**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.

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.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteLa libreria che utilizziamo è HIBERNATE che si basa sull’approccio JPA.

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamenteGli ORM danno la possibilità di mappare le nostre classi Java alle tabelle dei DB: definisco una classa 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.

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente

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.

Immagine che contiene testo, schermata, documento, Carattere

Descrizione generata automaticamenteConfigurare il progetto: passare password, port ecc…

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

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

1. Immagine che contiene testo, Carattere, schermata

   Descrizione generata automaticamenteNel persistence.xml (in META.INF) :

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Immagine che contiene testo, Software multimediale, schermata

Descrizione generata automaticamente

<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.2" 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\_2.xsd%22%3E

<persistence-unit name="u4d12">

<properties>

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

<property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>

<property name="hibernate.default\_schema" value="public"/>

<property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>

<property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/d12"/>

<property name="javax.persistence.jdbc.user" value="postgres"/>

<property name="javax.persistence.jdbc.password" value="1234"/>

</properties>

</persistence-unit>

</persistence>

1. Andiamo in PROPERTIES, Clicchiamo su Add e aggiungiamo le proprietà e i valori che andranno a configurare la connessione di Java al DB
2. Possiamo creare la nostra Class main (eventualmente col Logger)

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

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

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

Immagine che contiene testo, Carattere, numero, schermata

Descrizione generata automaticamente

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

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

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamenteNon 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

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

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.

Immagine che contiene testo, software, Software multimediale, Sistema operativo

Descrizione generata automaticamenteLa persistence-unit serve per raggruppare tutte le informazioni relative al nostro DB. Molto importante il name 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)

Immagine che contiene testo, software, Software multimediale, Software per la grafica

Descrizione generata automaticamente

Nel main:

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamenteL’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.

Immagine che contiene testo, schermata, Software multimediale, software

Descrizione generata automaticamente

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

Salvare un oggetto:

Immagine che contiene testo, schermata, software, schermo

Descrizione generata automaticamente

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,

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamenteImmagine che contiene testo, schermata, schermo, multimediale

Descrizione generata automaticamentecioè nel commit().

Immagine che contiene testo, schermata, schermo, software

Descrizione generata automaticamenteSd 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 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.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteNB 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.

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

Nella classe DAO:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente.find() vuole due parametri: la classe dove agire e l’id. .find() è come se facesse:

select \* from students where id=id

Nel main:



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.

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

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.

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

JAVA-JPA CONNESSIONE AL DATABASE

1-Creare un progetto maven o jpa in eclipse (new project -> maven project o jpa project per avere gia un template pronto -> nel caso abbiamo scelto il jpa, impostare il nome e lasciare tutto in default -> in jpa implementation selezionare "disable library configuration" e, in Persistent class management, spuntare "discover annotated classes automatically" -> confermare)

2-Il file persistence.xml è dove andremo a configurare il tutto

3-Cliccare su "Configur" e convertire in maven project o in jpa (La conversione è in base al tipo di progetto scelto),questo perchè avremo bisogno delle funzionalità di entrambi i progetti.

4-Avremo bisogno delle librerie "Hibernate" + "PostgreSql JDBC Driver" da mvnRepositery che andranno inserite in un blocco "dependencies" nel file "pom":

di seguito il codice da copiare e incollare nel pom.xml

<dependencies>

<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>42.6.0</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>5.6.15.Final</version>

</dependency>

<!-- https://mvnrepository.com/artifact/ch.qos.logback/logback-classic -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.4.8</version>

<scope>test</scope>

</dependency>

</dependencies>

5-Settiamo le proprietà del database nel file persistence.xml cliccando nella casella "Properties" e inserendo(vedi codice): driver,url, user, password + dialetto da utilizzare + opzioni per creare automaticamente tabelle(inserire nella colonna "name": hibernate.hbm2ddl.auto -> e nella colonna value "update")

6-Creare un main e le varie classi che useremo

7-Una volta configurato il tutto dobbiamo mappare le nostre classi tramite le varie annotazioni.

8-Creare una classe, ad esempio "Student" assegnandogli attributi, ad esempio id,name e username, e creiamo il costruttore, in questo caso l'id sarà una chiave primaria, quindi il valore non glielo passeremo noi, ma faremo un classico get e set per il resto degli attributi.

Importiamo l'annotazione @Entity //serve per dire che verrà gestita da jpa, quindi ogni classe a cui verrà assegnata questa annotazione, verrà importata nel database, l'annotazione @Table(name="Students")// serve a definire un nome custom alla nostra tabella, e @Id //serve per definire la chiave primaria(in questo caso l'id)

9-Creiamo una classe JpaUtil(Questa è una classe standard che si farà cosi in ogni progetto) dove creeremo un EntityManagerFactory(oggetto per manipolare gli oggetti).L'EntityManagerFactory è necessario per poter manipolare gli oggetti.Si porta dietro metodi già predefiniti.

10-Nel main andiamo a richiamare l'EntityManagerFactory.

11-Creiamo un database, nel caso non abbiamo il database dichiarato nell'url nel persistence.xm, andremo a crearlo e questo creerà automaticamente le classi(quelle a cui stata assegnata l'annotazaione @Entity) e le tabelle che abbiamo inserito in java.

12-è buona norma creare anche una classe DAO(ad esempio:"StudentsDao")per ogni classe che creo, dovrò creare una classe DAO.Le classi DAO raccolgono i metodi di cui abbiamo bisogno, la creo per separare i metodi e non inseririrli tutti nel main in modo da avere un codice pulito e per richiamarli dove serve. è simile ad un interfaccia. Dobbiamo passare al costruttore un riferimento all'entity manager, in modo da averlo già settato.Non ha senso creargli un set perchè l'entity manager ci servirà per tutti i metodi.

Esempio:

public class StudentsDAO {

private final EntityManager em;

public StudentsDao(EntityManager em){

this.em = em;

}

}

COME SALVARE UN NUOVO OGGETTO

Bisogna far partire una transazione : l'EntityTransaction, all'interno di un metodo save() nel DAO.

Esempio:

public void save(Student s){

EntityTransaction t = em.getTransaction

t.begin();//inizia la transizione

//in mezzo ai due andrò a fare la manipolazione dell'oggetto

t.persist()s // l'oggetto viene aggiunto al persistent content, non viene ancora salvato nel db, ma vien salvato quendo termino la transizione

t.commit();//termina la transizione

System.out.println("Studente salvato correttamente")

}

Dopodichè andremo a creare un nuovo oggetto (in questo caso uno studente) nel main e richiamando il metodo save() andremo a salvarlo nel database.

ricordarsi di chiudere l'entity manager e factory, una volta finite le transizioni di cui necessitiamo:

em.close()

emf.close()

ANNOTAZIONE PER FAR GENERARE SISTEMATICAMENTE L'ID AL DB

se dovessimo provare a salvare 2 volte un oggetto senza specificargli l'ID,questo ci tornerà un errore, ecco perche usiamo l'annotazione:

@Genereted value

Questa annotazione ci consentirà di generare un nuovo ID ad ogni salvataggio oggetto in modo da avere un id univoquo

METODO PER CERCARE UN OGGETTO

public Student findById(UUID id){

student found = em.find(Student.class, id);

return found;

}

Una volta richiamato il metodo findById() nel main, dobbiamo passargli l'UUID dell'oggetto che vogliamo cercare. Lo troviamo andando a leggere la tabella nel DB in pgAdmin4.

Dobbiamo anche richiamre il metodo toString

è necessario fare l'@Ovveride al to string,nella classe apposita, per farci restituire una stringa.

Vado a richiamarlo nel main:

Student aldoFromDB = sd.finById(UUID.fromString("codice univoco UUID");

syso(aldoFromDB)

METODO PER CANCELLARE UN OGGETTO

public void findByIdandDelete(UUID id) {

//1-faccio una find prima, per otterenere lo studente

Student found = em.find(Student.class,id);

if(found != null) {

//2-elimino lo studente

//2.1-ottengo la transizione

EntityTransaction t = em.getTransaction();

//2.2 faccio partire la transizione

t.begin();

//2.3 Rimuovo l'oggetto

em.remove(found)

//2.4 Faccio il commit della transizione

t.commit();

SystemOut.out.println("Studente eliminato correttamente")

}

}

}

Vado a richiamarlo nel main:

sd.findByIdAndDelete(UUID.fromString)("Codice univoco UUID"));

METODO REFRESH (metodo per ripristinare un oggetto prima della modifica, quindi viene richiamato dopo il commit)