Capisco che questa parte del corso possa sembrare un po' complessa, soprattutto se sei più abituato alla programmazione Java di base. Cercherò di fare chiarezza su tutte le tue domande, scomponendo i concetti uno per uno, e cercando di darti una visione chiara del processo che ti viene richiesto di seguire. Procediamo per ordine:

**Cos'è un Framework?**

Un **framework** è una sorta di "scaffale" di sviluppo, che fornisce un insieme di strumenti e librerie già pronte per facilitare la realizzazione di applicazioni.

In pratica, invece di partire da zero, un framework ti mette a disposizione una base strutturata e delle convenzioni da seguire. Tu devi semplicemente completare o adattare ciò che è già stato creato, seguendo delle linee guida.

Esempi di framework:

* **Spring** per il backend Java.
* **Angular** per il frontend.
* **Django** per Python.

Un framework può riguardare **diversi ambiti**: gestione delle risorse, autenticazione, interazione con il database, routing, ecc.

**Cos'è un Framework ORM?**

Un **framework ORM** (Object-Relational Mapping) è un tipo di framework che semplifica l'interazione tra il codice dell'applicazione e il database. Invece di scrivere manualmente SQL per ogni operazione (inserimento, aggiornamento, lettura, eliminazione), puoi utilizzare oggetti del linguaggio di programmazione, e l'ORM si occupa di tradurre automaticamente questi oggetti nelle operazioni SQL da eseguire sul database.

Esempi di **framework ORM**:

* **Hibernate** (per Java): mappa le classi Java alle tabelle del database.
* **MyBatis** (per Java): è simile a Hibernate, ma ti dà un maggiore controllo sulle query SQL.

**Cos'è MyBatis?**

**MyBatis** è un framework ORM, ma **a differenza di Hibernate**, non effettua una completa automazione del mapping (come ad esempio il mapping tra classi e tabelle del database). Invece, in MyBatis:

* Definisci tu le query SQL (in file XML o nel codice Java).
* MyBatis si occupa di eseguire queste query e di mappare i risultati in oggetti Java.

Con **MyBatis**, non devi scrivere codice SQL dentro il tuo codice Java, ma puoi definire le query in file XML separati e fare in modo che MyBatis gestisca l'esecuzione delle query. In pratica, MyBatis ti offre **maggiore flessibilità** rispetto ad altri ORM (come Hibernate), ma richiede che tu scriva manualmente più dettagli.

Un esempio di configurazione di **MyBatis** potrebbe essere:

|  |
| --- |
| <select id="getSitoById" resultType="Sito">    SELECT \* FROM sito WHERE id = #{id}  </select>  ​ |

**Cos'è un Mapper in MyBatis?**

In MyBatis, un **mapper** è un'interfaccia o una classe che definisce i metodi per l'accesso ai dati nel database. In pratica, un **mapper** contiene i metodi che corrispondono alle operazioni CRUD (Create, Read, Update, Delete) che vuoi eseguire. Ogni metodo di un mapper è associato a una query SQL (che può essere definita in un file XML o in annotazioni nel codice).

Esempio di **SitoMapper**:

|  |
| --- |
| public **interface** SitoMapper {     Sito getSitoById(int id);  // Metodo che ottiene un sito tramite id     void insertSito(Sito sito);  // Metodo che inserisce un sito nel database  }  ​ |

In pratica, il mapper:

* È **un'interfaccia** con metodi che corrispondono a operazioni sul database.
* Il framework MyBatis poi si occupa di eseguire le query e restituire i risultati in oggetti Java.

Quando scrivi:

java

Copia codice

SitoMapper mapper = sqlMapFactory.instance().getMapper(SitoMapper.class);

Stai ottenendo un'istanza di un **mapper** che MyBatis può utilizzare per eseguire operazioni sul database. La **sqlMapFactory.instance()** è il meccanismo tramite cui MyBatis crea e restituisce un'istanza del **mapper** definito (come SitoMapper.class).

**XML in MyBatis**

Hai detto che "con XML parliamo di un linguaggio per inserire dati". Ecco il punto: in MyBatis, **XML** viene utilizzato per definire le **query SQL**. Non è un linguaggio per "inserire dati" in modo generico, ma è proprio dove definisci **le query** che poi vengono eseguite.

In XML, definisci query come <select>, <insert>, <update>, <delete>, che poi verranno mappate a metodi nel **mapper**.

Esempio di un file XML di configurazione:

|  |
| --- |
| <mapper namespace="com.example.SitoMapper">     <select id="getSitoById" resultType="Sito">        SELECT \* FROM sito WHERE id = #{id}     </select>  </mapper>  ​ |

Questa è la definizione di una query SELECT in un file XML. Il metodo getSitoById nel **SitoMapper** utilizzerà questa query per ottenere un Sito dal database.

**Connessione vs Transazione**

* **Connessione**: è il "canale" fisico che permette la comunicazione tra l'applicazione e il database. È un'istanza di connessione che ti consente di inviare query e ottenere risultati. Ad esempio, una connessione può essere aperta, eseguita una query, e poi chiusa.
* **Transazione**: è un'operazione che raggruppa più query in un'unica unità di lavoro. Può comprendere più operazioni (come **inserimenti, aggiornamenti o cancellazioni**), e garantisce che o tutte vengano eseguite con successo, oppure che nessuna venga eseguita se c'è un errore (in tal caso si effettua un **rollback**).

In pratica, una **connessione** è il mezzo per inviare le query, mentre una **transazione** è il processo che garantisce che un insieme di operazioni venga eseguito in modo atomico e consistente.

**Il file pom.xml e le Dipendenze**

In un progetto Maven, il file pom.xml è fondamentale per gestire le **dipendenze** e la configurazione del progetto. Qui puoi definire tutte le librerie di cui il progetto ha bisogno.

Esempio di una sezione dependencies nel pom.xml:

|  |
| --- |
| <dependencies>     <dependency>         <groupId>org.mybatis</groupId>         <artifactId>mybatis</artifactId>         <version>3.5.4</version>     </dependency>     <dependency>         <groupId>mysql</groupId>         <artifactId>mysql-connector-java</artifactId>         <version>8.0.23</version>     </dependency>  </dependencies>  ​ |

Queste dipendenze indicano che il progetto utilizza **MyBatis** per il mapping ORM e il **connettore MySQL** per la connessione al database.

**Creare un Progetto Maven con MyBatis**

Se non ti è chiaro come **iniziare un progetto** con MyBatis e Maven, il processo in generale è questo:

1. **Crea un progetto Maven**: Usa un IDE come IntelliJ IDEA o Eclipse, o crea la struttura del progetto manualmente, e definisci il file pom.xml con le dipendenze necessarie.
2. **Crea i pacchetti**: Ad esempio, un pacchetto per le **classi di modello** (model), uno per i **mapper** (mapper), uno per la **logica di business** (service), ecc.
3. **Configura pom.xml**: Aggiungi le dipendenze per MyBatis e il connettore al database. Esempio:

|  |
| --- |
| <dependencies>     <dependency>         <groupId>org.mybatis</groupId>         <artifactId>mybatis</artifactId>         <version>3.5.4</version>     </dependency>     <dependency>         <groupId>mysql</groupId>         <artifactId>mysql-connector-java</artifactId>         <version>8.0.23</version>     </dependency>  </dependencies>  ​ |

1. **Scrivi le classi di modello e mapper**: Crea le tue classi Java (come Sito) e gli **interfaccia** mapper che definiscono le operazioni sul database.
2. **Configura MyBatis**: Crea un file di configurazione MyBatis (mybatis-config.xml) e definisci il mapping delle query.
3. **Esegui il programma**: MyBatis si occuperà del resto, connettendosi al database, eseguendo le query e restituendo i risultati sotto forma di oggetti Java.

**Conclusioni**

Non preoccuparti, capisco che la transizione da Java base a un framework come MyBatis possa sembrare difficile. In breve:

* MyBatis è una tecnologia per interagire con il database senza dover scrivere SQL manualmente nel codice.
* Usa **mapper** per definire le operazioni di accesso ai dati.
* L'XML in MyBatis definisce le **query SQL**.
* Il file pom.xml gestisce le **dipendenze** e la configurazione del progetto.

Il termine **DAO** sta per **Data Access Object**. Non è solo un nome, ma rappresenta un pattern di design molto comune in programmazione, in particolare nella gestione dell'accesso ai dati da parte di un'applicazione.

**Cos'è un DAO?**

Un **DAO (Data Access Object)** è una **classe** o un **oggetto** che fornisce un'interfaccia astratta per interagire con una fonte di dati (come un database, un file system o un altro tipo di storage). Il DAO nasconde i dettagli di accesso ai dati, come la gestione della connessione al database e l'esecuzione di query SQL, offrendo invece metodi specifici che l'applicazione può usare per ottenere o manipolare i dati.

In altre parole, un DAO è una **facciata** che centralizza la logica di accesso ai dati, facendo da intermediario tra il tuo codice di business e il database (o altro sistema di archiviazione).

**Caratteristiche principali di un DAO:**

* **Astrazione dell'accesso ai dati**: il DAO fornisce un'interfaccia che nasconde la logica specifica di accesso ai dati (ad esempio, query SQL).
* **Indipendenza dal tipo di storage**: un DAO può essere configurato per interagire con vari tipi di dati, come database relazionali (MySQL, PostgreSQL), NoSQL (MongoDB), file o altri servizi di persistenza.
* **Separation of Concerns (Separazione delle preoccupazioni)**: il DAO separa la logica di accesso ai dati dalla logica di business, migliorando la manutenibilità e la testabilità del codice.

**Esempio di un DAO in Java (con MyBatis)**

Immagina di avere una tabella Sito in un database, e di voler creare un DAO che gestisca le operazioni di accesso ai dati per questa entità.

1. **Definisci un modello (POJO) per la tabella Sito**:

|  |
| --- |
| public **class** Sito {  **private** int id;  **private** String nome;  **private** String url;  ​     // getter e setter  }  ​ |

1. **Crea un'interfaccia DAO per Sito**:

|  |
| --- |
| public **interface** SitoDao {     Sito getSitoById(int id);         // Legge un sito dal database     void insertSito(Sito sito);       // Inserisce un sito nel database     void updateSito(Sito sito);       // Aggiorna un sito     void deleteSito(int id);          // Elimina un sito  }  ​ |

1. **Implementazione del DAO con MyBatis (oppure un'altra tecnologia ORM)**:

|  |
| --- |
| public **class** SitoDaoImpl **implements** SitoDao {  **private** SqlSessionFactory sqlSessionFactory;  ​     // Costruttore, setter per configurare SqlSessionFactory  ​     @Override  **public** Sito getSitoById(int id) {  **try** (SqlSession session = sqlSessionFactory.openSession()) {             SitoMapper mapper = session.getMapper(SitoMapper.class);  **return** mapper.getSitoById(id);        }    }  ​     @Override  **public** void insertSito(Sito sito) {  **try** (SqlSession session = sqlSessionFactory.openSession()) {             SitoMapper mapper = session.getMapper(SitoMapper.class);             mapper.insertSito(sito);             session.commit();        }    }  ​     @Override  **public** void updateSito(Sito sito) {  **try** (SqlSession session = sqlSessionFactory.openSession()) {             SitoMapper mapper = session.getMapper(SitoMapper.class);             mapper.updateSito(sito);             session.commit();        }    }  ​     @Override  **public** void deleteSito(int id) {  **try** (SqlSession session = sqlSessionFactory.openSession()) {             SitoMapper mapper = session.getMapper(SitoMapper.class);             mapper.deleteSito(id);             session.commit();        }    }  }  ​ |

1. **Il Mapper (nel caso di MyBatis)**: Il **mapper** è l'interfaccia che contiene i metodi definiti per le query SQL (come nel caso di SitoMapper).

|  |
| --- |
| public **interface** SitoMapper {     @Select("SELECT \* FROM sito WHERE id = #{id}")     Sito getSitoById(int id);  ​     @Insert("INSERT INTO sito (nome, url) VALUES (#{nome}, #{url})")     void insertSito(Sito sito);  ​     @Update("UPDATE sito SET nome = #{nome}, url = #{url} WHERE id = #{id}")     void updateSito(Sito sito);  ​     @Delete("DELETE FROM sito WHERE id = #{id}")     void deleteSito(int id);  }  ​ |

**Perché usare il DAO?**

1. **Separazione delle preoccupazioni**: Il DAO separa il codice che gestisce i dati dal resto dell'applicazione, migliorando la **manutenibilità** e la **leggibilità** del codice.
2. **Centralizzazione della logica di accesso ai dati**: Tutte le operazioni sul database (come le query SQL) sono centralizzate nel DAO, rendendo il codice più modulare. Se in futuro decidi di cambiare il database o la tecnologia di persistenza, devi solo aggiornare il DAO senza toccare il resto del codice dell'applicazione.
3. **Testabilità**: Se hai un DAO separato, è più facile **mockarlo** durante i test unitari, evitando di interagire con il database vero e proprio.

**Quindi, il DAO è solo un nome?**

No, **DAO** non è solo un nome. È un **pattern di design** molto utile per la gestione dell'accesso ai dati. È una convenzione utilizzata per organizzare meglio il codice e separare le operazioni di persistenza (accesso ai dati) dalla logica di business dell'applicazione.

Nel contesto di MyBatis (o di qualsiasi altro ORM o tecnologia di persistenza), un DAO è il componente che **interagisce con il database** attraverso query predefinite e le **mappa** in oggetti Java.

Quindi, se vedi qualcosa come:

|  |
| --- |
| SitoDao sitoDao = **new** SitoDaoImpl();  sitoDao.insertSito(**new** Sito());  ​ |

Stai utilizzando il **DAO** per **astrarre** le operazioni di accesso al database, mantenendo il tuo codice di business **pulito e separato** dalla logica di persistenza.