più una Entity, ma avrà la notazione `@Embeddable`. Con la notazione `@Embeddable` non verranno più rappresentate 2 tabelle distinte, ma verrà tutto mappato nella tabella Task, come in figura.

task			
	id	integer	
	title	varchar(255)	
	duration	int	
	id_project	integer	
	title_project	integer	

L'annotazione `@Indexed` contrassegna Task come entità che deve essere indicizzata da Hibernate Search, necessario per effettuare le query sul Database.

```

@Embeddable
@Indexed
public class ProjectNoSQL extends BaseEntity{
    private Long id;

    @Override
    @GeneratedValue
    public Long getId() {
        return id;
    }
}

```

È necessario fare un override del metodo `getId` per evitare che esso acquisisca la notazione `@Id` di `BasicEntity`, in quanto `ProjectNoSQL` non avrà una chiave primaria.

La classe `TaskNoSQL` è invece molto simile alla classe `Task`, come possiamo vedere.

```

@Entity
@Indexed
public class TaskNoSQL extends BaseEntity{

    private ProjectNoSQL project;
    private int duration;

    @Embedded
    public ProjectNoSQL getProject() {
        return project;
    }

    public void setProject(ProjectNoSQL project) {

```

```

        this.project = project;
    }

    public int getDuration() {
        return duration;
    }

    public void setDuration(int duration) {
        this.duration = duration;
    }
}

```

Per far capire a Jpa come inserire l'elemento ProjectNoSQL senza che venga utilizzata una nuova tabella, si usa l'annotazione `@Embedded`.

Creiamo adesso il test per TaskNoSQL:

```

public class TaskNoSQLIntegrationTest {
    private static final String PERSISTENCE_UNIT_NAME = "infinispan-pu";
    private static EntityManagerFactory entityManagerFactory;

    @BeforeClass
    public static void setUpClass() {
        entityManagerFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);

        //accessing JBoss's Transaction can be done differently but this one works nicely
        transactionManager = com.arjuna.ats.jta.TransactionManager.transactionManager();
    }

    @AfterClass
    public static void tearDownClass() {
        entityManagerFactory.close();
    }
}

```

A differenza della versione relazionale, in cui il `transactionManager` veniva fornito dall'`entityManager`, adesso è necessario utilizzare quello fornito dalla libreria Jboss.

Creiamo un metodo `setUp()` per effettuare le operazioni di cancellazione, creazioni e settaggio di ProjectNoSQL e TaskNoSQL. E un metodo per liberare le risorse alla fine di ogni test.

```

@Before
public void setUp() throws Exception {
    entityManager = entityManagerFactory.createEntityManager();

    // make sure to have the TaskNoSQL table empty
    transactionManager.begin();
    query = entityManager.createQuery("select t from TaskNoSQL t");
    List<TaskNoSQL> allTasks = query.getResultList();
    for (TaskNoSQL element: allTasks) {
        entityManager.remove(element);
    }
    transactionManager.commit();

    transactionManager.begin();
    query = entityManager.createQuery("select p from TaskNoSQL p");
    assertTrue(query.getResultList().size() == 0);
    transactionManager.commit();

    // get a new EM to make sure data is actually retrieved from the store and
    not Hibernate's internal cache
}

```



```

entityManager.close();
entityManager = entityManagerFactory.createEntityManager();

project_another = new ProjectNoSQL();
project_another.setTitle("Another project");

task1 = new TaskNoSQL();
task1.setTitle("First task");
task1.setProject(project_another);
task1.setDuration(30);

task2 = new TaskNoSQL();
task2.setTitle("Second task");
task2.setDuration(20);
}

@After
public void tearDown() {
    task1 = null;
    task2 = null;
    entityManager.close();
}

```

Adesso andiamo ad effettuare i test veri e propri per controllare che i dati vengano correttamente persistiti.

```

@Test
public void testBasicPersistence() throws Exception {

    transactionManager.begin();
    entityManager.persist(task1);
    transactionManager.commit();

    // Perform a simple query for all the Task entities
    query = entityManager.createQuery("select p from TaskNoSQL p");

    // We should have only one task in the database
    assertTrue(query.getResultList().size() == 1);

    // We should have the same title
    assertTrue(((TaskNoSQL) query.getSingleResult())
        .getTitle()
        .equals(task1.getTitle()));

    // and the same id
    assertTrue(((TaskNoSQL) query.getSingleResult()).getId() ==
(task1.getId()));

    //and the same duration
    assertTrue(((TaskNoSQL) query.getSingleResult()).getDuration() ==
(task1.getDuration()));

    //and the same associate project
    assertTrue(((TaskNoSQL) query.getSingleResult()).getProject().equals(project_another));
}

```

- Creazione Dao di Task e testing

Come per TaskDao relazionale, TaskNoSQLDao offre un'interfaccia per la gestione della tabella TaskNoSQL.

L'unica differenza è la gestione del *transactionManager* che deve essere inizializzato e non è fornito dall'*entityManager*. Inoltre è necessario eseguire il metodo begin() e commit() rispettivamente prima e dopo la chiamata necessaria a svolgere azioni sul database.

```
public class TaskNoSQLDao {

    private EntityManager entityManager;
    private TransactionManager transactionManager;

    public TaskNoSQLDao(EntityManager entityManager) {
        this.entityManager = entityManager;
        //accessing JBoss's Transaction can be done differently but this one works
        nicely
        transactionManager =
com.arjuna.ats.jta.TransactionManager.transactionManager();
    }

    public void save(TaskNoSQL task) throws Exception{
        transactionManager.begin();
        entityManager.persist(task);
        transactionManager.commit();
    }

    public List<TaskNoSQL> getAll() throws Exception{
        transactionManager.begin();
        List<TaskNoSQL> result = entityManager.createQuery("select p from
TaskNoSQL p", TaskNoSQL.class).getResultList();
        transactionManager.commit();

        return result;
    }

    public TaskNoSQL findById(Long id) {
        return entityManager.find(TaskNoSQL.class, id);
    }

    public void update(TaskNoSQL task) throws Exception{
        transactionManager.begin();
        entityManager.merge(task);
        transactionManager.commit();
    }

    public ProjectNoSQL getProjectByTaskId(Long id) {
        TaskNoSQL tempTask = entityManager.find(TaskNoSQL.class, id);
        if (tempTask!=null) {
            return tempTask.getProject();
        }
        return null;
    }
}
```

La chiamata find() non ha la necessità che venga circondato da begin() e commit() perché è un'operazione che non influisce sui dati.

E adesso testiamo tutte le funzioni. Aggiungiamo quindi il metodo `setUpClass()` e liberiamo le risorse nel metodo `tearDownClass()`, inizializziamo l' `entityManagerFactory` e il `transactionManager`.

```
public class TaskNoSQLDaoIntegrationTest {

    private static final String PERSISTENCE_UNIT_NAME = "infinispan-pu";
    private static EntityManagerFactory entityManagerFactory;

    @BeforeClass
    public static void setUpClass() {
        entityManagerFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);

        //accessing JBoss's Transaction can be done differently but this one works nicely
        transactionManager = com.arjuna.ats.jta.TransactionManager.transactionManager();
    }

    @AfterClass
    public static void tearDownClass() {
        entityManagerFactory.close();
    }

}
```

Aggiungiamo quindi il metodo `setUp` per avere la tabella `TaskNoSQL` vuota e 2 oggetti `TaskNoSQL` e un oggetto `ProjectNoSQL` necessari per i test.

```
@Before
    public void setUp() throws Exception {
        entityManager = entityManagerFactory.createEntityManager();

        // make sure to have the TaskNoSQL table empty
        transactionManager.begin();
        query = entityManager.createQuery("select t from TaskNoSQL t");
        List<TaskNoSQL> allTasks = query.getResultList();
        for (TaskNoSQL element: allTasks) {
            entityManager.remove(element);
        }
        transactionManager.commit();

        transactionManager.begin();
        query = entityManager.createQuery("select p from TaskNoSQL p");
        assertTrue(query.getResultList().size() == 0);
        transactionManager.commit();

        // get a new EM to make sure data is actually retrieved from the store and
not Hibernate's internal cache
        entityManager.close();
        entityManager = entityManagerFactory.createEntityManager();

        project_another = new ProjectNoSQL();
        project_another.setTitle("Another project");

        task1 = new TaskNoSQL();
        task1.setTitle("First task");
        task1.setDuration(30);

        task2 = new TaskNoSQL();
        task2.setTitle("Second task");
        task2.setDuration(20);
    }
```

```

        taskDao = new TaskNoSQLDao(entityManager);
    }

```

Quindi andiamo a testare i singoli metodi di TaskNoSQLDao

```

@Test
    public void testSave() throws Exception{
        taskDao.save(task1);

        assertEquals(task1, entityManager.createQuery("from TaskNoSQL where id
=:id", TaskNoSQL.class).setParameter("id", task1.getId()).getSingleResult());
    }

@Test
    public void testEmptyGetAll() throws Exception{
        assertTrue(taskDao.getAll().size() == 0);
    }

@Test
    public void testOneGetAll() throws Exception{
        taskDao.save(task1);

        assertEquals(task1, taskDao.getAll().get(0));
        assertTrue(taskDao.getAll().size() == 1);
    }

@Test
    public void testMultipleGetAll() throws Exception{
        taskDao.save(task1);
        taskDao.save(task2);

        assertTrue(taskDao.getAll().size() == 2);
    }

@Test
    public void testEmptyFindById() {
        assertNull(taskDao.findById((long) -1));
    }

@Test
    public void testNotEmptyFindById() throws Exception{
        taskDao.save(task1);

        assertEquals(task1, taskDao.findById(task1.getId()));
    }

@Test
    public void testUpdate() throws Exception{
        taskDao.save(task1);
        task1.setDescription("new description!");
        taskDao.update(task1);

        assertEquals(task1.getDescription(),
taskDao.findById(task1.getId()).getDescription());
    }

@Test
    public void testGetProjectByTaskIdWithNonExistingId() {
        assertNull(taskDao.getProjectByTaskId((long) -1));
    }

```

```
@Test
public void testGetProjectByTaskIdWithExistingId() throws Exception{
    taskDao.save(task1);
    assertEquals(task1.getProject(),
taskDao.getProjectByTaskId(task1.getId()));
}
```