# Spring- Core\_Maven solutions

## Exercise 1: Configuring a Basic Spring Application

Set up a simple Spring application using XML configuration for managing a library system backend.

Steps:

1. Create Maven project named LibraryManagement.
2. Add Spring Core dependency in pom.xml.
3. Create applicationContext.xml in src/main/resources.
4. Define beans for BookService and BookRepository.

**Coding:**

// BookService.java  
package com.library.service;  
public class BookService {  
 public void display() {  
 System.out.println("BookService is working.");  
 }  
}  
  
// BookRepository.java  
package com.library.repository;  
public class BookRepository {  
 public void show() {  
 System.out.println("BookRepository is working.");  
 }  
}  
  
// applicationContext.xml (inside resources)  
<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">  
  
 <bean id="bookService" class="com.library.service.BookService"/>  
 <bean id="bookRepository" class="com.library.repository.BookRepository"/>  
</beans>  
  
// Main class  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
import com.library.service.BookService;  
  
public class LibraryManagementApplication {  
 public static void main(String[] args) {  
 ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
 BookService bookService = (BookService) context.getBean("bookService");  
 bookService.display();  
 }  
}

## Exercise 2: Implementing Dependency Injection

Use Spring’s IoC container to inject BookRepository into BookService.

Steps:

1. Add setter method for BookRepository in BookService.
2. Wire BookRepository into BookService in applicationContext.xml.
3. Test by calling repository method from service.

**Coding:**

// BookService.java  
private BookRepository bookRepository;  
public void setBookRepository(BookRepository bookRepository) {  
 this.bookRepository = bookRepository;  
}  
public void display() {  
 System.out.println("Service active.");  
 bookRepository.show();  
}  
  
// applicationContext.xml update  
<bean id="bookService" class="com.library.service.BookService">  
 <property name="bookRepository" ref="bookRepository"/>  
</bean>

## Exercise 3: Implementing Logging with Spring AOP

Add logging to track method execution time using Spring AOP.

Steps:

1. Add Spring AOP dependency to pom.xml.
2. Create LoggingAspect class with @Around advice.
3. Enable AspectJ auto-proxying in applicationContext.xml.

**Coding:**

// LoggingAspect.java  
@Aspect  
public class LoggingAspect {  
 @Around("execution(\* com.library.service.\*.\*(..))")  
 public Object log(ProceedingJoinPoint joinPoint) throws Throwable {  
 long start = System.currentTimeMillis();  
 Object result = joinPoint.proceed();  
 long elapsed = System.currentTimeMillis() - start;  
 System.out.println("Execution time: " + elapsed + "ms");  
 return result;  
 }  
}  
  
// applicationContext.xml additions  
<aop:aspectj-autoproxy/>  
<bean id="loggingAspect" class="com.library.aspect.LoggingAspect"/>

## Exercise 4: Creating and Configuring a Maven Project

Set up a new Maven project for the LibraryManagement system with proper Spring dependencies and plugin configuration.

Steps:

1. Create Maven project named LibraryManagement.
2. Add Spring Context, AOP, and WebMVC dependencies.
3. Configure Maven Compiler Plugin for Java 1.8.

**Coding:**

// pom.xml snippet  
<dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.3.20</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aop</artifactId>  
 <version>5.3.20</version>  
 </dependency>  
</dependencies>  
<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.8.1</version>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

## Exercise 5: Configuring the Spring IoC Container

Centralize bean configuration using the Spring IoC container.

Steps:

1. Create applicationContext.xml in resources.
2. Define beans for BookService and BookRepository.
3. Ensure BookService has setter for BookRepository.

**Coding:**

// BookService.java (with setter)  
public void setBookRepository(BookRepository bookRepository) {  
 this.bookRepository = bookRepository;  
}  
  
// applicationContext.xml  
<bean id="bookService" class="com.library.service.BookService">  
 <property name="bookRepository" ref="bookRepository"/>  
</bean>

## Exercise 6: Configuring Beans with Annotations

Use annotations to reduce XML configuration and enable component scanning.

Steps:

1. Annotate BookService with @Service.
2. Annotate BookRepository with @Repository.
3. Add component-scan in applicationContext.xml.

**Coding:**

// BookService.java  
@Service  
public class BookService { ... }  
  
// BookRepository.java  
@Repository  
public class BookRepository { ... }  
  
// applicationContext.xml  
<context:component-scan base-package="com.library"/>

## Exercise 7: Implementing Constructor and Setter Injection

Use both constructor and setter injection for flexible bean initialization.

Steps:

1. Add constructor to BookService with BookRepository.
2. Add setter method as well.
3. Configure both in applicationContext.xml.

**Coding:**

// BookService.java  
public BookService(BookRepository bookRepository) {  
 this.bookRepository = bookRepository;  
}  
public void setBookRepository(BookRepository bookRepository) {  
 this.bookRepository = bookRepository;  
}  
  
// applicationContext.xml  
<bean id="bookService" class="com.library.service.BookService">  
 <constructor-arg ref="bookRepository"/>  
 <property name="bookRepository" ref="bookRepository"/>  
</bean>

## Exercise 8: Implementing Basic AOP with Spring

Separate cross-cutting concerns like logging using basic Spring AOP.

Steps:

1. Create LoggingAspect with before and after advices.
2. Register the aspect in applicationContext.xml.
3. Enable AspectJ auto-proxying.

**Coding:**

// LoggingAspect.java  
@Aspect  
public class LoggingAspect {  
 @Before("execution(\* com.library.service.\*.\*(..))")  
 public void logBefore() {  
 System.out.println("Method execution started.");  
 }  
  
 @After("execution(\* com.library.service.\*.\*(..))")  
 public void logAfter() {  
 System.out.println("Method execution finished.");  
 }  
}  
  
// applicationContext.xml  
<aop:aspectj-autoproxy/>  
<bean id="loggingAspect" class="com.library.aspect.LoggingAspect"/>

## Exercise 9: Creating a Spring Boot Application

Develop a Spring Boot app to simplify deployment and configuration.

Steps:

1. Create project using Spring Initializr.
2. Add dependencies for Web, JPA, H2.
3. Define entity, repository, and controller.
4. Configure database in application.properties.

**Coding:**

// Book.java  
@Entity  
public class Book {  
 @Id @GeneratedValue  
 private Long id;  
 private String title;  
}  
  
// BookRepository.java  
public interface BookRepository extends JpaRepository<Book, Long> {}  
  
// BookController.java  
@RestController  
@RequestMapping("/books")  
public class BookController {  
 @Autowired  
 private BookRepository repo;  
  
 @GetMapping  
 public List<Book> getBooks() {  
 return repo.findAll();  
 }  
}  
  
// application.properties  
spring.datasource.url=jdbc:h2:mem:testdb  
spring.jpa.hibernate.ddl-auto=update