

Java EE – JPA

- JPA via Hibernate ORM
 - Gestione della persistenza in Java (usando JDBC per i RDBMS)
- Progetto di riferimento
 - <https://github.com/egalli64/jed>
 - Java SE 11, Java EE
 - Hibernate
 - Tomcat 9
 - JDBC per Oracle 18c o MySQL 8 in lib
 - In context.xml, DataSource JDBC come Resource

ORM

- Object Relational Mapping
- Integrazione tra due paradigmi
 - Object Oriented
 - Relazionale
- Alcuni problemi
 - Diverso approccio per
 - **Identità**: PK (*database*), == (*reference*), equals() (*uguaglianza tra oggetti*)
 - **Associazione**: FK vs has-a
 - **Navigazione** nei dati: JOIN vs reference
 - Tabelle e oggetti possono definire entità con diversa **granularità** (es.: indirizzo)
 - Come gestire **l'ereditarietà** in un RDBMS?

JPA

- Java Persistency API
 - Versione corrente 2.2
- Implementazioni basate su specifiche Oracle
 - Red Hat JBoss **Hibernate**
 - EclipseLink
 - ...
- JPQL: simile a SQL ma relativo a classi Java, dialetti per le implementazioni
- Nata come soluzione ORM più leggera rispetto a quella offerta da EJB
- Può essere usata in Java SE e EE

Hibernate

- Framework ORM – Your relational data. Objectively.
 - <https://hibernate.org/orm/>
 - Nato nel 2001 (Gavin King et al.) come alternativa più semplice a EJB
 - Dal ~2010 implementa JPA, wrappando l'architettura nativa
 - La versione stabile corrente è la 5.4 (Java 8/11, JPA 2.2)
- Mappa classi Java (JavaBean) e tabelle di database
 - File di configurazione XML
 - Annotazioni
- Definizione di un linguaggio simile a SQL: HQL

Entity

- Java Bean, POJO (Plain Old Java Object) annotato
- **@Entity**
 - Per default fa riferimento a una tabella con lo stesso nome
 - **@Table** name, nel caso l'entity abbia un nome diverso
- Una proprietà deve essere annotata come chiave con **@Id**
 - Riferimento alla PK
 - **@GeneratedValue** per generazione automatica dei valori (Identity → Identity / Autoincrement)
- Le proprietà sono mappate automaticamente a colonne della tabella
 - Per default si assume che proprietà e colonne abbiano lo stesso nome
 - **@Column** name, nel caso la proprietà abbia un nome diverso
- Eventuali proprietà non persistenti vanno annotate **@Transient**

Hibernate nativo

- Configurazione → SessionFactory
 - Programmatica, via Configuration e Service Registry
 - O via file di configurazione
 - hibernate.cfg.xml o hibernate.properties
- Esempio: lettura di tutte le entity Coder nel database
 - SessionFactory → Session
 - Gestione della connessione al database via JDBC
 - Query, via Session.createQuery(), passando come parametri
 - HQL ("SELECT c FROM Coder c")
 - Class dell'entity di riferimento
 - Estrazione della lista di entity, via Query.list()

Configurazione di JPA

- File persistence.xml, nel folder META-INF, in src/main/resources
- Definisce le persistency unit usate nell'app, ognuna con un nome univoco
 - Elemento persistence-unit, attributo name
- All'interno di persistence-unit si definisce il data source
 - Per Tomcat
 - non-jta-data-source, ex: java:comp/env/jdbc/me
 - Properties
 - hibernate.dialect
 - org.hibernate.dialect.MySQLDialect
 - org.hibernate.dialect.Oracle12cDialect // per 12 e successivi, Oracle10gDialect per 10 e 11
 - ...

Hibernate JPA

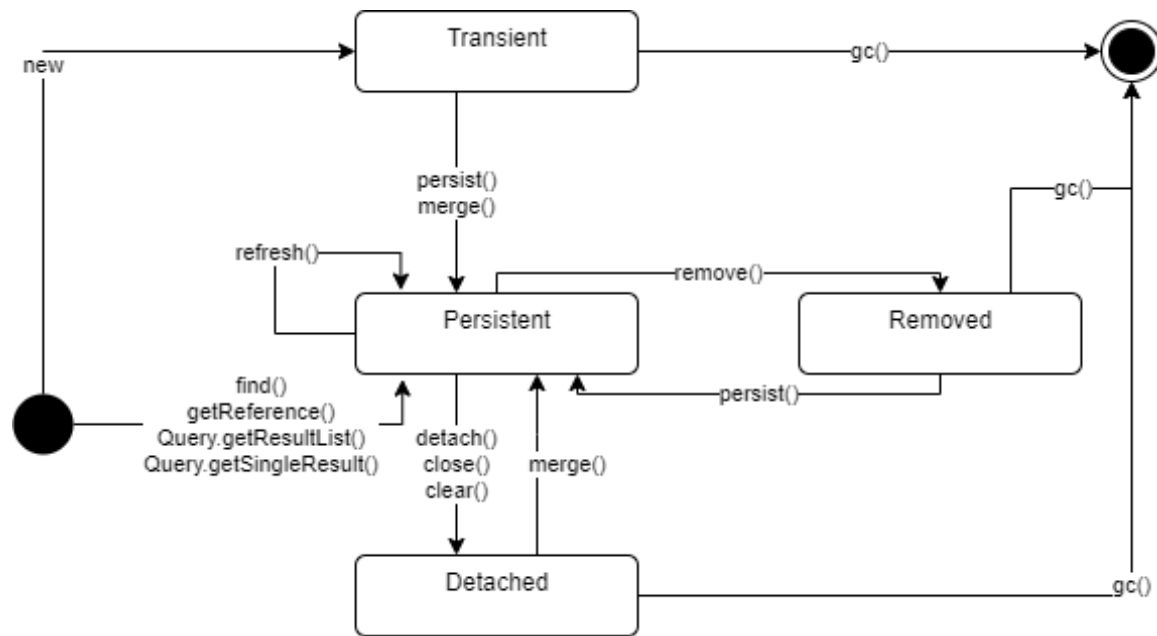
- EntityManagerFactory, da JPA Persistence.createEntityManagerFactory()
 - Il parametro è il nome della persistence unit come definito in configurazione
 - Un container full può gestirla direttamente, Tomcat *deve* gestirla via Persistence
- EntityManager, dal factory, via createEntityManager()
 - In Hibernate, è un wrapper di Session, accessibile via EntityManager.unwrap()
- TypedQuery, da EntityManager.createQuery(), passando come parametri
 - JPQL ("SELECT c FROM Coder c")
 - Class dell'entity di riferimento
- Lista di entity, via TypedQuery.getResultList()

EntityTransaction

- Il supporto alle transazioni da JPA è limitato
 - Andrebbe usato JTA, che però è disponibile in JEE full container o via terze parti
- Transazione relativa ad un EntityManager
 - `getTransaction()`
- Va esplicitamente aperta e chiusa
 - `begin()`
 - `commit()` / `rollback()`
 - Prima della chiusura dell'entity manager
- Le operazioni DML devono essere eseguite in una transazione



Diagramma di stato di un Entity



close() e clear() rendono tutte le entità persistenti detached

- Transient
 - non associata al DB
- Persistent
 - Rappresentata nel DB
 - Modifiche → update
- Detached
 - non più associata
- Removed
 - Non più persistente

CRUD via EntityManager

- **find()**
 - Data classe e id dell'entità, ritorna l'entità richiesta (persistente), o null
- **getReference()**
 - Simile a find, ma implementa il design pattern proxy
 - La select avviene al primo tentativo di accedere una proprietà
 - *L'entità deve essere ancora persistente*
 - `PersistenceUnitUtil.isLoaded()` verifica se l'entità è in memoria
 - `Hibernate.initialize()` helper per assicurare l'inizializzazione
- **refresh()**
 - Sincronizza una entità persistente in memoria leggendo dal database
 - Eccezione se nel frattempo i dati sono stati rimossi

CRUD via EntityManager

- Basati su comandi DML
 - Devono essere eseguiti in una transazione
- **persist()**
 - Prende come parametro una entità e la rende persistente
- **merge()**
 - Aggiorna l'entità sul database, se esiste l'id, altrimenti ne crea una nuova
- **remove()**
 - Prende come parametro una entità persistente e la elimina dal database

JPQL

- Java Persistence Query Language
- Simile a SQL ma basato sulle entità JPA
- Istruzioni viste come stringhe
- Il processo è organizzato in tre passi
 - Creazione della query
 - Preparazione, quando richiesta (sostituzione parametri, ...)
 - Esecuzione e ottenimento del risultato



JPQL Query

- Creazione di una query da uno statement JPQL
 - `Query EntityManager.createQuery(jpql)`
 - `TypedQuery<Entity> EntityManager.createQuery(jpql, Entity.class)`
- Preparazione di query parametrizzate
 - `Query.setParameter(pos/name, value)`
 - Posizionali (?1, ?2, ...) o con nome (:xyz)
- Esecuzione di select
 - `List<Entity> Query.getResultList()`
 - `Entity Query.getSingleResult()`
 - Tira una `NoResultException` nel caso la query non ritorni un risultato
- Esecuzione di update e delete
 - `int Query.executeUpdate()`

Generated value

- La creazione dell'id di una entità può delegata al DBMS
 - @GeneratedValue
 - Implica che l'utente non indicherà mai l'id
 - org.hibernate.PersistentObjectException: detached entity passed to persist
- Diverse strategie sono disponibili tra cui
 - Tecniche di auto incremento
 - GenerationType.IDENTITY
 - Supporto per mezzo di una sequenza
 - GenerationType.SEQUENCE
 - Richiede di indicare la sequenza via @SequenceGenerator

Disponibile in MySQL e Oracle 12+

Non disponibile in MySQL

Relazioni tra entità

- Definite per mezzo di annotazioni
 - **@OneToOne**
 - **@OneToMany**, **@ManyToOne**
 - **@ManyToMany**
- L'entità principale specifica la relazione con l'altra entità via
 - **@JoinColumn** e, nel caso molti a molti, **@JoinTable**
 - Attributi sull'annotazione Join... aggiungono dettagli, ad esempio
 - **name** specifica il nome della colonna (o tabella) SQL di riferimento



@OneToOne

- Esempio: relazione one to one tra Coder e Team
- L'entità Team ha una proprietà **leader** di tipo Coder
 - **@OneToOne(optional = false)**
 - Ogni team ha necessariamente un leader
 - **@JoinColumn(name="leader_id")**
 - Legame della FK: Teams.leader_id → Coders.coder_id
- L'entità Coder ha una proprietà di tipo Team
 - **@OneToOne(optional=true, mappedBy="leader")**
 - Un coder non è necessariamente un team leader
 - “leader” è il nome della proprietà di Team che mappa il coder

@OneToMany @ManyToOne

- Esempio: relazione many to one tra Country e Region
 - L'entità Country ha una proprietà **region** di tipo Region
 - **@ManyToOne @JoinColumn(name="region_id")**
- Relazione one to many tra Region e Country
 - L'entità Region ha una proprietà Set<Country>
 - **@OneToMany(mappedBy="region")**
 - La select è by default “lazy”, non vengono lette le entità associate
 - Hibernate.isInitialized() permette di verificare se un proxy è utilizzabile o meno
 - Comportamento “eager”
 - **@OneToMany(mappedBy="region", fetch=FetchType.EAGER)** – **da usare con attenzione**
 - **Preferito** l'uso di **SELECT DISTINCT ... JOIN FETCH** (quando necessario)

@ManyToMany

- Esempio: relazione many to many tra Coder e Team
 - Simmetrica, scegliamo noi il master → Team
- L'entità Team ha una proprietà **coders** di tipo Set<Coder> annotata
 - **@ManyToMany**
 - **@JoinTable**(
 - name = "TEAM_CODER",
 - joinColumns = **@JoinColumn**(name = "TEAM_ID"),
 - inverseJoinColumns = **@JoinColumn**(name = "CODER_ID"))
- L'entità Coder ha una proprietà teams di tipo Set<Team> annotata
 - **@ManyToMany**(mappedBy = "**coders**")
- Valgono le stesse considerazioni lazy-eager indicate per la OneToMany