**JPA: La base**

1. **Cos'è JPA?**  
   JPA è un'API standard di Java per gestire la persistenza degli oggetti in un database relazionale.  
   Permette di:
   * Mappare le classi Java a tabelle del database.
   * Gestire la persistenza con annotazioni come @Entity, @Table, @Column, @Id, ecc.
2. **Concetti principali di JPA:**
   * Entity: Mappa una classe a una tabella del database.
   * Table e Column: Specificano come mappare la classe e i suoi attributi nelle tabelle e colonne del database.
   * Id e GeneratedValue: Definiscono la chiave primaria e la sua strategia di generazione (es. AUTO, IDENTITY).

L'EntityManager è una delle principali interfacce di JPA e viene utilizzato per gestire la persistenza delle entità (cioè degli oggetti Java che rappresentano i dati). È l'oggetto centrale per interagire con il database quando si lavora con JPA. Attraverso l'EntityManager, puoi eseguire operazioni come:

* **Persistenza: Salvare, aggiornare e rimuovere oggetti dal database.**
* **Recupero: Recuperare oggetti dal database tramite query o identificatore.**
* **Gestione del contesto di persistenza: Gestire la sessione in cui vengono effettuate le operazioni di persistenza.**

EntityManager em = entityManagerFactory.createEntityManager();

em.getTransaction().**begin();**

em.persist(libreria); // Aggiunge l'oggetto libreria al contesto di persistenza

em.getTransaction().commit();

**import** javax.persistence.\*;

**import** java.util.List;

@Entity

@Table(name = "libri")

**public** **class** **Libro** {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** Long id;

@Column(name = "titolo", nullable = false)

**private** String titolo;

@Column(name = "prezzo", nullable = false)

**private** Double prezzo;

// Costruttori, getters e setters

**public** Libro() {}

**public** Libro(String titolo, Double prezzo) {

**this**.titolo = titolo;

**this**.prezzo = prezzo;

}

**public** Long getId() {

**return** id;

}

**public** void setId(Long id) {

**this**.id = id;

}

**public** String getTitolo() {

**return** titolo;

}

**public** void setTitolo(String titolo) {

**this**.titolo = titolo;

}

**public** Double getPrezzo() {

**return** prezzo;

}

**public** void setPrezzo(Double prezzo) {

**this**.prezzo = prezzo;

}

}

**class** **LibroManager** {

**private** EntityManagerFactory entityManagerFactory;

**private** EntityManager entityManager;

**public** LibroManager() {

// Crea un EntityManagerFactory usando il persistence unit definito nel persistence.xml

**this**.entityManagerFactory = Persistence.createEntityManagerFactory("nomePersistenceUnit");

}

// Metodo per aggiungere un libro

**public** void aggiungiLibro(Libro libro) {

entityManager = entityManagerFactory.createEntityManager(); // Crea un EntityManager

EntityTransaction transaction = entityManager.getTransaction(); // Ottieni la transazione

**try** {

transaction.begin(); // Inizia la transazione

entityManager.persist(libro); // Persiste il libro

transaction.commit(); // Conferma la transazione

System.**out**.println("Libro aggiunto con successo.");

} **catch** (RuntimeException e) {

**if** (transaction.isActive()) {

transaction.rollback(); // Annulla la transazione in caso di errore

}

e.printStackTrace();

} **finally** {

**if** (entityManager != **null** && entityManager.isOpen()) {

entityManager.close(); // Chiudi l'EntityManager

}

}

}

// Metodo per recuperare tutti i libri

**public** List<Libro> recuperaTuttiLibri() {

entityManager = entityManagerFactory.createEntityManager();

**try** {

// Recupera tutti i libri dal database

TypedQuery<Libro> query = entityManager.createQuery("SELECT l FROM Libro l", Libro.**class**);

**return** query.getResultList();

} **catch** (RuntimeException e) {

e.printStackTrace();

**return** **null**;

} **finally** {

**if** (entityManager != **null** && entityManager.isOpen()) {

entityManager.close(); // Chiudi l'EntityManager

}

}

}

// Metodo per aggiornare il prezzo di un libro

**public** void aggiornaPrezzo(Long libroId, Double nuovoPrezzo) {

entityManager = entityManagerFactory.createEntityManager();

EntityTransaction transaction = entityManager.getTransaction();

**try** {

transaction.begin();

Libro libro = entityManager.find(Libro.**class**, libroId); // Trova il libro per ID

**if** (libro != **null**) {

libro.setPrezzo(nuovoPrezzo); // Aggiorna il prezzo

entityManager.merge(libro); // Unisci la modifica

}

transaction.commit();

System.**out**.println("Prezzo aggiornato con successo.");

} **catch** (RuntimeException e) {

**if** (transaction.isActive()) {

transaction.rollback();

}

e.printStackTrace();

} **finally** {

**if** (entityManager != **null** && entityManager.isOpen()) {

entityManager.close();

}

}

}

// Metodo per rimuovere un libro

**public** void rimuoviLibro(Long libroId) {

entityManager = entityManagerFactory.createEntityManager();

EntityTransaction transaction = entityManager.getTransaction();

**try** {

transaction.begin();

Libro libro = entityManager.find(Libro.**class**, libroId); // Trova il libro per ID

**if** (libro != **null**) {

entityManager.remove(libro); // Rimuovi il libro

}

transaction.commit();

System.**out**.println("Libro rimosso con successo.");

} **catch** (RuntimeException e) {

**if** (transaction.isActive()) {

transaction.rollback();

}

e.printStackTrace();

} **finally** {

**if** (entityManager != **null** && entityManager.isOpen()) {

entityManager.close();

}

}

}

// Metodo per chiudere la connessione con il database

**public** void chiudiConnessione() {

**if** (entityManagerFactory != **null** && entityManagerFactory.isOpen()) {

entityManagerFactory.close(); // Chiudi l'EntityManagerFactory

System.**out**.println("Connessione chiusa.");

}

}

}

**public** **class** **Main** {

**public** static void main(String[] args) {

// Creazione del manager per la gestione dei libri

LibroManager libroManager = new LibroManager();

// Aggiungere un nuovo libro

Libro nuovoLibro = new Libro("Java Programming", 29.99);

libroManager.aggiungiLibro(nuovoLibro);

// Recuperare tutti i libri

List<Libro> libri = libroManager.recuperaTuttiLibri();

**if** (libri != **null**) {

**for** (Libro libro : libri) {

System.**out**.println("Titolo: " + libro.getTitolo() + ", Prezzo: " + libro.getPrezzo());

}

}

// Aggiornare il prezzo di un libro

libroManager.aggiornaPrezzo(1L, 24.99);

// Rimuovere un libro

libroManager.rimuoviLibro(1L);

// Chiudere la connessione

libroManager.chiudiConnessione();

}

}

**Introduzione: MyBatis e Hibernate**

**MyBatis** e **Hibernate** sono strumenti per lavorare con i database in applicazioni Java.

Entrambi consentono di eseguire query sul database, ma differiscono nel modo in cui operano:

* **MyBatis**: offre un controllo dettagliato delle query, dove ogni query deve essere definita esplicitamente tramite file XML o annotazioni.

È utile quando si ha bisogno di scrivere query SQL personalizzate e ottimizzare le prestazioni.

* **Hibernate**: è un framework ORM (Object-Relational Mapping) che mappa automaticamente le classi Java in tabelle del database (e viceversa), semplificando la gestione dei dati senza bisogno di scrivere query SQL complesse.

È utile per progetti in cui la gestione automatica delle entità è un requisito primario.

**Quando scegliere MyBatis vs Hibernate?**

* **MyBatis**: è preferibile quando hai bisogno di un controllo completo sulle query SQL, per ottimizzare le prestazioni o per query particolarmente complesse che non possono essere facilmente rappresentate tramite un ORM.
* **Hibernate**: è ideale quando si vuole ridurre la complessità del codice SQL, sfruttando le funzionalità ORM per la gestione automatica delle entità, relazioni e transazioni.

**Cos'è Hibernate?**

Hibernate è un framework ORM che implementa l’API JPA, facilita la mappatura delle tabelle di un database relazionale in oggetti Java (e viceversa).

**Hibernate può creare automaticamente le tabelle nel database** se configurato con hibernate.hbm2ddl.auto.

Una volta che le tabelle sono pronte (sia create automaticamente da Hibernate che manualmente), possiamo:

* Creare classi Java che rappresentano quelle tabelle (note come **entità** in Hibernate).
* Utilizzare annotazioni per mappare queste classi alle tabelle, definendo relazioni e proprietà.
* Eseguire le operazioni CRUD (Create, Read, Update, Delete) senza scrivere query SQL complesse.

**Configurazione di Hibernate in un progetto Java**

Per utilizzare Hibernate, è necessario configurare correttamente il progetto, includendo le **dipendenze** necessarie e il file di configurazione.

In un progetto **Maven**, aggiungiamo le dipendenze nel file *pom.xml.*

Per configurare Hibernate, è necessario creare un file *hibernate.cfg.xml* che definisce le informazioni di connessione al database e altre impostazioni.

**Mapping in Hibernate**

Il **mapping** è il processo di collegamento tra una classe Java e una tabella del database.

Ogni classe in Hibernate rappresenta una tabella del database e prende il nome di **entità**.

**Annotazioni principali**:

* @Entity: definisce una classe come entità.
* @Id: specifica che un attributo è una chiave primaria.
* @GeneratedValue: indica la strategia per generare valori nella colonna della chiave primaria (es. incremento automatico).
  + Una delle strategie comuni è GenerationType.IDENTITY, che utilizza il comportamento autoincrementale del database.
* @Column: permette di definire i dettagli di una colonna (es. nome, lunghezza, unicità).
* @Transient: esclude un attributo dal mapping (non rappresenta una colonna del database).

Es: passwordConferma

**Relazioni tra Entità**

Hibernate supporta diverse tipologie di relazioni tra le entità, che vengono mappate tramite annotazioni specifiche:

1. One-to-Many (Uno a Molti): Un'entità ha una relazione con molte altre entità. Esempio: Un autore può avere molti libri.
2. **Many-to-One** (Molti a Uno):

Molte entità sono associate a una sola entità.

Esempio: Più libri appartengono a un solo autore.

1. **One-to-One**

Un'entità è associata a una sola altra entità. Esempio: Ogni utente ha un profilo.

1. **Many-to-Many (Molti a Molti)**:

Due entità sono collegate tramite una tabella intermedia (bridge table). Esempio: Gli studenti possono iscriversi a più corsi, e ogni corso può avere più studenti.

**Annotazioni Utilizzate:**

* **@OneToMany** e **@ManyToOne**: utilizzate per rappresentare relazioni uno-a-molti e molti-a-uno.
* **@OneToOne**: utilizzata per relazioni uno-a-uno.
* **@ManyToMany**: utilizzata per relazioni molti-a-molti.
* **@JoinColumn**: specifica il nome della colonna utilizzata come **chiave esterna**.
* **mappedBy**: indica quale attributo, nella classe collegata, gestisce la relazione.

**il JoinColum specifica che name = autore\_id sia una chiave esterna**

Il mappedBy utilizzato in Autore evita che venga creata una chiave esterna anche in quella tabella in cui la sua chiave interna

**Esempio di relazione Uno-a-Molti:**

Se abbiamo una classe Autore e una classe Libro, la relazione potrebbe essere definita così:

|  |
| --- |
| @Entity  **public** **class** Autore {     @Id     @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;  ​     @OneToMany(mappedBy = "autore")  **private** List<Libro> libri;  }  ​  @Entity  **public** **class** Libro {     @Id     @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;  ​     @ManyToOne     @JoinColumn(name = "autore\_id")  **private** Autore autore;  }  ​ |

**Gestione delle Operazioni in Hibernate**

* **Aggiornamento e Cancellazione:**

Per aggiornare o cancellare un record, Hibernate richiede di caricare prima l'entità dal database

( Autore autore = session.get(Autore.class, 1L); session.delete(autore)).

Non possiamo passare direttamente il valore della chiave primaria.

* + Questo meccanismo permette a Hibernate di tenere traccia delle relazioni e di sincronizzare i cambiamenti.
* **Lazy Loading e Eager Loading:**

Hibernate gestisce il caricamento dei dati in modo lazy (caricamento ritardato) o eager (caricamento immediato), influenzando le prestazioni:

* + Lazy Loading: carica i dati solo quando necessari (potrebbe causare query N+1).
  + Eager Loading: carica i dati immediatamente, riducendo il numero di query.

**Metodi Utili in Hibernate**

* **Session.get():** Recupera un'entità dal database in base alla sua chiave primaria**.**
* **Session.find():** Simile a get(), ma può includere funzionalità aggiuntive come la gestione dei record non trovati.
* **flush():** Forza la sincronizzazione dei dati nel database**.**
* **clear():** Rimuove tutte le entità dalla sessione**.**
* **merge():** Unifica un'entità staccata con la sessione corrente.
* **Criteria API:** Consente di **costruire query dinamiche** in modo programmatico, **evitando di scrivere SQL direttamente.**

**Conclusione**

Hibernate semplifica notevolmente la gestione del database in applicazioni Java, **eliminando gran parte della necessità di scrivere query SQL manuali**.

Utilizzando le annotazioni e il mapping delle entità, è possibile creare relazioni complesse tra tabelle in modo più naturale e orientato agli oggetti.

**Vantaggi principali di Hibernate:**

* Automazione nella gestione delle entità e delle relazioni.
* Supporto per transazioni, caching e gestione delle sessioni.
* Minor quantità di codice SQL da scrivere, grazie all'uso delle annotazioni e dei metodi predefiniti di Hibernate.

1. **Cos'è Hibernate e perché viene utilizzato?**
   * Quali sono i principali vantaggi di Hibernate rispetto a JDBC?
   * Che cos'è un ORM e come Hibernate si inserisce in questo contesto?
2. **Cos'è un SessionFactory in Hibernate?**
   * Come si configura e qual è il suo ruolo?
   * In che modo differisce dalla Session?
3. **Cos'è una Session in Hibernate?**
   * Qual è il ciclo di vita di una Session?
   * Perché è necessario chiudere una Session dopo l'uso?

**Configurazione**

1. **Come si configura Hibernate in un progetto?**
   * Qual è il ruolo del file hibernate.cfg.xml?
   * Quali proprietà minime sono necessarie per configurare Hibernate?
2. **Cos'è un file di mapping in Hibernate?**
   * Quando è necessario utilizzarlo e cosa contiene?
   * Qual è la differenza tra il mapping XML e le annotazioni?
3. **Come si integrano Hibernate e un database?**
   * Come si specifica il dialect del database?
   * Quali driver sono necessari per connettersi a MySQL o altri database?

**Mapping delle entità**

1. **Cos'è un'entità in Hibernate?**
   * Qual è il ruolo dell'annotazione @Entity?
   * Cosa succede se un'entità non ha un ID?
2. **Come si definisce una chiave primaria in Hibernate?**
   * Qual è la differenza tra @Id e @GeneratedValue(strategy = GenerationType.IDENTITY)?
   * Cosa sono le altre strategie di generazione dell'ID, come AUTO o SEQUENCE?
3. **Come si mappano le colonne di un'entità?**
   * Qual è il ruolo dell'annotazione @Column?
   * Come si mappano attributi opzionali o esclusi dal database (es. @Transient)?
4. **Cos'è l'annotazione @Table?**
   * Quando è necessario utilizzarla?
   * Cosa succede se non la includi in un'entità?

**Relazioni tra entità**

1. **Quali sono i tipi di relazioni supportate in Hibernate?**
   * Come si implementano relazioni OneToOne, OneToMany, ManyToOne e ManyToMany?
   * Qual è il ruolo di @JoinColumn e @MappedBy?
2. **Cos'è il FetchType in Hibernate?**
   * Qual è la differenza tra LAZY ed EAGER fetching?
   * Quali sono i vantaggi e gli svantaggi di ciascun approccio?
3. **Come si gestiscono relazioni bidirezionali?**
   * Come evitare cicli infiniti durante la serializzazione (es. con JSON)?
   * Qual è il ruolo del mappedBy nelle relazioni bidirezionali?

**Operazioni CRUD**

1. **Come si salvano e si aggiornano i dati in Hibernate?**
   * Qual è la differenza tra save() e persist()?
   * Quando si utilizza merge() e come funziona?
2. **Come si rimuove un'entità in Hibernate?**
   * Qual è la differenza tra delete() e remove()?
   * Come si assicura che un'entità esista prima di eliminarla?
3. **Come si recuperano i dati in Hibernate?**
   * Qual è la differenza tra get() e load()?
   * Quando è preferibile utilizzare una query HQL rispetto a get()?

**Query e HQL**

1. **Cos'è HQL (Hibernate Query Language)?**
   * Qual è la differenza tra HQL e SQL?
   * Come si scrive una query HQL per recuperare tutte le entità di una tabella?
2. **Come funzionano le named queries in Hibernate?**
   * Come si definiscono le named queries in XML e con annotazioni?
   * Quali sono i vantaggi di utilizzare named queries?
3. **Cos'è un CriteriaQuery?**
   * Come si costruiscono query dinamiche utilizzando l'API dei criteri?
   * Qual è il vantaggio di Criteria rispetto a HQL?

**Caching**

1. **Come funziona il caching in Hibernate?**
   * Qual è la differenza tra cache di primo livello e di secondo livello?
   * Quali provider di cache di secondo livello supporta Hibernate (es. Ehcache)?
2. **Cos'è la cache di primo livello?**
   * Come funziona e quali dati vengono memorizzati?
   * È possibile disabilitarla per una sessione specifica?
3. **Cos'è la cache di secondo livello?**
   * Come si configura un provider di cache in Hibernate?
   * Qual è il vantaggio principale di utilizzare la cache di secondo livello?

**Transazioni**

1. **Come si gestiscono le transazioni in Hibernate?**
   * Qual è il ruolo del metodo beginTransaction()?
   * Cosa succede se non si commette o si fa rollback di una transazione?
2. **Qual è la differenza tra commit e rollback?**
   * Quando è necessario utilizzare rollback()?
   * Cosa succede se un'eccezione viene lanciata durante una transazione?

**Avanzato**

1. **Cos'è un Interceptor in Hibernate?**
   * Come si utilizza per aggiungere funzionalità personalizzate a livello di Session?
   * Quali sono gli scenari tipici di utilizzo degli Interceptor?
2. **Come funzionano le relazioni ereditate in Hibernate?**
   * Quali strategie di mapping sono disponibili per classi ereditate (TABLE\_PER\_CLASS, SINGLE\_TABLE, JOINED)?
   * Quali sono i vantaggi e gli svantaggi di ciascuna strategia?
3. **Come si esegue la validazione delle entità in Hibernate?**
   * Qual è il ruolo di Hibernate Validator?
   * Come si utilizzano annotazioni come @NotNull o @Size?
4. **Quali sono le principali differenze tra Hibernate e JPA?**
   * Perché si potrebbe scegliere di utilizzare Hibernate anziché solo JPA?
   * Quali funzionalità offre Hibernate che non sono presenti in JPA?