**https://docs.google.com/document/d/1rCzehj4kG9y1VBEyDuUli8TlggVrFVyo66XO0hXqigc/edit?usp=sharing**

**Spring Core**

**Spring Framework Introduction**

The Spring Framework is an open-source application framework and inversion of control container for the Java platform. It’s one of the most popular frameworks for building Java applications, especially enterprise-level ones. Here’s a brief introduction to its key features:

* Core Container: At the heart of Spring is the Inversion of Control (IoC) container, which manages Java objects, called beans, through dependency injection.
* Aspect-Oriented Programming (AOP): Spring supports AOP, allowing you to define cross-cutting concerns (like logging or security) that can be applied across your application without modifying the main business logic.
* Data Access: Spring provides extensive support for interacting with databases through JDBC, Hibernate, JPA, and other data access technologies.
* Transaction Management: It offers a consistent abstraction for transaction management that scales down to local transactions and scales up to global transactions (JTA).
* Spring MVC: A model-view-controller framework that uses the Spring Framework’s features to develop web applications.
* Spring Boot: An extension of the Spring Framework that simplifies the initial setup and development of new Spring applications.
* Spring Security: A powerful and highly customizable authentication and access-control framework.
* Spring Cloud: Provides tools for developers to quickly build some of the common patterns in distributed systems.

**Spring Framework Architecture**

The Spring framework consists of seven modules which are shown in the above Figure. These modules are Spring Core, Spring AOP, Spring Web MVC, Spring DAO, Spring ORM, Spring context, and Spring Web flow. These modules provide different platforms to develop different enterprise applications; for example, you can use Spring Web MVC module for developing MVC-based applications.

**Spring Core Module:**   
 The Spring Core module, which is the core component of the Spring framework, provides the IoC container There are two types of implementations of the Spring container, namely, bean factory and application context. Bean factory is defined using the org.springframework.beans.factory.BeanFactory interface and acts as a container for beans.

**Bean Lifecycle and Factory**

In the Spring Framework, the lifecycle of a bean is managed by the Spring container. Here’s a simplified view of the lifecycle steps for a Spring bean:

1. **Instantiation**: The container creates an instance of the bean.
2. **Populate Properties**: The container injects the bean’s properties as specified in the Spring configuration file or annotations.
3. **Set Bean Name**: If the bean implements the BeanNameAware interface, the container passes the bean’s ID to the setBeanName() method.
4. **Set Bean Factory**: If the bean implements the BeanFactoryAware interface, the container calls the setBeanFactory() method, passing an instance of itself.
5. **Pre-Initialization**: Before handing over the bean to the client, the container calls any BeanPostProcessors configured in the context.
6. **Initialize Bean**: If the bean implements InitializingBean, the afterPropertiesSet() method is called. Also, any custom init-method defined in the bean configuration is called.
7. **Post-Initialization**: After the initialization, the BeanPostProcessors can modify the bean.
8. **Ready for Use**: Now the bean is ready to be used by the application.
9. **Destroy**: When the container shuts down, if the bean implements DisposableBean, the destroy() method is called. Also, any custom destroy-method is invoked.

The **Bean Factory** is the actual container that instantiates, configures, and manages a number of beans. It provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality. The BeanFactory is a sophisticated implementation of the factory pattern and is able to maintain a registry of different beans and their dependencies.

**XML**

<bean id="myBean" class="com.example.MyBean">

<property name="myProperty" value="someValue"/>

</bean>

**Java**

package com.example;

public class MyBean {

private String myProperty;

public void setMyProperty(String myProperty) {

this.myProperty = myProperty;

}

// other methods

}

The Spring container will manage the lifecycle of myBean, from creation to destruction, according to the bean definition and the interfaces it implements.

**What Is the Spring Container?**

The Spring container is a core component of the Spring Framework. It is responsible for creating and managing the lifecycle and configuration of application objects (beans). The container reads configuration metadata to know how to instantiate, configure, and assemble these beans. This metadata can be supplied in various forms, such as XML, annotations, or Java code.

**Responsibilities of the IoC Container**

* **Instantiating beans**: Creating instances of the beans defined in the configuration.
* **Wiring beans**: Managing dependencies between beans.
* **Configuring beans**: Setting properties and managing initialization and destruction callbacks.
* **Managing the bean lifecycle**: Handling the complete lifecycle of a bean from creation to destruction.

Spring provides two main types of containers:

1. **BeanFactory**: The simplest container providing basic dependency injection features.
2. **ApplicationContext**: An enhanced container providing more enterprise-specific functionality such as event propagation, declarative mechanisms to create a bean, and various ways to look up.

The org.springframework.beans and org.springframework.context packages are fundamental to the Spring Framework’s IoC container.

**How the Spring IoC Container Works**

1. **Read Dependency and Configuration Metadata**: The Spring IoC container reads configuration metadata, which can be supplied in XML, annotations, or Java-based configurations.
2. **Create Dependency Objects and Inject Them**: Based on the configuration metadata, the container creates and injects dependencies into business objects (POJOs).

**What is Configuration Metadata?**

Configuration metadata is how you instruct the Spring container about the objects it should manage. It defines how beans are created, configured, and assembled within the Spring IoC container. There are three primary ways to provide configuration metadata to the Spring container:

1. [**XML-based configuration**](https://www.javaguides.net/2018/10/spring-ioc-container-xml-config-example.html): The traditional way to define beans and dependencies in XML files.
2. [**Annotation-based configuration**](https://www.javaguides.net/2018/07/spring-annotation-based-container-configuration.html): Uses annotations in Java classes to define beans and their dependencies.
3. [**Java-based configuration**](https://www.javaguides.net/2018/06/spring-java-based-configuration-basics.html): Uses Java classes annotated with @Configuration and methods annotated with @Bean to define beans and dependencies.

**How to Create a Spring Container?**

Spring provides various implementations of the ApplicationContext interface, each suited for different scenarios. Here are the common ones:

1. **AnnotationConfigApplicationContext**: Used for standalone Java applications with annotation-based configuration.
2. **ClassPathXmlApplicationContext**: Used for standalone applications with XML configuration loaded from the classpath.
3. **FileSystemXmlApplicationContext**: Similar to ClassPathXmlApplicationContext, but the XML configuration file can be loaded from anywhere in the file system.
4. **AnnotationConfigWebApplicationContext**: Used for web applications with annotation-based configuration.
5. **XmlWebApplicationContext**: Used for web applications with XML configuration.

**Creating a Spring Container with XML Configuration**

**Create a Maven project with the dependency.**

<dependencies>  
<dependency>  
<groupId>org.springframework</groupId>  
<artifactId>spring-context</artifactId>  
<version>6.0.1</version>  
</dependency>  
</dependencies>

**Message.java**

package org.example;  
  
 public class Message  
 {  
 private String msg;  
  
 public void getMsg() {  
 System.*out*.println("This is the message :"+msg);  
 }  
  
 public void setMsg(String msg) {  
 this.msg = msg;  
 }  
 }

**Create applicationConetext.xml file in the Resource folder**

**applicationContext.xml**

*<?*xml version="1.0" encoding="UTF-8" *?>*<beans xmlns = "[http://www.springframework.org/schema/beans"](http://www.springframework.org/schema/beans%22) xmlns:xsi = "[http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance%22) xsi:schemaLocation="<http://www.springframework.org/schema/beans>[http://www.springframework.org/schema/beans/spring-beans.xsd">](http://www.springframework.org/schema/beans/spring-beans.xsd%22%3E)<bean id="message" class="org.example.Message">  
<property name="msg" value="Welcome"/>  
</bean>  
  
</beans>

**Application.java**

package org.example;  
 import org.springframework.context.ApplicationContext;  
 import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
 public class Application {  
 public static void main(String args[])  
 {  
 ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
 Message ob = (Message)context.getBean("message");  
 ob.getMsg();  
 Demo obj = (Demo)context.getBean("demo");  
 obj.getMsg();  
 }  
 }

has context menu

has context menu

**Dependency injection (DI) Overview**

Dependency injection is a fundamental concept in the Spring framework. It refers to a design pattern where objects are provided with their dependencies rather than creating them internally.  
  
In Spring, Dependency Injection (DI) is achieved through the Inversion of Control (IoC) container. The container manages the creation and lifecycle of objects and injects the required dependencies into the object when it is created.  
  
The DI mechanism is implemented through constructor injection, setter injection, or field injection. Constructor injection involves passing the required dependencies through the constructor of the object. Setter injection involves injecting dependencies through the object's setter methods, and field injection involves injecting dependencies directly into the object's fields.  
  
The advantages of using dependency injection in Spring include increased modularity, flexibility, and testability of the application. By externalizing the configuration of dependencies, it becomes easier to swap out implementations and test individual components in isolation.

**Types of Dependency Injection**

1. Constructor-based dependency injection
2. Setter-based dependency injection
3. Field-based dependency injection
4. **Constructor-based dependency injection**

package org.example;  
  
public interface Message  
{  
 public void sendings(String message);  
}

package org.example;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Component;

@Component

public class ConstructorBased

{

private Message message;

@Autowired

public ConstructorBased(@Qualifier("A")Message message) {

super();

this.message = message;

}

public void receiving(String message) {

this.message.sendings("message");

}

}

package org.example;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Application

{

public static void main(String args[])

{

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfiguration.class);

ConstructorBased constructorBased = applicationContext.getBean(ConstructorBased.class);

constructorBased.receiving("This is from C");

}

}

package org.example;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("org.example")

public class AppConfiguration

{

}

package org.example;

import org.springframework.stereotype.Service;

@Service("A")

public class A implements Message

{

public void sendings(String message)

{

System.out.println("This is A class sending"+message);

}

}

package org.example;

import org.springframework.stereotype.Service;

@Service("B")

public class B implements Message

{

public void sendings(String message)

{

System.out.println("This is B class sending"+message);

}

}

package org.example;

import org.springframework.stereotype.Service;

@Service("C")

public class C implements Message

{

public void sendings(String message)

{

System.out.println("This is C class sending"+message);

}

}

[5:58 PM] Madabattula Rani (Unverified) (External)

**Day 2: Spring Core - Policy Administration Backend**

**Task 1**: Refactor policy-related operations to utilize Spring Beans and Dependency Injection.

**Task 2**: Implement Spring validation on the server side to ensure policy data integrity.

**Task 3**: Set up Application Context and Bean Factory for a scalable backend structure.