UNIT 10 CONFIGURATION OF HIBERNATE(ORM)

Structure **Page No.**

10.0 INTRODUCTION

In computer science, object-relational mapping (ORM, O/RM, and O/R mapping tool) is a programming technique for converting data between incompatible type systems using object-oriented programming languages. ORM makes it possible to perform ***CRUD (Create, Read, Update and Delete)*** operations without considering how those objects relate to their data source. It manages the mapping details between a set of objects and the underlying database. It hides and encapsulates the changes in the data source itself. Thus, when data sources or their APIs change, only ORM change is required rather than the application that uses ORM.

Hibernate is a pure Java object-relational mapping (ORM) and persistence framework which maps the POJOs to relational database tables. 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. There are several advantages of Hibernate, such as:

- Open Source
- High Performance
- Light Weight
- Database independent
- Caching
- Scalability
- Lazy loading
- Hibernate Query Language (HQL)

Java Persistence API (JPA) is a specification that defines APIs for object-relational mappings and management of persistent objects. JPA is a set of interfaces that can be used to implement the persistence layer. JPA itself doesn't provide any implementation classes. In order to use the JPA, a provider is required which implements the specifications. Hibernate and EclipseLink are popular JPA providers.

Spring Data JPA is one of the many sub-projects of **Spring Data**that simplifies the data access to 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 a complete abstraction over the DAO layer into the project.

This unit explains only Hibernate in detail with its architecture and sample examples. This unit also covers the relationship between Spring Data JPA, JPA and Hibernate. There are many similar Crud operations available in Hibernate, such as **save, persist, saveOrUpdate, get and load**. Hibernate entity state defines the behavior of the operations. Thus, one should study the Hibernate entity states as **transient, persistent, detached and removed** very carefully in order to have in-depth insights for Hibernate ***CRUD*** operations.

This unit also covers the overview of **Spring Data JPA** with its advantages and explains how this reduces the effort of a developer to implement the data access layer. The next unit covers Spring Data JPA in detail.

10.1 OBJECTIVES

After going through this unit, you will be able to:

- describe Hibernate as ORM tool,
- explain Hibernate architecture,
- use Annotations in Hibernate and Java-based Hibernate configuration,
- use Hibernate CRUD operations such as save, persist, saveOrUpdate,
- differenciate between same task performing operations such asget Vs load etc.,
- establish the relationship between Spring Data JPA, JPA and Hibernate, and
- use Spring Data JPA.

10.2 HIBERNATE OVERVIEW

Hibernate is an open-source Java persistence framework created by Gavin King in 2011. It simplifies the development of Java applications to interact with databases. Hibernate is a lightweight, powerful and high-performance ORM (Object Relational Mapping) tool that simplifies data creation, data manipulation and data access.

The Java Persistence API (JPA) is a specification that defines how to persist data in Java applications. Hibernate is a popular ORM which is a standard implementation of the JPA specification with a few additional features specific to Hibernate. Hibernate lies between Java Objects and database servers to handle all persistence and retrieval of those objects using appropriate mechanisms and patterns. A schematic diagram is shown in Figure 10.1

Figure10.1 Hibernate ORM

10.2.1 Hibernate Architecture

Hibernate has four-layered architecture. The Hibernate includes many objects such as persistent objects, session factory, session, connection factory, transaction factory, transaction etc. High-level architecture diagram is shown in Figure 10.2

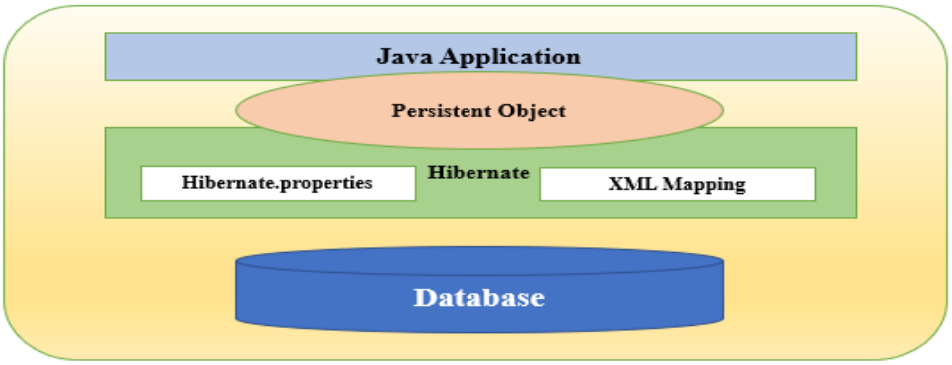


Figure 10.2 Hibernate Architecture

A detailed view of the Hibernate Application Architecture with some of the core classes is shown in Figure 10.3

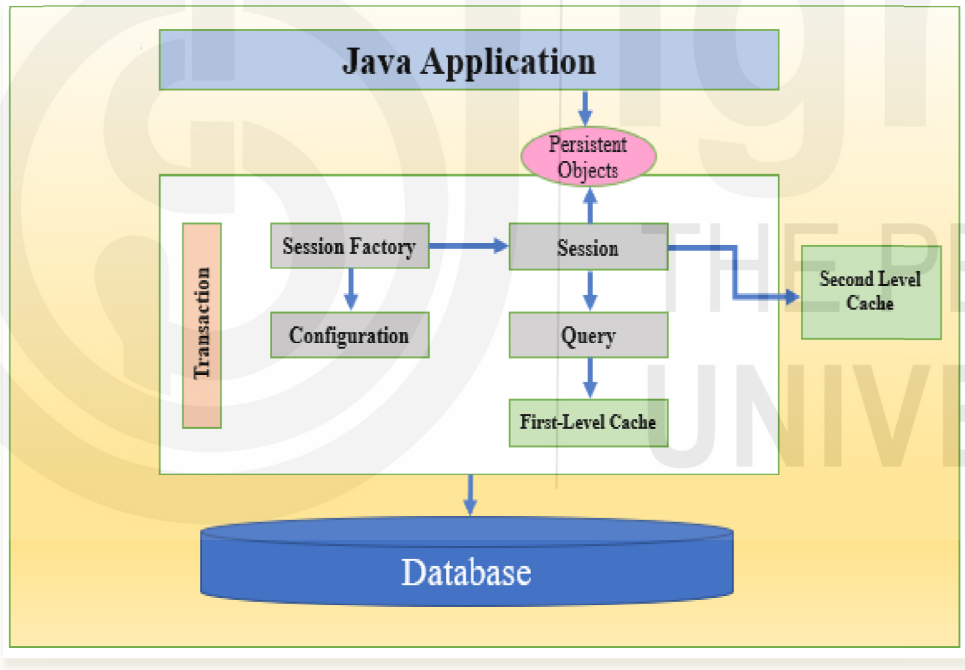


Figure10.3 Hibernate Detailed Architectural View

Hibernate uses Java APIs like Java Transaction API (JTA), Java Database Connectivity (JDBC) and Java Naming and Directory Interface (JNDI). The knowledge of Hibernate architecture elements helps you to understand the internal working of Hibernate.

Configuration

An instance of Configuration allows the application to specify properties and mapping documents to be used when creating a SessionFactory. The Configuration object is the first object which is being created in the Hibernate application. The Configuration is

only an initialization-time object. Usually, a Hibernate application creates a single Configuration, builds a single immutable SessionFactory and then instantiate Sessions in threads servicing client requests. Configuration represents an entire set of mappings of an application's Java types to an SQL database. The Configuration can be configured either programmatically or using configuration file. The configuration file can be either an XML file such as hibernate.cfg.xml or a properties file such as hibernate.properties.

Xml based hibernate configuration for Mysql database is shown below-

```
<?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>
```

Properties file base hibernate configuration for Mysql database is shown below-

```
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
```

SessionFactory

The Hibernate creates an immutable SessionFactory object using the Configuration object. SessionFactory is used to instantiate the session object in a thread to service client requests. SessionFactory object is thread safe and used by all the threads of an application.

SessionFactory object is per database using a separate configuration file. Thus, for multiple databases, we will require to create multiple SessionFactory objects. The SessionFactory is a heavyweight object, and it is created at the time of application startup.

Session

The session 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

Cache is a mechanism that improves the performance of any application. Hibernate provides a caching facility also. First-level cache is enabled by default, and it can't be disabled. Hibernate ORM reduces the number of queries made to a database in a single transaction using First-level cache. First-level cache is associated with **Session object** and it is available till the session object is alive. First-level cache associated with session objects is not accessible to any other session object in other parts of the application. Important facts about First-level cache is as followings-

- First-level cache scope is session. Once the session object is closed, the first-level cache is also destroyed.
- First-level cache can't be disabled.
- First time query for an entity into a session is retrieved from the database and stored in the First-level cache associated with the Hibernate session.
- Query for an object again with the same session object will be loaded from the cache.
- Session evict() method is used to remove the loaded entity from Session.
- Session clear() method is used to remove all the entities stored in the cache.

Second-level Cache

Second-level cache is used globally in **Session Factory** scope. **Once the session factory is closed, all cache associated with it dies**, and the cache manager is also closed down. Working of Second-level cache is as followings-

- At first, the hibernate session tries to load an entity, then First-level cache is looked up for cached copy of entity.
- If an entity is available in First-level cache, it is returned as a result of the load method.
- If an entity is not available in First-level cache, Second-level cache is looked up for cached copy of the entity.
- If an entity is available into Second-level cache, it is returned as a result of the load method. But before returning the result, the First-level cache is being updated so that the next invocation for the same entity can be returned from the First-level cache itself.
- If an entity is not found in any of cache, the database query is executed, and both the cache is being updated.
- Second level cache validates itself for modified entities, if the modification has been done through hibernate session APIs.

☛Check Your Progress 1

1) What is Hibernate Framework? List down its advantages.

……

……

……

……

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.

……

……

……

……

10.3 HIBERNATE CONFIGURATION WITH

ANNOTATION

This section explains various annotations available in Hibernate. Later a "Hello World" hibernate example will be created using explained annotation.

@Entity

In JPA, POJOs are entities that represent the data that can be persisted into a database. An entity represents the **table** in a database. Every instance of an entity represents a **row** in the table. Let's say there is a POJO named Student, which represents the data of **Student**. In order to store the student records into a database, **Student** POJO must be defined as an entity so that JPA is aware of it. This can be done by annotating POJO class with **@Entity** annotation. The annotation must be defined at **class level**. The Entity must have 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(name = "student")

public class Student

 {

}
```

Entity classes must not be declared final since JPA implementations try to subclass the entity in order to provide their functionality.

@Id

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.***

```
@Entity
public class Student

{

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

}
```

@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.

```
@Entity
```

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

publicclass Student

{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

privateintid;

}
```

@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***.

```
@Entity

@Table(name="STUDENT")

publicclass Student

{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

privateintid;


@Column(name = "firstname", length=50, nullable=false, unique=false)

private String firstName;

}
```

@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 as shown below-

```
@Entity

@Table(name=”STUDENT”)

publicclass 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
privateintage;
}
```

@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.

```
@Entity
@Table(name="STUDENT")
publicclass 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
privateintage;
@Temporal(TemporalType.DATE)
@Column(name = "dateofbirth", nullable = false)
private Date dob;
}
```

The types from the *java.time* package in **Java 8**is directly mapped to corresponding SQL types. So there's no need to explicitly specify *@Temporal* annotation:

- *LocalDate* is mapped to *DATE*

- *Instant*, *LocalDateTime*, *OffsetDateTime* and *ZonedDateTime* are mapped to *TIMESTAMP*
- *LocalTime* and *OffsetTime* are mapped to *TIME*

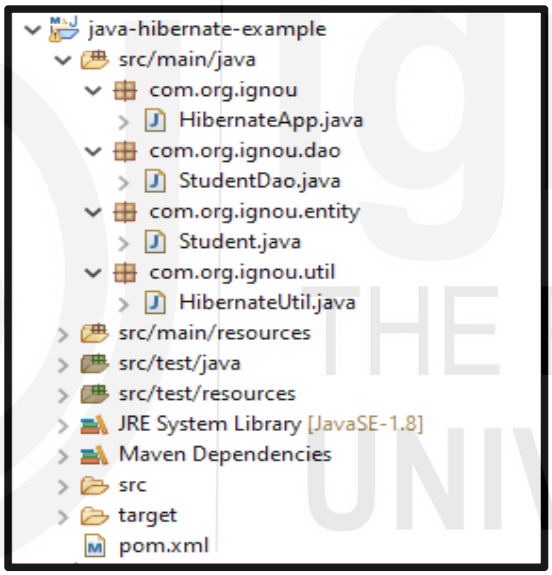
10.3.1 A Complete Hibernate Example

This section explains all steps required to create Hibernate Application with Mysql using Java configuration without using hibernate.cfg.xml. In the application, the Data Access Object (DAO) layer persists the Student entity and retrieves the persisted entity from the database. Required tools and technologies are as follows:

- IDE – Eclipse
- Hibernate 5.3.7.Final
- Maven
- MySQL 8
- JavaSE 1.8

Perform the following steps to create and run the Hibernate Application.

Step 1: Maven project is created into Eclipse IDE. The directory structure of the project with various layers is shown below-



Step 2: Create a maven project with the following dependencies in 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>
```

Step 3: Create Hibernate configuration file named as **HibernateUtil** with Java configuration. This contains the information about the database and JPA entity mapping.

```
publicclassHibernateUtil
{
    privatestaticSessionFactorysessionFactory;
    publicstaticSessionFactorygetSessionFactory()
    {
        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;
    }
}
```

In above configuration, list of possible options for **Environment.HBM2DDL_AUTO** are as followings-

- *validate*: validate the schema, makes no changes to the database.
- *update*: update the schema.

- *create*: creates the schema, destroying previous data.
- *create-drop*: drop the schema when the SessionFactory is closed explicitly, typically when the application is stopped.
- *none*: does nothing with the schema, makes no changes to the database

Step 4: Create a JPA Entity/Persistent class. This example creates the Student persistent class, which is mapped to a ***student*** database table. A Persistent class should follow some rules:

- **Prefer non-final class:** Hibernate uses proxies to apply some performance optimization. JPA entity or Hibernate Persistent class as final limits the ability of hibernate to use proxies. Without proxies, the application loses lazy loading which will cost performance.
- **A no-arg constructor:** Hibernate creates an instance of a persistent class using newInstance() method. Thus, a persistent class should have a default constructor at least package visibility.
- **Identifier Property:** Each JPA entity should have a Primary Key. Thus an attribute should be annotated with @Id annotation.
- **Getter and Setter methods:** Hibernate recognizes the method by getter and setter method names by default.

```
/**
 * @authorRahulSingh
 */
@Entity
@Table(name="student")
publicclass 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
privateintage;
@Temporal(TemporalType.DATE)
@Column(name = "dateofbirth", nullable = false)
private Date dob;
private String email;
publicStudent()
  {
  }
publicStudent(String firstName, String lastName, String email, Date dob)
  {
this.firstName = firstName;
this.lastName = lastName;
this.email = email;
this.dob = dob;
  }
publicintgetId()
  {
returnid;
  }
publicvoidsetId(intid)
  {
this.id = id;
  }
public String getFirstName()
  {
returnfirstName;
  }
publicvoidsetFirstName(String firstName)
  {
this.firstName = firstName;
  }
public String getLastName()
{
returnlastName;
```

```
    }
publicvoidsetLastName(String lastName)
  {
this.lastName = lastName;
    }
public String getEmail()
  {
returnemail;
    }
publicvoidsetEmail(String email)
  {
this.email = email;
    }
publicintgetAge()
  {
      returnage;
    }
   publicvoidsetAge(intage)
  {
this.age = age;
    }
public Date getDob()
  {
       return dob;
    }
publicvoidsetDob(Date dob)
  {
this.dob = dob;
    }
@Override
   public String toString()
  {
return"Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + ", age=" + age + ", dob="
                                                + dob + ", email=" + email + "]";
    }
}
```

Step 5: Create a DAO layer that performs the operations to the database. StudentDao code is given below, which performs two operations.

- save(Student student) – saves a given Student into the database.
- getStudentList() – retrieves all students from the database.

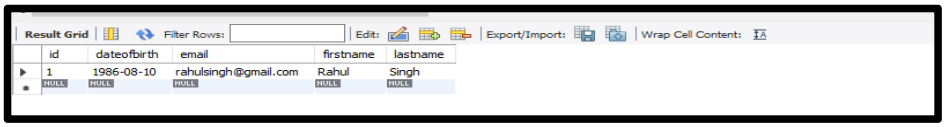
```
publicclassStudentDao

{

      publicvoidsave(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())
            {
                returnsess.createQuery("from Student", Student.class).list();
            }
        }
}
```

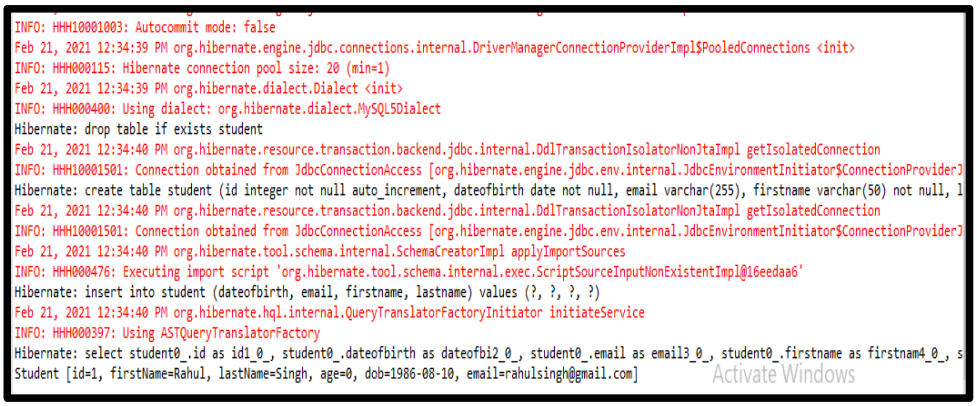
Step 6: Create the main class named as HibernateApp

```
publicclassHibernateApp
{
publicstaticvoidmain(String[] args) throws Exception
  {
StudentDaostudentDao = newStudentDao();
SimpleDateFormatdateFormat = newSimpleDateFormat("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));
  }
}
```

Step 7: Run the application from eclipse IDE or command line. Check the logs in the console and record into the database. The application is executed with eclipse IDE. It saves student records into a database. Console log with insert and select query generated by Hibernate as shown below:



Output



10.4 HIBERNATE CRUD (CREATE, READ, UPDATE AND DELETE) FEATURES

This section describes the persistence context. It also explains how hibernate uses the concept of persistence context to solve the problem of managing the entities at runtime.

The persistence context sits between the application and database store. Persistence context keeps monitoring the managed entity and marks the entity as dirty if some modification has been done during a transaction. Persistence context can be considered as a staging area or first-level cache where all the loaded and saved objects to the database reside during a session.

The instance of ***org.hibernate.Session***represents the persistence context in Hibernate. Similarly, the instance of ***javax.persistence.EntityManager***represents the persistence context in JPA. When Hibernate is used as a JPA provider, EntityManager interface is used to perform database related operations. In this case, basically ***java.persistence.EntityManager*** wraps the session object. Hibernate Session has more capability than JPA EntityManager. Thus, sometimes it is useful to work directly with Session.

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 and depicted in Figure 10.4.

1. Transient
2. Persistent
3. Detached
4. Removed

Transient: An object of a POJO class is in a transient state when created using a new

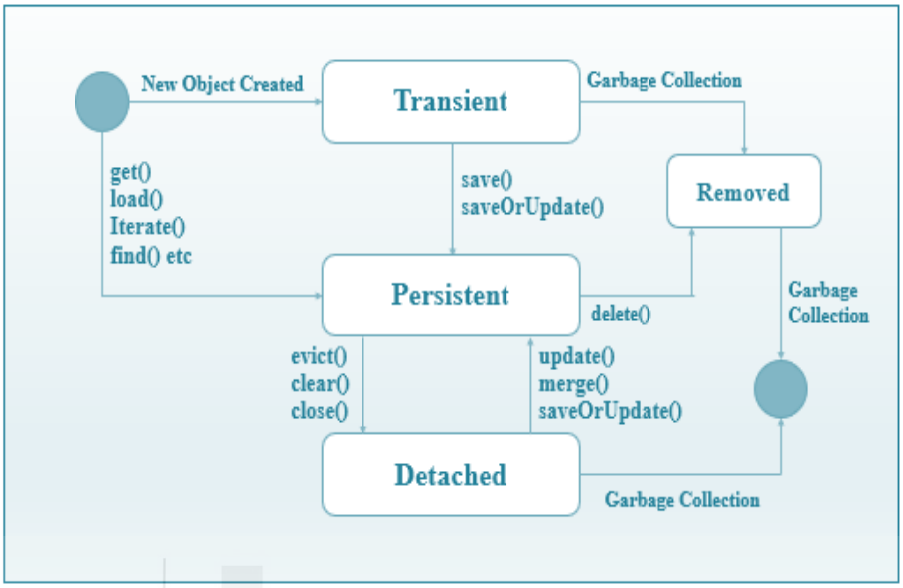


Figure 10.4: Hibernate Entity Lifecycle States

operator in Java. In the transient state, the object is not related to any database table. An object in a transient state is not managed by session or Entity Manager. The transient state object is not tracked for any modification from the persistence context. Thus, any modification of a transient state object does not impact the database.

Persistent: When a Transient entity is saved, it is associated with a unique session object and it enters into a Persistent state. A Persistent state entity is a representation of a row into a database, although the row might not exist in the database yet; upon flushing the session, the entity is guaranteed to have a corresponding row into the database. Session manages the Persistent state objects and keeps track of every modification done. Session propagates all the modifications into the database automatically. The Hibernate session's following methods make an entity into Persistent state.

- session.save()
- session.update()
- session.saveOrUpdate()
- session.lock()
- session.merge()

```
publicclassPersistentStateExample
{
        publicstaticvoidmain(String[] args)
        {
                Transaction txn = null;
                try (Session sess = HibernateUtil.getSessionFactory().openSession())
                {
                        txn = sess.beginTransaction(); // start a transaction
                        SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
                        Date dob = dateFormat.parse("10-09-1995");
                        Student student = newStudent("Rahul", "", "rahulsingh@gmail.com", dob); // 1
                        sess.saveOrUpdate(student); // 2
                        List<Student>students = sess.createQuery("from Student", Student.class).list();
                        students.forEach(s ->System.out.println(s)); // 3
                        student.setLastName("Singh"); //4
                        students = sess.createQuery("from Student",Student.class).list();
                        students.forEach(s ->System.out.println(s)); //5
                        txn.commit(); // 6
                }
                catch (Exception e)
```

```
                {
                        if (txn != null)
                        {
                                txn.rollback();
                        }
                        e.printStackTrace();
                }
        }
}
```

Output:

```
Hibernate: drop table if exists student
Feb 24, 2021 1:01:59 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Hibernate: create table student (id integer not null auto_increment, dateofbirth date not null, email varchar(255), firstname varchar(50) not null, l
Feb 24, 2021 1:01:59 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Feb 24, 2021 1:02:00 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@77d680e6'
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
Feb 24, 2021 1:02:00 PM org.hibernate.hql.internal.QueryTranslatorFactoryInitiator initiateService
INFO: HHH000397: Using ASTQueryTranslatorFactory
Hibernate: select student0_.id as id1_0_, student0_.dateofbirth as dateofbi2_0_, student0_.email as email3_0_, student0_.firstname as firstnam4_0_, s
Student [id=1, firstName=Rahul, lastName=, age=0, dob=Sun Sep 10 00:00:00 IST 1995, email=rahulsingh@gmail.com]
Hibernate: update student set dateofbirth=?, email=?, firstname=?, lastname=? where id=?
Hibernate: select student0_.id as id1_0_, student0_.dateofbirth as dateofbi2_0_, student0_.email as email3_0_, student0_.firstname as firstnam4_0_, s
Student [id=1, firstName=Rahul, lastName=Singh, age=0, dob=Sun Sep 10 00:00:00 IST 1995, email=rahulsingh@gmail.com]
```

Check the following code into the above example code.

1. The ***student*** object created using the ***new*** operator is in a transient state.
2. session.saveOrUpdate() method makes the **student** object into persistent state. Once the object is in persistent state, Session keeps track of any modification and automatically flushes the changes into the database.
3. All the records of the student table are being rendered. Notice that at this point lastName as empty for id 1.
4. The student's last name is updated. The student entity is in the persistent state, and Session keeps track of this change and updates the database automatically. The update statement is issued and can be found in the log.
5. All the records of the student table are being rendered. Check at this point lastNameis updated for id 1.
6. The transaction is committed, and all required changes are synced with the database.

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.

In order to persist the changes of entity, detached entities must be re-attached to the valid Hibernate Session. The detached state entity can be re-attached to the Hibernate Session with the call of following methods-

- session.update()
- session.merge()
- session.saveOrUpdate()

```
publicclassDetachedStateExample
{
        publicstaticvoidmain(String[] args)
        {
                try
                {
                        Session sess = HibernateUtil.getSessionFactory().openSession();
                        SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
                        Date dob = dateFormat.parse("10-09-1995");
                        Student student = newStudent("Rahul", "", "rahulsingh@gmail.com", dob);
```

```
                    sess.saveOrUpdate(student); //student entity is in persistent state
                    List<Student>students = sess.createQuery("from Student",
Student.class).list();
                    sess.close(); // session close() makes student entity in detached state.
                    students.forEach(s ->System.out.println(s));
                    student.setLastName("Singh"); //This change is not tracked by Hibernate
session.
                    sess = HibernateUtil.getSessionFactory().openSession();
                    sess.update(student); // session update() makes the detached object into
persistent state.
                    students = sess.createQuery("from Student",Student.class).list();
                    students.forEach(s ->System.out.println(s));
                    sess.close();
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
        }
}
```

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. At this state, java instance exists but any change made to the instance will not be saved into the database. If the removed entities are not referenced anywhere, then it is eligible for garbage collection.

The following sections explain the various Hibernate operations and differences between the same types of operations.

10.4.1 Hibernate Get Entity – get vs load

The Hibernate provides session.load() and session.get() methods to fetch the entity by id. This section explains both the methods with examples and differences between them.

Session load() method: 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. Some important overloaded methods are as follows:

- public Object load(Class<T>theClass, Serializable id) throws HibernateException
- public Object load(String entityName, Serializable id) throws HibernateException
- public void load(Object object, Serializable id) throws HibernateException

As we see that return type of load() method is either Object or void. Thus, we need to typecast the returned object with a suitable type of class. There are other overloaded methods of load(), which requires lock mode as an argument too. Hibernate picks the correct lock mode for us. Thus, very rarely an overloaded load() method with lock mode argument is used.

```
publicclassEntityLoadExample

{

    publicstaticvoidmain(String[] args)
    {
        try
        {
```

```
            Session sess = HibernateUtil.getSessionFactory().openSession();
            SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
            Date dob = dateFormat.parse("10-09-1995");
            Student student = newStudent("Rahul", "Singh", "rahulsingh@gmail.com", dob);
            sess.save(student); //student entity is in persistent state
            sess.close();

            intstudentId = student.getId(); //Id has been generated and set on call of save method.

            // First Overloaded method example
            Session sess1 = HibernateUtil.getSessionFactory().openSession();
            Student student1 = (Student)sess1.load(Student.class, studentId);
            System.out.println(student1);
            sess1.close();

            // Second Overloaded method example
            Session sess2 = HibernateUtil.getSessionFactory().openSession();
            Student student2 = (Student)sess2.load("com.org.ignou.entity.Student", studentId);
            System.out.println(student2);
            sess2.close();

            // Third Overloaded method example
            Session sess3 = HibernateUtil.getSessionFactory().openSession();
            Student student3 = newStudent();
            sess3.load(student3, studentId);
            System.out.println(student3);
            sess3.close();

        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Session get() method: Theget() 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. Some important overloaded methods are as follows:

- public Object get(Class<T>clazz, Serializable id) throws HibernateException
- public Object get(String entityName, Serializable id) throws HibernateException

Difference between load() and get() methods

Although load() and get() methods perform the same task, still the differences exist in their return values in case the identifier does not exist in the database.

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

10.4.2 Hibernate Insert Entity – persist, save and saveOrUpdate

The Session interface into Hibernate provides a couple of methods to transit an object from transient state or detached state to persistent state e.g. persist(), save(), and saveOrUpdate(). These methods are used to store an object in a database. There are significant differences among these methods.

Session persist() method: 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**. The semantics of ***persist()*** method is as follows:

- A transient state object becomes a persistent object, and the operation cascades all of its relation with cascade=PERSIST or cascade=ALL. The spec does not say that persist() method will generate the **Id** right away but on commit or flush,**non-null Id** is guaranteed. The persist() method does not promise the Id as non-null just after calling this method. Thus one should not rely upon it.
- The persist() method has no impact on persistent objects, but the operation still cascaded all of its relation with cascade=PERSIST or cascade=ALL.
- The persist() method on detached objects throws an exception, either upon calling this method or upon committing or flushing the session.

A sample example explains the above points with output. For better understanding, have hands-on with the following code.

```
publicclassEntityPersistExample
{
    publicstaticvoidmain(String[] args)
    {
        try
        {
            Session sess = HibernateUtil.getSessionFactory().openSession();
            Transaction txn = sess.beginTransaction();
            SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
            Date dob = dateFormat.parse("10-09-1995");
            Student student = newStudent("Abhishek", "Singh", "abhisheksingh@gmail.com", dob);
            sess.persist(student); //student entity is transitioned from transit -> persistent
            sess.persist(student); // No impact
            sess.evict(student); // student entity is transitioned from persistent -> detached

            sess.persist(student); // Exception. Since persistent state object is being persisted.
            txn.commit();
            sess.close();
        }
         catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Output:

```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate: drop table if exists student
Feb 27, 2021 5:45:56 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProvider
Hibernate: create table student (id integer not null auto_increment, dateofbirth date not null, email varchar(255), firstname varchar(50) not null,
Feb 27, 2021 5:45:56 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProvider
Feb 27, 2021 5:45:57 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@77d680e6'
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
javax.persistence.PersistenceException: org.hibernate.PersistentObjectException: detached entity passed to persist: com.org.ignou.entity.Student
        at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:154)
        at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:181)
        at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:188)
        at org.hibernate.internal.SessionImpl.firePersist(SessionImpl.java:807)
        at org.hibernate.internal.SessionImpl.persist(SessionImpl.java:785)
        at com.org.ignou.EntityPersistExample.main(EntityPersistExample.java:26)
Caused by: org.hibernate.PersistentObjectException: detached entity passed to persist: com.org.ignou.entity.Student
        at org.hibernate.event.internal.DefaultPersistEventListener.onPersist(DefaultPersistEventListener.java:127)
        at org.hibernate.event.internal.DefaultPersistEventListener.onPersist(DefaultPersistEventListener.java:62)
        at org.hibernate.internal.SessionImpl.firePersist(SessionImpl.java:800)
```

Session save() method: 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. The semantics of ***save()*** method is as follows:

- A transient state object becomes a persistent object and the operation cascades all of its relation with cascade=PERSIST or cascade=ALL. At first it assigns the generated id to the identifier and then persists the entity.
- The ***save()*** method has no impact on persistent objects, but the operation still cascaded all of its relations with cascade=PERSIST or cascade=ALL.
- The ***save()*** method on detached instances creates a new persistent instance. It assigns the new generated Id to the identifier, which results in a duplicate record in the database upon committing or flushing.

A sample example explains the above points with output as shown below:

```
publicclassEntitySaveExample
{
    publicstaticvoidmain(String[] args)
    {
        try
        {
            Session sess = HibernateUtil.getSessionFactory().openSession();
            Transaction txn = sess.beginTransaction();
            SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
            Date dob = dateFormat.parse("10-09-1995");
            Student student = newStudent("Abhishek", "Singh", "abhisheksingh@gmail.com", dob);
            Integer id1 = (Integer) sess.save(student); //student entity is transitioned from transit ->
persistent
            System.out.println("Id1: "+id1);
            Integer id2 = (Integer) sess.save(student); // No impact
            System.out.println("Id2: "+id2);
            sess.evict(student); // student entity is transitioned from persistent -> detached

            Integer id3 = (Integer) sess.save(student); // New Id generated and assigns it to identifier
            System.out.println("Id3: "+id3);
            txn.commit();
            sess.close();
        }
        catch (Exception e)
        {
           e.printStackTrace();
        }
    }
}
```

Execution Result:

```
INFO: HHH10001003: Autocommit mode: false
Feb 27, 2021 7:30:21 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
Feb 27, 2021 7:30:21 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate: drop table if exists student
Feb 27, 2021 7:30:22 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProvider]
Hibernate: create table student (id integer not null auto_increment, dateofbirth date not null, email varchar(255), firstname varchar(50) not null, l
Feb 27, 2021 7:30:22 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProvider]
Feb 27, 2021 7:30:22 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@77d680e6'
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
Id1: 1
Id2: 1
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
Id3: 2
```


Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

id	dateofbirth	email	firstname	lastname
1	1995-09-10	abhisheksingh@gmail.com	Abhishek	Singh
2	1995-09-10	abhisheksingh@gmail.com	Abhishek	Singh
NULL	NULL	NULL	NULL	NULL

It can be observed in the above log that the detached instance is saved into the database with a newly generated id and which results in a duplicate record into the database as shown in the database screenshot.

Session saveOrUpdate() method: 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. This method can be considered as a universal tool to make an object persistent regardless of its state, whether it is transient or detached.

```
publicclassEntitySaveOrUpdateExample
{
    publicstaticvoidmain(String[] args)
    {
        try
        {
            Session sess = HibernateUtil.getSessionFactory().openSession();
            Transaction txn = sess.beginTransaction();
            SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
            Date dob = dateFormat.parse("10-09-1995");
            Student student = newStudent("Abhishek", "Singh", "abhisheksingh@gmail.com", dob);
            sess.saveOrUpdate(student);    //student entity is transitioned from transit -> persistent
            System.out.println("Id1: "+student.getId());
            sess.saveOrUpdate(student);   // No impact
            System.out.println("Id2: "+student.getId());

            sess.evict(student);   // student entity is transitioned from persistent -> detached

            sess.saveOrUpdate(student); // Since Id exist, thus only update operation is performed on
detached entity
            System.out.println("Id3: "+student.getId());
            txn.commit();
            sess.close();
        }
        catch (Exception e)
        {
           e.printStackTrace();
        }
    }
```

```
}
```

Program Output:

```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate: drop table if exists student
Feb 27, 2021 11:57:31 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProvide
Hibernate: create table student (id integer not null auto_increment, dateofbirth date not null, email varchar(255), firstname varchar(50) not null,
Feb 27, 2021 11:57:32 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProvide
Feb 27, 2021 11:57:32 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@77d680e6'
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
Id1: 1
Id2: 1
Id3: 1
Hibernate: update student set dateofbirth=?, email=?, firstname=?, lastname=? where id=?
```


As in output, you can see that saveOrUpdate() operation does not result in duplicate records even for the detached entity . Thus, this operation can be used as a universal tool to make an object persistent regardless of its state.

10.4.3 Hibernate Query Language (HQL)

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 queries for Select, Update, Delete and Insert is explained below.

HQL Select query to retrieve the student data of id 1.

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

HQL Update query to update the student's last name as 'Singh' whose id 1.

```
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 to delete the student where id is 1.

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

HQL Insert

In HQL, only the INSERT INTO … SELECT … is supported; there is no INSERT INTO … VALUES. HQL only support insert from another table. For example, insert student records from another student_backup table. This is also called bulk-insert statement.

```
Query query = session.createQuery("Insert into Student(firstName, lastName, emailId)"+
                                  "Select firstName, lastName, emailed from
```

```
student_backup");
int result = query.executeUpdate();
```

☛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.

……

……

……

……

10.5 SPRING DATA JPA OVERVIEW

A well-structured codebase of a web project consists of various layers such as Controller, Service and DAO (data access object) layer. Each layer has its own responsibilities. While developing a web application, business logic should be focused instead of technical complexity and boilerplate code. The DAO layer usually consists of lots of boilerplate code, and that can be simplified. The simplification reduces the number of artifacts that need to be defined and maintained. It also makes the data access pattern and configuration consistent.

Different storage technologies require different API's to access data, different configurations and different methods. JPA handles the object-relational mappings and technical complexities in JDBC based database interaction. The JPA provides standardization across SQL data stores. The Java Persistence API (JPA) specification and Spring Data JPA are very popular to simplify the DAO layer. Before the explanation of Spring Data JPA, the following section explains the relationship between Spring Data JPA, JPA and Hibernate.

Java Persistence API (JPA) is just a specification that defines APIs for object-relational mappings and management of persistent objects. It is a set of interfaces that can be used to implement the persistence layer. It itself doesn't provide any

implementation classes. In order to use the JPA, a provider is required which implements the specifications. Hibernate and EclipseLink are popular JPA providers.

Spring Data JPA is one of the many sub-projects of **Spring Data**that 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 a complete abstraction over the DAO layer into the project.

Spring Data JPA introduces the concept of **JPA Repositories**. JPA Repositories are a set of Interfaces that defines query methods. It does not require writing native queries anymore. Sometimes custom queries may be required in order to fetch the data from the database. These queries are JPQL (Java Persistence Query Language), not native queries. 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. The interfaces only define query methods and spring provides proxy implementation at runtime. With Spring Data JPA, we don't have to write SQL statement, instead we just need to extend the interface defined by Spring Data JPA for one of the entities. Based on JPA specification, the underlying JPA implementation enables the *Entity* objects and their metadata mapping. It also enables the entity manager who is responsible for persisting and retrieving entities from the database.

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. If the custom query isn't complex, only a method is required to be defined in the repository interface with a name starting with ***find…By***. Here is an example of a Student Repository to find a student by id and list of students based on firstName and lastName.

```
public interface StudentRepository extends CrudRepository<Student, Long>
{
   Optional<Student>findById(Long studentId);
   List<Student>findByFirstNameAndLastName(String firstName, String lastName);
}
```

1. The first method corresponds to **select * from student where id = ?**
2. The second method corresponds to **select * from student where firstName= ?andlastName = ?**

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 your DAO layer integrating and interacting 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. We only need to provide the data source details and the rest of the things like configuring and using *EntityManager*.

☛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.

……

……

……

……

10.6 SUMMARY

This unit has explained the concepts of ORM, Hibernate as ORM tool and Spring Data JPA, which uses Hibernate as the default JPA provider. Below are the important things which have been explained.

- Many different data storage technologies are available in this world, and each one comes with its own data access API and driver.
- ORM stands for Object Relational Mapping, where a Java object is mapped to a database table and with APIs.We need to work with objects and not with native queries.
- The Java Persistence API (JPA) is a specification that defines how to persist data in Java applications
- The Hibernate architecture elements includ such as SessionFactory, Session, Query, First Level Cache and Second Level Cache. The first level cache can't be disabled.
- In order to use optimized performance of Hibernate, a POJO should follow some rules such as-
  - Prefer non-final class
  - A no-arg constructor
  - Identifier property
  - Getter and Setter method
- The Hibernate entity lifecycle states as transient, persistent, detached and removed.

- The Hibernate Session persist(), and the save() methods perform the same job still; their implementation differs in terms of generated Id assignment.
- The **saveOrUpdate()** method can be used as a universal tool for persisting an entity irrespective of entity state.
- Spring Data JPA takes one step forward and enables developers to use data access layers without any implementation.

10.7 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) Hibernate is an open-source Java persistence framework created by Gavin King in 2011. It simplifies the development of Java applications to interact with databases. Hibernate is a lightweight, powerful, and high-performance ORM (Object Relational Mapping) tool that simplifies data creation, manipulation, and access.
The Java Persistence API (JPA) is a specification that defines how to persist data in Java applications. Hibernate is a popular ORM which is a standard implementation of the JPA specification with a few additional features specific to Hibernate. Hibernate lies between Java Objects and database servers to handle all persistence and retrieval of those objects using appropriate mechanisms and patterns. There are several advantages of Hibernate, such as:
    - Open Source
    - High Performance
    - Light Weight
    - Database independent
    - Caching
    - Scalability
    - Lazy loading
    - Hibernate Query Language (HQL)

2) Hibernate has four-layered architecture. The Hibernate includes many objects such as persistent objects, sessionfactory, session, connection factory, transaction factory, transaction etc. A few important interfaces have been explained.

SessionFactory
The Hibernate creates an immutable SessionFactory object using the Configuration object. SessionFactory is used to instantiate the session object in thread to service client requests. SessionFactory object is thread-safe and used by all the threads of an application.
SessionFactory object is per database using a separate configuration file. Thus, for multiple databases, we will require to create multiple SessionFactory objects. The SessionFactory is a heavyweight object, and it is created at the time of application startup.

Session
The session object provides an interface to interact with the database from an application. A Session object 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 methods for transaction management. Transaction objects enable data consistency, and rollback in case something goes wrong.

3) The cache is a mechanism that improves the performance of any application. Hibernate provides a caching facility also. First-level cache is enabled by default, and it can't be disabled. Hibernate ORM reduces the number of queries made to a database in a single transaction using First-level cache. First-level cache is associated with **Session object,** and it is available till session object is alive. First-level cache associated with session objects is not accessible to any other session object in other parts of the application. Important facts about First-level cache is as follows:
    - First-level cache scope is session. Once the session object is closed, the first-level cache is also destroyed.
    - First-level cache can't be disabled.
    - First time query for an entity into a session is retrieved from the database and stored in the First-level cache associated with the hibernate session.
    - Query for an object again with the same session object will be loaded from cache.
    - Session evict() method is used to remove the loaded entity from Session.

    Session's clear() method is used to remove all the entities stored into cache.

4) Second-level cache is used globally in **Session Factory** scope. **Once session factory is closed, all cache associated with it die**, and the cache manager also closed down. **Check section 9.2.1 for the working of Second-level cache.**

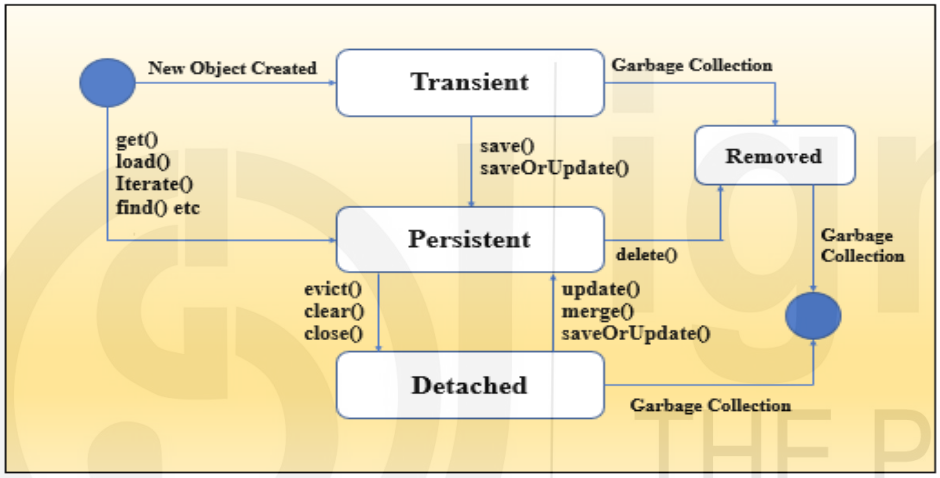
Check Your Progress 2

1) 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.
2) A Persistent class should follow the following guidelines:
    - **Prefer non-final class:** Hibernate uses proxies to apply some performance optimization. JPA entity or Hibernate Persistent class as final limits the ability of hibernate to use proxies. Without proxies, application loses lazy loading, which will cost performance.
    - **A no-arg constructor:** Hibernate creates an instance of a persistent class using newInstance() method. Thus, a persistent class should have default constructor at least package visibility.
    - **Identifier Property:** Each JPA entity should have a primary Key. Thus, an attribute should be annotated with @Id annotation.
    - **Getter and Setter methods:** Hibernate recognizes the method by getter and setter method names by default.

3) Hibernate works with POJO. Without any Hibernate specific annotation and mapping, Hibernate does not recognize these POJOs. 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.
Same as Figure 10.5: Hibernate Entity Lifecycle States

1. Transient
2. Persistent
3. Detached
4. Removed

4) Check section 10.4.3



Check Your Progress 3

1) **Difference between load() and get() methods**

Although load() and get() methods perform the same task, still the differences exist in their return values in case the identifier does not exist in the database.

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

2) Differences between save() and persist() methods are listedin the below table.

S.NO	Key	Save()	Persist()
1	Basic	It stores object in database	It also stores object in database
2	Transaction Boundaries	It can save object within boundaries and outside boundaries	It can only save object within the transaction boundaries
3	Return Type	It returns generated id and return type is serializable	It does not return anything. Its return type is void.

4	Detached Object	It will create a new row in the table for detached object	It will throw persistence exception for detached object
5	Supported by	It is only supported by Hibernate	It is also supported by JPA

3) This saveOrUpdate() method either performs**save()** or **update()** based on identifier existence. E.g. If an identifierexists 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. This method can be considered as a universal tool to make an object persistent regardless of its state,whether it is transient or detached.

4) 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.

10.8 REFERENCES/FURTHER READING

- Craig Walls, "Spring Boot in action" Manning Publications, 2016. (https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf)

- Christian Bauer, Gavin King, and Gary Gregory, "**J**ava Persistence with Hibernate",Manning Publications, 2015.
- Ethan Marcotte, "Responsive Web Design", Jeffrey Zeldman Publication, 2011(http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf)
- Tomcy John, "Hands-On Spring Security 5 for Reactive Applications",Packt Publishing,2018
- https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/tutorial.html
- https://www.w3spoint.com/get-vs-load-hibernate
- https://www.baeldung.com/learn-jpa-hibernate
- https://www.javatpoint.com/hibernate-with-annotation
- https://www.tutorialspoint.com/jpa/index.htm
- https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/
- https://docs.jboss.org/hibernate/orm/3.6/quickstart/en-US/html/hibernate-gsg-tutorial-jpa.html
- https://www.baeldung.com/hibernate-save-persist-update-merge-saveorupdate
- https://www.journaldev.com/3472/hibernate-session-get-vs-load-difference-with-examples
- https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa
- https://dzone.com/articles/spring-data-jpa