Spring Framework & Spring Data JPA - Library Systems

# Document 1: Spring Framework - Core & Maven Setup (Library Management)

## Introduction

This document explains how to build a basic Spring Framework application using Maven. The goal is to manage a library system using Spring Core features like Dependency Injection (DI) and XML-based configuration.

**Software Prerequisites**

• MySQL Server 8.0+  
• MySQL Workbench 8.x  
• Eclipse IDE for Java EE Developers (2019 or later)  
• Maven 3.6.2 or newer

**Step 1: Initialize Spring Boot Project**

1. Visit https://start.spring.io/  
2. Set Group as: com.library.digital  
3. Set Artifact: library-data-jpa  
4. Add Description: Spring Boot demo using Spring Data JPA & Hibernate ORM  
5. Select Dependencies: Spring Boot DevTools, Spring Data JPA, MySQL Driver  
6. Click Generate to download the project ZIP  
7. Extract and import into Eclipse: File > Import > Maven > Existing Maven Projects > Select the folder

Open MySQL terminal and create schema:

> mysql -u root -p  
mysql> create schema library\_jpa;

**Step 2: Configure application.properties**

Navigate to src/main/resources/application.properties and configure the database:

# Logging  
logging.level.org.springframework=info  
logging.level.com.library=debug  
logging.level.org.hibernate.SQL=debug  
logging.level.org.hibernate.type.descriptor.sql=trace  
  
# Console Pattern  
logging.pattern.console=%d{HH:mm:ss.SSS} %-5level %logger{36} - %msg%n  
  
# MySQL Configuration  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
spring.datasource.url=jdbc:mysql://localhost:3306/library\_jpa  
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

**Step 3: Maven Build Command**

Use terminal to build the project:

mvn clean package -Dhttp.proxyHost=proxy.server.com -Dhttp.proxyPort=8080 -Dhttps.proxyHost=proxy.server.com -Dhttps.proxyPort=8080

**Step 4: Add Logger to Main Class**

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
private static final Logger LOGGER = LoggerFactory.getLogger(LibraryJpaApplication.class);  
  
public static void main(String[] args) {  
 ApplicationContext context = SpringApplication.run(LibraryJpaApplication.class, args);  
 LOGGER.info("Application started");  
}

**Project Structure Overview**

1. src/main/java - Contains all source code including entity, repository, and service  
2. src/main/resources - Configuration files including application.properties  
3. src/test/java - JUnit test files  
4. LibraryJpaApplication.java - Contains the main() method to bootstrap the Spring context  
5. @SpringBootApplication - Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan  
6. pom.xml - Contains all Maven dependencies and plugin configurations

**Step 5: Create Country Table**

create table country (  
 co\_code varchar(2) primary key,  
 co\_name varchar(50)  
);  
  
insert into country values ('IN', 'India');  
insert into country values ('UK', 'United Kingdom');

**Step 6: Entity Class - Country**

package com.library.digital.model;  
  
import javax.persistence.\*;  
  
@Entity  
@Table(name = "country")  
public class Country {  
  
 @Id  
 @Column(name = "co\_code")  
 private String code;  
  
 @Column(name = "co\_name")  
 private String name;  
  
 // Getters and Setters  
  
 // toString()  
}

**Step 7: Country Repository Interface**

package com.library.digital.repository;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
import com.library.digital.model.Country;  
  
@Repository  
public interface CountryRepository extends JpaRepository<Country, String> {  
}

**Step 8: Country Service Implementation**

package com.library.digital.service;  
  
import java.util.List;  
import javax.transaction.Transactional;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import com.library.digital.model.Country;  
import com.library.digital.repository.CountryRepository;  
  
@Service  
public class CountryService {  
  
 @Autowired  
 private CountryRepository countryRepository;  
  
 @Transactional  
 public List<Country> fetchAllCountries() {  
 return countryRepository.findAll();  
 }  
}

**Step 9: Test Service in Main Class**

## @Autowired private static CountryService countryService; private static void testCountries() { LOGGER.info("Begin test"); List<Country> countries = countryService.fetchAllCountries(); LOGGER.debug("Countries found: {}", countries); LOGGER.info("End test"); } public static void main(String[] args) { ApplicationContext context = SpringApplication.run(LibraryJpaApplication.class, args); countryService = context.getBean(CountryService.class); testCountries(); } Conclusion

This setup introduces Spring’s XML-based bean configuration and IoC container, enabling modular and scalable Java applications.

# Document 2: Spring Data JPA – Digital Library System

## Introduction

This guide demonstrates building a backend for a Digital Library System using Spring Boot, Spring Data JPA, and MySQL.

### 1. Database Setup

CREATE DATABASE library\_jpa;  
  
CREATE TABLE country (  
 co\_code VARCHAR(2) PRIMARY KEY,  
 co\_name VARCHAR(50)  
);  
  
INSERT INTO country VALUES ('IN', 'India'), ('US', 'United States');

### 2. application.properties

spring.datasource.url=jdbc:mysql://localhost:3306/library\_jpa  
spring.datasource.username=root  
spring.datasource.password=root  
spring.jpa.hibernate.ddl-auto=validate  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

### 3. Entity: Country.java

@Entity  
@Table(name = "country")  
public class Country {  
 @Id  
 @Column(name = "co\_code")  
 private String code;  
  
 @Column(name = "co\_name")  
 private String name;  
}

### 4. Repository

@Repository  
public interface CountryRepository extends JpaRepository<Country, String> {  
 List<Country> findByNameContainingIgnoreCase(String keyword);  
}

### 5. Service

@Service  
public class CountryService {  
 @Autowired  
 private CountryRepository countryRepository;  
  
 public List<Country> getAllCountries() {  
 return countryRepository.findAll();  
 }  
  
 public List<Country> searchCountries(String keyword) {  
 return countryRepository.findByNameContainingIgnoreCase(keyword);  
 }  
}

### 6. Main Method

public static void main(String[] args) {  
 ApplicationContext context = SpringApplication.run(LibraryJpaApplication.class, args);  
 CountryService service = context.getBean(CountryService.class);  
 service.getAllCountries();  
 service.searchCountries("in");  
}

## Comparison Table: Hibernate vs Spring Data JPA

|  |  |
| --- | --- |
| Feature | Hibernate vs Spring Data JPA |
| Setup Complexity | Medium vs Low |
| Boilerplate Code | High vs Minimal |
| Querying | HQL vs Method naming / JPQL |
| Integration | Manual vs Spring Boot Integrated |
| Transactions | Manual or AOP vs Auto-managed with @Transactional |
| Repository Handling | Manual DAO Layer vs JpaRepository Interface |

## Conclusion

Spring Data JPA makes data interaction seamless using repositories, eliminating the need for complex SQL or manual transactions. It’s ideal for building maintainable and scalable applications.