Modulo 4

Integration Tier

Hibernate

Presented by

Paolo Locorotondo

IBM Client Innovation Center - Italy

05/12/2023

IBM Client Innovation Center
Italy

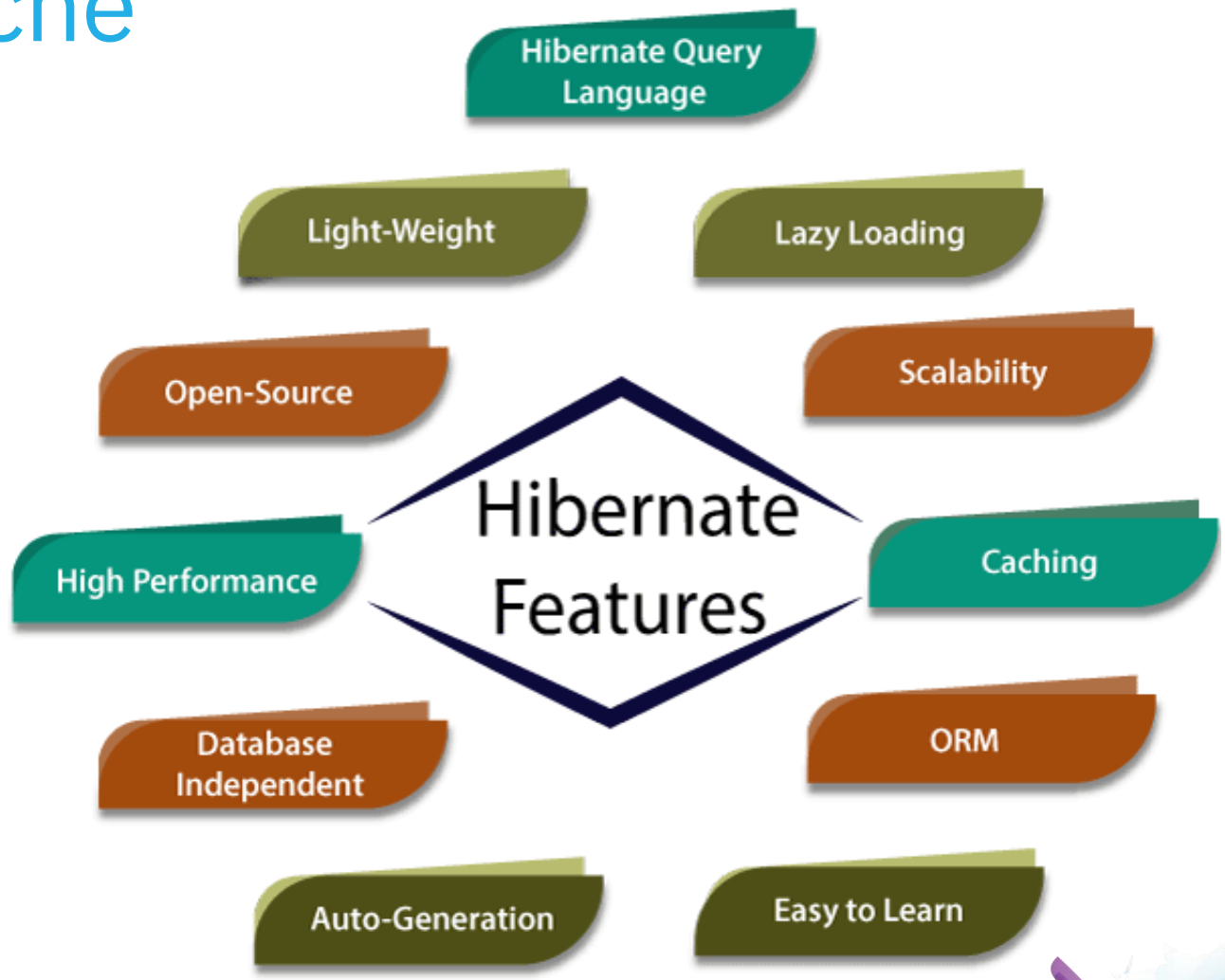
Introduzione

Framework per object-relational mapping

Favorisce l’integrazione di sistemi software che si basano sul paradigma della programmazione ad oggetti con sistemi per la gestione di basi dati relazionali

La versione più recente del framework consente di gestire anche database NoSQL

Caratteristiche



Caratteristiche

- **ORM**: aiuta nell'interazione tra le classi Java e il database relazionale
- **Open-Source**: Hibernate è un software open source, il che significa che è disponibile per tutti senza alcun costo. Il suo codice sorgente è reso disponibile in modo che chiunque possa utilizzarlo e sviluppare applicazioni
- **High Performance**: supporta diverse tecniche di fetch. Memorizzazione nella cache, Lazy Loading e molte altre per ottenere prestazioni elevate

Caratteristiche

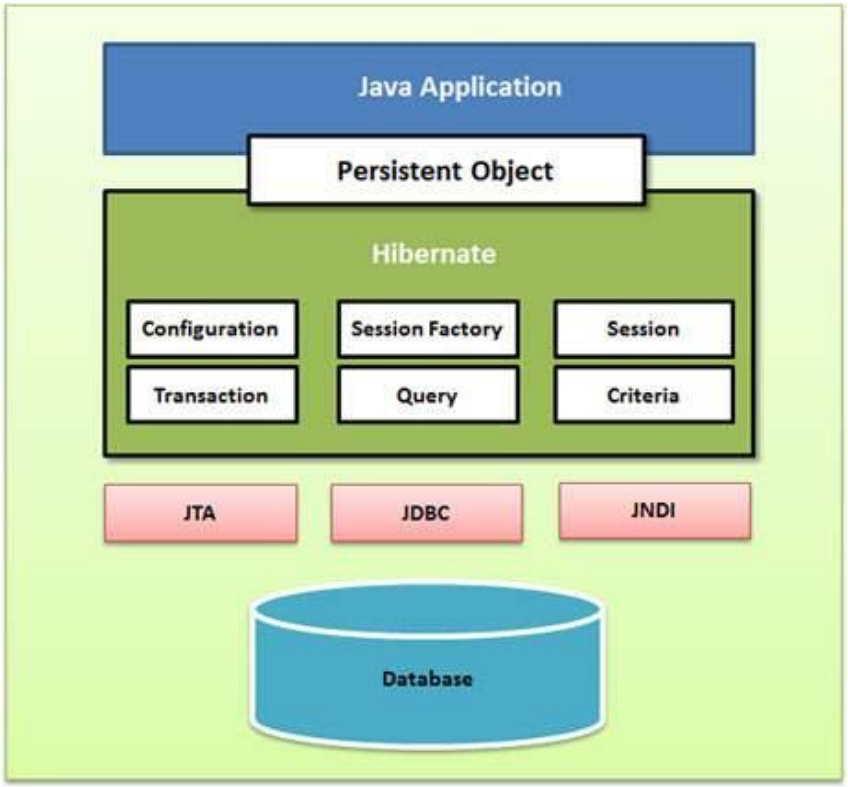
- **Lazy Loading**: recupera l'unico oggetto necessario per l'esecuzione, successivamente gli altri (solo se richiesti)
- **Hibernate Query Language**: linguaggio di query di Hibernate, indipendente dal database
- **Auto-Generation**: fornisce la possibilità di generare automaticamente le tabelle in base al mapping

Caratteristiche

- **Easy to learn**: facile da imparare e implementare
- **Scalability**: può essere utilizzato sia per applicazioni su piccola scala che su larga scala
- **Database Independent**: è indipendente dal database in quanto genera/traduce query in SQL utilizzando il database dialect specificato

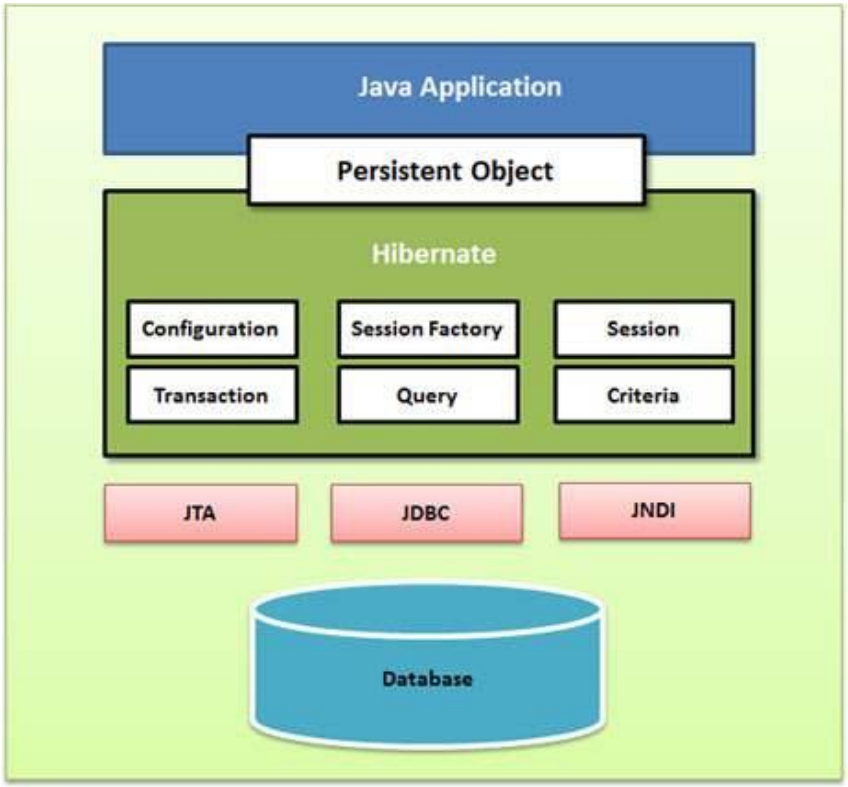
Architettura

- **Configuration**: creato durante l'inizializzazione dell'applicazione. Rappresenta un file di configurazione richiesto da Hibernate in cui sono presenti due componenti chiave:
  - Parametri connessione al database
  - Mapping classi Java e tabelle database
- **SessionFactory**: costruito tramite l'oggetto Configuration ed utilizzato per ottenere un oggetto Session. Creato all'avvio dell'applicazione e conservato per usi successivi



Architettura

- **Session**: connessione fisica con il database
- **Transaction**: unità di lavoro
- **Query**: usa SQL o HQL per recuperare/modificare dati. Un'istanza di Query viene utilizzata per impostare i parametri della query, limitare il numero di risultati restituiti dalla query ed eseguirla
- **Criteria**: query object-oriented per recuperare dati



Configurazione

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN« "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
        <session-factory>
                <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
                <property name="hibernate.connection.url">jdbc:mysql://localhost/persons</property>
                <property name="connection.username">root</property>
                <property name="connection.password">root</property>
                <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
                <property name=”hibernate.hbm2ddl.auto”>create</property>
                <property name="show_sql">true</property>
                <property name="format_sql">true</property>

                <!-- List of XML mapping files -->
                <mapping resource="Person.hbm.xml" />
        </session-factory>
</hibernate-configuration>
```

Auto-Generation

Sono supportate diverse strategie per la generazione di tabelle:

- **create**: elimina lo schema esistente, quindi lo ricrea in base ai mapping
- **update**: il modello creato in base ai mapping viene confrontato con lo schema esistente, quindi vengono applicate eventuali modifiche. Non vengono eliminate tabelle/colonne esistenti anche se non più richieste dall'applicazione
- **create-drop**: simile a create, ma lo schema viene eliminato alla chiusura del SessionFactory
- **validate**: verifica la correttezza dello schema, non applica modifiche
- **none**: disattiva la generazione ddl

Mapping

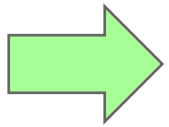
Sono disponibili due modalità per il mapping delle classi con il modello relazionale:

- File XML
- Annotazioni Java Persistence API (JPA)

Mapping XML

SQL

```
create table PERSON (
   ID INT NOT NULL auto_increment,
   FIRST_NAME VARCHAR(64),
   LAST_NAME VARCHAR(64),
   AGE INT,
   PRIMARY KEY (id)
)
```



XML

Person.hbm.xml

```
<?xml version = "1.0" encoding = "utf-8"?>
<hibernate-mapping>
   <class name="Person" table = "PERSON">
      <meta attribute = "class-description">
         This class contains the person detail.
      </meta>
      <id name="id" type = "int" column = “ID">
         <generator class=“native“/>
      </id>
      <property name = "firstName" column = "FIRST_NAME" type = "string"/>
      <property name = "lastName" column = "LAST_NAME" type = "string"/>
      <property name = "age" column = "AGE" type = "int"/>
   </class>
</hibernate-mapping>
```

Ogni classe da mappare avrà il proprio file XML

filename: *<classname>.hbm.xml*

Mapping relazioni

ONE-TO-ONE

Ogni istanza di una entità è legata ad una singola istanza di un’altra entità

```
<one-to-one name=“address" class=“Address" />
```

ONE-TO-MANY

Un’istanza di una entità può essere correlata a più istanze di altre entità

```
<set name = "documents" cascade="all">
    <key column = "person_id"/>
    <one-to-many class="Document"/>
</set>
```

Mapping relazioni

MANY-TO-ONE

Più istanze di una entità possono essere correlate ad una singola istanza dell'altra entità

```
<many-to-one name="person" column="person_id" class="Person" not-null="true"/>
```

MANY-TO-MANY

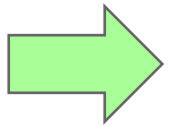
Più istanze di una entità possono essere correlate a più istanze di un'altra entità

```
<set name = "shifts" table="PERSON_WORKSHIFT">
    <key column = "person_id"/>
    <many-to-many column = "workshift_id" class="Workshift" />
</set>
```

Mapping con annotazioni JPA

SQL

```
create table PERSON (
   ID INT NOT NULL auto_increment,
   FIRST_NAME VARCHAR(64),
   LAST_NAME VARCHAR(64),
   AGE INT default NULL,
   PRIMARY KEY (id)
)
```



```
@Entity
@Table(name = "PERSON")
public class Person {

   public Person () {}

   @Id
   @Column(name = "ID")
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Integer id;

   @Column(name = "FIRST_NAME")
   private String firstName;

   @Column(name = "LAST_NAME")
   private String lastName;

   @Column
   private Integer age;

...
}
```

Metodi

L’interfaccia Session fornisce diversi metodi per effettuare operazioni di CRUD sugli oggetti mappati

#	Method name	Return	Description	Issued SQL statement
1	`beginTransaction()`	`Transaction`	Creates a Transaction object or returns an existing one, for working under context of a transaction.	
2	`getTransaction()`	`Transaction`	Returns the current transaction.	
3	`get(Class class, Serializable id)`	`Object`	Loads a persistent instance of the given class with the given id, into the session.	SELECT
4	`load(Class class, Serializable id)`	`Object`	Does same thing as `get()` method, but throws an `ObjectNotFound` error if no row with the given id exists.	SELECT
5	`persist(Object)`	`void`	saves a mapped object as a row in database	INSERT
6	`save(Object)`	`Serializable`	Does same thing as `persist()` method, plus returning a generated identifier.	INSERT
7	`update(Object)`	`void`	Updates a detached instance of the given object and the underlying row in database.	UPDATE
8	`saveOrUpdate(Object)`	`void`	Saves the given object if it does not exist, otherwise updates it.	INSERT or UPDATE
9	`delete(Object)`	`void`	Removes a persistent object and the underlying row in database.	DELETE
10	`close()`	`void`	Ends the current session.	
11	`flush()`	`void`	Flushes the current session. This method should be called before committing the transaction and closing the session.	
12	`disconnect()`	`void`	Disconnects the session from current JDBC connection.	

JPA vs Hibernate

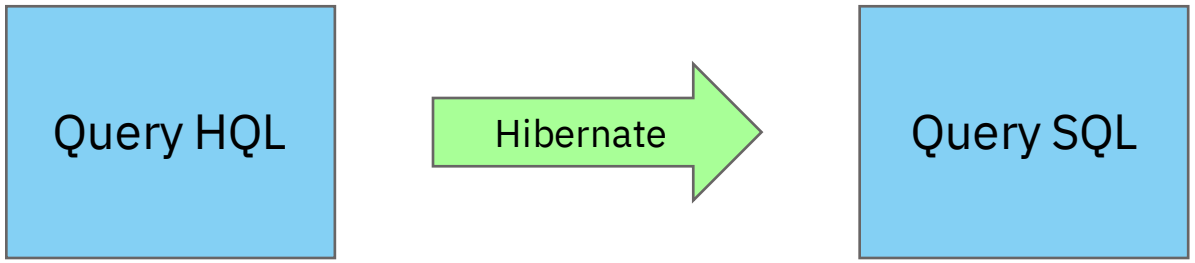
Action	Hibernate methods	JPA methods
create/save new entity	save(...)/persist(...)	persist(...)
retrieve entity by id	get(...)/load(...)	find(...)
save or update entity	saveOrUpdate(...)	merge(...)
delete entity	delete(...)	remove(...)

HQL

Hibernate Query Language (HQL) è un linguaggio di interrogazione object-oriented simile a SQL

Lavora su entità e proprietà invece di tabelle e colonne

Indipendente dal database



JPQL vs HQL

HQL estende JPQL aggiungendo delle caratteristiche specifiche per Hibernate

Una query JPQL è sempre una query HQL valida

Hibernate

HQL

```
SELECT p FROM Person p
```

```
FROM Person
```

```
SELECT p.firstName FROM Person p
```

positional parameter

```
FROM Person p WHERE p.lastName = ?0
```

named parameter

```
FROM Person p WHERE p.lastName = :lastName
```

```
SELECT p.age, count(p) FROM Person p
  GROUP BY p.age ORDER BY p.age
```

HQL

```
UPDATE Person p SET p.age = :age WHERE p.id = :person_id
```

```
DELETE FROM Person p WHERE e.id = :person_id
```

INSERT supporta solo l'inserimento di record provenienti da un altro oggetto

```
INSERT INTO Person(firstName, lastName, age)
  SELECT firstName, lastName, age FROM old_person
```

HQL

Aggregazione	avg(property)	SELECT avg(p.age) FROM Person p
	count(property or *)	SELECT count(p) FROM Person p
	max(property)	SELECT max(p.age) FROM Person p
	min(property)	SELECT min(p.age) FROM Person p
	sum(property)	SELECT sum(p.age) FROM Person p

Paginazione	Query setFirstResult(int startPosition)
	Query setMaxResults(int maxResult)

```
Query query = session.createQuery("FROM Person");
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```

Vediamo un esempio

(hibernate_esempio)