FPT Education
FPT UNIVERSITY

Java

Introduction to Spring Framework

Objectives

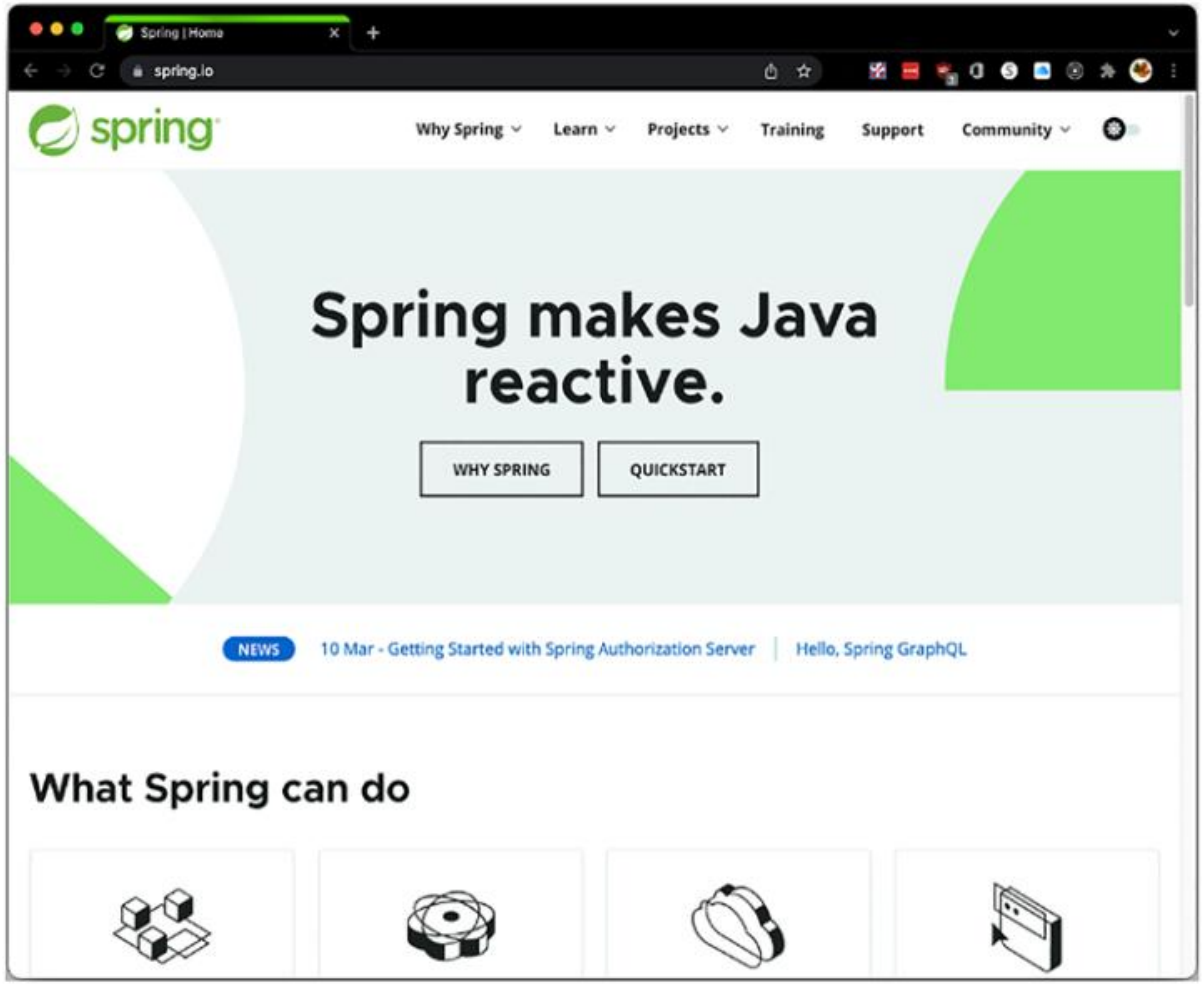
- What is Spring Framework?
- Advantages of using Spring Framework
- Key features of Spring Framework
  - Dependency Injection and Inversion of Control
  - Aspect Oriented Programming

Spring Framework Overview

What is Spring Framework?

- Spring is the most popular application development framework for enterprise Java.
- Open source Java platform since 2003.
- Spring supports all major application servers and JEE standards.
- Spring handles the infrastructure so you can focus on your application.
- Spring is the most popular application development framework for enterprise Java.
- Millions of developers around the world use Spring Framework to create high performing, easily testable, reusable code.

What is Spring Framework?



Spring Version History

Version	Year	Note
	2002	First version was written by Rob Johnson
1.0	March 2004	
2.0	October 2006	
2.5	November 2017	
3.0	December 2009	
4.0	December 2013	Support for Java SE (Standard Edition) 8, Groovy 2, some aspects of Java EE 7, and Web Socket
4.3	June 2016	
5.0	March 2017	5.2.2 (December 2019)
6.0	November 2022	6.1.5 (March 2024), JDK 17

Spring, Java and Java EE compatibility

Spring	Java Version	Java EE Version
1.x	Full Support for JDK 1.3, 1.4	Java EE 1.3, 1.4 are fully supported.
2.0.x	Full Support for JDK 1.3, 1.4 and 1.5	Java EE 1.3, 1.4 are fully supported.
2.5	Full support for Java SE 1.4.2 or later, Early Support for Java SE 6.	Java EE 1.3, 1.4 are fully supported. Early support for Java EE 5.
3.x	Java SE 5 and 6 are fully supported	Java EE 1.4, 5 are fully supported. Early support for Java EE 6.
4.x	Java SE 6, 7 are fully supported. Early support for Java SE 8.	Java EE 6 is fully supported Early support for Java EE 7
5.x	Java SE 6, 7, 8 are fully supported Early Support for Java SE 9	Java EE 6, 7 are fully supported
6.x	JDK 17	Jakarta EE

Spring – Evolution over Intelligent Design

- The Dependency Inversion Principle (DIP) is a fundamental concept in software design and architecture, and it's a key principle behind the Spring Framework.

- The Dependency Inversion Principle
  - High level modules should not depend upon low level modules. Both should depend upon abstractions.
  - Abstractions should not depend upon details, details should depend upon abstractions. Depend upon Abstractions, Do not depend upon Concrete Classes

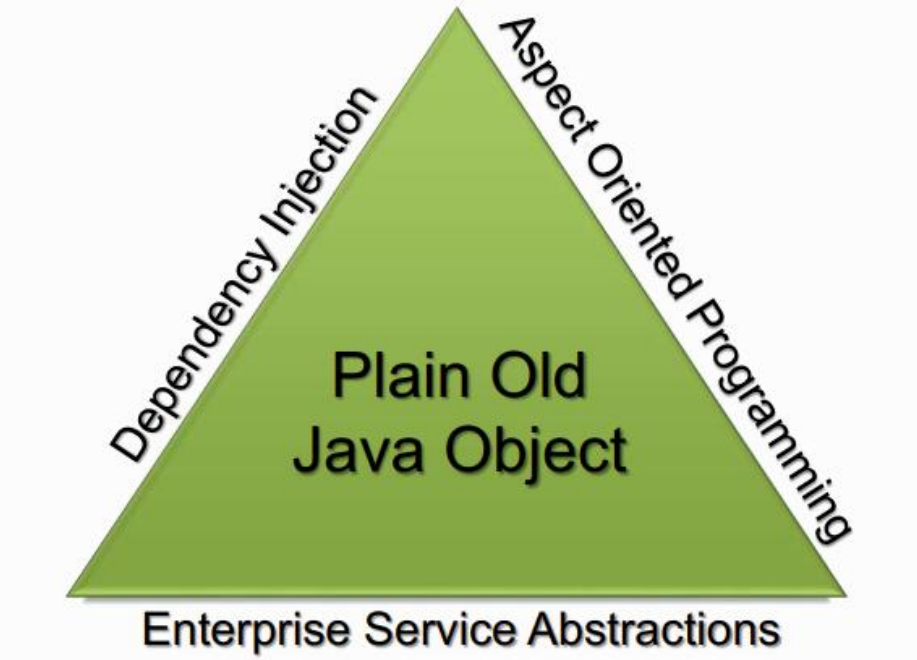
Spring Features

- Core technologies: dependency injection, events, resources, i18n, validation, data binding, type conversion, SpEL, AOP.
- Testing: mock objects, TestContext framework, Spring MVC Test, WebTestClient.
- Data Access: transactions, DAO support, JDBC, ORM, Marshalling XML.
- Spring MVC and Spring WebFlux web frameworks.
- Integration: remoting, JMS, JCA, JMX, email, tasks, scheduling, cache and observability.
- Languages: Kotlin, Groovy, dynamic languages.

Maven

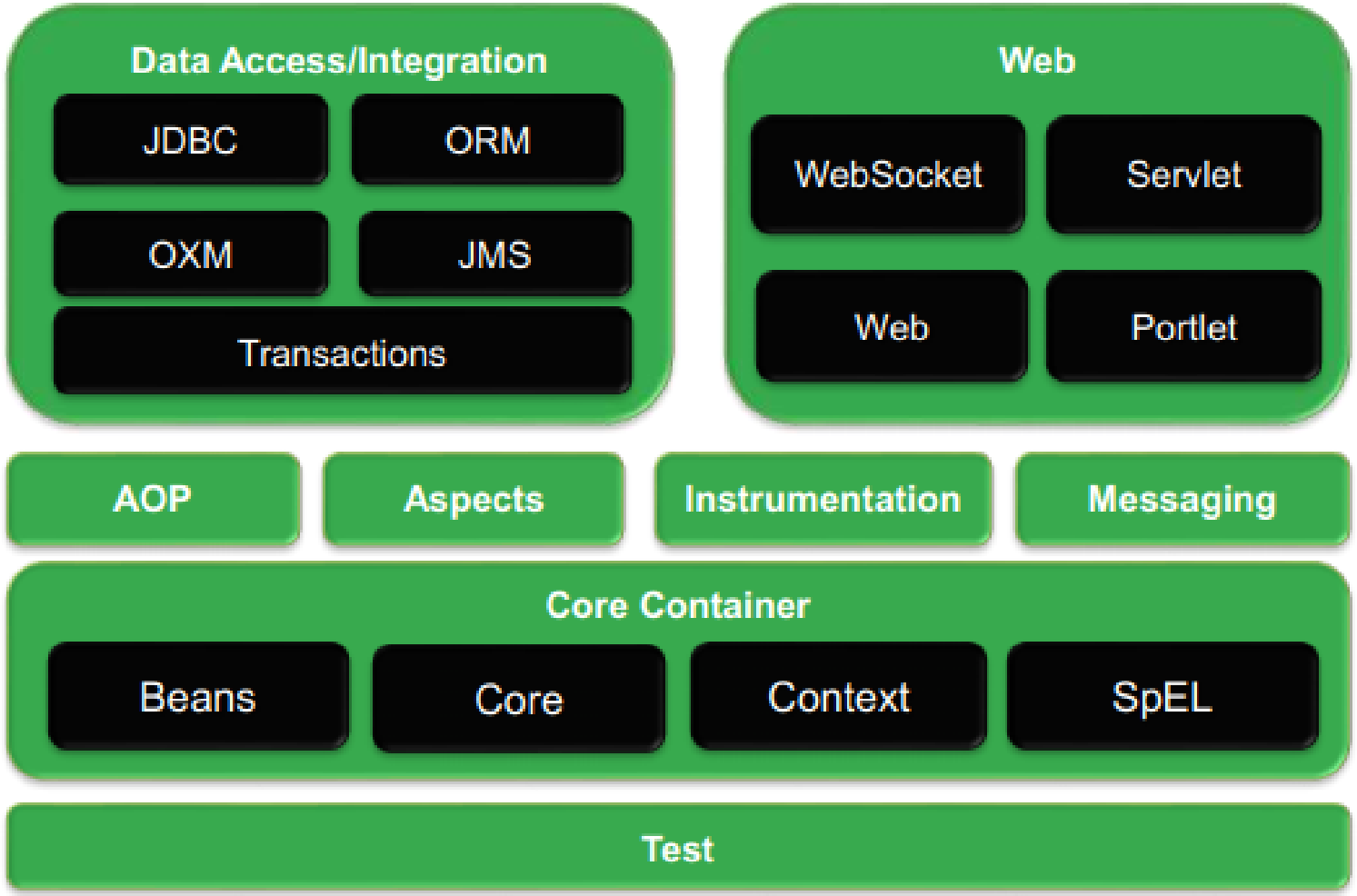
- Using Maven for Spring Framework development streamlines the dependency management, build process, project structure, and overall development workflow.
- It enhances collaboration, reduces configuration overhead, and ensures a standardized approach to building Spring applications.
- Reasons to use Maven for Spring applications
  - Dependency Management
  - Build Automation
  - Consistent Project Structure
  - Dependency Scope Management
  - Plugin Ecosystem
  - Transitive Dependency Resolution
  - Easy Project Configuration

The Spring Triangle



- *"Spring's main aim is to make J2EE easier to use and promote good programming practice. It does this by enabling a POJO-based programming model that is applicable in a wide range of environments." – Rod Johnson*

Architectural overview of Spring Framework



Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details:

- The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The **Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
- The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
- The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration

The Data Access/Integration layer consists of the modules:

- The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service **JMS** module contains features for producing and consuming messages.
- The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules:

- The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
- The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Spring Framework 6

- **Java 17 Support** - Spring Framework 6 is compatible with Java 17, providing developers with access to the newest Java functionalities and improvements.
- **Reactive Programming** - Enhanced support for reactive programming is a significant highlight of Spring Framework 6, enabling developers to build responsive, scalable applications efficiently.
- **Modularization** - The framework is modularized, allowing developers to choose and include only the necessary components, reducing the overhead of unused features.
- **Microservices Architecture** - Spring Framework 6 caters to the needs of building microservices-based applications, promoting flexibility and scalability in the development process.

Spring Framework 6

- Kotlin Support - Kotlin, a programming language that runs on the Java Virtual Machine and is fully interoperable with Java, is well-supported in Spring Framework 6 for developers who prefer it as an alternative language for their projects.
- Containerization and Cloud-native Development - With enhanced containerization support, Spring Framework 6 facilitates cloud-native development, aligning well with modern deployment practices such as container orchestration.
- Simplified Configuration - Spring Framework 6 simplifies configuration options, making it easier for developers to set up and manage their applications with less complexity.

Spring Framework 6

- **Developer Productivity** - The framework aims to enhance developer productivity by offering streamlined tools, integrations, and documentation to expedite the development process.
- **Spring Boot 3.0** is also the first Spring Boot GA release to support Spring Framework 6.0. As a developer, we need to be aware of these updates in order to work smoothly with Spring Framework. Undoubtedly, one of the biggest turns in the Spring Framework 6 release was the dropping of support for older versions of Java.
- The Spring Framework 6 is migrated towards Jakarta EE from Java EE. Hence, It will use the 'jakarta' packages namespace in place of 'javax' namespace.

Dependency Injection & Inversion of Control

Understanding Dependency Injection

- SOLID is a set of five object-oriented design principles that help in creating more maintainable, flexible, and scalable software.
  - Single Responsibility Principle (SRP)
  - Open/Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)
- Dependency Injection (DI) is a design pattern used to facilitate Inversion of Control (IoC) and improve the modularity and maintainability of software systems.

Benefits of Dependency Injection

- **Improved Testability** - By injecting dependencies into classes, it becomes easier to isolate components for unit testing without the need for complex setup or mocking frameworks. This promotes a more reliable and efficient testing process.
- **Reduced Coupling** - Dependency Injection helps reduce tight coupling between classes by allowing dependencies to be provided from external sources.
- **Enhanced Modularity** - Implementing Dependency Injection promotes modular design by clearly defining and managing the dependencies between various components.
- **Flexibility and Scalability** - Dependency Injection enables a more flexible and scalable architecture, as components can be easily replaced or extended without modifying the existing codebase.

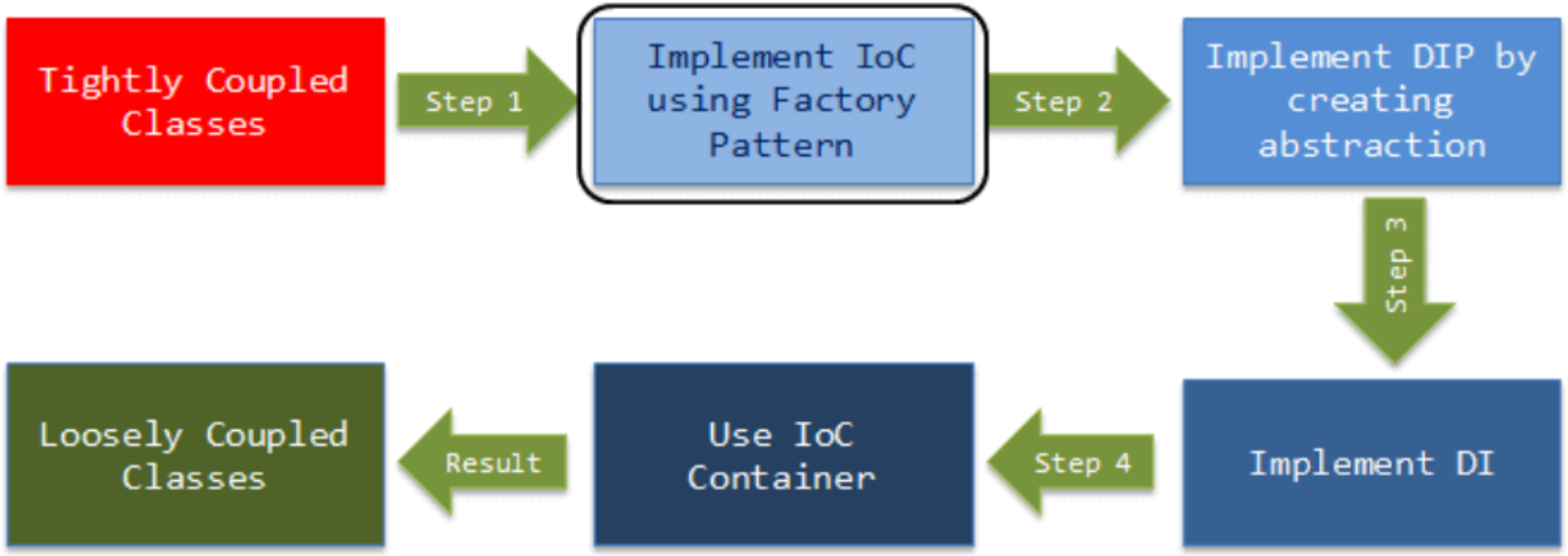
Benefits of Dependency Injection

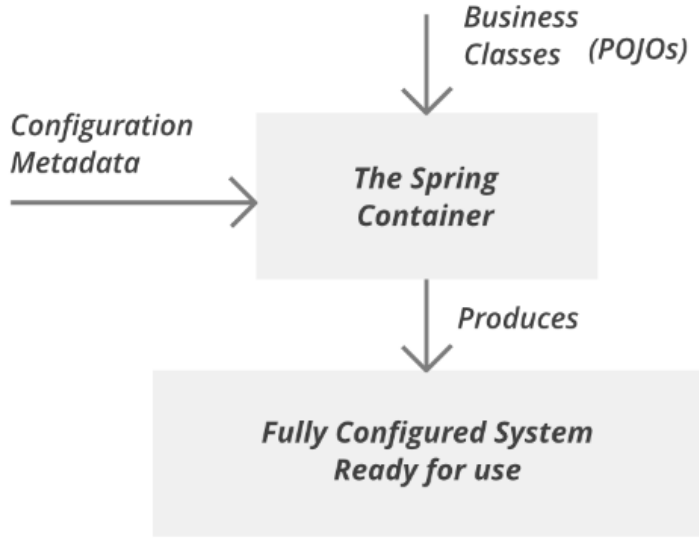
- **Promotes Design Patterns** - Dependency Injection encourages the adoption of design patterns such as Inversion of Control (IoC) and Dependency Inversion Principle (DIP). These patterns contribute to better code organization, improved readability, and overall software design quality.
- **Encourages Separation of Concerns** - Dependency Injection supports the separation of concerns principle by clearly defining the roles and responsibilities of different components.
- **Integration with DI Containers** - Dependency Injection can be utilized in conjunction with Dependency Injection Containers (such as Spring Framework or Google Guice), which help manage the injection of dependencies and further streamline the development process.

Inversion of Control

- Inversion of Control (IoC) is a design principle where the control over object creation and management is taken away from the objects themselves and inverted (controlled externally).
- In traditional programming, objects are responsible for creating and managing their dependencies, leading to tightly coupled and less maintainable code. With Inversion of Control, the responsibility of creating and managing objects and their dependencies is shifted to an external entity.

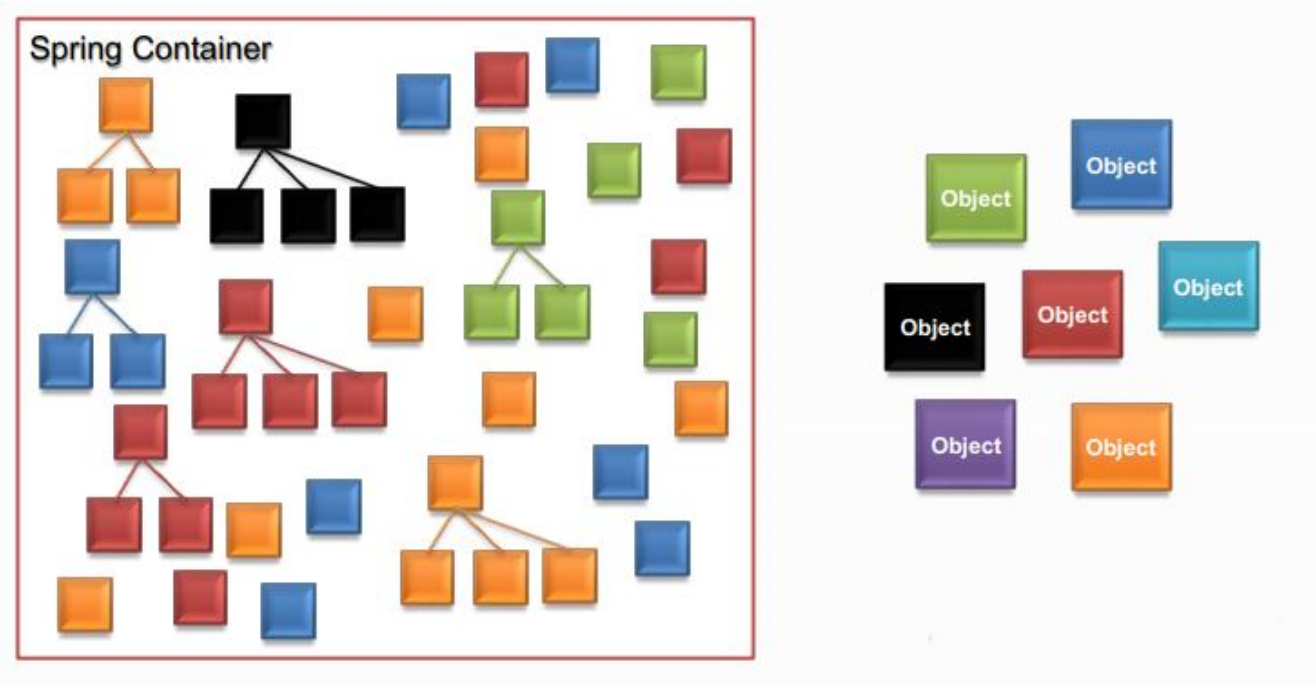
Inversion of Control



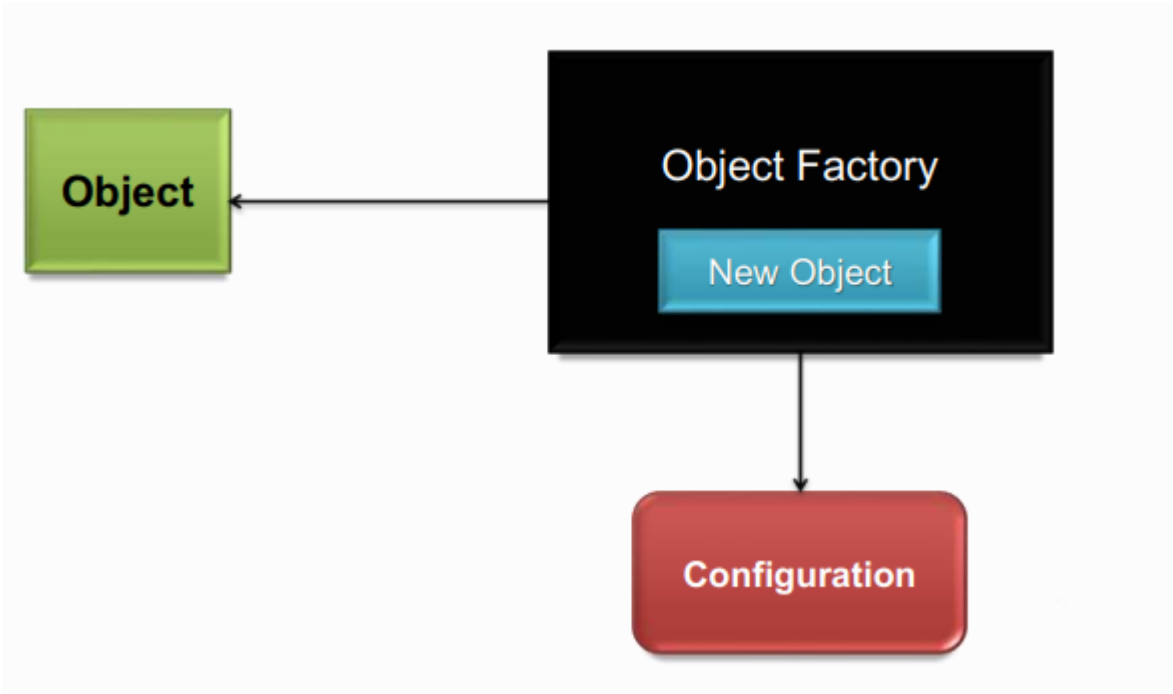


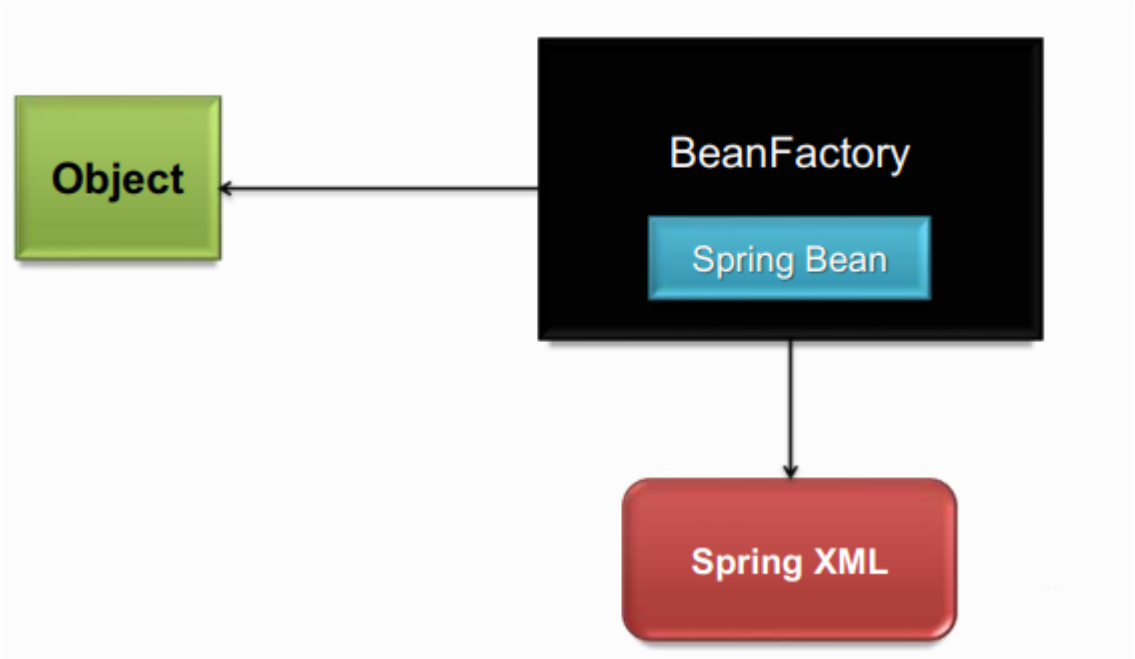
Spring Container

- Built-in Spring Containers
  - BeanFactory Container
  - ApplicationContext Container



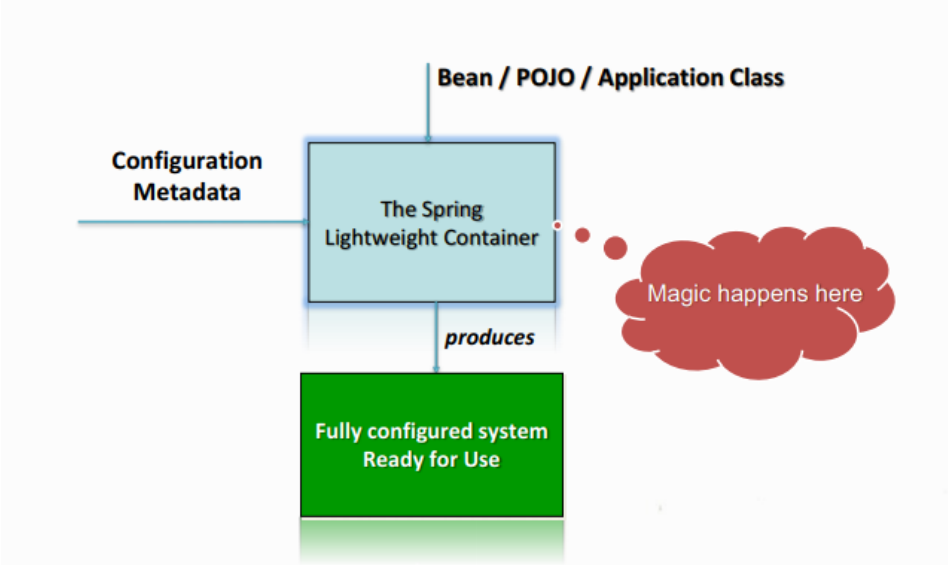
Factory Pattern - Spring Bean Factory





Spring IoC Container

- The Spring IoC creates the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction.
- The Spring container uses dependency injection (DI) to manage the components that make up an application.



Spring IoC Container

- BeanFactory Container
  - A BeanFactory is an implementation of the factory pattern that applies Inversion of Control to separate the application's configuration and dependencies from the actual application code. The most commonly used BeanFactory implementation is the XmlBeanFactory class.
  - This is the simplest container providing basic support for DI. The BeanFactory is usually preferred where the resources are limited like mobile devices or applet based applications

Spring IoC Container

- ApplicationContext Container
  - In addition to the capabilities of a BeanFactory Container, this container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

Instantiating a Spring IoC Container

Packages that are basis for the Spring IOC Container

1. Spring **BeanFactory** Container
   - **org.springframework.beans.factory.BeanFactory**
   - The BeanFactory interface provides a configuration mechanism capable of managing any type of object
2. Spring **ApplicationContext** Container
   - **org.springframework.context.ApplicationContext**
   - ApplicationContext is a sub-interface of BeanFactory.

Instantiating a Spring IoC Container

Can instantiate a Spring IOC container using any one of the below two classes.

- Spring Bean Factory Container

  org.springframework.beans.factory.BeanFactory

- Spring ApplicationContext Container

  org.springframework.context.ApplicationContext

Spring ApplicationContext Container

- Spring's more advanced container.
- Similar to BeanFactory it can load bean definitions, wire beans together and dispense beans upon request.
- Additionally it adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.
- Defined by the **org.springframework.context.ApplicationContext** interface.

Spring ApplicationContext Container

- The most commonly used ApplicationContext implementations are:
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - XmlWebApplicationContext

Spring ApplicationContext Container

- ClassPathXmlApplicationContext
  - This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
- FileSystemXmlApplicationContext
  - This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
- XmlWebApplicationContext
  - This container loads the XML file with definitions of all beans from within a web application.

BeanFactory vs ApplicationContext

BeanFactory	ApplicationContext
The simplest factory, mainly for DI	The advanced and more complex factory
Saving Resources: Used when resources are limited, e.g., mobile, applets etc.	Used elsewhere and has the below features, 1> Enterprise aware functions 2> Publish application events to listeners 3> Wire and dispose beans on request
Package: org.springframework.beans.factory.BeanFactory	Package: org.springframework.context.ApplicationContext
Implementation: BeanFactory factory = new XmlBeanFactory (new ClassPathResource(“wildLife.xml"));	Implementation: ApplicationContext context = new ClassPathXmlApplicationContext(“wildLife.xml”);

BeanFactory vs ApplicationContext

The additional features of Application Context:

- Provide a means for resolving text messages, including support for internationalization (i18N) of those messages, where as BeanFactory doesn't support. provide a generic way to load file resources, such as images.
- Publish events to beans that are registered as listeners.
- Certain operations on the container or beans in the container, which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context.
- ApplicationContext provides integration with Spring AOP.

Types of Dependency Injection

- *Dependency injection (DI)* comes with two flavors
  - Constructor-based Dependency Injection
    - Accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.
    - Calling a static factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to a static factory method similarly.
  - Setter-based Dependency Injection
    - Accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

Beans

- The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans.
- A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.
- These beans are created with the *configuration metadata* that you supply to the container, for example, in the form of XML <bean/> definitions

Beans

- The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans.
- A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.
- These beans are created with the *configuration metadata* that you supply to the container, for example, in the form of XML <bean/> definitions

Beans - Definition

Property	Description
class*	The bean class to be used to create the bean.
name	The unique bean identifier.
scope	The scope of the objects created from a particular bean definition.
autowiring mode	Used to specify autowire mode for a bean definition
lazy-initialization mode	Tells the IoC container to create a bean instance when it is first requested, rather
constructor-arg	Used to inject the dependencies into the class through a class constructor
properties	Used to inject the dependencies into the class through setter methods
initialization method	A callback to be called just after all necessary properties on the bean have been set by the container.
destruction method	A callback to be used when the container containing the bean is destroyed.

Bean Scopes

Scope	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request *	This scopes a bean definition to an HTTP request.
session *	This scopes a bean definition to an HTTP session.
global-session *	This scopes a bean definition to a global HTTP

Naming Spring Beans

- Every bean has one or more identifiers.
- Identifiers must be unique within the container that hosts the bean.
- A bean usually has only one identifier, but if it requires more than one, the extra ones can be considered *aliases*.
- The id attribute allows you to specify exactly one id.
- If you want to introduce other aliases to the bean, you can specify them in the name attribute, separated by a comma (,), semicolon (;), or white space.
- You are not required to supply a name or id for a bean. If no name or id is supplied explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name you must provide a name.
- Motivations for not supplying a name are related to using inner beans and *autowiring* collaborators.

Beans in Spring Configuration File

Tag Name	Description	Example
id	Unique Id	<bean id="person" … />
name	Unique Name	<bean name="lion" … />
class	Fully qualified Java class name	<bean class="a.b.C" … />
scope	Bean object type	<bean scope="singleton" … />
constructor-arg	Constructor injection	<constructor-arg value="a" />
property	Setter injection	<property name="a" … />
autowire	Automatic Bean referencing	<bean autowire="byName" … />
lazy-init	Create a bean lazily (at its first request)	<bean lazy-init="true" … />
init-method	A callback method just after bean creation	<bean init-method="log" … />
destroy-method	A callback just before bean destruction	<bean destroy-method="log" … />

Beans – Definition Inheritance

- Spring Bean definition inheritance has nothing to do with Java class inheritance.
- You can define a parent bean definition as a template and other child beans can inherit required configuration from the parent bean.
- A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.
- When you use XML-based configuration metadata, you indicate a child bean definition by using the parent attribute, specifying the parent bean as the value of this attribute.

Beans - Lifecycle

- The life cycle of a Spring bean is easy to understand.
- When a bean is instantiated, it may be required to perform some initialization to get it into a usable state.

- When the bean is no longer required and is removed from the container, some cleanup may be required.

Beans – Lifecycle – Initialization

- The org.springframework.beans.factory.InitializingBean interface specifies a single method:

```
void afterPropertiesSet() throws Exception;
```

```
public class ExampleBean implements InitializingBean {
   public void afterPropertiesSet() {
      // do some initialization work
   }
}
```

Beans – Lifecycle – Initialization

- In the XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature.

```
<bean id="..." class="..." init-method="init"/>
```

```
<bean id = "exampleBean" class = "examples.ExampleBean" init-method = "init"/>
```

```
public class ExampleBean {
   public void init() {
      // do some initialization work
   }
}
```

Beans – Lifecycle – Initialization

- Annotate the method with @PostConstruct

```
@PostConstruct
public void init() {
    ...
}
```

- A PostConstruct interceptor method must not throw application exceptions, but it may be declared to throw checked exceptions including the java.lang.Exception if the same interceptor method interposes on business or timeout methods in addition to lifecycle events.
- If a PostConstruct interceptor method returns a value, it is ignored by the container.

Beans – Lifecycle - Destruction

- The org.springframework.beans.factory.DisposableBean interface specifies a single method:

```
void destroy() throws Exception;
```

- Can simply implement the above interface and finalization work can be done inside destroy() method as follows

```
public class ExampleBean implements DisposableBean {
   public void destroy() {
      // do some destruction work
   }
}
```

Beans – Lifecycle – Destruction

- In the XML-based configuration metadata, you can use the destroy-method attribute to specify the name of the method that has a void no-argument signature.

```
<bean id="..." class="..." destroy-method="destroy"/>
```

- Example

```
<bean id = "exampleBean" class = "examples.ExampleBean" destroy-method = "destroy"/>
public class ExampleBean {
   public void destroy() {
      // do some destruction work
   }
}
```

Beans – Lifecycle – Destruction

- Annotate the method with @PreDestroy

```
@PreDestroy
public void destroy() {
    ...
}
```

Default initialization and destroy methods

- If you have too many beans having initialization and/or destroy methods with the same name, you don't need to declare **init-method** and **destroy-method** on each individual bean.
- Instead, the framework provides the flexibility to configure such situation using **default-init-method** and **default-destroy-method** attributes on the <beans> element as follows

```
<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-3.0.xsd"
   default-init-method = "init"
   default-destroy-method = "destroy">

   <bean id = "..." class = "...">
      <!-- collaborators and configuration for this bean go here -->
   </bean>

</beans>
```

Instantiating Beans

- Instantiating with a Constructor

```
<bean id="exampleBean" class="examples.ExampleBean" />
<bean name="anotherExample" class="examples.ExampleBeanTwo" />
```

- Instantiating with a Static Factory Method

```
<bean id="clientService“ class="examples.ClientService“ factory-method="createInstance"/>
public class ClientService {
        private static ClientService clientService = new ClientService();
        private ClientService() {}
        public static ClientService createInstance() {
                return clientService;
}
```

Instantiating Beans

- Instantiation using an instance factory method

```
<!-- the factory bean, which contains a method called createClientServiceInstance () -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
        <!-- inject any dependencies required by this locator bean -->
</bean>
<!-- the bean to be created via the factory bean -->
<bean id="clientService" factory-bean="serviceLocator"
factory-method="createClientServiceInstance" />
```

```
public class DefaultServiceLocator {
    private ClientService clientService = new ClientServiceImpl();

    private DefaultServiceLocator() {}

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

Instantiating Beans

- Instantiating more than one instance factory methods

```
public class DefaultServiceLocator {
    private static ClientService    clientService = new ClientServiceImpl();
    private static AccountService accountService = new AccountServiceImpl();
    private DefaultServiceLocator() {}
    public ClientService createClientServiceInstance() {
        return clientService;
    }
    public AccountService createAccountServiceInstance() {
        return accountService;
    }
}
```

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator" />
<bean id="clientService" factory-bean="serviceLocator" factory-method=
"createClientServiceInstance" />
<bean id="accountService" factory-bean="serviceLocator" factory-method=
"createAccountServiceInstance" />
```

Bean Aliasing

- In a bean definition, you can supply more than one name for the bean
  - by using a combination of up to one name specified by the id attribute, and any number of other names in the name attribute.
    <bean id="employee" name= "emp1, emp2, emp3" ... />
  - These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.
    <alias name="fromName" alias="toName"/>
  - This is useful in case we need to use beans that exist in a different sub-system.

Spring - Bean Definition Inheritance

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

```
<bean id = "helloWorld" class = "com.HelloWorld">
   <property name = "message1" value = "Hello 1"/>
   <property name = "message2" value = "Hello 2"/>
</bean>
<bean id = "helloWorld2" class = "com.HelloWorld2" parent = "helloWorld">
   <property name = "message1" value = "Hello 1 from HelloWorld2"/>
   <property name = "message3" value = "Hello 3 from HelloWorld2"/>
</bean>
```

Configuration & Advanced Features

The @Autowired annotation in the Spring

- Dependency Injection: The @Autowired annotation is used to automatically inject dependent beans into the associated fields, constructors, or methods within a Spring component.
- Field Injection: In the case of field injection, you can directly annotate the field to be injected using @Autowired.

```
@Component
public class MyComponent {
    @Autowired
    private MyDependency dependency;
    // ...
}
```

The @Autowired annotation in the Spring

- Constructor Injection: Alternatively, you can use @Autowired on a constructor to automatically inject the dependencies.

```
@Service
public class MyService {
    private final MyRepository repository;
    @Autowired
    public MyService(MyRepository repository) {
        this.repository = repository;
    }
    // ...
}
```

The @Autowired annotation in the Spring

- Method Injection: The @Autowired annotation can also be used on methods, allowing for method-level dependency injection.

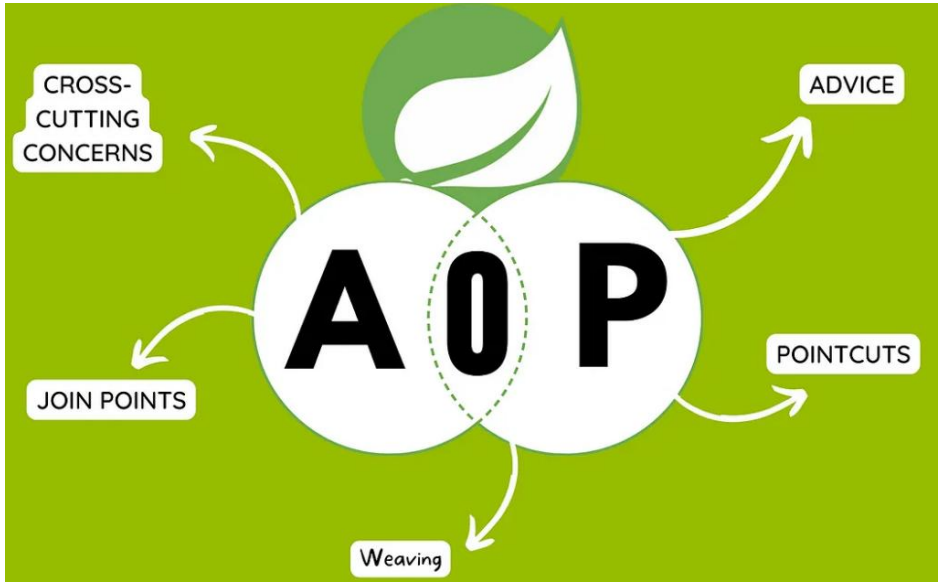
```
@Controller
public class MyController {
    private MyService service;
    @Autowired
    public void setService(MyService service) {
        this.service = service;
    }
    // ...
}
```

The @Autowired annotation in the Spring

- Qualifiers: In situations where there are multiple beans of the same type, the @Qualifier annotation can be combined with @Autowired to specify the exact bean to be injected.
- Optional Injection: The @Autowired annotation supports optional injections, meaning that if the specified bean is not found, the injection will be gracefully handled.

Aspect-Oriented Programming (AOP)

- AOP is a programming paradigm that enables modularization of cross-cutting concerns in software systems.
- In Spring, AOP complements OOP by providing a way to dynamically add behavior to the existing code without modifying it directly.



AOP Core Concepts

- **Aspect:** A module that encapsulates behaviors affecting multiple classes. In Spring AOP, aspects are implemented using regular classes annotated with @Aspect.
- **Advice:** Defines the additional behavior to be applied at a particular join point. Types of advice include "before", "after", "around", etc.
- **Join Point:** A point during the execution of a program where an aspect can be plugged in.
- **Pointcut:** A set of join points where advice should be executed. It defines the expressions that target specific methods.

Benefits of AOP

- Modularity: Separation of cross-cutting concerns into aspects promotes cleaner and more maintainable code.
- Reusability: Aspects can be applied to multiple classes or components.
- Cleaner Code: Business logic remains clean and focused, undisturbed by cross-cutting concerns like logging, security, etc.
- Dynamic Application: Aspects can be added or removed without modifying the core application code.

Summary

Concepts were introduced:

- Spring Framework
- Advantages of using Spring Framework
- Key features of Spring Framework
  - Dependency Injection and Inversion of Control
  - Aspect Oriented Programming