Experiment 9

Student Name: Swati Joshi
UID: 22BCS10183
Branch: BE/CSE
Section/Group: 22BCS_IOT-618/A
Semester: 6th
Date of Performance: 21/03/25
Subject Name: Project Based Learning in JAVA with Lab
Subject Code: 22CSH-359

1. Aim:

9.1 : Create a simple Spring application using Java-based configuration to demonstrate Dependency Injection (DI).

Code:

```
// File: Course.java

package com.example;

public class Course {

    private String courseName;

    private int duration;

    public Course(String courseName, int
duration) {

        this.courseName = courseName;

        this.duration = duration;

    }

    public String getCourseName() {

        return courseName;

    }

    public int getDuration() {

        return duration;

    }
```

```
    @Override
    public String toString() {
        return "Course{name='" +
courseName + "', duration=" + duration +
" weeks}";
    }
}
// File: Student.java
package com.example;

public class Student {
    private String name;
    private Course course;

    public Student(String name, Course
course) {
        this.name = name;
        this.course = course;
    }
    public void displayInfo() {
        System.out.println("Student Name: "
+ name);
        System.out.println("Enrolled in: " +
course);
```

```
    }
}
// File: AppConfig.java
package com.example;
import
org.springframework.context.annotation.B
ean;
import
org.springframework.context.annotation.C
onfiguration;

@Configuration
public class AppConfig {

    @Bean
    public Course course() {
        return new Course("Spring
Framework", 6);
    }
    @Bean
    public Student student() {
        return new Student("Alice",
course());
    }
}
```

```
// File: MainApp.java

package com.example;

import

org.springframework.context.Application

Context;

import

org.springframework.context.annotation.A

nnotationConfigApplicationContext;

public class MainApp {

    public static void main(String[] args) {

        ApplicationContext context = new

AnnotationConfigApplicationContext(App

Config.class);

        Student student =

context.getBean(Student.class);

        student.displayInfo();

    }}
```

Output:

```
Student Name: John
Course Name: Java Programming
Course Duration: 3 months
```

9.2: Develop a Hibernate-based application to perform CRUD operations on a Student entity with MySQL.

Code:

```
package com.example;

import jakarta.persistence.*;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

@Entity
@Table(name = "students")
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;
    }

    // Getters and setters

    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;
        }

    @Override
    public String toString() {
        return "Student{id=" + id + ", name='" + name + "', age=" + age + "}";
    }
}

public class MainApp {

    public static void main(String[] args) {

        SessionFactory factory = new Configuration()
                .configure("hibernate.cfg.xml")
                .addAnnotatedClass(Student.class)
                .buildSessionFactory();

        Session session = null;

        try {
            // CREATE
            session = factory.getCurrentSession();
            Student newStudent = new Student("Alice", 22);
            session.beginTransaction();
            session.save(newStudent);
            session.getTransaction().commit();
            System.out.println("Inserted: " + newStudent);

            // READ
            session = factory.getCurrentSession();
            session.beginTransaction();
            Student fetchedStudent = session.get(Student.class, newStudent.getId());
            session.getTransaction().commit();
            System.out.println("Fetched: " + fetchedStudent);

            // UPDATE
            session = factory.getCurrentSession();
            session.beginTransaction();
            fetchedStudent.setAge(23);
            session.update(fetchedStudent);
            session.getTransaction().commit();
            System.out.println("Updated: " + fetchedStudent);

            // DELETE
            session = factory.getCurrentSession();
            session.beginTransaction();
            session.delete(fetchedStudent);
            session.getTransaction().commit();
            System.out.println("Deleted student with ID: " + fetchedStudent.getId());
```

```
            } finally {
                factory.close();
            }
        }
    }
```

Output:

```
Inserted: Student{id=1, name='Alice', age=22}
Fetched: Student{id=1, name='Alice', age=22}
Updated: Student{id=1, name='Alice', age=23}
Deleted student with ID: 1
```

9.3: Create a banking system with Spring and Hibernate to manage money transfers using transactions.

Code:

```
package com.example.banking;
import jakarta.persistence.*;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import java.util.Date;
import java.util.List;

@Entity
@Table(name = "accounts")
class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String owner;
    private double balance;

    public Account() {}
    public Account(String owner, double balance) {
        this.owner = owner;
        this.balance = balance;
    }

    public int getId() { return id; }
    public String getOwner() { return owner; }
    public double getBalance() { return balance; }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    public String toString() {
        return "Account{id=" + id + ", owner='" + owner + "', balance=" + balance + '}';
    }
}

@Entity
```

```
@Table(name = "transactions")
class TransferRecord {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private int fromAccount;
    private int toAccount;
    private double amount;
    private Date timestamp;

    public TransferRecord() {}
    public TransferRecord(int from, int to, double amount) {
        this.fromAccount = from;
        this.toAccount = to;
        this.amount = amount;
        this.timestamp = new Date();
    }

    public String toString() {
        return "Transfer{id=" + id + ", from=" + fromAccount + ", to=" + toAccount +
                ", amount=" + amount + ", time=" + timestamp + '}';
    }
}

interface BankService {
    void transferMoney(int fromId, int toId, double amount);
    void showAccounts();
}

class BankServiceImpl implements BankService {

    private final SessionFactory sessionFactory;

    public BankServiceImpl(SessionFactory factory) {
        this.sessionFactory = factory;
    }

    @Override
    @Transactional
    public void transferMoney(int fromId, int toId, double amount) {
        var session = sessionFactory.getCurrentSession();
```

```
        Account from = session.get(Account.class, fromId);
        Account to = session.get(Account.class, toId);

        if (from == null || to == null) throw new RuntimeException("Invalid account ID(s)");

        if (from.getBalance() < amount) throw new RuntimeException("Insufficient funds");

        from.setBalance(from.getBalance() - amount);
        to.setBalance(to.getBalance() + amount);

        session.persist(new TransferRecord(fromId, toId, amount));
    }

    @Override
    @Transactional(readOnly = true)
    public void showAccounts() {
        var session = sessionFactory.getCurrentSession();
        List<Account> accounts = session.createQuery("from Account", Account.class).list();
        accounts.forEach(System.out::println);
    }
}

@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = "com.example.banking")
class AppConfig {

    @Bean
    public SessionFactory sessionFactory() {
        return new Configuration()
                .configure("hibernate.cfg.xml")
                .addAnnotatedClass(Account.class)
                .addAnnotatedClass(TransferRecord.class)
                .buildSessionFactory();
    }

    @Bean
    public BankService bankService() {
        return new BankServiceImpl(sessionFactory());
    }
```

```
    @Bean
    public HibernateTransactionManager transactionManager() {
        return new HibernateTransactionManager(sessionFactory());
    }
}
package com.example.banking;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {
    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext(AppConfig.class);
        var service = context.getBean(BankService.class);
session.beginTransaction();
        session.save(new Account("Alice", 1000));
        session.save(new Account("Bob", 500));
        session.getTransaction().commit();
        session.close();

        System.out.println("Initial state:");
        service.showAccounts();

        System.out.println("\n--- Performing successful transfer ---");
        try {
            service.transferMoney(1, 2, 200);
        } catch (Exception e) {
            System.out.println("Failed: " + e.getMessage());
        }
        service.showAccounts();

        System.out.println("\n--- Performing failed transfer (insufficient funds) ---");
        try {
            service.transferMoney(1, 2, 5000); // should fail
        } catch (Exception e) {
            System.out.println("Failed: " + e.getMessage());
        }
        service.showAccounts();

        context.close();
    }
}
```

Output:

```
Initial state:
Account{id=1, owner='Alice', balance=1000.0}
Account{id=2, owner='Bob', balance=500.0}

--- Performing successful transfer ---
Account{id=1, owner='Alice', balance=800.0}
Account{id=2, owner='Bob', balance=700.0}

--- Performing failed transfer (insufficient funds) ---
Failed: Insufficient funds
Account{id=1, owner='Alice', balance=800.0}
Account{id=2, owner='Bob', balance=700.0}
```

2. Learning Outcomes:

- Database Connectivity with JDBC – Understood how to connect Java programs to MySQL databases and perform CRUD operations.
- MVC Architecture in Java – Implemented the Model-View-Controller (MVC) pattern for better separation of concerns in database applications.
- Data Processing & Aggregation – Used Java streams to group, filter, and compute statistics (like average price and max values) on datasets.
- Functional Programming Concepts – Practiced method references, functional interfaces, and stream operations for cleaner and more efficient code.