Week - 3
Spring Core and Maven

Exercise 1: Configuring a Basic Spring Application

Scenario:
Your company is developing a web application for managing a library. You need to use the Spring Framework to handle the backend operations.
Steps:

- **Set Up a Spring Project:**
  - Create a Maven project named **LibraryManagement**.
  - Add Spring Core dependencies in the **pom.xml** file.
- **Configure the Application Context:**
  - Create an XML configuration file named **applicationContext.xml** in the **src/main/resources** directory.
  - Define beans for **BookService** and **BookRepository** in the XML file.
- **Define Service and Repository Classes:**
  - Create a package **com.library.service** and add a class **BookService**.
  - Create a package **com.library.repository** and add a class **BookRepository**.
- **Run the Application:**
  - Create a main class to load the Spring context and test the configuration.

✅ 1. Create a Maven Project

Project Name: `LibraryManagement`

✅ 2. `pom.xml`

MAKE SURE YOUR `POM.XML` CONTAINS SPRING CORE DEPENDENCIES :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
         xsi:schemaLocation="http://maven.apache.org/POM/
4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.library</groupId>
    <artifactId>LibraryManagement</artifactId>
    <version>1.0-SNAPSHOT</version>
```

Week - 3

```
    <dependencies>
        <!-- Spring Core Dependency -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.32</version> <!-- You can adjust the
version -->
        </dependency>
    </dependencies>
</project>
```

✅ 3. `applicationContext.xml` (inside `src/main/resources`)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-
beans.xsd">

    <!-- Repository Bean -->
    <bean id="bookRepository"
class="com.library.repository.BookRepository" />

    <!-- Service Bean -->
    <bean id="bookService"
class="com.library.service.BookService">
        <property name="bookRepository" ref="bookRepository"/
>
    </bean>

</beans>
```

✅ 4. Java Classes

`com.library.repository.BookRepository.java`

Week - 3

```
package com.library.repository;

public class BookRepository {
    public void saveBook(String bookName) {
        System.out.println("Book saved: " + bookName);
    }
}
```

com.library.service.BookService.java

```
package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

    private BookRepository bookRepository;

    // Setter Injection
    public void setBookRepository(BookRepository
bookRepository) {
        this.bookRepository = bookRepository;
    }

    public void addBook(String bookName) {
        System.out.println("Adding book: " + bookName);
        bookRepository.saveBook(bookName);
    }
}
```

✅ 5. Main Class to Load Spring Context

com.library.MainApp.java

```
package com.library;

import com.library.service.BookService;
import org.springframework.context.ApplicationContext;
```

Week - 3

```
import
org.springframework.context.support.ClassPathXmlApplicationCo
ntext;


public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context =
                new
ClassPathXmlApplicationContext("applicationContext.xml");

        BookService bookService = (BookService)
context.getBean("bookService");
        bookService.addBook("Spring in Action");
    }
}
```

✅ Folder Structure

```
LibraryManagement/
│
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── library/
│   │   │           ├── MainApp.java
│   │   │           ├── repository/
│   │   │           │   └── BookRepository.java
│   │   │           └── service/
│   │   │               └── BookService.java
│   │   └── resources/
│   │       └── applicationContext.xml
│   │
└── pom.xml
```

✅ To Run:

- Use `mvn clean install`
- Run `MainApp.java`

Week - 3

✅ Output of the Application

WHEN YOU RUN THE `MAINAPP.JAVA` FILE, THIS WILL BE THE CONSOLE OUTPUT :

```
Adding book: Spring in Action
Book saved: Spring in Action
```

💡 Explanation:

- `BookService.addBook("Spring in Action")` is called.
- It prints: `Adding book: Spring in Action`
- Then it calls `BookRepository.saveBook("Spring in Action")`, which prints: `Book saved: Spring in Action`

Exercise 2: Implementing Dependency Injection

Scenario:
In the library management application, you need to manage the dependencies between the BookService and BookRepository classes using Spring's IoC and DI.
Steps:
- **Modify the XML Configuration:**
  - Update **applicationContext.xml** to wire **BookRepository** into **BookService**.
- **Update the BookService Class:**
  - Ensure that **BookService** class has a setter method for **BookRepository**.
- **Test the Configuration:**
  - Run the **LibraryManagementApplication** main class to verify the dependency injection.

✅ Step-by-Step Walkthrough

✅ 1. Modify `applicationContext.xml`

Week - 3

You already partially did this in Exercise 1, but here's a clearer and cleaner version with proper dependency wiring using **setter-based injection**:

`src/main/resources/applicationContext.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/
spring-beans.xsd">

    <!-- Define Repository Bean -->
    <bean id="bookRepository"
class="com.library.repository.BookRepository" />

    <!-- Define Service Bean and inject bookRepository using
setter -->
    <bean id="bookService"
class="com.library.service.BookService">
        <property name="bookRepository"
ref="bookRepository" />
    </bean>

</beans>
```

✅ 2. Update `BookService` Class for Setter Injection

Ensure that the `BookService` class has a public setter for `bookRepository`.

`com.library.service.BookService.java`

```
package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

    private BookRepository bookRepository;

    // Setter for dependency injection
```

Week - 3

```
    public void setBookRepository(BookRepository
bookRepository) {
        this.bookRepository = bookRepository;
    }

    public void addBook(String bookName) {
        System.out.println("Adding book: " + bookName);
        bookRepository.saveBook(bookName);
    }
}
```

✅ 3. Test the Configuration

Run the `MainApp` class (renamed here for clarity as `LibraryManagementApplication`) to verify that Spring correctly wires the dependencies.

`com.library.LibraryManagementApplication.java`

```
package com.library;

import com.library.service.BookService;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationCo
ntext;

public class LibraryManagementApplication {
    public static void main(String[] args) {
        ApplicationContext context =
                new
ClassPathXmlApplicationContext("applicationContext.xml");

        BookService bookService = (BookService)
context.getBean("bookService");
        bookService.addBook("Effective Java");
    }
}
```

✅ 4. Expected Console Output

```
Adding book: Effective Java
```

Week - 3

```
Book saved: Effective Java
```

🔁 Summary

- ✅ Spring IoC container loads the `applicationContext.xml`.
- ✅ It creates a `BookRepository` bean and injects it into the `BookService` bean.
- ✅ Running the main app confirms that DI is working.

Exercise 4: Creating and Configuring a Maven Project

Scenario:
You need to set up a new Maven project for the library management application and add Spring dependencies.
Steps:
- **Create a New Maven Project:**
  - Create a new Maven project named **LibraryManagement**.
- **Add Spring Dependencies in pom.xml:**
  - Include dependencies for Spring Context, Spring AOP, and Spring WebMVC.
- **Configure Maven Plugins:**
  - Configure the Maven Compiler Plugin for Java version 1.8 in the pom.xml file.

✅ 1. Create a New Maven Project

Project Name: `LibraryManagement`
Directory Structure (after setting it up):

```
LibraryManagement/
├── src/
│   ├── main/
│   │   ├── java/
│   │   └── resources/
├── pom.xml
```

You can create this using an IDE like IntelliJ or Eclipse, or by running:

Week - 3

```
mvn archetype:generate -DgroupId=com.library
-DartifactId=LibraryManagement -DarchetypeArtifactId=maven-
archetype-quickstart -DinteractiveMode=false
```

✅ 2. `pom.xml` with Spring Dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
         xsi:schemaLocation="http://maven.apache.org/POM/
4.0.0
                             http://maven.apache.org/xsd/
maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.library</groupId>
    <artifactId>LibraryManagement</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <!-- Spring Context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.32</version>
        </dependency>

        <!-- Spring AOP -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>5.3.32</version>
        </dependency>

        <!-- Spring Web MVC -->
        <dependency>
```

```
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.3.32</version>
        </dependency>

        <!-- Servlet API (provided by container like Tomcat)
-->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.1</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <!-- Maven Compiler Plugin -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</
artifactId>
                <version>3.10.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

✅ 3. What This Setup Does

- Adds **Spring Context** for core DI and bean management.
- Adds **Spring AOP** for cross-cutting concerns (you can use this later for logging, security, etc.).
- Adds **Spring WebMVC** if you plan to add controllers and web layers.

Week - 3

- Configures **Java 1.8 compatibility** via Maven compiler plugin.

✅ Run `mvn compile` Output :

WHEN YOU RUN THIS MAVEN COMMAND IN THE TERMINAL :

```
mvn compile
```

YOU SHOULD SEE OUTPUT LIKE :

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< com.library:LibraryManagement
>------------------
[INFO] Building LibraryManagement 1.0-SNAPSHOT
[INFO] --- maven-resources-plugin:3.2.0:resources (default-
resources) @ LibraryManagement ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /
LibraryManagement/src/main/resources
[INFO] --- maven-compiler-plugin:3.10.1:compile (default-
compile) @ LibraryManagement ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 0 source files to /LibraryManagement/target/
classes
[INFO]
-------------------------------------------------------------
-----------
[INFO] BUILD SUCCESS
[INFO]
-------------------------------------------------------------
-----------
```

✅ This indicates:

- Your Maven configuration is correct.
- All dependencies downloaded successfully.
- Project is ready to add source code (controllers, services, etc.).

Week - 3

Spring Data JPA with Spring Boot, Hibernate

Spring Data JPA - Quick Example

Software Pre-requisites

- MySQL Server 8.0
- MySQL Workbench 8
- Eclipse IDE for Enterprise Java Developers 2019-03 R
- Maven 3.6.2

Create a Eclipse Project using Spring Initializer

- Go to https://start.spring.io/
- Change Group as “com.cognizant”
- Change Artifact Id as “orm-learn”
- In Options > Description enter "Demo project for Spring Data JPA and Hibernate"
- Click on menu and select "Spring Boot DevTools", "Spring Data JPA" and "MySQL Driver"
- Click Generate and download the project as zip
- Extract the zip in root folder to Eclipse Workspace
- Import the project in Eclipse "File > Import > Maven > Existing Maven Projects > Click Browse and select extracted folder > Finish"
- Create a new schema "ormlearn" in MySQL database. Execute the following commands to open MySQL client and create schema.

```
> mysql -u root -p


mysql> create schema ormlearn;
```

- In orm-learn Eclipse project, open src/main/resources/application.properties and include the below database and log configuration.

```
# Spring Framework and application log
logging.level.org.springframework=info
```

Week - 3

```
logging.level.com.cognizant=debug


# Hibernate logs for displaying executed SQL, input and output
logging.level.org.hibernate.SQL=trace
logging.level.org.hibernate.type.descriptor.sql=trace


# Log pattern
logging.pattern.console=%d{dd-MM-yy} %d{HH:mm:ss.SSS} %-20.20thread %5p
%-25.25logger{25} %25M %4L %m%n


# Database configuration
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/ormlearn
spring.datasource.username=root
spring.datasource.password=root


# Hibernate configuration
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

- Build the project using 'mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456' command in command line
- Include logs for verifying if main() method is called.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


private static final Logger LOGGER =
LoggerFactory.getLogger(OrmLearnApplication.class);


public static void main(String[] args) {
    SpringApplication.run(OrmLearnApplication.class, args);
    LOGGER.info("Inside main");
}
```

- Execute the OrmLearnApplication and check in log if main method is called.

SME to walk through the following aspects related to the project created:

- src/main/java - Folder with application code

Week - 3

- src/main/resources - Folder for application configuration
- src/test/java - Folder with code for testing the application
- OrmLearnApplication.java - Walkthrough the main() method.
- Purpose of @SpringBootApplication annotation
- pom.xml
  - Walkthrough all the configuration defined in XML file
  - Open 'Dependency Hierarchy' and show the dependency tree.

Country table creation

- Create a new table country with columns for code and name. For sample, let us insert one country with values 'IN' and 'India' in this table.

```
create table country(co_code varchar(2) primary key, co_name varchar(50));
```

- Insert couple of records into the table

```
insert into country values ('IN', 'India');

insert into country values ('US', 'United States of America');
```

Persistence Class - com.cognizant.orm-learn.model.Country

- Open Eclipse with orm-learn project
- Create new package com.cognizant.orm-learn.model
- Create Country.java, then generate getters, setters and toString() methods.
- Include @Entity and @Table at class level
- Include @Column annotations in each getter method specifying the column name.

```
import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;


@Entity

@Table(name="country")

public class Country {


    @Id

    @Column(name="code")
    private String code;

    @Column(name="name")
    private String name;

    // getters and setters

    // toString()

}
```

Notes:

- @Entity is an indicator to Spring Data JPA that it is an entity class for the application
- @Table helps in defining the mapping database table
- @Id helps is defining the primary key
- @Column helps in defining the mapping table column

Repository Class - com.cognizant.orm-learn.CountryRepository

- Create new package com.cognizant.orm-learn.repository
- Create new interface named CountryRepository that extends JpaRepository<Country, String>
- Define @Repository annotation at class level

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.cognizant.ormlearn.model.Country;

@Repository
public interface CountryRepository extends JpaRepository<Country, String> {

}
```

Service Class - com.cognizant.orm-learn.service.CountryService

- Create new package com.cognizant.orm-learn.service

Week - 3

- Create new class CountryService
- Include @Service annotation at class level
- Autowire CountryRepository in CountryService
- Include new method getAllCountries() method that returns a list of countries.
- Include @Transactional annotation for this method
- In getAllCountries() method invoke countryRepository.findAll() method and return the result

Testing in OrmLearnApplication.java

- Include a static reference to CountryService in OrmLearnApplication class

```
private static CountryService countryService;
```

- Define a test method to get all countries from service.

```
private static void testGetAllCountries() {
    LOGGER.info("Start");
    List<Country> countries = countryService.getAllCountries();
    LOGGER.debug("countries={}", countries);
    LOGGER.info("End");
}
```

- Modify SpringApplication.run() invocation to set the application context and the CountryService reference from the application context.

```
        ApplicationContext context =
SpringApplication.run(OrmLearnApplication.class, args);
        countryService = context.getBean(CountryService.class);


        testGetAllCountries();
```

- Execute main method to check if data from ormlearn database is retrieved.

✅ What You Have Set Up

You have:

- Created a Spring Boot app using Spring Initializer with:
  - `Spring Boot DevTools`
  - `Spring Data JPA`

Week - 3

    - `MySQL Driver`
- Set up MySQL schema `ormlearn`
- Configured `application.properties`
- Created:
    - `Country` entity
    - `CountryRepository`
    - `CountryService`
    - `testGetAllCountries()` in `OrmLearnApplication`

✅ What to Check/Verify

1. ✅ MySQL Setup

Run:

```
CREATE SCHEMA ormlearn;

CREATE TABLE country (
  co_code VARCHAR(2) PRIMARY KEY,
  co_name VARCHAR(50)
);

INSERT INTO country VALUES ('IN', 'India');
INSERT INTO country VALUES ('US', 'United States of
America’);
```

2. ✅ Expected `application.properties`

```
# Logging
logging.level.org.springframework=info
logging.level.com.cognizant=debug
logging.level.org.hibernate.SQL=trace
logging.level.org.hibernate.type.descriptor.sql=trace
logging.pattern.console=%d{dd-MM-yy} %d{HH:mm:ss.SSS}
%-20.20thread %5p %-25.25logger{25} %25M %4L %m%n
```

Week - 3

```
# DB Config
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/ormlearn
spring.datasource.username=root
spring.datasource.password=root

spring.jpa.hibernate.ddl-auto=validate
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect
.MySQL5Dialect
```

3. ✅ **Country.java**

```
@Entity
@Table(name="country")
public class Country {
    @Id
    @Column(name="co_code")
    private String code;

    @Column(name="co_name")
    private String name;

    // Getters, Setters, toString()
}
```

4. ✅ **CountryRepository.java**

```
@Repository
public interface CountryRepository extends
JpaRepository<Country, String> {
}
```

5. ✅ **CountryService.java**

```
@Service
public class CountryService {

    @Autowired
    private CountryRepository countryRepository;

    @Transactional
    public List<Country> getAllCountries() {
         return countryRepository.findAll();
    }
}
```

6. ✅ `OrmLearnApplication.java`

```
@SpringBootApplication
public class OrmLearnApplication {

    private static final Logger LOGGER =
LoggerFactory.getLogger(OrmLearnApplication.class);
    private static CountryService countryService;

    public static void main(String[] args) {
        ApplicationContext context =
SpringApplication.run(OrmLearnApplication.class, args);
        LOGGER.info("Inside main");

        countryService =
context.getBean(CountryService.class);
        testGetAllCountries();
    }

    private static void testGetAllCountries() {
        LOGGER.info("Start");
        List<Country> countries =
countryService.getAllCountries();
        LOGGER.debug("countries={}", countries);
        LOGGER.info("End");
    }
}
```

✅ Build & Run

Run this in terminal (only if behind proxy):

```
mvn clean package -Dhttp.proxyHost=proxy.cognizant.com
-Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com
-Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456
```

Then, run the `OrmLearnApplication.java` main method.

Week - 3

✅ Expected Console Output

You should see logs like:

```
03-07-25 10:15:12.123 main  INFO OrmLearnApplication     main
25 Inside main
03-07-25 10:15:12.124 main  INFO OrmLearnApplication
testGetAllCountries 30 Start
03-07-25 10:15:12.150 main DEBUG OrmLearnApplication
testGetAllCountries 31 countries=[Country{code='IN',
name='India'}, Country{code='US', name='United States of
America'}]
03-07-25 10:15:12.151 main  INFO OrmLearnApplication
testGetAllCountries 32 End
```

🧠 Spring Concepts Covered

Component	Role
`@Entity`, `@Table`	ORM mapping
`@Repository`	DAO layer via Spring Data
`@Service`, `@Transactional`	Business logic and transaction management
`@SpringBootApplication`	Marks entry point & enables auto-configuration
`JpaRepository`	Inherits ready-made CRUD functionality

Difference between JPA, Hibernate and Spring Data JPA

Java Persistence API (JPA)

- JSR 338 Specification for persisting, reading and managing data from Java objects
- Does not contain concrete implementation of the specification
- Hibernate is one of the implementation of JPA

Hibernate

- ORM Tool that implements JPA

Week - 3

Spring Data JPA

- Does not have JPA implementation, but reduces boiler plate code
- This is another level of abstraction over JPA implementation provider like Hibernate
- Manages transactions

Refer code snippets below on how the code compares between Hibernate and Spring Data JPA

Hibernate

```
/* Method to CREATE an employee in the database */
public Integer addEmployee(Employee employee){
   Session session = factory.openSession();
   Transaction tx = null;
   Integer employeeID = null;

   try {
      tx = session.beginTransaction();
      employeeID = (Integer) session.save(employee);
      tx.commit();
   } catch (HibernateException e) {
      if (tx != null) tx.rollback();
      e.printStackTrace();
   } finally {
      session.close();
   }
   return employeeID;
}
```

Spring Data JPA

EmployeeRespository.java

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {


}
```

EmployeeService.java

Week - 3

```
@Autowire
    private EmployeeRepository employeeRepository;

    @Transactional
    public void addEmployee(Employee employee) {
        employeeRepository.save(employee);
    }
```

Reference Links:
https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1
https://www.javaworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html

Here's a clear and structured comparison between **JPA**, **Hibernate**, and **Spring Data JPA**, along with explanation and example code:

🔁 Difference Between JPA, Hibernate, and Spring Data JPA

Feature	JPA	Hibernate	Spring Data JPA
Definition	Java specification for object-relational mapping (ORM)	Implementation of JPA (ORM tool)	Abstraction layer over JPA implementations
Type	Specification (no implementation)	Library (concrete implementation)	Spring-based framework
Implements	—	JPA (JSR 338)	Uses JPA + Hibernate under the hood
Boilerplate Code	Medium	Medium	Very low (most code auto-generated)
Query Language	JPQL	HQL (Hibernate Query Language)	JPQL / method names / `@Query`
Transactions	Must manage manually or via container	Needs manual or declarative management	Spring manages via `@Transactional`
Who Provides?	Oracle/Sun (via JSR)	Red Hat (formerly JBoss)	Spring (Pivotal/VMware)
Setup Complexity	High	Medium	Low

Week - 3

🧠 Analogy

Think of **JPA** as the **interface**, **Hibernate** as a **concrete class**, and **Spring Data JPA** as a **helper library that uses the class without making you write everything manually.**

✅ Code Comparison

✍️ Hibernate Example (Manual ORM)

```
public Integer addEmployee(Employee employee) {
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;

    try {
        tx = session.beginTransaction();
        employeeID = (Integer) session.save(employee);
        tx.commit();
    } catch (HibernateException e) {
        if (tx != null) tx.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }

    return employeeID;
}
```

✅ Spring Data JPA Example (Minimal Boilerplate)

```
public interface EmployeeRepository extends
JpaRepository<Employee, Integer> {
}
@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Transactional
    public void addEmployee(Employee employee)
```

Week - 3

```
{
            employeeRepository.save(employee);   // One-liner!
      }
}
```

📘 Summary

Concept	JPA	Hibernate	Spring Data JPA
Role	API/ Specification	ORM Implementation	Framework abstraction
Core Advantage	Standardization	Full JPA + extra features	Simplifies development
Usage Style	Interface-based	Code-heavy	Auto-repository, annotations
Learning Curve	Medium	Medium	Easy