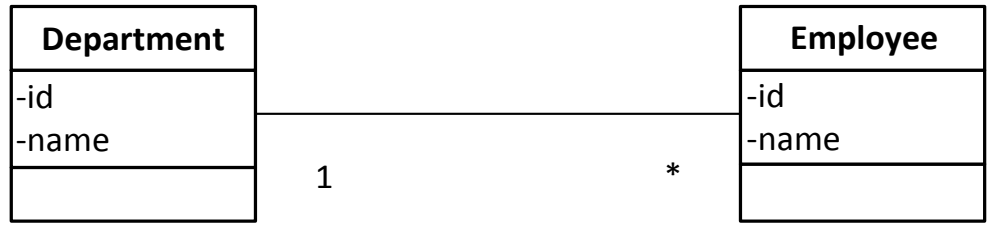
Hibernate

Index

What is ORM?

- Enterprise Applications typically save state in relational database. E,g., eCommerce portals like Flipkart.com store purchase order information
- However, applications are developed using object oriented language like Java, C#, etc.
- That is application developers code using objects which reside in computer memory
- At the end of the transaction they need to be persisted into RDBMS so that orders can be retrieved and fulfilled later.
- Object Relational Mapping (ORM) is all about manipulating persistent data to Relational Database

What is ORM?



- A Department can have multiple employees
- An employee can belong to one department only

	Physical Name	Data Type	Req'd	PK	
▶	dept_id	INTEGER	☑	☑	dept_id identifies Department
	dept_name	VARCHAR(125)	☑	☐	name of Department

	Physical Name	Data Type	Req'd	PK	
▶	emp_id	INTEGER	☑	☑	emp_id identifies Employee
	dept_id	INTEGER	☑	☐	Foreign Key to Department
	emp_name	VARCHAR(125)	☑	☐	Name of Employee

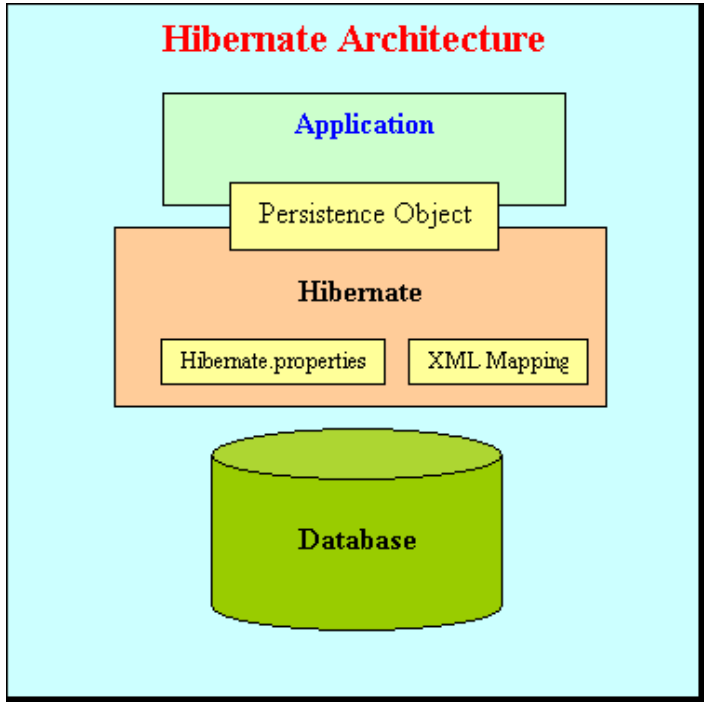
Saving data with JDBC

- Obtain a java.sql.Connection, java.sql.Statement objects
- Generate the SQL query, manually
- Execute the Query !
- Remember to close all objects (else face problems)

```
Connection conn = null;
Statement stmt = null;
try {
    Class.forName("org.hsqldb.jdbcDriver");
    conn = DriverManager.getConnection("jdbc:hsqldb:file:d:/test2db", "sa", "");
    stmt = conn.createStatement();
    String sql = "INSERT INTO <TABLE> (col1, col2, col3) VALUES(colVal1, colVal2, colVal3)";
    stmt.executeUpdate(sql);
} catch (Exception e) {
    //handle exception
} finally {
    if(stmt!=null)
        stmt.close();
    if(conn!null)
        conn.close();
}
```

What is Hibernate?

- ORM Tool - O-R mapping using ordinary JavaBeans
- Map metadata either with Annotations or in XML configuration files
- Used in data layer of application
- Transaction management with rollback
- Implements JPA



What is Hibernate?

- Advantages of Hibernate
  - Transparent persistence without ever coding in JDBC
  - Developer can concentrate on business logic rather than writing JDBC code
  - Optimized performance - can cache frequently read data in application
- Disadvantages
  - Extra learning curve

Important Hibernate API

- **Session** (org.hibernate.Session)
  - A single-threaded, short-lived object representing a conversation between the application and the persistent store.
  - Wraps a JDBC connection. Factory for Transaction.
- **SessionFactory** (org.hibernate.SessionFactory)
  - A factory for Session
- **Transaction** (org.hibernate.Transaction)
  - A single-threaded, short-lived object used by the application to specify atomic units of work.
  - Abstracts application from underlying JDBC, JTA or CORBA transaction.
  - A Session might span several Transactions
- Persistent objects and collections
  - Short-lived, single threaded objects containing persistent state and business function like **Department**, **Employee**, etc.

Hibernate Annotations

- Hibernate requires metadata that governs the transformation of data from java object database representation and vice versa. This can by specified using annotations on entity bean class.

Annotation	Description
@Entity	To declare a pojo class as an ENTITY bean
@Table	Declare the database TABLE name for the entity bean
@Column	Declare COLUMN for an attribute of entity beans
@Id	Declares the identifier property the entity bean (PK in the table)
@GeneratedValue	Identifier generation strategy - AUTO, TABLE, SEQUENCE, etc
@OneToOne	Declare One-to-one relationship with another object
@OneToMany	Declare One-to-many relationship with another object
@OneToMany	Declare Many-to-many relationship with another object
@JoinColumn	Declare join column, (foreign key in table)

Hibernate Configuration

- Declare hibernate **dependencies** in pom.xml or classpath make sure right libraries are downloaded and configured
- Configure database properties in hibernate.cfg.xml
  - Make note of **show_sql** property

```
<hibernate-configuration>
<session-factory>
<property
name="hibernate.connection.driver_class">or
acle.jdbc.driver.OracleDriver</property>
<property
name="hibernate.connection.url">jdbc:oracle
:thin:@127.0.0.1:1521:xe</property>
<property
name="hibernate.connection.username">root1<
/property>
<property
name="hibernate.connection.password">root1<
/property>
<property
name="hibernate.dialect">org.hibernate.dial
ect.Oracle10gDialect</property>
```

mysql

```
 <property
name="hibernate.connection.driver_class">co
m.mysql.jdbc.Driver</property>
        <property
name="hibernate.connection.url">jdbc:mysql:
//localhost:3306/testdb</property>
        <property
name="hibernate.connection.username">root</
property>
         <property
name="hibernate.connection.password">root</
property>
        <property
name="hibernate.dialect">org.hibernate.dial
ect.MySQLDialect</property>
        <property
name="hibernate.hbm2ddl.auto">update</prope
rty>
```

Create DB objects or have them generated

- Use the sql script to create database Tables (MySQL)

```
drop table if exists DEPT;
drop table if exists EMP;

create table DEPT (
    dept_id integer not null auto_increment,
    dept_name varchar(255),
    primary key (dept_id)
);

create table EMP (
    emp_id integer not null auto_increment,
    emp_name varchar(255),
    dept_id integer,
    primary key (emp_id)
);

alter table EMP
    add constraint FK0001
foreign key (dept_id)
references DEPT (dept_id);
```

Implement Entity Classes as POJOs

- Specify O-R metadata with Annotations

```
import javax.persistence.Column;
import javax.persistence.Entity;
 import javax.persistence.GeneratedValue;
 import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
@Entity
 @Table(name="employee")
public class Employee{
@Id @Column(name="id") @GeneratedValue(strategy =
GenerationType.AUTO)

 private long id; @Column(name="firstname")
private String firstName;

@Column(name = "salary") private double salary;
@JoinColumn(name="department")

@ManyToOne private Department department;
```

Implement test application

- Create, save and retrieve department & employee objects

```
    public static void main(String[] agrs) {
        Session session = getSessionFactory().openSession();
        //Begin Transaction
        session.beginTransaction();
        try {
            Department dept = new Department("java");
            session.save(dept);

            Employee joe = new Employee("Joe", dept);
            session.save(joe);

            Employee dan = new Employee("Dan", dept);
            session.save(dan);

            //Commit transaction
            session.getTransaction().commit();
        } catch (Exception ex) {
            session.getTransaction().rollback();
        }

        org.hibernate.Query q = session.createQuery("From Employee ");

        List<Employee> resultList = q.list();
        System.out.println("num of employess:" + resultList.size());
        for (Employee next : resultList) {
            System.out.println("next employee: " + next.getName());
        }

    }

}
```

Run test application

- To run the application, execute HibernateTest class
  - You shall see outputs on the console similar to the following
  - Make note of sql code is automatically generated by Hibernate

```
Hibernate: insert into DEPT (dept_name) values (?)
Hibernate: insert into EMP (dept_id, emp_name) values (?, ?)
Hibernate: insert into EMP (dept_id, emp_name) values (?, ?)
Hibernate: select employee0_.emp_id as emp1_0_, employee0_.dept_id as dept3_0_,
        employee0_.emp_name as emp2_0_ from EMP employee0_
number of employess:2
Employee: Joe
Employee: Dan
```

Building a session factory

- SessionFactory is a factory of sessions.
- A SessionFactory has the following features:
- It's an interface implemented using the singleton pattern.
- It's created using the configuration provided by the configuration file.
- It's thread-safe, so it's created once during the application's lifetime, and multiple users or threads can access it at the same time without any concurrency issue.

Session Factory

- As a SessionFactory object is immutable, changes made to the configuration will not affect the existing factory object.
- It's a factory class, and its main duty is to create, manage, and retrieve a session on request. A Session is used to get a physical connectivity with the database.

- If we want to connect two different databases in an application, we need to create two different SessionFactory objects in it.
- SessionFactory *factory = **new Configuration().configure().buildSessionFactory***

- ***Create a generic Session factory provider and place it in a Hibernate util class***

Opening a new Session

- A Session is also known as an interface that is used to get a physical connectivity with a database.
- It is instantiated every time we need to interact with the database for the CRUD (Create, Read, Update, Delete) operations.
- Persistent objects always travel from the application to the database and vice versa only through the Session.

Session

SessionFactory sessionFactory = HibernateUtil.getSessionFactory();

Session session = sessionFactory.openSession();

alternatively

Session session = sessionFactory.getCurrentSession() requires you to provide additional configuration if you have to reuse a Session:

<property name="hibernate.current_session_context_class">

Thread

</property>

where we associate the same session for a particular thread.

Saving an object to DB

- Get the SessionFactory.
- Open a new session.
- Begin a transaction.
- Create an employee object.
- Save an employee.
- Commit the transaction.
- Close the session.
- Close the SessionFactory.

```
SessionFactory sessionFactory =
HibernateUtil.getSessionFactory();
Session session = sessionFactory.openSession();

// begin a transaction
session.getTransaction().begin();


//creating an employee object
Employee employee = new Employee();
employee.setFirstName("aarush");
employee.setSalary(35000);

// save employee object
session.save(employee);


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

session.close();
HibernateUtil.shutdown();
```

Retrieve

- SessionFactory sessionFactory = HibernateUtil.getSessionFactory();
- Session session = sessionFactory.openSession();
- Employee employee = (Employee) session.get(Employee.class, 1l);
- if(employee != null){
- System.out.println(employee.toString());
- }

- session.close();
- HibernateUtil.shutdown();

Get, load methods

- get() method hits the db retrieves the actual record eagerly
- load() returns a proxy lazily and queries based on invocations on tht object – good for performance
- When there is no data in db for the given id, get returns null
- When there is no data in db for the given id, load throws ObjectNotFoundException

Removing an object

```
SessionFactory sessionFactory =
HibernateUtil.getSessionFactory();
Session session = sessionFactory.openSession();

session.getTransaction().begin();
Employee employee = (Employee) session.get(Employee.class,
new Long(1));
session.delete(employee);
session.getTransaction().commit();

session.close();
HibernateUtil.shutdown();
```

update

```
SessionFactory sessionFactory =
HibernateUtil.getSessionFactory();
Session session = sessionFactory.openSession();

Employee employee = (Employee) session.get(Employee.class,
new Long(2));


session.getTransaction().begin();
 employee.setFirstName("aarush_updated");
 session.update(employee);
session.getTransaction().commit();

session.close();
HibernateUtil.shutdown();
```

Criteria

- A criteria is an interface; it provides an API to perform WHERE, ORDER BY, LIMIT, result transformation

Select * from employees

```
Criteria criteria = session.createCriteria(Employee.class);
List<Employee> employees = criteria.list();
for(Employee employee : employees){
System.out.println(employee.toString());
}
```

restrictions

SELECT * FROM employee WHERE salary > 35000;

- Criteria criteria = session.createCriteria(Employee.class);
- criteria.add(Restrictions.gt("salary", 35000));
- List<Employee> employees = criteria.list();
- for (Employee employee : employees) {
- System.out.println(employee.toString());
- }

Operators

- There are many functions available in the class Restrictions.
- You can use logical operators such as:

- gt(>, greater than)
- ge(>=, greater than or equal to)
- lt(<, less than)
- le(<=, less than or equal to)
- eq(=, equal to)
- ne(<>, !=, not equal to)

More..

- like (to perform Like operation)
- iLike (to perform Like operation with ignore case)
- Not
- Between
- In
- Or
- isNull
- isNotNull
- isEmpty
- isNotEmpty

Basic annotations

- Declaring a class as an entity and creating a table in the database – @Entity and @Table
- Creating a column in the table – @Column
- Creating a primary key and composite primary key column – @Id and @IdClass
- Creating an autogenerator column

Primary key

- To declare a column as a primary key column, we use the @Id annotation

```
@Id
private long id;
```

- To declare it as a composite primary key, we will consider creating a composite primary key using the employee's first name and phone.

```
@Entity
@IdClass(Employee.class)
public class Employee implements Serializable {
  @Id
  private String firstName;

  @Id
  private String phone;
}
```

Generating a surrogate key

- Using a default generation strategy
- Using a sequence generator
- Using a table generator

Auto

```
@Id
@GeneratedValue
private long id;
```

By default, hibernate uses the GenerationType.AUTO strategy if no strategy is supplied; so, @GeneratedValue is equal to @GeneratedValue(strategy=GenerationType.AUTO).

Still, as it is database–specific, it's the responsibility of the database to provide a value for this column, and the same rule is applied for @GeneratedValue(strategy=GenerationType.IDENTITY).

Sequence

SEQUENCE GENERATOR
Here, we are using GenerationType.SEQUENCE in the @GeneratedValue annotation

```
@Id
@SequenceGenerator(name="seq",
sequenceName="DB_SEQ")
@GeneratedValue(strategy=GenerationType.SEQUENCE,
generator="seq")
private long id;
```

TABLE GENERATOR

In a table generator, the value for the primary key column is stored in one table.
Hibernate uses this table to get the next value for the primary key column in the particular class.

```
@Id
@Column(name = "id")
@GeneratedValue(strategy = GenerationType.TABLE, generator
= "gen_tbl")
@TableGenerator(name = "gen_tbl", table = "gen_table",
pkColumnName = "pk", valueColumnName = "id",
pkColumnValue = "employee0", initialValue = 0, allocationSize =
1)
private long id;
```

Save or saveOrUpdate or persist

- Hibernate Session class provides couple of ways to save object into database by methods like save, saveOrUpdate and persist.
- You can use either save(), saveOrUpdate() or persist() based upon your requirement for persisting object into Database.

- Main difference between save and saveOrUpdate method is that save() generates a new identifier and INSERT record into database while saveOrUpdate can either INSERT or UPDATE based upon existence of record.
- Clearly saveOrUpdate is more flexible in terms of use but it involves an extra processing to find out whether record already exists in table or not.

Save or persist ?

- Save returns a serialized object, persist returns none
- persist() method doesn't guarantee that the identifier value will be assigned to the persistent instance immediately, the assignment might happen at flush time.

Persisting collections

- Hibernate lets you persist collections that are not entities.
- Just annotate @ElementCollection over a list or set attribute.

Associations

- One-to-one mapping
- One-to-many mapping or many-to-one mapping
- Many-to-many mapping

One-to-One

- Person holds a passport detail
- In Person class

```
@OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "passport_detail_id")
  private PassportDetail passportDetail;
```

The @JoinColumn annotation is used to define the relationship between tables

Passport_detail_id is the column of passportdetail table

One-to-many

- Eg: A book has many reviews

Book class

```
@OneToMany(cascade=CascadeType.ALL,mappedBy="book")
private Collection<Review> reviews= new ArrayList<Review>();
```

- Review class

```
@ManyToOne
@JoinColumn(name="bookId")
private Book book;
```

Many-many

- One student –taught by- Many teachers
- One teacher – teaches – many students

Student

@ManyToMany(cascade=CascadeType.***ALL)***

private List<Teacher> teachers = new ArrayList<Teacher>();

Teacher

- @ManyToMany(mappedBy = "teachers",cascade=CascadeType.***ALL)***
- **private List<Student> students = new ArrayList<Student>();**

HQL

- HQL stands for Hibernate Query Language and is a fully object-oriented language.
- In HQL, we have to use a class name instead of a table name and a field name instead of a column name.

- HQL queries are converted to SQL queries by hibernate

```
Query query = session.createQuery("FROM Category");
List<Category> list = query.list();
System.out.println("Category size: " + list.size());
```

Inheritance

- three inheritance mapping strategies defined in hibernate:

- Table per class hierarchy
- Table per subclass
- Table per concrete class

Table per class

- Single table
- Parent, child objects mapped to different columns of same table
- Discriminator column specifies the type of record whether parent or child

```
@Inheritance(strategy =InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn( name="worker_type" , discriminatorType=DiscriminatorType.STRING
                , length=2)
            @DiscriminatorValue(value="W")
Public class Worker {}
In subclass
 @DiscriminatorValue(value="SW")
Public class SalariedWorker extends Worker {}
```

Table per subclass

- There are individual tables for every class and they are joined by a primary key relationship

@Inheritance(strategy=InheritanceType.***JOINED)***

public class Resource{ }

@PrimaryKeyJoinColumn(name="resId")

public class FullTime extends ITResource{ }

Table per concrete class

- One class per type, and fields are repeated, overriding fields are specified in the subclass

- @Inheritance(strategy=InheritanceType.***TABLE_PER_CLASS)***
- **public class Emp { }**

- @AttributeOverrides({
- @AttributeOverride(name="id", column = @Column(name="id")),
- @AttributeOverride(name="name", column = @Column(name="name"))
- })
- **public class Temp extends Emp{ }**

Thank you