Experiment-9

Student Name: Gagnesh Kakkar **UID:** 23BCS11196
Branch: B.E-C.S.E **Section/Group:** 23KRG-2B
Semester: 5th **Date of Performance:** 24/10/2025
Subject Name: PBLJ **Subject Code:** 23CSH-304

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.

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

2. **Objective:** Understand Spring IoC, DI concepts, and Java-based bean configuration using annotations.

3. **Input/Apparatus Used:** Spring Core, Java-based Config, IntelliJ/Eclipse.

4. **Procedure:**
   1. Create a `Course` class with attributes courseName and duration, and override the toString() method.
   2. Create a `Student` class with attributes name and a reference to Course.
   3. Create a configuration class annotated with @Configuration and define @Bean methods for Course and Student.
   4. In the main method, use AnnotationConfigApplicationContext to load the Spring context and fetch the Student bean.
   5. Call the getter methods to display Student and Course details.

5.
Sample Output:
Student: Rahul, Course: Java Full Stack, Duration: 6 months

6. Code:

1. Course class

```
public class Course {
    private String courseName;
    private String duration;

    public Course(String courseName, String duration) {
        this.courseName = courseName;
        this.duration = duration;
    }

    @Override
    public String toString() {
        return courseName + ", Duration: " + duration;
    }
}
```

2. Student class

```
public class Student {
    private String name;
    private Course course; // Dependency

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

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

3. Java Configuration (No XML)

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public Course course() {
        return new Course("Java Full Stack", "6 months");
    }

    @Bean
    public Student student() {
        return new Student("Rahul", course());
    }
}
```

4. Main Class

```
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(AppConfig.class);

        Student stu = context.getBean(Student.class);
        stu.display();

        context.close();
    }
}
```

7. Output:

```
Student: Gagnesh, Course: Java Full Stack, Duration: 6 months
```

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.

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

2. Objective: Learn Hibernate ORM configuration, entity mapping, and CRUD operations with MySQL.

3. Input/Apparatus Used: Hibernate, MySQL, hibernate.cfg.xml, Java Application.

4. Procedure:

1. Create a Hibernate configuration file (hibernate.cfg.xml) with MySQL DB properties.
2. Create a Student entity class with annotations @Entity, @Id, and fields id, name, age.

3. Use HibernateUtil to manage SessionFactory setup.
4. Implement CRUD operations using session.save(), session.get(), session.update(), session.delete().
5. Write a main class to test insert, fetch, update, and delete actions.
6. Use try-catch-finally to handle exceptions and resource closing.

5.

Sample Output :

```
Insert Success!
Student List:
ID: 1, Name: John, Age: 22
```

6. Code:

1. hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory>
   <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
   <property
name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
   <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/yourdb</property>
   <property name="hibernate.connection.username">root</property>
   <property name="hibernate.connection.password">yourpass</property>
   <property name="show_sql">true</property>
   <mapping class="Student"/>
 </session-factory>
</hibernate-configuration>
```

2. Student Entity

```java
import javax.persistence.*;

@Entity
public class Student {
    @Id
    private int id;
    private String name;
    private int age;

    public Student() {}
    public Student(int id, String name, int age) {
        this.id=id; this.name=name; this.age=age;
    }
}
```

3. HibernateUtil

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

public class HibernateUtil {
    private static SessionFactory factory =
        new Configuration().configure().buildSessionFactory();
    public static SessionFactory getFactory() { return factory; }
}
```

4. CRUD Test

```java
import org.hibernate.Session;
import org.hibernate.Transaction;
import java.util.List;
```

```
public class TestStudentCRUD {
    public static void main(String[] args) {

        // CREATE
        try(Session session = HibernateUtil.getFactory().openSession()) {
            Transaction tx = session.beginTransaction();
            Student s = new Student(1, "John", 22);
            session.save(s);
            tx.commit();
            System.out.println("Insert Success!");
        }

        // READ
        try(Session session = HibernateUtil.getFactory().openSession()) {
            List<Student> list = session.createQuery("from Student").list();
            System.out.println("Student List:");
            list.forEach(st -> System.out.println(
                    "ID: "+st.id+", Name: "+st.name+", Age: "+st.age));
        }
    }
}
```

7. Output:

```
Insert Success!

Student List:

ID: 1, Name: Gagnesh, Age: 20
```

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.

1. Use Spring configuration with Hibernate ORM.
2. Implement two entity classes (Account.java and Transaction.java).
3. Use Hibernate Transaction Management to ensure atomic operations.
4. If a transaction fails, rollback should occur.

Demonstrate successful and failed transactions.

2. Objective:

Understand transaction management in Spring-Hibernate apps and implement atomic operations.

3. Input/Apparatus Used:

Spring, Hibernate, MySQL, Spring ORM, ApplicationContext.

4. Procedure:

1. Create entity classes: Account (id, name, balance) and Transaction (id, fromAccount, toAccount, amount, date).
2. Configure Spring with Hibernate and DataSource in applicationContext.xml or JavaConfig.
3. Create DAO layer with methods to debit from one account and credit to another.
4. Annotate service methods with @Transactional for transaction boundaries.
5. Use Spring context to call service methods from a main class or test class.
6. Simulate a successful and failed transaction and confirm rollback on failure.

Sample Output:

```
Transaction Successful: ₹500 transferred from Acc101 to Acc102
Transaction Failed: Insufficient balance — transaction rolled back
```

5. Code:

1. Account Entity

```
import javax.persistence.*;

@Entity
public class Account {
    @Id
    private int id;
    private String name;
    private double balance;
    public void credit(double amt) { this.balance += amt; }
    public void debit(double amt) { this.balance -= amt; }
}
```

2. Service Layer with Transaction

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import javax.persistence.*;

@Service
public class BankService {

    @PersistenceContext
    EntityManager em;

    @Transactional
    public void transfer(int fromAcc, int toAcc, double amount) {
        Account acc1 = em.find(Account.class, fromAcc);
        Account acc2 = em.find(Account.class, toAcc);

        if(acc1.getBalance() < amount)
            throw new RuntimeException("Insufficient balance");

        acc1.debit(amount);
        acc2.credit(amount);

        System.out.println("Transaction Successful: ₹" + amount +
                " transferred from " + fromAcc + " to " + toAcc);
    }
}
```

3. Main Runner

```
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class BankApp {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new
AnnotationConfigApplicationContext(SpringHibernateConfig.class);

        BankService service = context.getBean(BankService.class);

        try {
            service.transfer(101, 102, 500);
        } catch(Exception e) {
            System.out.println("Transaction Failed: " + e.getMessage());
        }
        context.close();
    }
}
```

6. Output:

```
Transaction Successful: ₹500 transferred from Acc101 to Acc102

Transaction Failed: Insufficient balance — transaction rolled
back
```