Experiment-9

Student Name: Anushka Kotiyal **UID: 22BCS13559**
Branch: BE-CSE **Section/Group: 22BCS_KRG_IOT-3/B**
Semester: 6th **Date of Performance:16/04/25**
Subject: Project-Based Learning with Java **Subject Code: 22CSH-359**

Easy -Level

1. **Aim:** Create a simple Spring application that demonstrates Dependency Injection (DI) using Java-based configuration instead of XML. Define a Student class that depends on a Course class. Use Spring's @Configuration and @Bean annotations to inject dependencies.

   Requirements:
   - Define a Course class with attributes courseName and duration.
   - Define a Student class with attributes name and a reference to Course.
   - Use Java-based configuration (@Configuration and @Bean) to configure the beans.
   - Load the Spring context in the main method and print student details.

2. **Objective:** To build a simple Spring application using Java-based configuration that demonstrates Dependency Injection by injecting a Course object into a Student object using @Configuration and @Bean annotations.

3. **Implementation/Code:**

Course.java

```
 public class Course
 { private String
 courseName;
 private String
 duration;
 public Course(String courseName, String
   duration) { this.courseName = courseName;
   this.duration = duration;
}
public String
   getCourseName() {
   return courseName;
}
public String
   getDuration() {
   return duration;
}
public String toString() {
   return "Course: " + courseName + ", Duration: " + duration;
}
  }
```

Student.java

```
public class
Student { private
String name;
private Course
course;
 public Student(String name, Course course)
   { this.name = name;
   this.course = course;
 }

 public void showDetails() {
    System.out.println("Student: " + name);
    System.out.println(course);
 }
   }
```

AppConfig.java

```
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration
; @Configuration
public class AppConfig
{ @Bean
public Course course() {
   return new Course("Java", "3 months");
}
@Bean
public Student student() {
   return new Student("Anshika", course());
}
}
```

MainApp.java

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContex
t; public class MainApp {
   public static void main(String[] args) {
     ApplicationContext context = new
     AnnotationConfigApplicationContext(AppConfig.class); Student student =
     context.getBean(Student.class);
    student.showDetails();
}
}
```

4. **Output:**

```
Student: Anushka
Course: Java, 3 months
```

1. **Learning Outcomes:**

- Understand the concept of Dependency Injection (DI) in Spring Framework.
- Learn how to configure beans using Java-based configuration with @Configuration and @Bean.
- Gain practical experience in setting up and initializing a Spring application without XML.
- Learn how to manage object dependencies and lifecycle using the Spring container.
- Understand the relationship between components (Student and Course) and how DI promotes loose coupling.

Medium -Level

1. **Aim:** Develop a Hibernate-based application to perform CRUD (Create, Read, Update, Delete) operations on a Student entity using Hibernate ORM with MySQL.

   Requirements:
   - Configure Hibernate using hibernate.cfg.xml.
   - Create an Entity class (Student.java) with attributes: id, name, and age.
   - Implement Hibernate SessionFactory to perform CRUD operations.
   - Test the CRUD functionality with sample data.

2. **Objective:** To develop a Hibernate-based application that performs CRUD operations on a Student entity using Hibernate ORM with MySQL, configured through hibernate.cfg.xml, and managing data with the SessionFactory.

3. **Implementation/Code:**

hibernate.cfg.xml

```
<hibernate-configuration>
<session-factory>
  <property name="hibernate.connection.driver_class">com.mysql.cj.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">password</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
  <property name="hibernate.hbm2ddl.auto">update</property>
  <mapping class="Student"/>
</session-factory>
</hibernate-configuration>
```

Student.java

```
import javax.persistence.*;
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private int age;
    public Student() {}
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
```

```
    public void setAge(int age) {
        this.age = age;
    }
    public String toString() {
        return "Student [id=" + id + ", name=" + name + ", age=" + age + "]";
    }
    }
```

HibernateUtil.java

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateUtil {
    private static final SessionFactory sessionFactory;
    static {
        sessionFactory = new Configuration().configure().buildSessionFactory();
    }
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
    }
```

MainCRUD.java

```
import org.hibernate.*;
public class MainCRUD {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction tx = session.beginTransaction();
        Student s1 = new Student("Sallu", 22);
        session.save(s1);
        tx.commit();
        Student student = session.get(Student.class, 1);
        System.out.println(student);
        tx = session.beginTransaction();
        student.setAge(23);
        session.update(student);
        tx.commit();
        tx = session.beginTransaction();
        session.delete(student);
        tx.commit();
        session.close();
    }
}
```

4. Output:

```
Student{id=1, name=Anushka',
age=21)
Updated age to 23
Deleted student with id 1
```

5. Learning Outcomes:

- Understand how to configure Hibernate with hibernate.cfg.xml.
- Learn how to create and annotate entity classes for use with Hibernate ORM.
- Gain practical knowledge of performing CRUD operations (Create, Read, Update, Delete) using Hibernate.
- Understand the role of SessionFactory in managing Hibernate sessions for database interaction.
- Practice integrating Hibernate with MySQL to persist and retrieve data from the database.

Hard -Level

1. **Aim:** Develop a Spring-based application integrated with Hibernate to manage transactions. Create a banking system where users can transfer money between accounts, ensuring transaction consistency.

   Requirements:
   - Use Spring configuration with Hibernate ORM.
   - Implement two entity classes (Account.java and Transaction.java).
   - Use Hibernate Transaction Management to ensure atomic operations.
   - If a transaction fails, rollback should occur.
   - Demonstrate successful and failed transactions.

2. **Objective:** To develop a Spring-based banking application integrated with Hibernate ORM, managing money transfers between accounts with transaction consistency, rollback on failure, and demonstrating successful and failed transactions.

3. **Implementation/Code:**

Account.java

```
import javax.persistence.*;
@Entity
public class Account {
    @Id
    private int accountId;
    private String holderName;
    private double balance;
    public Account() {}
    public Account(int accountId, String holderName, double balance) {
        this.accountId = accountId;
        this.holderName = holderName;
        this.balance = balance;
    }
    public int getAccountId() {
        return accountId;
    }
    public void setAccountId(int accountId) {
        this.accountId = accountId;
    }
    public String getHolderName() {
        return holderName;
    }
    public void setHolderName(String holderName) {
        this.holderName = holderName;
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }
  }
```

BankTransaction.java

```
import javax.persistence.*;
```

```
import java.util.Date;
@Entity
public class BankTransaction {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int txnId;
    private int fromAcc;
    private int toAcc;
    private double amount;
    private Date txnDate = new Date();
    public BankTransaction() {}
    public BankTransaction(int fromAcc, int toAcc, double amount) {
        this.fromAcc = fromAcc;
        this.toAcc = toAcc;
        this.amount = amount;
    }
    public int getTxnId() {
        return txnId;
    }
    public void setTxnId(int txnId) {
        this.txnId = txnId;
    }
    public int getFromAcc() {
        return fromAcc;
    }
    public void setFromAcc(int fromAcc) {
        this.fromAcc = fromAcc;
    }
    public int getToAcc() {
        return toAcc;
    }
    public void setToAcc(int toAcc) {
        this.toAcc = toAcc;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public Date getTxnDate() {
        return txnDate;
    }
    public void setTxnDate(Date txnDate) {
        this.txnDate = txnDate;
    }
}
```

BankService.java

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.transaction.annotation.Transactional;
public class BankService {
    private SessionFactory sessionFactory;
    public BankService(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
```

```
    }
    @Transactional
    public void transferMoney(int fromId, int toId, double amount) {
        Session session = sessionFactory.getCurrentSession();
        Account from = session.get(Account.class, fromId);
        Account to = session.get(Account.class, toId);
        if (from.getBalance() < amount) {
            throw new RuntimeException("Insufficient Balance");
        }
        from.setBalance(from.getBalance() - amount);
        to.setBalance(to.getBalance() + amount);
        session.update(from);
        session.update(to);
        BankTransaction txn = new BankTransaction(fromId, toId, amount);
        session.save(txn);
    }
}
```

AppConfig.java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import javax.sql.DataSource;
import java.util.Properties;
@Configuration
@EnableTransactionManagement
public class AppConfig {
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("com.mysql.cj.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/testdb");
        ds.setUsername("root");
        ds.setPassword("password");
        return ds;
    }
    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean lsf = new LocalSessionFactoryBean();
        lsf.setDataSource(dataSource());
        lsf.setPackagesToScan("your.package");
        Properties props = new Properties();
        props.put("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");
        props.put("hibernate.hbm2ddl.auto", "update");
        lsf.setHibernateProperties(props);
        return lsf;
    }
    @Bean
    public HibernateTransactionManager transactionManager(SessionFactory sf) {
        return new HibernateTransactionManager(sf);
    }
    @Bean
    public BankService bankService(SessionFactory sf) {
```

```
        return new BankService(sf);
    }
}
```

MainApp.java

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class MainApp {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
        BankService service = ctx.getBean(BankService.class);
        try {
            service.transferMoney(101, 102, 500);
            System.out.println("Transaction Successful!");
        } catch (Exception e) {
            System.out.println("Transaction Failed: " + e.getMessage());
        }
        ctx.close();
    }
}
```

4. **Output:**

```
Transaction Successful!
OR
Transaction Failed: Insufficient Balance
```

5. **Learning Outcomes:**

- Understand how to integrate Spring with Hibernate ORM for transaction management.
- Learn how to design entity classes (Account and Transaction) and map them to database tables using Hibernate annotations.
- Gain hands-on experience with Hibernate Transaction Management to ensure consistency in banking operations.
- Understand how to manage transactions in a Spring-based application using @Transactional.
- Learn how to implement rollback functionality to revert changes in case of transaction failures.
- Practice handling both successful and failed transactions in a real-world banking system scenario.