Configuration of Hibernate

Object-Relational Mapping

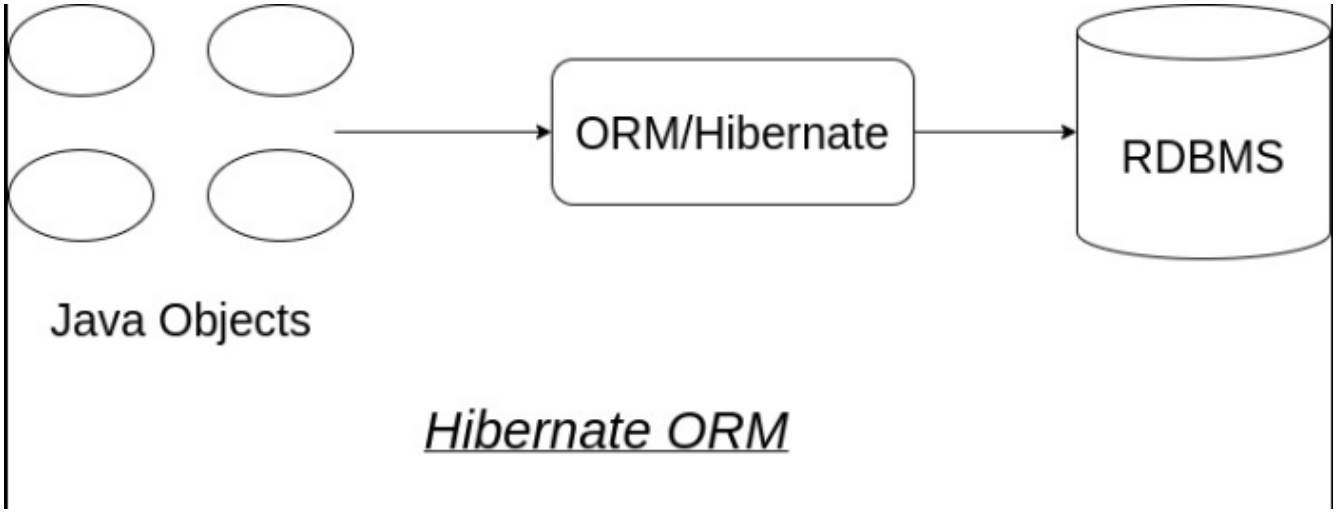
- **Object-Relational Mapping (ORM)** is a technique that lets you query and manipulate data from a database using an object-oriented paradigm.
- An ORM library is a completely ordinary library **written in your language of choice** that **encapsulates the code needed to manipulate the data, so you don't use SQL anymore**; you interact directly with an object in the same language you're using.

Hibernate

- Hibernate is a Java framework that **simplifies the development of Java application to interact with the database by mapping the POJOs to relational database tables**. It is an open source, lightweight, ORM (Object

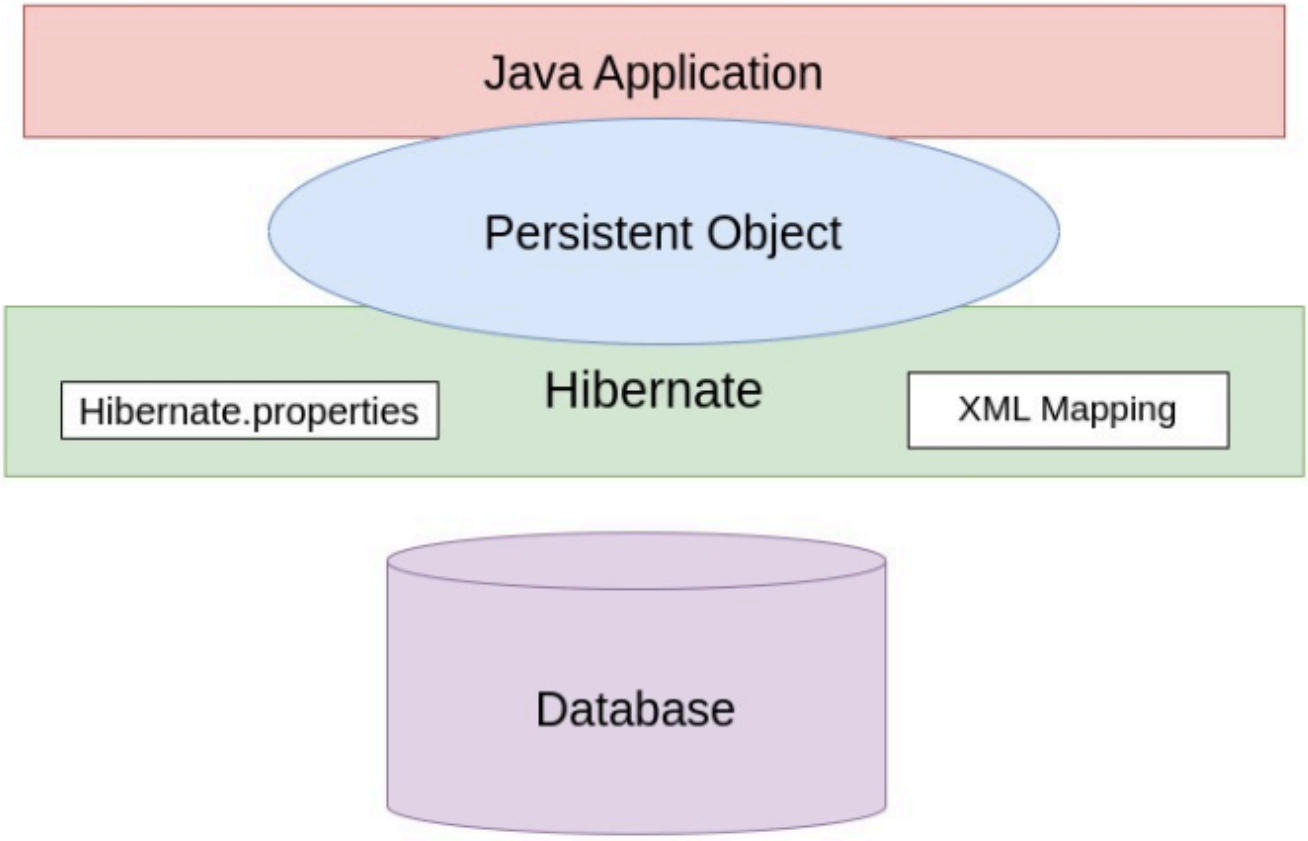
Relational Mapping) tool. Hibernate *implements the specifications of JPA (Java Persistence API)* for data persistence.

- **JPA**:- The JPA is a specification that defines how to persist data in Java Applications.
- The usage of Hibernate as a persistence framework enables the developers to concentrate more on the business logic instead of making efforts on SQLs and writing boilerplate code.

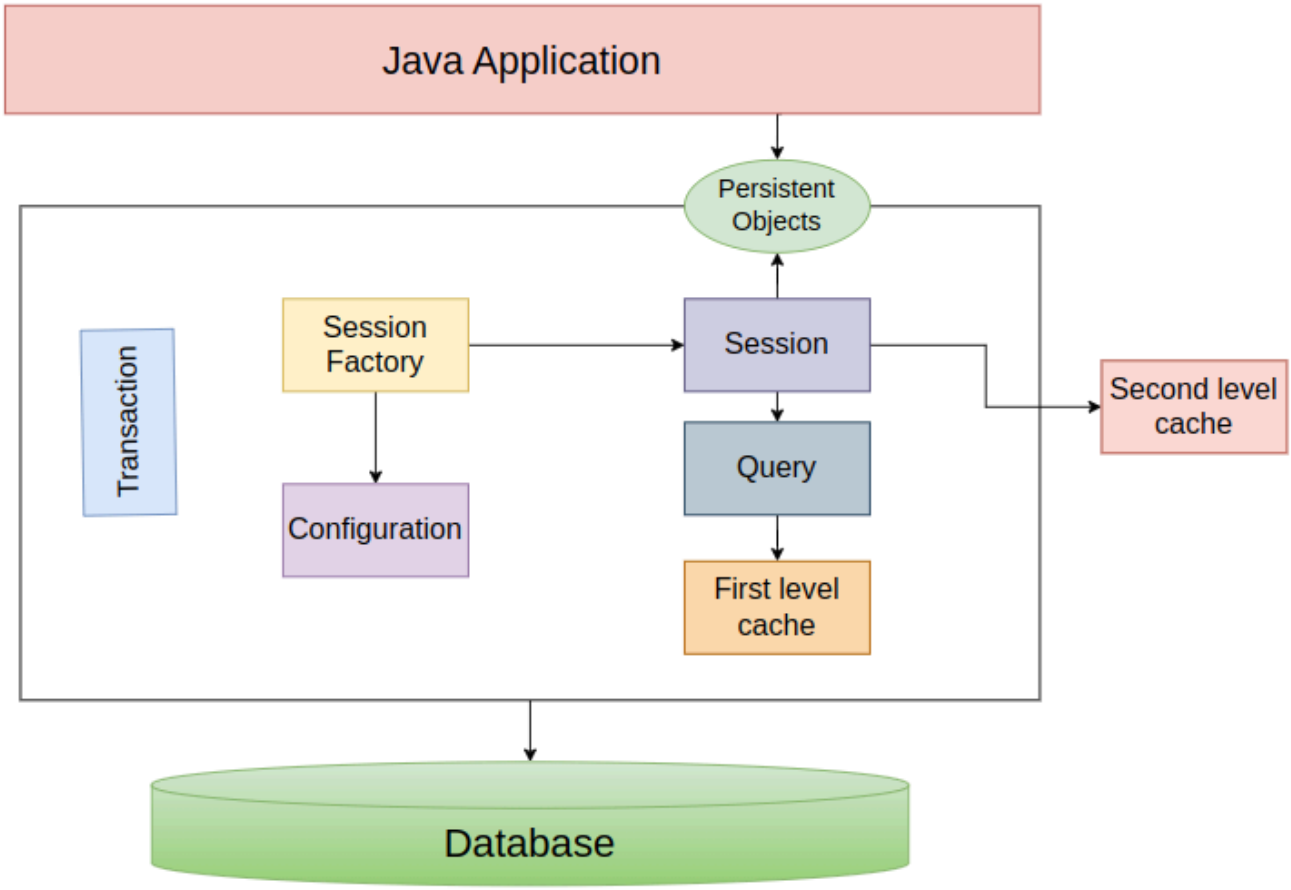


Hibernate Architecture

- 4 Layered architecture



Hibernate Architecture



Hibernate Detailed Architecture

- **Configuration**: It is the first object which is being created in the Hibernate Appliation.
- **Session**:
  - This object provides an interface to interact with the database from an application.
  - It is lightweight, which instantiates each time an interaction is required with the database.
  - Session objects are used to retrieve and save persistent objects. It is a short-lived object which wraps the JDBC connection.
  - It also provides a first-level cache of data.
- **Transaction**:
  - A Transaction represents a unit of work with the database, and most of the RDBMS supports transaction functionality.
  - Transaction object provides a method for transaction management.
  - It enables data consistency and rollback in case something wrong.
- **Query**:
  - Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects.
  - A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.
- **Persistent Objects**:
  - These are plain old java objects (POJOs), which get persisted into the database by hibernate. Persistent objects can be configured in configurations files(hibernate.cfg.xml or hibernate.properties) or annotated with @Entity annotation.

- **First level cache**
  - **Caching**is a mechanism that improves the performance of an application by reducing the number of database queries. Hibernate ORM provides built-in caching facilities, primarily the First-level cache and the Second-level cache.
  - The First-level cache is associated with the Hibernate Session object and is enabled by default. It cannot be disabled. The cache's purpose is to reduce the number of database queries within a single transaction.
  - ***Key Characteristics***
    - **Scope**: The First-level cache scope is limited to the session. It exists as long as the session object is alive.
    - **Default Behavior**: The First-level cache is enabled by default and cannot be disabled.
    - **Query Mechanism**:
      - The first time an entity is queried within a session, it is retrieved from the database and stored in the First-level cache.
      - Subsequent queries for the same entity within the same session are loaded from the cache.
    - **Eviction Methods**:
      - session.evict(entity): Removes a specific entity from the session cache.
      - session.clear(): Removes all entities from the session cache.
- **Second-level cache**
  - The Second-level cache is associated with the SessionFactory and provides a global caching mechanism across sessions. This cache is optional and can be configured based on application needs.
  - ***Key Characteristics***
    - **Scope**: The Second-level cache is scoped to the SessionFactory. It persists as long as the SessionFactory is active.
    - **Configuration**: It needs to be explicitly enabled and configured.
    - **Query Mechanism**:
      - When an entity is queried, Hibernate first looks up the First-level cache.
      - If the entity is not found, the Second-level cache is checked.
      - If found in the Second-level cache, the entity is returned and added to the First-level cache.
      - If not found in either cache, the entity is loaded from the database and both caches are updated.

XML based Configuration

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-5.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/test</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>
        <property name="hibernate.connection.pool_size">10</property>
        <property name="show_sql">true</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.current_session_context_class">thread</property>
    </session-factory>
</hibernate-configuration>
```

`application.properties`

```
hibernate.dialect= org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class= com.mysql.jdbc.Driver
hibernate.connection.url= jdbc:mysql://localhost:3306/test
hibernate.connection.username= root
hibernate.connection.password=root
hibernate.show_sql=true
hibernate.hbm2ddl=update
```

Hibernate Configuration with Annotation

@Entity

- In JPA, POJOs are entities that represent the data that can be persisted into the database.
- An entity represents the **table** in a DB. Every instance of an entity represents a **row** in the table.
- This annotation must be done at **class level**.
- The entity must have a **no arg constructor** and a **primary key**.
- The entity name **defaults to the name of the class**. Its name **can be changed using the name element** into @Entity.
- Entity classes **must not be declared final** since JPA implementations try to subclass the entity in order to provide their functionality

@Id and @GeneratedValue

- The primary key uniquely identifies the instances of the entity.
- The primary key is a must for each JPA entity. The @Id annotation is used to define the primary key in a JPA entity.
- Identifiers are generated using @GeneratedValue annotation. There are four strategies to generate id.
- The strategy element in @GeneratedValue can have value from any of **AUTO**, **TABLE**, **SEQUENCE**, or **IDENTITY**

@Table

- Bydefault, the name of the table in the database will be the same as the entity name. @Table annotation is used to define the name of the table in the database.

@Column

- Similar to @Table annotation, @Column annotation is used to provide the details of columns into the database. Check the other elements defined into @Column, such as length, nullable, unique.

@Transient

- @Transient annotation is used to make a field into an entity as non-persistent. For example, the age of a student can be calculated from the date of birth. Thus, age field can be made as non-persistent into Student class.

@Temporal

- The types in the `java.sql package` such as `java.sql.Date , java.sql.Time , java.sql.Timestamp` are in line with SQL data types. Thus, its mapping is straightforward, and either the **@basic** or **@column** annotation can be used.
- The type `java.util.Date` contains ***both date and time*** information. This is not directly related to any SQL data type. For this reason, another annotation is required to specify the desired SQL type.
- **@Temporal** annotation is used to specify the desired SQL data type. It has a single element TemporalType. TemporalType can be DATE, TIME or TIMESTAMP. The type of `java.util.Calendar` also requires @Temporal annotation to map with the corresponding SQL data type.

Example:

```
@Entity

@Table(name = "student")

public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    privateintid;

    @Column(name = "firstname", length = 50, nullable = false, unique = false)
    private String firstName;

    @Column(name = "lastname", length = 50, nullable = true, unique = false)
    private String lastName;

    @Transient
    private int age;

    @Temporal(TemporalType.DATE)
    @Column(name = "dateofbirth", nullable = false)
    private Date dob;

    private String email;

    public Student() {}

    public Student(String firstName, String lastName, String email, Date dob) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.dob = dob;
    }
    public int getId() {
        returnid;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        returnfirstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
```

```
    public String getLastName() {
        returnlastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getEmail() {
        returnemail;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public int getAge() {
        returnage;
    }
    public void setAge(intage) {
        this.age = age;
    }
    public Date getDob() {
        return dob;
    }
    public void setDob(Date dob) {
        this.dob = dob;
    }
    @Override
    public String toString() {
        return "Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + ", age="
            dob + ", email=" + email + "]";
    }
}
```

Complete Hibernate Example

1. Create Project using maven.
2. Add ***dependencies*** - **Mysql**, **Hibernate**, **xml bind**.

   pom.xml

```
    <dependencies>

         <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
         <dependency>
             <groupId>mysql</groupId>
             <artifactId>mysql-connector-java</artifactId>
             <version>8.0.13</version>
         </dependency>
         <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -
         ->

         <dependency>
             <groupId>org.hibernate</groupId>
             <artifactId>hibernate-core</artifactId>
             <version>5.3.7.Final</version>
         </dependency>

         <dependency>
```

```
            <groupId>jakarta.xml.bind</groupId>
            <artifactId>jakarta.xml.bind-api</artifactId>
            <version>2.3.2</version>
        </dependency>

    </dependencies>
```

3. Create ***Hibernate Configuration File***

`HibernateUtil.java`

```
public class HibernateUtil
 {
     private static SessionFactory sessionFactory;
     public static SessionFactory getSessionFactory()
     {
         if (sessionFactory == null)
         {
             try
             {
                 Configuration configuration = newConfiguration();

                 // Hibernate settings equivalent to hibernate.cfg.xml's properties
                 Properties settings = newProperties();
                 settings.put(Environment.DRIVER, "com.mysql.cj.jdbc.Driver");
                 settings.put(Environment.URL,"jdbc:mysql://localhost:3306/test?useSSL=false");
                 settings.put(Environment.USER, "root");
                 settings.put(Environment.PASS, "root");
                 settings.put(Environment.DIALECT,"org.hibernate.dialect.MySQL5Dialect");
                 settings.put(Environment.SHOW_SQL, "true");
                 settings.put(Environment.CURRENT_SESSION_CONTEXT_CLASS, "thread");
                 settings.put(Environment.HBM2DDL_AUTO, "create-drop");
                 configuration.setProperties(settings);
                 configuration.addAnnotatedClass(Student.class);
                 ServiceRegistryserviceRegistry = newStandardServiceRegistryBuilder()
                 .applySettings(configuration.getProperties()).build();
                 sessionFactory = configuration.buildSessionFactory(serviceRegistry);
             }
             catch (Exception e)
             {
                 e.printStackTrace();
             }
         }
     returnsessionFactory;
     }
 }
```

4. Create a **JPA Entity/Persistent class**
    - Non final class
    - A no arg constructor
    - Identifier Property
    - Getter and Setter methods

`Student.java`

```
@Entity
```

```
@Table(name = "student")

public class Student {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id")
privateintid;

@Column(name = "firstname", length = 50, nullable = false, unique = false)
private String firstName;

@Column(name = "lastname", length = 50, nullable = true, unique = false)
private String lastName;

@Transient
private int age;

@Temporal(TemporalType.DATE)
@Column(name = "dateofbirth", nullable = false)
private Date dob;

private String email;

public Student() {}

public Student(String firstName, String lastName, String email, Date dob) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    this.dob = dob;
}
public int getId() {
    returnid;
}
public void setId(int id) {
    this.id = id;
}
public String getFirstName() {
    returnfirstName;
}
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
public String getLastName() {
    returnlastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
}
public String getEmail() {
    returnemail;
}
public void setEmail(String email) {
    this.email = email;
}
public int getAge() {
    returnage;
}
public void setAge(intage) {
    this.age = age;
```

```
}
public Date getDob() {
    return dob;
}
public void setDob(Date dob) {
    this.dob = dob;
}
@Override
public String toString() {
    return "Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + ", age="
        dob + ", email=" + email + "]";
}
}
```

5. Create a **DAO layer** that performs the operations to the database.

StudentDao.java

```
public class StudentDao
 {
     public void save(Student student)
     {
         Transaction txn = null;
         try (Session sess = HibernateUtil.getSessionFactory().openSession())
         {
             txn = sess.beginTransaction(); // start a transaction
             sess.save(student); // save the student object
             txn.commit(); // commit transaction
         }
         catch (Exception e)
         {
             if (txn != null)
             {
                 txn.rollback();
             }
                 e.printStackTrace();
             }
         }

     public List<Student>getStudentList()
     {
         try (Session sess = HibernateUtil.getSessionFactory().openSession())
         {
             return sess.createQuery("from Student", Student.class).list();
         }
     }
 }
```

6. Create the **main class**

HibernateApp.java

```
publicclassHibernateApp
 {
     public static void main(String[] args) throws Exception
     {
         StudentDao studentDao = newStudentDao();
         SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy");
```

```
            Date dob = dateFormat.parse("10-08-1986");
            Student student = newStudent("Rahul", "Singh", "rahulsingh@gmail.com", dob);
            studentDao.save(student);

            List < Student >students = studentDao.getStudentList();
            students.forEach(s ->System.out.println(s));
        }
    }
```

7. Run the App

Hibernate CRUD Features

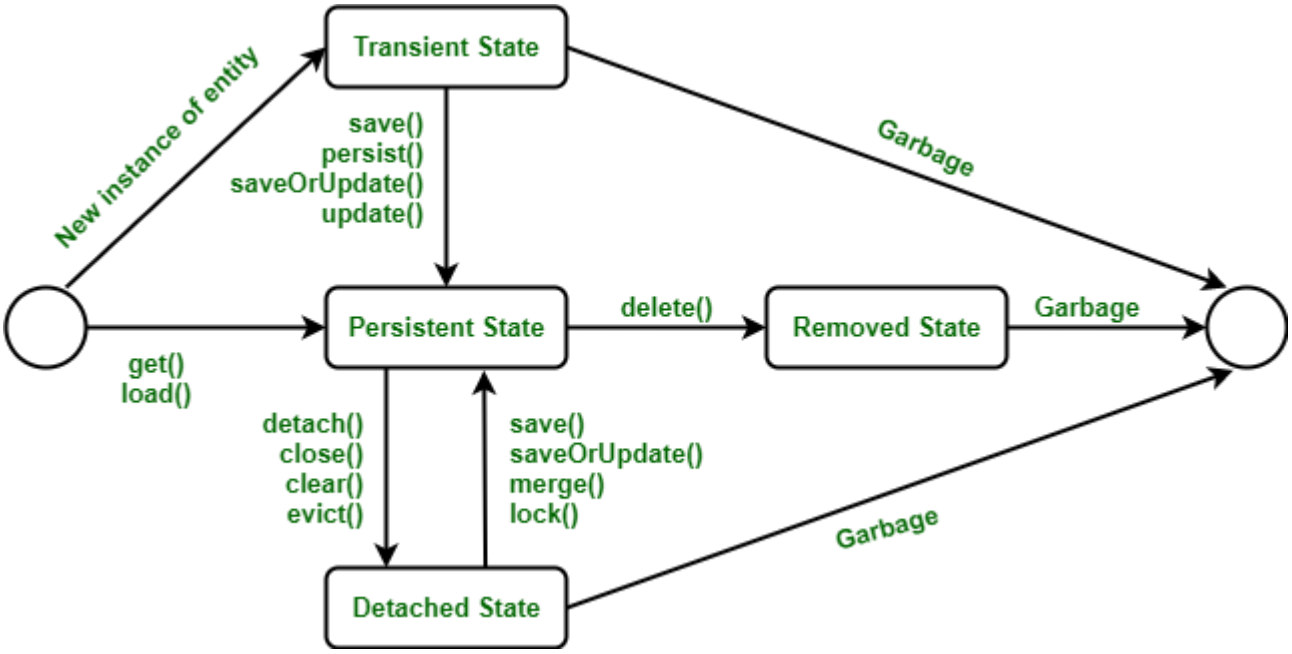
Hibernate Entity Lifecycle states

- Hibernate works with POJO.Without any Hibernate specific annotation and mapping,Hibernate does not recognize these POJO's. Once properly annotated with required annotation, hibernate identifies them and keeps track of them to perform database operations such as create, read, update and delete. These POJOs are considered as mapped with Hibernate.
- An instance of a class mapped with Hibernate can be any of the **four persistence states** which are known as ***hibernate entity lifecycle states***
    i. ***Transient***
    ii. ***Persistent***
    iii. ***Detached***
    iv. ***Removed***

1. **Transient**
    - An object of a POJO class is in Transient state when created using new operator in java.
    - In this, the object is not related to any database table.
    - Object is not managed by session or Entity manager.
    - Object is not tracked for any modification.
    - So any changes in the object doesn't impact the database.
2. **Persistent**
    - When a transient entity is saved, it is associated with a unique session object and it enters into a Persistent state.
    - It is a representation of a row into a database.
    - Session manages the Persistent state objects and keeps track of every modification done. It automatically propagates all the modifications into database.
    - The following methods take an object from Transient to Persistent
        - session.save()
        - session.update()
        - session.saveOrUpdate()
        - session.lock()
        - session.merge()
3. **Detached**
    - On call of any session method such as **clear()**, **evict()** or **close()** make the persistent state objects into the detached state.

- Session does not keep track of any modification to detached state objects. Detached entity has a corresponding row in the database but changes to the entity will not be reflected in the database.
- The detached state entity can be re-attached to the Hibernate Session with the call of following methods
  - session.update()
  - session.merge()
  - session.saveOrUpdate()

4. **Removed**
   - Once a persistent object passes through session's delete() method, it enters into Removed state. When an application commits the session the entries in the database that correspond to remove entities are deleted.
   - If the removed entities are not referenced anywhere, then it is eligible for garbage collection.




```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateEntityLifecycle {

    public static void main(String[] args) {
        // Create a configuration object
        Configuration configuration = new Configuration();
        configuration.configure("hibernate.cfg.xml");
        configuration.addAnnotatedClass(User.class);

        // Build a SessionFactory
        SessionFactory sessionFactory = configuration.buildSessionFactory();

        // Open a session
        Session session = sessionFactory.openSession();

        // Transient state
        User user = new User(1L, "John Doe");
        System.out.println("User is in Transient state: " + user);

        // Begin transaction
        session.beginTransaction();
```

```
        // Persistent state
        session.save(user);
        System.out.println("User is now in Persistent state: " + user);

        // Make some changes in Persistent state
        user.setName("Jane Doe");
        System.out.println("User's name changed in Persistent state: " + user);

        // Commit the transaction
        session.getTransaction().commit();

        // Detaching the entity
        session.evict(user);
        System.out.println("User is now in Detached state: " + user);

        // Making changes in Detached state
        user.setName("John Smith");
        System.out.println("User's name changed in Detached state: " + user);

        // Reattach the entity to the session and make it Persistent again
        session.beginTransaction();
        session.update(user);
        System.out.println("User is back in Persistent state: " + user);

        // Commit the transaction
        session.getTransaction().commit();

        // Removing the entity
        session.beginTransaction();
        session.delete(user);
        System.out.println("User is now in Removed state: " + user);

        // Commit the transaction
        session.getTransaction().commit();

        // Close the session
        session.close();

        // Close the SessionFactory
        sessionFactory.close();
    }
}
```

Hibernate Get Entity – get vs load

- The Hibernate provides session.load() and session.get() methods to fetch the entity by id.
- **session.load()**
  - There are several overloaded methods of load() into the Hibernate Session interface. Each load() method requires the object's primary key as an identifier to load the object from the database. Along with the identifier, hibernate also requires to know which class or entity name to use to find the object.
- **session.get()**
  - The get() method is similar to the load() method to fetch the entity from the database. Like load() method, get() methods also take an identifier and either an entity name of a class.
- **Difference between load() and get() methods**
  - The get() method returns NULL if the identifier does not exist.

- The load() method throws a runtime exception if the identifier does not exist.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateExample {

    public static void main(String[] args) {
        // Create a configuration object
        Configuration configuration = new Configuration();
        configuration.configure("hibernate.cfg.xml");
        configuration.addAnnotatedClass(User.class);

        // Build a SessionFactory
        SessionFactory sessionFactory = configuration.buildSessionFactory();

        // Open a session
        Session session = sessionFactory.openSession();

        // Start a transaction
        session.beginTransaction();

        // Assuming the database has a User with id 1
        Long existingUserId = 1L;
        Long nonExistingUserId = 999L;

        // Using session.get()
        User userGet = session.get(User.class, existingUserId);
        if (userGet != null) {
            System.out.println("User fetched with get(): " + userGet.getName());
        } else {
            System.out.println("No user found with id " + existingUserId + " using get()");
        }

        User userGetNonExisting = session.get(User.class, nonExistingUserId);
        if (userGetNonExisting != null) {
            System.out.println("User fetched with get(): " + userGetNonExisting.getName());
        } else {
            System.out.println("No user found with id " + nonExistingUserId + " using get()");
        }

        // Using session.load()
        try {
            User userLoad = session.load(User.class, existingUserId);
            System.out.println("User fetched with load(): " + userLoad.getName());
        } catch (org.hibernate.ObjectNotFoundException e) {
            System.out.println("No user found with id " + existingUserId + " using load()");
        }

        try {
            User userLoadNonExisting = session.load(User.class, nonExistingUserId);
            System.out.println("User fetched with load(): " + userLoadNonExisting.getName());
        } catch (org.hibernate.ObjectNotFoundException e) {
            System.out.println("No user found with id " + nonExistingUserId + " using load()");
        }

        // Commit the transaction
        session.getTransaction().commit();
```

```
        // Close the session
        session.close();

        // Close the SessionFactory
        sessionFactory.close();
    }
}
```

Hibernate Insert Entity – persist, save and saveOrUpdate

- These methods are used to store an object in a database. There are significant differences among these methods.
- **persist()**
  - The persist() method is used to add a record into the database. This method adds a new entity instance to persistent context. On call of persist() method, the transit state object moves into a persistent state object but not yet saved to the database.
  - The insert statement generates only upon committing the transaction, flushing or closing the session.
  - This method has a return type as void.
- **save()**
  - The save() method is similar to the persist() method which stores the record into the database. This is the original Hibernate Session method which does not conform to JPA specification.
  - The save() method differs from the persist() method in terms of its implementation.
  - The documentation of this method states that at first, it assigns the generated id to the identifier and then persists the entity.
  - The save() method returns the Serializable value of this identifier.
- **saveOrUpdate()**
  - This method either performs save() or update() on the basis of identifier existence. E.g. If an identifier exists for the instance, update() method will be called otherwise save() will be performed.
  - The saveOrUpdate() method handles the cases where we need to save a detached instance. Unlike the save() operation on detached instances, the saveOrUpdate() method does not result in a duplicate record. Similar to update(), this method is used to reattach an instance to the session.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateInsertEntityExample {

    public static void main(String[] args) {
        // Create a configuration object
        Configuration configuration = new Configuration();
        configuration.configure("hibernate.cfg.xml");
        configuration.addAnnotatedClass(User.class);

        // Build a SessionFactory
        SessionFactory sessionFactory = configuration.buildSessionFactory();

        // Open a session
        Session session = sessionFactory.openSession();

        // Begin transaction
        session.beginTransaction();
```

```
        // Using persist()
        User userPersist = new User("John Persist");
        session.persist(userPersist);
        System.out.println("User persisted but not yet saved to DB: " + userPersist);

        // Commit the transaction to save the user to the database
        session.getTransaction().commit();
        System.out.println("User saved to DB with persist(): " + userPersist);

        // Open a new session and transaction for next operations
        session = sessionFactory.openSession();
        session.beginTransaction();

        // Using save()
        User userSave = new User("John Save");
        Serializable userId = session.save(userSave);
        System.out.println("User saved to DB with save(): " + userSave + " with ID: " + userId);

        // Commit the transaction
        session.getTransaction().commit();

        // Open a new session and transaction for next operations
        session = sessionFactory.openSession();
        session.beginTransaction();

        // Using saveOrUpdate() with a new user (acts like save)
        User userSaveOrUpdateNew = new User("John SaveOrUpdate New");
        session.saveOrUpdate(userSaveOrUpdateNew);
        System.out.println("New user saved to DB with saveOrUpdate(): " + userSaveOrUpdateNew);

        // Commit the transaction
        session.getTransaction().commit();

        // Open a new session and transaction for next operations
        session = sessionFactory.openSession();
        session.beginTransaction();

        // Using saveOrUpdate() with an existing user (acts like update)
        User userSaveOrUpdateExisting = session.get(User.class, userSaveOrUpdateNew.getId());
        userSaveOrUpdateExisting.setName("John SaveOrUpdate Updated");
        session.saveOrUpdate(userSaveOrUpdateExisting);
        System.out.println("Existing user updated in DB with saveOrUpdate(): " + userSaveOrUpdateExist

        // Commit the transaction
        session.getTransaction().commit();

        // Close the session
        session.close();

        // Close the SessionFactory
        sessionFactory.close();
    }
}
```

Hibernate Query Language

- Hibernate Query Language (HQL) is an object-oriented query language. HQL is similar to the database SQL language. The main difference between HQL and SQL is HQL uses class name instead of table name, and property names instead of column name.

HQL Select

```
Query query = session.createQuery("from Student where id = :id ");
query.setParameter("id", "1");
List list = query.list();
```

HQL Update

```
Query query = session.createQuery("Update Student set lastName = :lastName where id = :id");
query.setParameter("lastName", "Singh");
query.setParameter("id", "1");
int result = query.executeUpdate();
```

HQL Delete

```
Query query = session.createQuery("Delete Student where id = :id ");
query.setParameter("id", "1");
int result = query.executeUpdate();
```

HQL Insert

```
Query query = session.createQuery("Insert into Student(firstName, lastName, emailId)"+"Select firstNam
int result = query.executeUpdate();
```

Spring Data JPA and its advantages

- Spring Data JPA is one of the many sub-projects of Spring Data which simplifies the data access for relational data stores. Spring Data JPA is not a JPA provider; instead it wraps the JPA provider and adds its own features like a no-code implementation of the repository pattern. Spring Data JPA uses Hibernate as the default JPA provider. JPA provider is configurable, and other providers can also be used with Spring Data JPA. Spring Data JPA provides complete abstraction over the DAO layer into the project.
- The advantages of using Spring Data JPA are as follows:
  - **DAO Abstraction with No-Code Repositories**
    - Spring Data JPA defines many Repository Interfaces such as **CrudRepository**, **PagingAndSortingRepository**, **JpaRepository** having methods to store, retrieve, sorted retrieval, paginated result and many more.With Spring Data JPA, we don’t have to write SQL statements; instead, we just need to extend the interface defined by Spring Data JPA for one of the entities.
  - **Query Methods**

        - Another robust and comfortable feature of Spring Data JPA is the Query Methods. Based on the name of methods declared in the repository interfaces are converted to low level SQL queries at runtime.
    - **Seamless Integration**
        - Spring Data JPA seamlessly integrates JPA into the Spring stack, and its repositories reduce the boilerplate code required to the JPA specification.Spring Data JPA also helps the DAO layer integration and interaction with other Spring based components in your application, like accessing property configurations or auto-wiring the repositories into the application service layer. It also works perfectly with Spring Boot auto-configuration.

Check Your Progress - 1

1. What is Hibernate Framework? List down its advantages.
    - Advantages
        - Open Source
        - High Performance
        - Light Weighh
        - Database Independent
        - Caching
        - Scalability
        - Lazy Loading
        - HQL
2. Explain some important interfaces of Hibernate framework.
3. What is Hibernate caching? Explain Hibernate first level cache.
4. Explain the working of second level cache.

Check Your Progress - 2

1. Describe @Temporal annotation with its usage.
2. List down the prescribed guidelines for a persistent class.
3. Explain Hibernate entity lifecycle state.
4. Explain HQL with examples.

Check Your Progress - 3

1. What is the difference between load() and get() method of Hibernate Session?
2. What is the difference between persist() and save() method of Hibernate Session?
3. Why is Hibernate Session's saveOrUpdate() method considered a universal tool to make an object persistent?
4. Explain Spring Data JPA with its advantages.