Why is Spring so popular in developing Java Applications ?

1. Simplifies overall java development

2. Allows the developers to create loosely coupled applications.

- Spring Manages the dependencies, so that dependent objects are completely de coupled from the dependencies.

3. Supplies readymade implementation of multiple design patterns

- singleton , factory , proxy , MVC , front controller , API Gateway , Service Registry …
- Spring supports the MVC pattern for developing monolithic web application as well as RESTful web services.

4. Reduces Boilerplate code.

5. Helps to build Easy, Simple, and Lightweight applications.

6. Excellent support for JDBC as well as ORM , along with automatic transaction management support.

7. It has modular and non intrusive design.

8. Provides smooth integration with other frameworks such as Hibernate , Struts , EJB.

9. Supports AOP (Aspect oriented programming) for the separation & modularization of cross cutting concerns - repetitive tasks. It separates such recurring tasks from the core tasks

Eg. Allows clean separation between request handling tasks (in Controller) and exception handling (in Controller advice)

10. Supports Unit testing as well as integration testing.

11. Developing & Securing web applications as well as RESTful web services becomes easy .

12. Excellent support for building Micro services based distributed architecture.

13. Provides Spring AI as a thin layer of abstraction over different **AI / LLM providers**

And many more reasons

What is Spring ?

- It is a container, since it can manage the life cycle of any java object.

What is a spring bean ?

- It is a java class (object) whose life cycle is completely managed by SC(spring container)

Eg. of Spring beans - REST controller, Controller, Service,DAO.

- It is Framework of frameworks | modules .

- It is a framework since -
- It provides readymade implementation of standard design
- Example of Spring modules
  - Spring core , Web , ORM , Transactions, Spring boot , Spring security , Spring cloud …..

3. Overview of Spring framework modules

- Diagram
  - spring-framework-modules.png

4. The main reason of popularity behind Spring is

- Spring helps us build **loosely coupled applications** using **Inversion of Control** (IoC), which is achieved through **Dependency Injection** (DI).
- Traditionally , our **code creates objects**
  - Dependent Classes decide *what* dependencies to use and *how* to create them
- With **IoC**:
- **Spring creates and manages objects**
- Our code only *declares what it needs*
- This is how, Control is *inverted* from application code to the Spring container.

- **Dependency Injection (DI) — the mechanism**
- **DI is how IoC is implemented in Spring.**
- Spring:
  creates objects (beans)
  injects their dependencies automatically.

- Example:
- Appointment Controller depends upon Appointment Service , to book ,cancel , list of appointments. So instead of Controller creating the service, Spring creates it for us.
- Code sample –
- @Controller
- public class AppointmentController {

```
private final AppointmentService appointmentService;

    public AppointmentService(AppointmentService service) {

  this. appointmentService = service;   //ctreated & injected by Spring

}

}
```

- Here:

- AppointmentService **does not create** AppointmentRepository
- Spring injects it → **loose coupling**

Why this leads to loose coupling

- Dependent Classes depend on **interfaces**, not **implementations**.
- Object creation is **externalized**
- Dependencies can be changed without changing business logic

In the earlier **traditional programming model**, For example

- Hibernate based DAO layer was dependent upon SessionFactory.
- To build SessionFactory , a programmer has to write HibernateUtils class & build SessionFactory from the Configuration class.

This is the example of tight coupling, since

- Any time the nature of the dependency changes , dependent object is affected

(Meaning - you will have to make changes in dependent object as well)

Tight coupling is strongly undesirable

- Since it's difficult to maintain or extend the application.

What is D.I ?

(**Dependency injection**=**wiring**=**collaboration** between dependent & dependency)

- It is a design pattern that tries to decouple dependent objects from their dependencies.
- Instead of creating their own dependencies explicitly, dependent objects receive the dependencies from the 3rd party (Eg Spring Container SC) directly at run time.
- This approach leads to loose coupling ,which leads further to maintainable code .

Summary -

Hollywood principle states

- You don't call us , We will call you!

SC (Spring Container) is telling these dependent objects

- not to manage their dependencies , since it will be automatically provided by SC.

Diagram –

" spring_ioc_container.jpg"

Development steps, for standalone spring application using pure XML based configuration

- Create Maven based standalone Java project with spring

- Identify & Create dependent & dependency classes
- Under <src>/<main>/<resources> , create spring bean configuration xml file
- Add <beans> namespace.
- Add <bean> tags in the xml file

Important Attributes of <bean> tag

- **id** - mandatory attribute , refers to bean unique identifier
- **class** - mandatory attribute , refers to Fully qualified bean class name
- **scope** - optional

  Default **scope = singleton**

  In a standalone spring application , supported scopes are

  - singleton | prototype

  In a web application

  - singleton | prototype | request | session | global session (global session is supported ONLY under portlet based web application)
    For Singleton scoped bean
    - SC will create & share single bean instance for multiple requests/demands.
  - For prototype scoped bean
    - SC creates a NEW bean instance per request/demand.

- **lazy-init**

  - boolean attribute . Default value=false.

  - Applicable only to singleton beans.

- Prototype scoped beans are **always** created lazily one per demand.
- By default, SC will automatically create singleton spring bean instance at the application start up.
- **init-method**
  - to specify the name of custom initiali method
  - default pattern of the init method

  - public void anyMethodName() throws Exception

  {init logic}

  - It will be automatically called by SC after D.I (after calling the setters)

  - It will be called for singleton as well as prototype beans.

- **destroy-method**

  -to specify the name of custom destroy method

-default pattern of the destroy method

- public void anyName() throws Exception

{shutdown / cleanup logic}

- It will be automatically called by SC **before garbage collection of singleton** spring beans & won't be called for Prototype beans.

- Design **singleton beans as stateless always** .Otherwise you may risk:

- race conditions
- data leakage between users
- unpredictable bugs

- Can design **prototype scoped beans as stateful**

Spring Container API

Diagram

" SC-API-help.png"

The Spring container is implemented using BeanFactory and ApplicationContext interfaces where ApplicationContext is the fully featured container responsible for bean lifecycle, dependency injection, events, and AOP integration.

To start SC , in the **standalone environment**

- Create instance of o.s.context.support.**ClasspathXmlApplicationContext**(String configFile) throws BeansException
- Represents a standalone SC started using XML based instructions , loaded from run time class path (<resources>)
- To get ready to use spring bean from SC ?

  API of BeanFactory

  public <T> T getBean(String beanId,Class<T> beanClass) throws BeansException

  T - type of the spring bean

Spring Bean Life Cycle

Diagram

- spring-bean-life-cycle.png

- singleton vs prototype.png"

Modes of wiring (D.I) supported by SC

1.**Explicit wiring**

Programmer has to supply - java code(setters | parameterized constructors | factory method)

- xml configuration

1.1 **Constructor Based D.I**

- Used for injecting mandatory dependencies
- Dependency is created first & then dependent object.
- Steps
- In the dependent bean class
  - Add parameterized constructor with argument per dependency
- In xml file

  - <constructor-arg name|type|index value|ref/>

  one per dependency

  use name & value - for injecting values (primitive types, strings, enums , simple constants)

  use ref - for injecting another dependency bean reference.

1.2 **Setter based D.I**

- Used for injecting optional dependencies
- Dependent is created first & then dependency object
- Steps

  - In dependent bean class - add a setter per dependency.

  - In xml file

  Add **<property name value|ref/> ,** per dependency

  use name & value - for injecting values (primitive types, strings, enums , simple constants)

  use ref - for injecting another dependency bean reference.

1.3 **Factory method based D.I**

- SC can provide the dependencies even without any parameterized constructor or setters ,but using a factory method

Factory method=public static method returning bean instance

-Steps

- Add factory method in bean class

In xml

- Add factory-method="nameOfFactroyMethod" attribute in the <bean> tag.

- To supply arguments , to the factory method

<constructor-arg name|type|index value|ref/>

Since lot of efforts are needed in explicit wiring , enter **auto wiring.**

2. **auto wiring (implicit wiring)**

Add autowire attribute in <bean> tag.

- <bean ..... autowire="no|byName|byType|constructor"/>

default value of autowire="no"

2.1 Common features of autowire="**byName|byType**"

- Represents setter based D.I - for injecting optional dependencies.
- Programmer still needs to supply a setter per dependency in dependent bean class
- No need of <property> tags
- In the absence of a match , SC still continues , BUT causes B.L(Business Logic) failure (NullPointerException)

autowire="byName"

- SC tries to match the property name (derived via setter) with dependency bean id

autowire="byType"

- SC tries to match data type of the property (via setter) with data type of dependency bean
- In case of multiple matches
  - SC throws the exception **NoUniqueBeanDefinitionException** & it aborts.

2.2 **autowire="constructor"**

- Represents constructor based D.I - for injecting **mandatory** dependencies.
- Still needs to supply parameterized constructor with constructor argument per dependency in dependent bean class.
- No need of <constructor-arg> tags in bean configuration xml file
- SC tries to match data type of constructor argument with data type of dependency bean.

What will happen ?

1. In the absence of a match - SC throws exception - **NoBeanDefinitionFoundException** & aborts!

2. In case of multiple matches - SC throws the exception NoUniqueBeanDefinitionException & aborts!

3. In case of exact match - D.I succeeds.