🌱 **Spring Framework**

Spring is a **lightweight, open-source framework** for building enterprise-grade Java applications. It promotes **loose coupling** through **Dependency Injection (DI)** and **Inversion of Control (IoC)**, making your code more modular and testable.

* **Why it matters:** It abstracts away boilerplate code and integrates seamlessly with other technologies like Hibernate, JPA, and JMS.
* **Use cases:** Web apps, microservices, batch processing, and mor

🧩 **Spring Core**

This is the **foundation module** of the Spring Framework.

* **Key components:**
  + **BeanFactory** and **ApplicationContext**: Containers that manage object lifecycles and dependencies.
  + **Dependency Injection (DI)**: Injects required objects at runtime.

🗃️ **Spring Data JPA**

This module simplifies **database access** using **Java Persistence API (JPA)**.

* **What it does:**
  + Eliminates boilerplate DAO code.
  + Auto-generates queries from method names (e.g., findByUsername()).
  + Supports pagination, sorting, and custom queries with @Query.

🌐 **Spring REST (Spring Web / Spring MVC)**

This is how you build **RESTful APIs** using Spring.

* **Core features:**
  + @RestController, @GetMapping, @PostMapping, etc., to handle HTTP requests.
  + JSON/XML serialization out of the box.
  + Exception handling, validation, and content negotiation.
* **Why it’s essential:** It’s the go-to for building scalable APIs in microservice architectures.

Spring automatically creates and injects the objects of dependencies into a class at run-time (dynamically), i.e Spring takes care of dependency injection

**Dependency Injection:**

* It’s a design pattern where a class receives its dependencies (like services or components) from the outside instead of creating them internally.
* **Dependency:** The WeatherService **depends** on a WeatherApi to perform its primary function (getting weather reports). Without a WeatherApi, WeatherService cannot do its job.
* **Injection:** Instead of the WeatherService *creating* its own WeatherApi (which would make it tightly coupled and less flexible and harder to test), the WeatherApi is **"injected"** or **"provided"** to the WeatherService from the *outside*. The WeatherService doesn't know or care *how* the WeatherApi was created; it just receives it.

**Inversion of Control:**

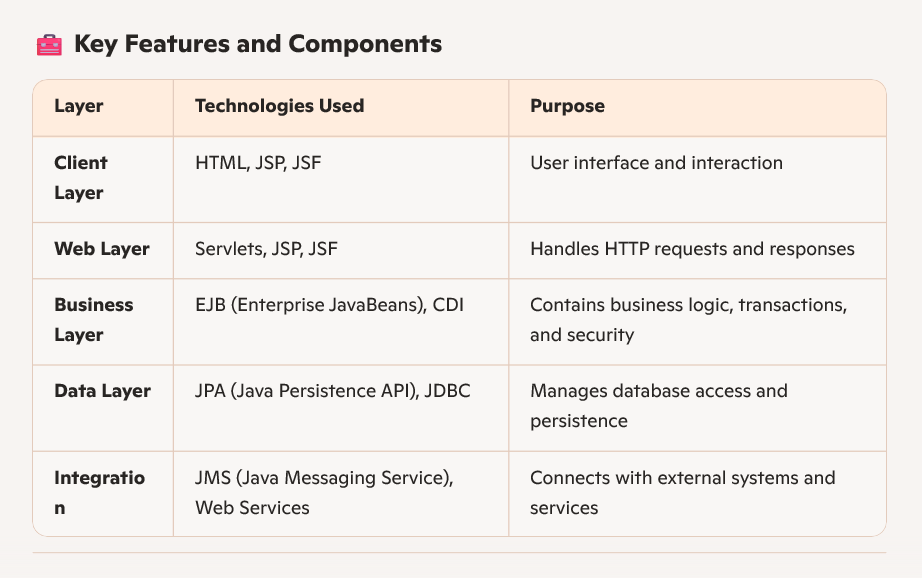
🔄 What is Inversion of Control?

At its heart, **IoC means flipping the usual flow of control** in a program. Instead of your code manually creating and managing dependencies (like calling new to instantiate objects), you delegate that responsibility to a framework or container.

In Spring, this container is the **IoC container**, which manages the lifecycle and dependencies of your objects (called *beans*).

**JAVA EE**

Java EE—now known as **Jakarta EE**—is a powerful platform for building **enterprise-level Java applications**. It extends the capabilities of Java SE (Standard Edition) by adding APIs and tools specifically designed for large-scale, distributed, and transactional systems.



**USE OF SPRING IN JAVA EE**

**Java EE Layers:**

UI Layer: To handles Requests java classes (accepts requests, uses business layer logic/class)

Business/Service Layer To write business logic class with business logic

Data Access Aceess data through DB class which contacts DB (if business logic class is reqd of data)

DB (database) Accessible to Data Access Layer

Here, classes are inter-dependent of each other, this is how DI is used in java ee,   
old method of creating objects with new and using them makes code tightly coupled.

Annotations, XML.. are used for injecting objects

Other than this, spring also provides, some api’s, modules which are very useful:

1. Spring JDBC, ORM in Data Access Layer
2. MVC in UI Layer
3. Security and Transaction Management in Business Layer

Spring JDBC: module which works with DB

ORM: used to work with other ORM tools (hibernate)

MVC: Model View Controller (module) used to develop UI

**Spring Modules:**

**Spring Core:**

core, beans, context, spELs (modules in spring core)

They form the basics for DI and IOC (Especially core and beans)

Context: Inherits lots of features from beans, also event propagation, resource loading, adding context in servelts, java j2ee features (ejb, jms..etc), basic remoting.

spEL: used to query and manipulate the Expression Language, mainly when to modify (query and manipulate) the graph (value) of object at run time.

**AOP : Aspect Oriented Programming:** Allows us to define method interceptors (works done before or after method), point cuts (used to decouple code, functionlaity should be separated)

Instrumentation: provides class loader implementation support.

Messaging: gives foundation for message based services.

**Data Integration Layer (Related to DB)**

JDBC: allows easy way to access dbms

ORM: integration tool which allows any other orm tool(eg. Hibernate) to be used in our project

JMS: Java Messaging Services : to produce and consume messages.

OXM: supports Object XML Mapping (OXM)

**Web: (used for creating web projects / REST api’s)**

Web: File Uploading, creating REST Apis’

Servelt, Portlet, Web Socket: All are related to creating web projects

**TEST**: supports unit testing and integrated testing (provides junit and testNG, mock objects)

**IOC: (What does it do) (Student is dependent on Address)**

1. Creates an object of Address and add (inject) its variables values at run time
2. Also, creates an object of student and injects variables values of student class and automatically at run time also injects address object in student class as well.
3. And finally returns the student object which we can use

All these are done by IOC (Spring) automatically

**Types of Dependency Injections:**

1. Setter Injection (Property Injection)
2. Constructor Injection

**Setter Injection:**

Setter Injection is a technique used in **dependency injection (DI)** where an object’s dependencies (values of variables of class) are provided through **public setter methods of class** after the object is constructed.

Setters are created at the time of object creation

**Constructor Injection:**

Uses constructor to provide dependencies.

Constructor is called at the time of object creation.

Which type is used, we have to mention in config file (xml file)

Class details need to be provided to xml file to create objects and inject dependencies

Those classes whose details are provided are called **“BEANS”**

**XML FILE:**

<beans>

<bean>

</bean>

</beans>

Spring uses a **bean configuration file** or annotations to define which classes require what dependencies.

The IoC container uses this configuration to inject objects automatically, making the application loosely coupled and more testable

**DATA TYPES INJECTED BY IOC:**

1. Primitive Data Types (Byte, Short, Int, Long, Char, Float, Double, Boolean)
2. Collection Type (List, Set, Map and Properties)
3. Reference Type (Other class object) :Address (User defined class) object is being injected into student class.

FOR EXERCISE 1

|  |  |  |  |
| --- | --- | --- | --- |
| Feature/Step | Plain Eclipse (with M2Eclipse) | STS (using File > New > Maven Project) | STS (using File > New > Spring Starter Project) |
| Initial IDE Setup | Download Eclipse. Manually install M2Eclipse (for Maven) and Spring Tools plugins via Marketplace. | Download STS (includes Eclipse, M2Eclipse, Spring Tools pre-installed). No extra setup for these. | Download STS (includes Eclipse, M2Eclipse, Spring Tools pre-installed). No extra setup for these. |
| 1. Create Project | File > New > Maven Project. Choose maven-archetype-quickstart. | File > New > Maven Project. Choose maven-archetype-quickstart | **Not ideal for this task.** This creates a Spring Boot project, which is Java-config by default. You *can* modify it, but it's counter-intuitive for an XML-based exercise. |
| 1. Add Spring Core Dependencies | M**anual:** Edit pom.xml to add <dependency> for spring-context, spring-beans, spring-core, etc. | **Manual:** Edit pom.xml to add <dependency> for spring-context, spring-beans, spring-core, etc. | **No automatic direct spring-core add.** This wizard adds spring-boot-starter-\* dependencies, which then transitively pull in core. You'd still need to ensure your pom.xml aligns with the exercise's intent if using this path. |
| Define Beans in XML | **Manual:** Write <bean> definitions. Eclipse's XML editor provides assist. | **Manual:** Write <bean> definitions. STS's XML editor provides excellent assist. | **Manual:** You would still need to write these. |
| Define Service/Repo Classes | **Manual:** Create packages/classes in src/main/java. | **Manual:** Create packages/classes in src/main/java | **Manual:** Create packages/classes in src/main/java. (Basic main class is generated, but not these specific ones). |
| . Run Application | **Manual:** Create main method to load ClassPathXmlApplicationContext. Run as Java App. | **Manual:** Create main method to load ClassPathXmlApplicationContext. Run as Java App. | **Manual:** Create main method to load ClassPathXmlApplicationContext (or use Spring Boot's main class but override its config). Run as Java App. |

**Brief Explanation:**

* **Plain Eclipse:** You start with a very generic Java IDE. You must *manually add all the necessary plugins* (Maven integration, Spring-specific tools) to make it suitable for Spring development. Once configured, it behaves very similarly to STS for traditional Spring projects.
* **STS (using File > New > Maven Project):** This is the **most straightforward and recommended path for your specific exercise**. STS comes pre-packaged with all the tools you need (Maven, Spring Tools). You're using the general-purpose Maven project creation, and then manually adding the XML configuration and traditional Spring dependencies as the exercise requires. STS's editors will still assist you greatly with auto-completion for XML and Java.
* **STS (using File > New > Spring Starter Project):** This wizard is **specifically for Spring Boot projects**. Spring Boot is a modern approach that prefers Java-based configuration and convention over XML. While technically you *could* create a Spring Boot project and then *add* applicationContext.xml to it, it goes against the design philosophy of Spring Boot and the clear intent of your exercise, making it unnecessarily complicated. It would also automatically pull in Spring Boot-specific dependencies that your exercise doesn't explicitly mention.

Without maven, need to download jar files of all these spring core, context. Maven handles those if added, dependencies.

Always update project after adding dependencies, best practice

**FOR SPRING DEPENDENCIES**

1. GO FOR MAVEN REPOSITORY, AND COPY PASTE IN IDE
   1. Spring core
   2. Spring context
2. For config xml file, go to spring documentation and find brief template.(or ask chatgpt) i.e add namespaces
   1. <https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/xsd-configuration.html>
   2. The 'xsi:schemaLocation' fragment is not actually required, but can be included to reference a local copy of a schema (which can be useful during development).

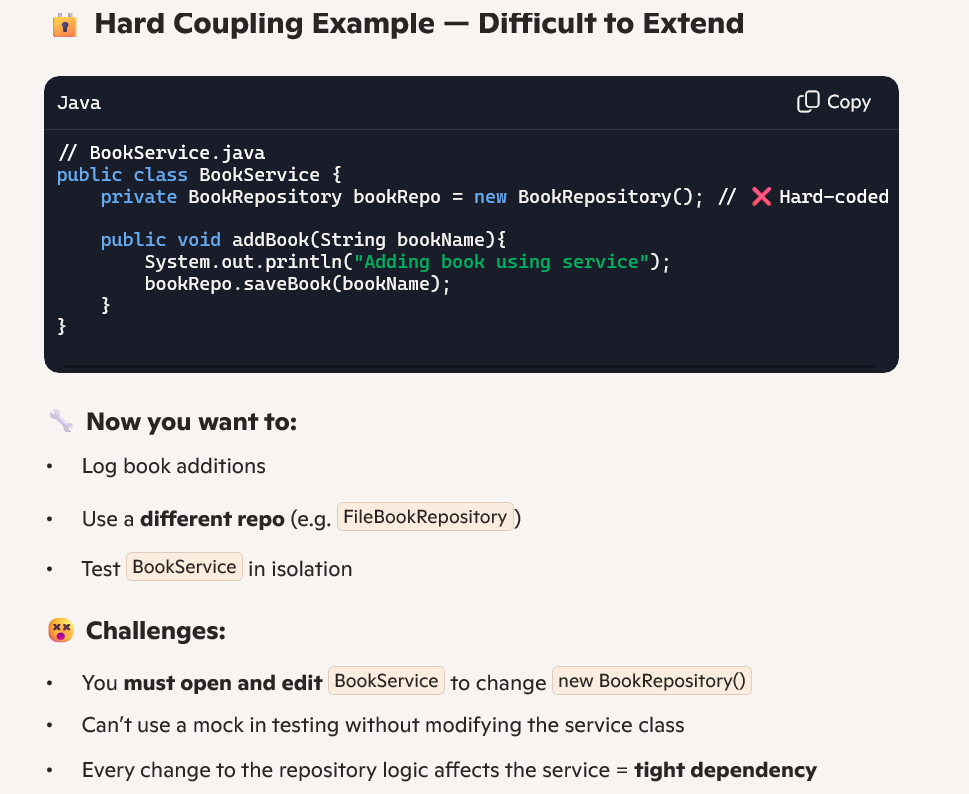
**XML CONFIG FILE**

Those schema links in your Spring XML file—like http://www.springframework.org/schema/beans and http://www.springframework.org/schema/beans/spring-beans.xsd—are part of Spring’s **XML Schema-based configuration** system. You don’t need to manually download them or search for them online.

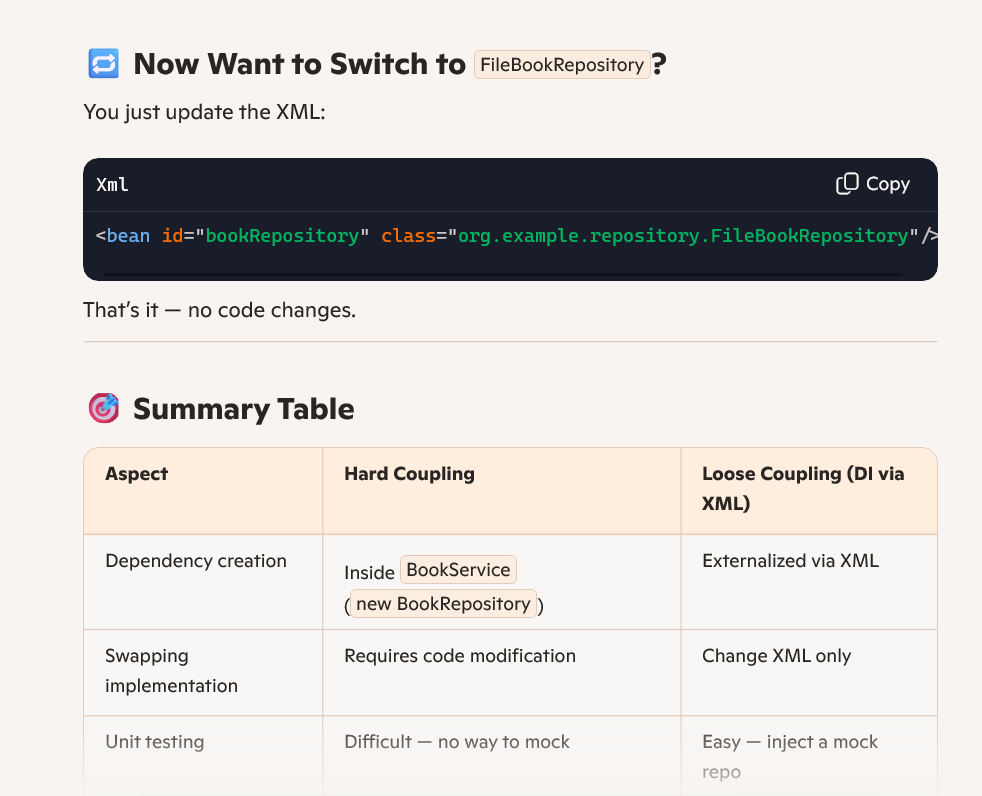
🔗 **Where Do These Links Come From?**

* They’re **standard URLs** used to identify Spring’s XML namespaces and schemas.
* Spring automatically maps these URLs to **local copies** of the .xsd files inside the Spring JARs.
* Your IDE (like IntelliJ or Eclipse) uses these links to provide **auto-completion** and **validation**.
* 💡 **Pro Tip:**  
  Use **versionless URLs** like spring-beans.xsd instead of spring-beans-5.3.xsd. They automatically match the version of Spring you’re using, thanks to internal mapping.

Generally, those two namespaces are **enough for basic bean configuration** in Spring XML. But if you want to use additional features like annotation scanning, utility tags, or p-namespace shortcuts, you’ll need to declare extra namespaces and schema locations.



****

****

**Code Notes:**

1. That line of XML:

<property name="bookRepository" ref="bookRepository"/>

is used for setter injection in Spring.

Let's break down what it means:

* <property>: This tag tells Spring that you want to set a property (a field) of the bean being defined.
* name="bookRepository": This specifies the name of the property (field) in your Java class that you want to set. So, in your BookService class, it expects a method like setBookRepository(BookRepository bookRepository). Spring will call this setter method.
* ref="bookRepository": This indicates that the value to be injected into the bookRepository property is another Spring bean. The ref attribute points to the id of the bean that Spring should inject. In this case, it's referring to the bean with the ID "bookRepository" (which would be your com.library.repository.BookRepository instance defined elsewhere in the applicationContext.xml).

1. 🧪 getBean() = Retrieve Already-Created Bean

* getBean() **doesn't create a new object every time** — it *retrieves an existing bean* from the container.
* Beans are usually singletons by default (unless configured otherwise), so calling getBean() repeatedly will return the **same object**.

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml"); BookService service1 = context.getBean("bookService", BookService.class); BookService service2 = context.getBean("bookService", BookService.class); System.out.println(service1 == service2); // true (singleton)



🔎 Bonus Insight:

* Behind the scenes, Spring initializes all singleton beans during context startup.
* So getBean() is more like saying: “Hey Spring, hand me the bean you’ve already prepared!”

1. getBean syntax

✅ **Basic Syntax with Bean Name**

Object bean = context.getBean("beanId");

* Returns a generic Object, so you’ll usually need to cast it.

✅ **Preferred: Type-Safe Syntax**

BookService service = context.getBean("bookService", BookService.class);

* 💡 Gives you compile-time type checking and avoids explicit casting.

✅ **By Type Only (if bean is unique)**

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

* Works only if there is **a single bean** of that type in the context.

⚠️ Common Pitfall

If there are multiple beans of the same type and you use the by-type version, Spring will throw:

NoUniqueBeanDefinitionException

Want a quick recap of how Spring resolves bean IDs versus types at runtime? Or maybe how to test getBean() with different scopes like singleton vs prototype?

**Spring Context:**

Spring Context is the **container** that wires everything together and manages your beans and dependencies throughout the application.

🧠 **What Is Spring Context?**

* It’s part of the **Spring ApplicationContext**, which extends BeanFactory and adds *extra features* like internationalization, event propagation, and AOP integration.
* It **loads bean definitions** from various sources (XML, annotations, or Java config) and wires them up based on your configuration.
* It **controls the lifecycle** of those beans—instantiation, dependency injection, initialization, and destruction.

🔧 **Key Responsibilities**

* Bean creation and injection (@Component, @Autowired, or via XML)
* Dependency management through **IoC** (Inversion of Control)
* Resource loading (like reading from a config file)
* Application events (publishing and listening to custom or built-in events)
* Internationalization (loading message bundles for different languages)

🗂️ **Types of Contexts You’ll Encounter**  
ClassPathXmlApplicationContext: Loads beans from XML in classpath AnnotationConfigApplicationContext: Uses @Configuration classes

WebApplicationContext: Used in Spring MVC for web apps

🔍 **Spring Core Contributions**

* Supplies core interfaces: BeanFactory, ApplicationContext, etc.
* Defines lifecycle management, scopes, and DI concepts
* Is the library that makes inversion of control *possible*

🛠️ **ApplicationContext Responsibilities**

* Loads bean definitions (XML, annotations, Java config)
* Instantiates and wires beans using **DI**
* Manages bean lifecycle, scopes, and events