Why Spring

Spring Framework is an open source Java application development framework that supports developing all types of Java applications such as enterprise applications, web applications, cloud based applications, and many more.

Java applications developed using Spring are simple, easily testable, reusable, and maintainable.

Spring modules do not have tight coupling on each other, the developer can pick and choose the modules as per the need for building an enterprise application.

Spring is considered to be a secure, low-cost and flexible framework that improves coding efficiency and reduces overall application development time through efficient use of system resources.

Spring removes tedious configuration work so that developers can focus on writing business logic. Spring handles the infrastructure so developers can focus on the application

Reasons why we use spring(FEATURES)

Light Weight: Spring JARs are relatively small.

A basic Spring framework would be lesser than 10MB.

It can be deployed in Tomcat and they do not require heavy-weight application servers.

Non-Invasive: The application is developed using POJOs.

No need to extend/implement any pre-defined classes.

Loosely Coupled: Spring features like Dependency Injection and Aspect Oriented Programming help in loosely coupled code.

Inversion of Control(IoC): IoC takes care of the application object's life cycle along with their dependencies.

Spring Container: Spring Container takes care of object creation, initialization, and managing object dependencies.

Aspect Oriented Programming(AOP): Promotes separation of supporting functions(concerns) such as logging, transaction, and security from the core business logic of the application.

MVC Pattern: MVC is a pattern and methodology in software design that stands for Model View Controller which helps in separating implementation and business logic so that developers can focus on their code for better performance of the application

Dependency Injection : Dependency is neither good for human beings nor the classes in projects. Dependency Injection helps in decreasing coupling and dependency between classes in Spring projects so that the program becomes maintainable and reusable.

What is Spring Framework

Spring Framework is an open source Java application development framework that supports developing all types of Java applications such as enterprise applications, web applications, cloud based applications, and many more.

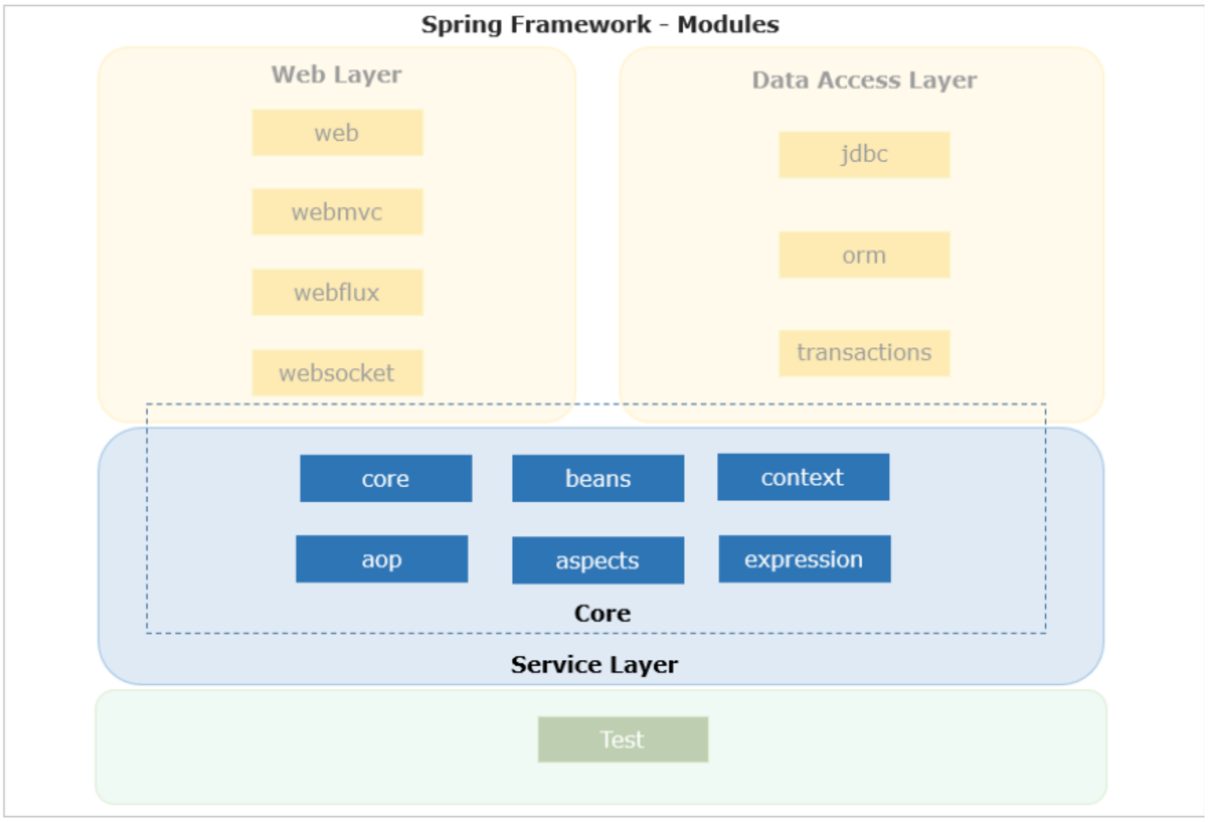
Java applications developed using Spring are simple, easily testable, reusable, and maintainable.

Spring modules do not have tight coupling on each other, the developer can pick and choose the modules as per the need for building an enterprise application.

Spring is considered to be a secure, low-cost and flexible framework that improves coding efficiency and reduces overall application development time through efficient use of system resources.

Spring removes tedious configuration work so that developers can focus on writing business logic. Spring handles the infrastructure so developers can focus on the application

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the following diagram.



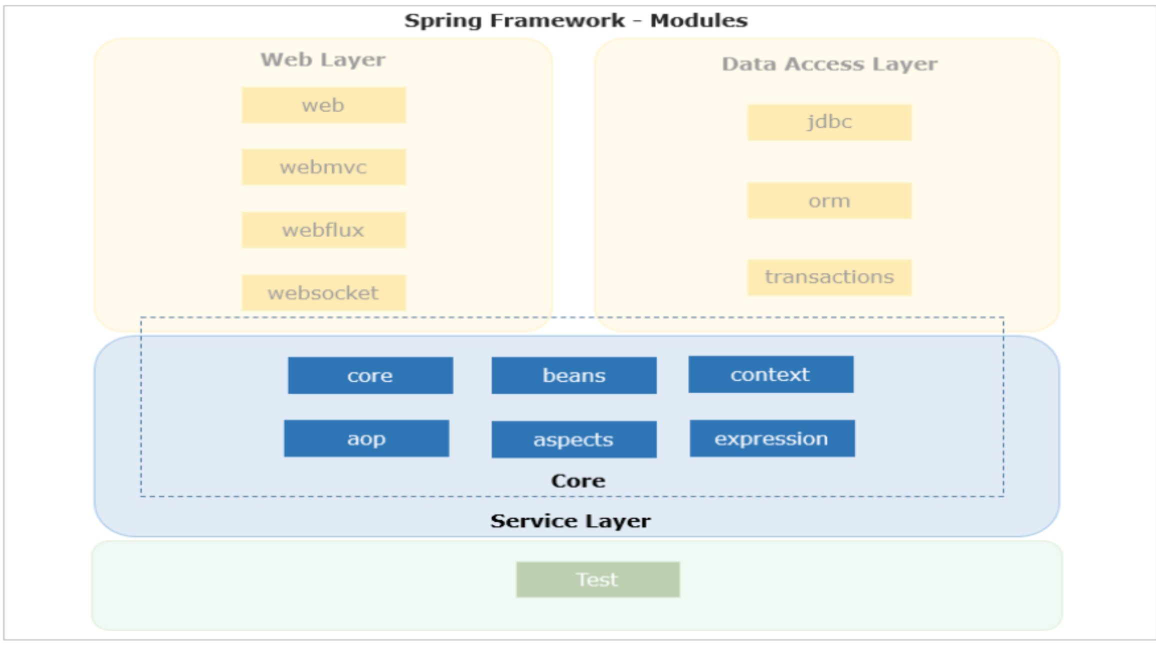
- Core Container: These are core modules that provide key features of the Spring framework.
- Data Access/Integration: These modules support JDBC and ORM data access approaches in Spring applications.
- Web: These modules provide support to implement web applications.
- Others: Spring also provides few other modules such as the Test for testing Spring applications.

Modules

Spring Framework 5.x has the following key module groups:

- Core Container: These are core modules that provide key features of the Spring framework.
- Data Access/Integration: These modules support JDBC and ORM data access approaches in Spring applications.
- Web: These modules provide support to implement web applications.
- Others: Spring also provides few other modules such as the Test for testing Spring applications.

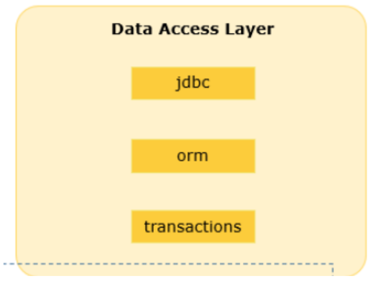
Core Container of Spring framework provides the Spring IoC container and Dependency Injection features.



Core container has the following modules:

- Core: This is the key module of Spring Framework which provides fundamental support on which all other modules of the framework are dependent.
- Bean: This module provides a basic Spring container called BeanFactory.
- Context: This module provides one more Spring container called ApplicationContext which inherits the basic features of the BeanFactory container and also provides additional features to support enterprise application development.
- Spring Expression Language (SpEL): This module is used for querying/manipulating object values.
- AOP (Aspect Oriented Programming) and aspects: These modules help in isolating cross-cutting functionality from business logic.

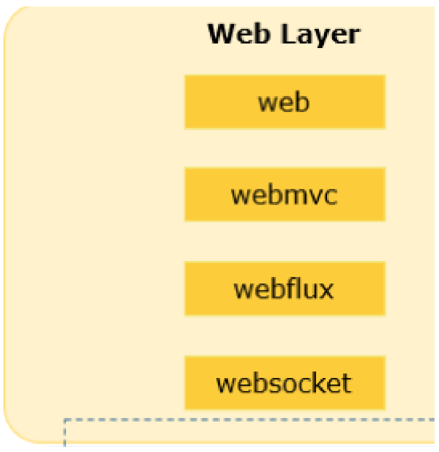
The Data Access/Integration module of Spring provides different data access approaches.



The following modules support Data Access/Integration:

- Java Database Connectivity (JDBC): It provides an abstract layer to support JDBC calls to relational databases.
- Object Relational Mapping (ORM): It provides integration support for popular ORM(Object-Relational Mapping) solutions such as Hibernate, JPA, etc.
- Transactions: It provides a simple transaction API which abstracts the complexity of underlying repository specific transaction API's from the application.

Spring Framework Web module provides basic support for web application development. The Web module has a web application context that is built on the application context of the core container. Web module provides complete Model-View-Controller(MVC) implementation to develop a presentation tier of the application and also supports a simpler way to implement RESTful web services.

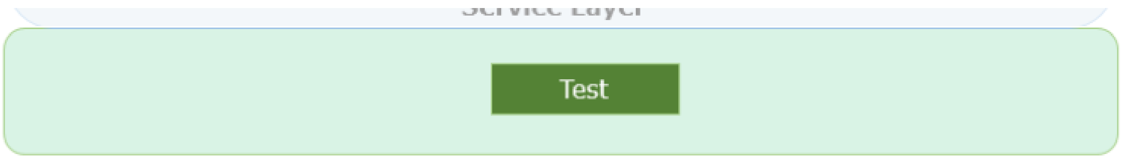


Spring Framework provides the following modules to support web application development:

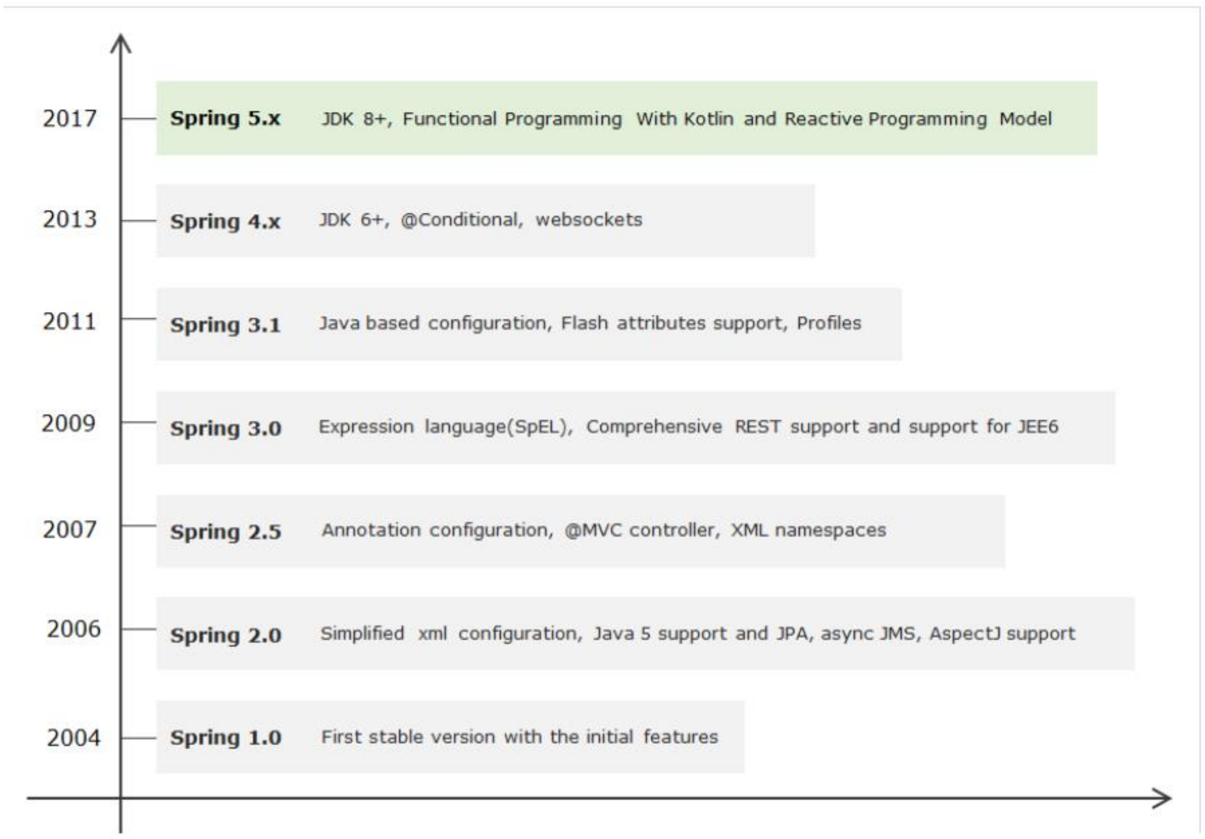
- Web: This module has a container called web application context which inherits basic features from ApplicationContext container and adds features to develop web based applications.
- Webmvc: It provides the implementation of the MVC(model-view-controller) pattern to implement the serverside presentation layer and also supports features to implement RESTful Web Services.
- WebFlux: Spring 5.0 introduced a reactive stack with a web framework called Spring WebFlux to support Reactive programming in Spring's web layer and runs on containers such as Netty, Undertow, and Servlet 3.1+.
- WebSocket: It is used for 2 way communication between client and server in WebSocket based web applications.

Spring Framework has few additional modules, test module is one of the most commonly used ones for testing Spring applications.

- Test: This module provides the required support to test Spring applications using TestNG or JUnit.



History of spring


2017
Spring 5.x
JDK 8+, Functional Programming With Kotlin and Reactive Programming Model
2013
Spring 4.x
JDK 6+, @Conditional, websockets
2011
Spring 3.1
Java based configuration, Flash attributes support, Profiles
2009
Spring 3.0
Expression language(SpEL), Comprehensive REST support and support for JEE6
2007
Spring 2.5
Annotation configuration, @MVC controller, XML namespaces
2006
Spring 2.0
Simplified xml configuration, Java 5 support and JPA, async JMS, AspectJ support
2004
Spring 1.0
First stable version with the initial features

The current version of Spring Framework is 5.x, the framework has been enhanced with new features keeping core concepts the same as Spring 4.x.

At a high level, the new features of Spring Framework 5.x are:

- JDK baseline update
- Core framework revision

- Reactive Programming Model: Introduces a new non-blocking web framework called Spring WebFlux
- Functional programming using Kotlin language support
- Testing improvements by supporting integration with JUnit5

Let us look at Spring core relevant changes in detail:

JDK baseline update

The entire Spring framework 5.x codebase runs on Java 8 and designed to work with Java 9. Therefore, Java 8 is the minimum requirement to work on Spring Framework 5.x

Core framework revision

The core Spring Framework 5.x has been revised, one of the main changes is Spring comes with its own commons-logging through spring-jcl jar instead of standard Commons Logging.

There are few more changes in Spring 5.x with respect to library support and discontinued support, you can refer to **the Spring documentation** for additional details.

Spring IoC

Inversion of Control (IoC) helps in creating a more loosely coupled application. IoC represents the inversion of the responsibility of the application object's creation, initialization, and destruction from the application to the third party such as the framework. Now the third party takes care of application object management and dependencies thereby making an application easy to maintain, test, and reuse.

There are many approaches to implement IoC, Spring Framework provides IoC implementation using Dependency Injection(DI).

Spring Container managed application objects are called beans in Spring.

We need not create objects in dependency injection instead describe how objects should be created through configuration.

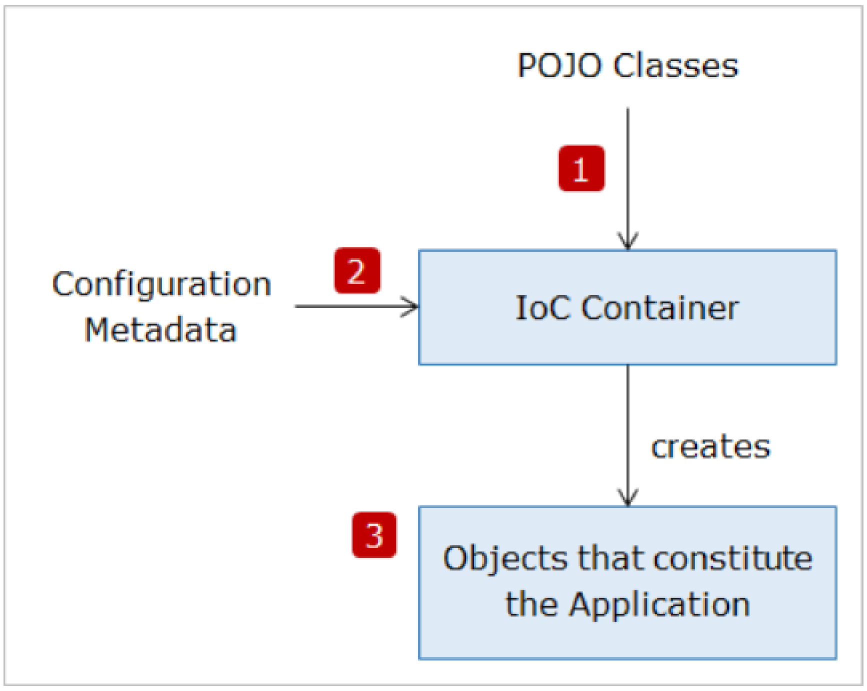
DI is a software design pattern that provides better software design to facilitate loose coupling, reuse, and ease of testing.

Benefits of Dependency Injection(DI):

- Helps to create loosely coupled application architecture facilitating re-usability and easy testing.
- Separation of responsibility by keeping code and configuration separately. Hence dependencies can be easily modified using configuration without changing the code.
- Allows to replace actual objects with mock objects for testing, this improves testability by writing simple JUnit tests that use mock objects.

The core container module of the Spring Framework provides IoC using Dependency Injection.

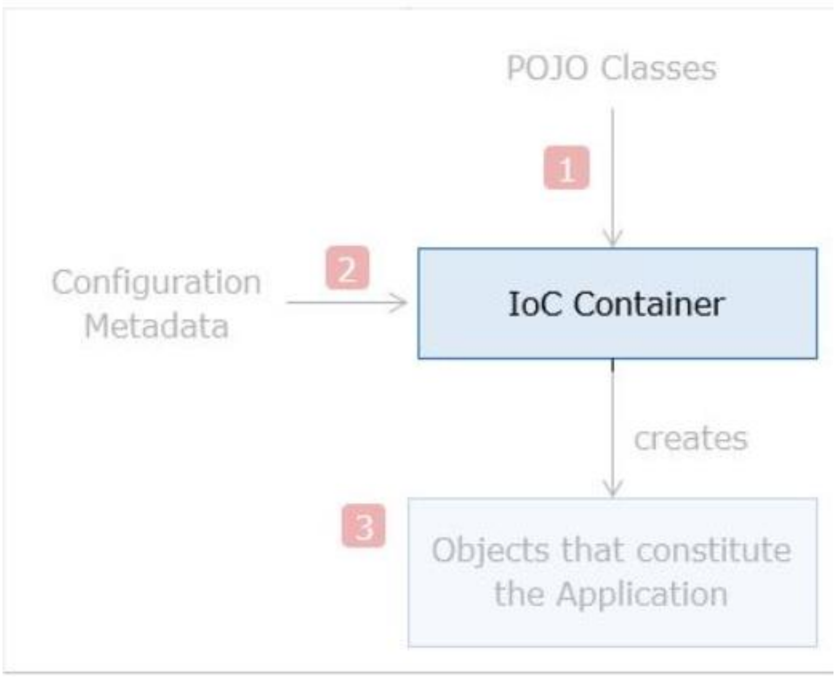
The Spring container knows which objects to create and when to create through the additional details that we provide in our application called **Configuration Metadata.**



1. Application logic is provided through POJO classes.
2. Configuration metadata consists of bean definitions that the container must manage.
3. IoC container produces objects required by the application using POJO classes and configuration metadata. IoC container is of two types – **BeanFactory and ApplicationContext**.

Let us understand IoC containers and configuration metadata in detail on the go.

Spring provides two types of containers



BeanFactory:

- It is the basic Spring container with features to instantiate, configure and manage the beans.
- org.springframework.beans.factory.BeanFactory is the main interface representing a BeanFactory container.

ApplicationContext:

- ApplicationContext is another Spring container that is more commonly used in Spring applications.
- org.springframework.context.ApplicationContext is the main Interface representing an ApplicationContext container.
- It inherits the BeanFactory features and provides added features to support enterprise services such as internationalization, validation, etc.

ApplicationContext is the preferred container for Spring application development. Let us look at more details on the ApplicationContext container.

Let us now understand the differences between BeanFactory and ApplicationContext containers.

BeanFactory	ApplicationContext
It does not support annotation based Dependency Injection.	Support annotation based Dependency Injection.
It does not support enterprise services.	Support enterprise services such as validations, internationalization, etc.
By default, it supports Lazy Loading.	By default, it supports Eager Loading. Beans are instantiated during load time.

//Loading BeanFactory
BeanFactory factory = new AnnotationConfigApplicationContext(SpringConfiguration.class);
 	
// Instantiating bean during first access using getBean()
CustomerServiceImpl service = (CustomerServiceImpl) factory.getBean("customerService");
	// Loading ApplicationContext and instantiating bean
	ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);
	
	// Instantiating bean during first access using getBean()
	CustomerServiceImpl service = (CustomerServiceImpl) context.getBean("customerService");

org.springframework.context.annotation.AnnotationConfigApplicationContext is one of the most commonly used implementation of ApplicationContext.

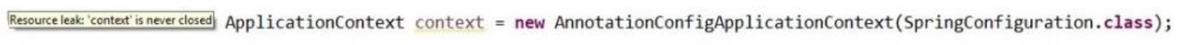
Example: ApplicationContext container instantiation.

```
ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);

Object obj = context.getBean("customerService");
```

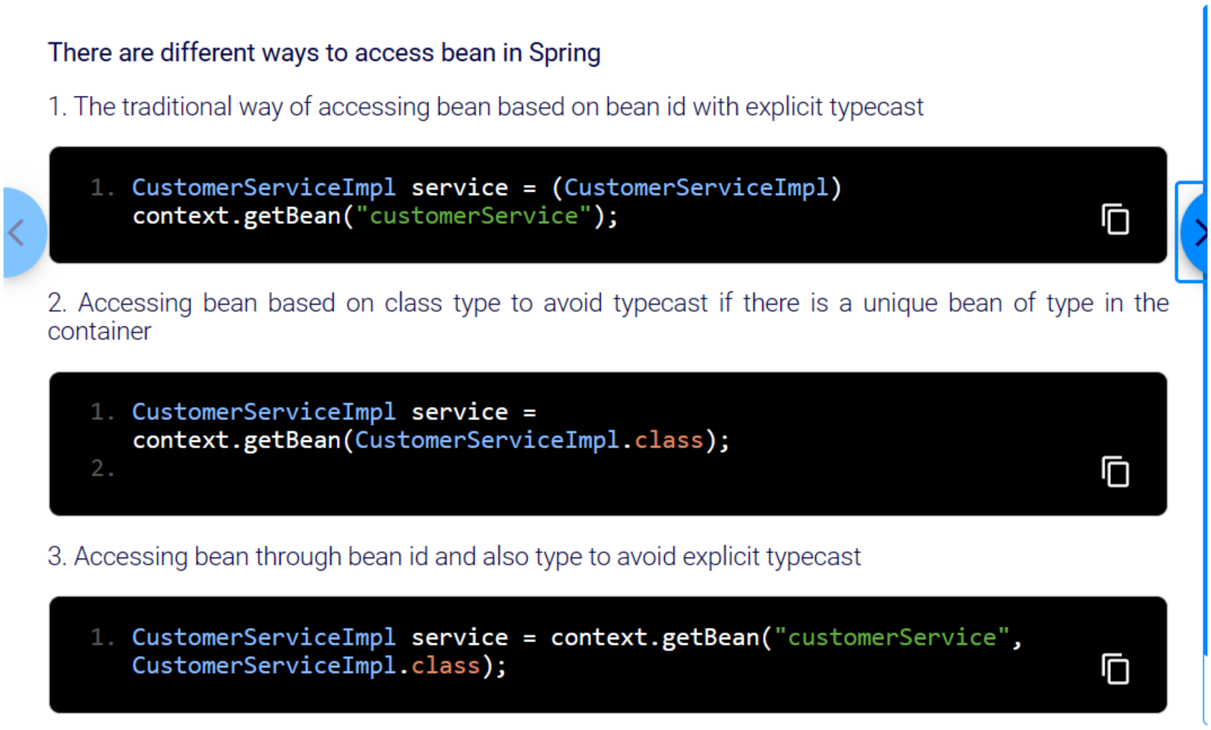
1. ApplicationContext container is instantiated by loading the configuration from the SpringConfiguration.class which is available in the utility package of the application.
2. Accessing the bean with id "customerService" from the container.

AbstractApplicationContext



```
ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);
```

You can see a warning message as Resource leak: 'context' is never closed while using the ApplicationContext type. This is for the reason that you don't have a close method with BeanFactory or even ApplicationContext. AbstractApplicationContext is an abstract implementation of the ApplicationContext interface and it implements Closeable and AutoCloseable interfaces. To close the application context and destroy all beans in its abstractApplicationContext has a close method that closes this application context.



There are different ways to access bean in Spring

1. The traditional way of accessing bean based on bean id with explicit typecast

```
CustomerServiceImpl service = (CustomerServiceImpl)
context.getBean("customerService");
```

2. Accessing bean based on class type to avoid typecast if there is a unique bean of type in the container

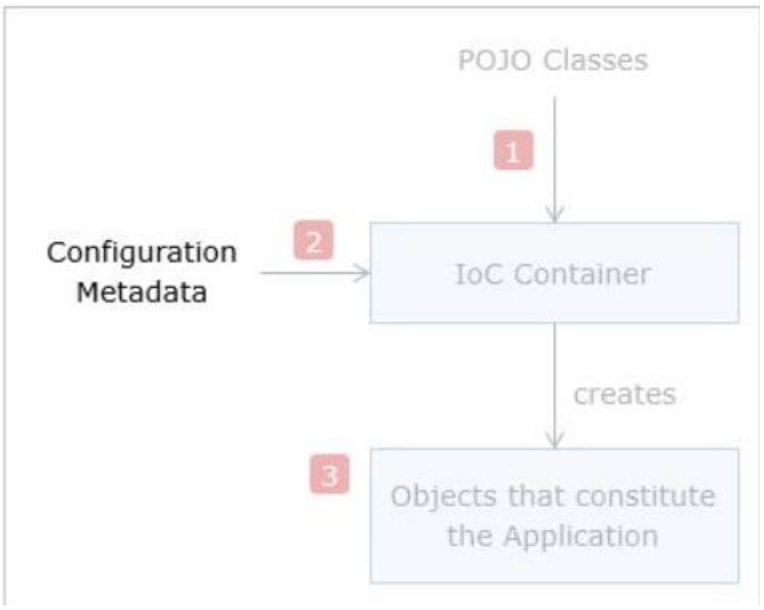
```
CustomerServiceImpl service =
context.getBean(CustomerServiceImpl.class);

```

3. Accessing bean through bean id and also type to avoid explicit typecast

```
CustomerServiceImpl service = context.getBean("customerService",
CustomerServiceImpl.class);
```

The Spring configuration metadata consists of definitions of the beans that the container must manage



Spring allows providing the configuration metadata using :

- XML Configuration
- Annotation Based configuration
- Java Based configuration

We will cover the **Java based configuration** in this course.

Configuring IoC container using Java-based configuration

The Java-based configuration metadata is provided in the Java class using the following annotations:

@Configuration: The Java configuration class is marked with this annotation. This annotation identifies this as a configuration class, and it's expected to contain details on beans that are to be created in the Spring application context.

@Bean: This annotation is used to declare a bean. The methods of configuration class that creates an instance of the desired bean are annotated with this annotation. These methods are called by the Spring containers during bootstrap and the values returned by these methods are treated as Spring beans. By default, only one bean instance is created for a bean definition by the Spring Container, and that instance is used by the container for the whole application lifetime.

For example, the SpringConfiguration class can be configured in a Java class using the above annotations as follows :

```
@Configuration
public class SpringConfiguration {

    @Bean
    public CustomerServiceImpl customerService() {

        return new CustomerServiceImpl();
    }
}
```

By default, the bean name is the same as the name of the method in which the bean is configured. So in the above code bean name is customerService. If you want to change the bean name then you can either rename the method or provide a different name with the name attribute as follows:

```
@Configuration
public class SpringConfiguration {

    @Bean(name="service")
    public CustomerServiceImpl customerService() {

        return new CustomerServiceImpl();
    }
}
```

Introduction To Dependency Injection

Let us now understand in detail how to implement Dependency Injection in Spring.

For the previously discussed InfyTel Customer application, we defined CustomerService bean as shown below

```
@Bean

public CustomerService customerService() {


          return new CustomerService();

}
```

This is the same as the below Java code wherein an instance is created and initialized with default values using the default constructor.

```
CustomerService customerService = new CustomerService();
```

How do we initialize beans with some specific values in Spring?

This can be achieved through **Dependency Injection** in Spring.

Inversion of Control pattern is achieved through Dependency Injection (DI) in Spring. In Dependency Injection, the developer need not create the objects but specify how they should be created through configuration.

Spring container uses one of these two ways to initialize the properties:

- **Constructor Injection**: This is achieved when the container invokes a parameterized constructor to initialize the properties of a class
- **Setter Injection**: This is achieved when the container invokes setter methods of a class to initialize the properties after invoking a default constructor.

Let us consider the CustomerService class of InfyTel Customer application to understand constructor injection.

CustomerService class has a count property, let us now modify this class to initialize count property during bean instantiation using the constructor injection approach.

```
package com.infy.service;

public class CustomerServiceImpl implements CustomerService {

          private int count;

          public CustomerServiceImpl(int count) {
                this.count = count;

        }

}
```

How do we define bean in the configuration to initialize values?

```
@Configuration

public class SpringConfiguration {

        @Bean // CustomerService bean definition with bean dependencies through constructor injection

        public CustomerServiceImpl customerService() {

                return new CustomerServiceImpl(20);

        }
```

What is mandatory for constructor injection?

A parameterized constructor is required in the CustomerService class as we are injecting the values through the constructor argument.

Can we use constructor injection to initialize multiple properties?

Yes, we can initialize more than one property.

So far, we learned how to inject primitive values using constructor injection in Spring. Now we will look into inject object dependencies using Constructor injection.

Consider our InfyTel Customer application. **CustomerServiceImpl.java** class which is dependent on CustomerRepository(class used to in persistence layer to perform CRUD operations) object type to call fetchCustomer() method.

```
package com.infy.service;

public class CustomerServiceImpl implements CustomerService {

// CustomerServiceImpl needs to contact CustomerRepository, hence injecting the customerRepository dependency
```

```
        private CustomerRepository customerRepository;

    private int count;

        public CustomerServiceImpl() {

        }

        public                    CustomerServiceImpl(CustomerRepository
customerRepository, int count) {

                this.customerRepository = customerRepository;

      this.count=count;

        }

   public String fetchCustomer() {

                return customerRepository.fetchCustomer(count);

        }

        public String createCustomer() {

                return customerRepository.createCustomer();

        }



}
```

Observe in the above code, CustomerRepository property of CustomerSevice class has not been initialized with any value in the code. This is because Spring dependency is going to be taken care of in the configuration.

How do we inject object dependencies through configuration using constructor injection?

```
package com.infy.util;

@Configuration

public class SpringConfiguration {
        @Bean// customerRepository bean definition

        public CustomerRepository customerRepository() {

                return new CustomerRepository();

        }

        @Bean // CustomerServic bean definition with bean dependencies
through constructor injection

        public CustomerServiceImpl customerService() {

                return new CustomerServiceImpl(customerRepository(),20);

        }

}
```

Setter Injection

Let us now understand Setter Injection in Spring.

In Setter Injection, Spring invokes setter methods of a class to initialize the properties after invoking a default constructor.

How can we use setter injection to inject values for the primitive type of properties?

Consider the below example to understand setter injection for primitive types.

Following the CustomerServiceImpl class has a count property, let us see how to initialize this property during bean instantiation using the setter injection approach.

```
        package com.infy.service;

        public class CustomerServiceImpl implements CustomerService {

                private int count;

                public int getCount() {
```

```
                return count;
            }
            public void setCount(int count) {
                this.count = count;
            }
          public CustomerServiceImpl(){
      }
    }
```

How do we define bean in the configuration to initialize values?

```
package com.infy.util;

@Configuration

public class SpringConfiguration {


        @Bean // CustomerService bean definition using Setter Injection

        public CustomerServiceImpl customerService() {

                CustomerServiceImpl customerService = new CustomerServiceImpl();

                customerService.setCount(10);

                return customerService;

        }

}
```

What is mandatory to implement setter injection?

- Default constructor and setter methods of respective dependent properties are required in the CustomerServiceImpl class. For setter injection, Spring internally uses the default constructor to create a bean and then invokes a setter method of the respective property based on the name attribute in order to initialize the values.

So far, we learned how to inject primitive values using setter injection in Spring.

How do we inject object dependencies using setter injection?

- Consider the CustomerServiceImpl class of InfyTel Customer application.

```
package com.infy.service;

public class CustomerServiceImpl implements CustomerService {

    private CustomerRepository customerRepository;

  private int count;

    public CustomerRepository getCustomerRepository() {

        return customerRepository;

    }

    public void setCustomerRepository(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public int getCount() {

        return count;

    }

    public void setCount(int count) {

        this.count = count;

    }

}
```

How do we inject object dependencies through configuration using setter injection?

```
package com.infy.util;

@Configuration

public class SpringConfiguration {

        @Bean

        public CustomerRepository customerRepository() {

                return new CustomerRepository();

        }

        @Bean // Setter Injection

        public CustomerServiceImpl customerService() {

                CustomerServiceImpl customerService = new CustomerServiceImpl();

            customerService.setCount(10);


        customerService.setCustomerRepository(customerRepository());

                return customerService;

        }

}
```

What is AutoScanning

As discussed in the previous example as a developer you have to declare all the bean definition in SpringConfiguration class so that Spring container can detect and register your beans as below

```
@Configuration
```

```
public class SpringConfiguration {

        @Bean

        public CustomerRepository customerRepository() {

                return new CustomerRepository();

        }

        @Bean

        public CustomerServiceImpl customerService() {

                return new CustomerServiceImpl();

        }

}
```

Is any other way to eliminate this tedious beans declaration?

Yes, Spring provides a way to automatically detect the beans to be injected and avoid even the bean definitions within the Spring configuration file through Auto Scanning. In Auto Scanning, Spring Framework automatically scans, detects, and instantiates the beans from the specified base package, if there is no declaration for the beans in the SpringConfiguration class.

So your SpringConfiguration class will be looking as below:

```
@Configuration

@ComponentScan(basePackages="com.infy")

public class SpringConfiguration {

}
```

Component scanning isn't turned on by default, however. You have to annotate the configuration class with **@ComponentScan** annotation to enable component scanning as follows:

```
@Configuration

@ComponentScan
```

```
public class SpringConfiguration  {

}
```

In the above code, Spring will scan the package that contains SpringConfig class and it subpackages for beans. But if you want to scan a different package or multiple packages then you can specify this with the basePackages attribute as follows:

```
@Configuration

@ComponentScan(basePackages =
"com.infy.service,com.infy.repository")

public class SpringConfiguration {

}
```

Spring uses @ComponentScan annotation for the auto scan feature. It looks for classes with the stereotype annotations and creates beans for such classes automatically.

So what are **Stereotype annotations**?

- Stereotype annotations denote the roles of types or methods at the conceptual level.
- Stereotype annotations are @Component, @Service, @Repository, and @Controller annotations.
- These annotations are used for auto-detection of beans using @ComponentScan.
- The Spring stereotype @Component is the parent stereotype.
- The other stereotypes are the specialization of @Component annotation.

Annotation	Usage
@Component	It indicates the class(POJO class) as a Spring component.
@Service	It indicates the Service class(POJO class) in the business layer.
@Repository	It indicates the Repository class(POJO class in Spring DATA) in the persistence layer.
@Controller	It indicates the Controller class(POJO class in Spring MVC) in the presentation layer.

- @Component is a generic stereotype for any Spring-managed component.
- @Repository, @Service, and @Controller are specializations of @Component for more specific use cases.
- @Component should be used when your class does not fall into either of three categories i.e. Controllers, Services, and DAOs.

@Component: It is a general purpose annotation to mark a class as a Spring-managed bean.

```
@Component

public class CustomerLogging{

    //rest of the code

}
```

@Service - It is used to define a service layer Spring bean. It is a specialization of the @Component annotation for the service layer.

```
@Service

public class CustomerSeviceImpl implements CustomerService {


    //rest of the code

}
```

@Repository - It is used to define a persistence layer Spring bean. It is a specialization of the @Component annotation for the persistence layer.

```
@Repository

public class CustomerRepositoryImpl implements
CustomerRepository {

    //rest of the code

}
```

@Controller - It is used to define a web component. It is a specialization of the @Component annotation for the presentation layer.

```
@Controller

public class CustomerController {

        //rest of the code

}
```

By default, the bean names are derived from class names with a lowercase initial character. Therefore, your above defined beans have the names customerController, customerServiceImpl, and customerRepositoryImpl. It is also possible to give a specific name with a value attribute in those annotations as follows:

```
@Repository(value="customerRepository")

public class CustomerRepositoryImpl implements CustomerRepository {

        //rest of the code

}
```

Note: As a best practice, use @Service for the service layer, @Controller for the Presentation layer, and @Repository for the Persistence layer.