

Hibernate

Overview

- Implementation of Java EE's **Java Persistence API (JPA)** specification.
 - JPA classes and annotations
 - Java Persistence Query Language (JPQL)
- Additionally provides its own native API.
 - Hibernate classes and annotations to complement JPAs.
 - Hibernate Query Language (HQL) as extension to JPQL.

Common usage scenario

```
Configuration configuration = new Configuration();
ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings(
    configuration.getProperties()).build();
sessionFactory = configuration.addAnnotatedClass(Book.class).buildSessionFactory(serviceRegistry);

Session session = sessionFactory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    // Do some work...
    tx.commit();
} catch (RuntimeException e) {
    if (tx != null) { tx.rollback(); }
    // Handle exception...
} finally {
    session.close();
}

sessionFactory.close();
```

Creating and saving an entity

```
Session session = sessionFactory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    Book book = new Book();
    book.setIsbn("isbn");
    book.setTitle("title");
    session.save(book);
    tx.commit();
} catch (RuntimeException e) {
    if (tx != null) { tx.rollback(); }
    throw e;
} finally {
    session.close();
}
```

Finding and updating an entity

```
Session session = sessionFactory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    Book book = (Book) session.get(Book.class, 8L);
    book.setTitle("Corrected Title");
    session.save(book);
    tx.commit();
} catch (RuntimeException e) {
    if (tx != null) { tx.rollback(); }
} finally {
    session.close();
}
```

Persistent classes and mapping

Persistent classes

Java classes whose objects or instances will be stored in database tables are called **persistent classes** in Hibernate. The classes need to conform with the following rules:

- All Java classes that will be persisted need a default constructor.
- All classes should contain an ID in order to allow easy identification of your objects within Hibernate and the database. This property maps to the primary key column of a database table.
- A central feature of Hibernate, proxies, depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods.

Mapping

- Defines the relationship between the application's object data model to the database's relational data model.
 - How classes map to tables
 - How class attributes map to table columns
 - Which class attribute is going to be used as a primary key
 - What primary key generation strategy is to be used
 - Etc.
- Hibernate provides two mapping models:
 - Through XML configuration files
 - By using annotations

Annotation-based mapping

Annotation	Annotates	Used for
<code>@javax.persistence.Entity</code>	Class	Marks a class as a persistent class i.e. an entity bean. The class must have a no-argument constructor.
<code>@javax.persistence.Table</code>	Class	Specifies the details of the table that will be used to persist the table in the database.
<code>@javax.persistence.Id</code>	Field	Marks the field that will be used as a primary key.
<code>@javax.persistence.GeneratedValue</code>	Field	Used alongside <code>@Id</code> to specify that Hibernate should take care of generating the primary key value using some strategy.
<code>@javax.persistence.Column</code>	Field	Used to specify the details of the column to which the field will be mapped. It provides attributes such as "name", "length", "nullable", "unique".

Mapping an entity

```
@Entity
@Table(name = "book")
public class Book {

    @Id @GeneratedValue private Long id;
    @Column(name = "isbn") private String isbn;
    @Column(name = "title") private String title;

    public Book() { }
    public String getIsbn() { return isbn; }
    public void setIsbn(String isbn) { this.isbn = isbn; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}
```

Spring Data JPA

Spring Boot Maven dependency

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>
```

Core concepts

- Makes it easy to implement JPA repositories hiding away all of the plumbing code.
- Central abstraction is **Repository**.
 - **CrudRepository** abstraction extends Repository to provide CRUD functionalities.
 - **PagingAndSortingRepository** abstraction extends CrudRepository to ease pagination and sorting.
- Spring Data JPA automatically creates JPA implementations when the application starts, so called proxy instances.

CrudRepository

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {
    <S extends T> S save(S entity);
    Optional<T> findById(ID primaryKey);
    Iterable<T> findAll();
    long count();
    void delete(T entity);
    boolean existsById(ID primaryKey);
    // ... more functionality omitted.
}
```

PagingAndSortingRepository

```
public interface PagingAndSortingRepository<T, ID extends  
    Serializable> extends CrudRepository<T, ID> {  
    Iterable<T> findAll(Sort sort);  
    Page<T> findAll(Pageable pageable);  
}
```

Defining and using repositories

1. Declare an interface extending Repository or any of its subinterfaces specifying the entity class and the ID type, one for each entity.

```
interface PersonRepository extends Repository<Person, Long> { ... }
```

2. Add @EnableJpaRepositories, best if added to the “main” application class alongside @SpringBootApplication.
3. Inject the repositories using @Autowired where needed and use them.

Defining query methods

The query is derived from a repository method name.

Example:

```
interface UserRepository extends Repository<User, Long> {  
    User findByEmail(String email);  
    User findByFirstNameAndLastName(String firstName, String lastName);  
    List<User> findByZipCode(String zipCode);  
}
```

Defining query methods with JPQL

```
public interface UserRepository extends JpaRepository<User, Long> {  
    @Query("select u from User u where u.emailAddress = ?1")  
    User anyMethodNameOne(String emailAddress);  
  
    @Query("select u from User u where u.firstName = ?1 and u.lastName = ?2")  
    User anyMethodNameTwo(String firstName, String lastName);  
  
    @Query("select u from User u where u.zipCode = ?1")  
    List<User> anyMethodNameThree(String zipCode);  
}
```

Hibernate Query Language

Hibernate Query Language (HQL)

- Superset of the Java Persistence Query Language (JPQL).
- Similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.
- Understands notions such as inheritance, polymorphism and associations.
- HQL queries are translated into RDBMS specific SQL upon execution.

FROM clause

Used to load objects into memory.

Example:

```
"FROM Book"
```

SELECT clause

Provides more control over the result set than the FROM clause.

Example:

```
"SELECT B.title FROM Book B"
```

WHERE clause

Used to narrow the list of objects that should be returned from the storage.

Example:

```
"FROM Book B WHERE B.id = 10"
```

ORDER BY clause

- Used to sort the results returned from an HQL query.
- The results can be ordered by any property of the objects in the result set either ascending (ASC) or descending (DESC).
- Examples:
 - `"FROM Book B WHERE B.id > 10 ORDER BY B.title DESC"`
 - `"FROM Book B WHERE B.id > 10 ORDER BY B.title DESC, B.isbn ASC"`

GROUP BY clause

Allows Hibernate to pull information from the database and group it based on a value of an attribute, and typically, use the result to include an aggregate value.

Example:

```
"SELECT COUNT(LB.id), LB.isbn FROM LentBook LB GROUP BY LB.isbn"
```

UPDATE clause

Example:

```
"UPDATE Book set title = ?1 WHERE id = ?2"
```

DELETE clause

Example:

```
"DELETE FROM Book WHERE id = ?1"
```

Exercise

Exercise: Library

Create a Library application that allows a library administrator to:

- Register books
- List registered books

The following information is kept for each book:

- International Standard Book Number (ISBN)
- Title

* See next page for additional instructions.

Exercise: Library

- Adding dependency on a H2 in-memory database

```
dependency>  
  <groupId>com.h2database</groupId>  
  <artifactId>h2</artifactId>  
</dependency>
```

- Adding dependency on spring-boot-starter-web to enable access to the H2 console

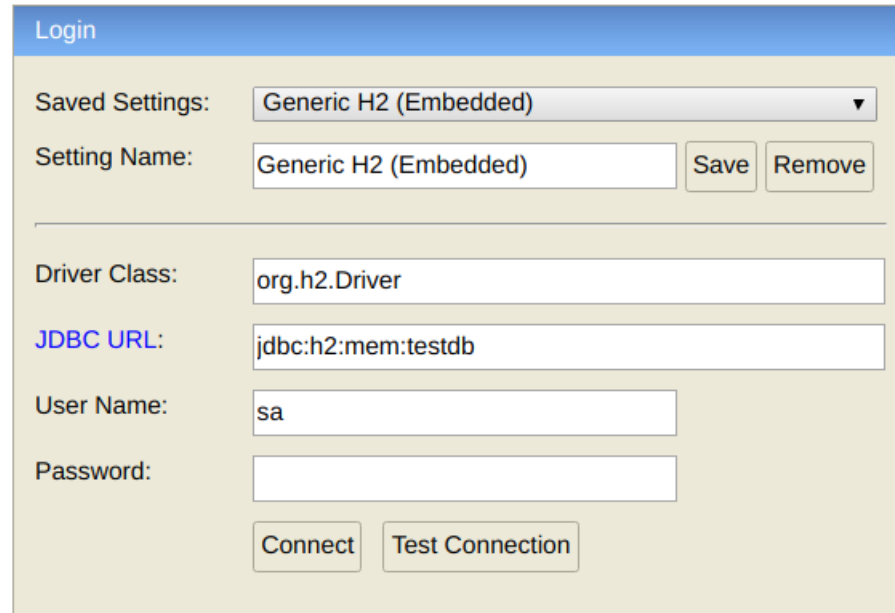
```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-web</artifactId>  
</dependency>
```

- Enabling the H2 console by adding the following line to the application.properties file:

```
spring.h2.console.enabled = true
```

Accessing the H2 console

Navigate to <http://localhost:8080/h2-console> and sign in using the settings displayed below.



The screenshot shows the 'Login' page of the H2 console. It features a blue header with the word 'Login'. Below the header, there are several input fields and buttons. The 'Saved Settings' section includes a dropdown menu currently showing 'Generic H2 (Embedded)'. Below this, the 'Setting Name' field also contains 'Generic H2 (Embedded)', with 'Save' and 'Remove' buttons to its right. A horizontal line separates this section from the connection configuration fields. These fields include 'Driver Class' with the value 'org.h2.Driver', 'JDBC URL' with the value 'jdbc:h2:mem:testdb', 'User Name' with the value 'sa', and an empty 'Password' field. At the bottom, there are two buttons: 'Connect' and 'Test Connection'.

Login	
Saved Settings:	Generic H2 (Embedded) ▼
Setting Name:	Generic H2 (Embedded) [Save] [Remove]
<hr/>	
Driver Class:	org.h2.Driver
JDBC URL:	jdbc:h2:mem:testdb
User Name:	sa
Password:	
[Connect] [Test Connection]	

Mapping entity associations

Mapping entity associations

- Supported association/relationship types:
 - One-to-one
 - Many-to-one / one-to-many
 - Many-to-many
- Direction:
 - Unidirectional
 - Bidirectional

One-to-one

```
@Entity
public class Employee {
    @OneToOne
    private Desk desk;
}
```

```
@Entity
public class Desk {
    @OneToOne(mappedBy = "desk")
    private Employee employee;
}
```

Many-to-one / one-to-many

```
@Entity
public class Fan {
    @ManyToOne
    private Singer favoriteSinger;
}
```

```
@Entity
public class Singer {
    @OneToMany(mappedBy = "favoriteSinger")
    private Collection<Fan> fans;
}
```

Many-to-many

```
@Entity
public class Developer {
    @JoinTable(name = "developer_project",
        joinColumns = @JoinColumn(name = "developer_id"),
        inverseJoinColumns = @JoinColumn(name = "project_id"))
    private Collection<Project> projects;
}
```

```
@Entity
public class Project {
    @ManyToMany(mappedBy = "projects")
    private Collection<Developer> developers;
}
```

Exercises

Exercise: Library

Modify the Library application to support:

- Member registration
- Member listing
- Book lending

When listing members, display information about books lent to that user. Likewise, when listing books, display information about the member the book has been lent to if any.

Application

Application: Twitter

- Modify the Twitter application to use a H2 database leveraging the Spring Data JPA technology.
- Include functionality for:
 - Signing up
 - Only email is stored
 - Signing in
 - By specifying the email
 - Signing out
 - Following other twitters
 - Modify the functionality for listing tweets to only include those which belong to twitters being followed by the signed in twitter.