Spring Framework

Provides core support for dependency injection, transaction management, web apps, data access, messaging and more.

* The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications – on any kind of deployment platform.
* A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

Features:

1. **Core Technologies** **–** dependency injection, events, resources, i18n, validation, data binding,

type Conversion, SpEL, AOP.

1. **Testing –** mock objects, TestContext Framework, Spring MVC Test, WebTestClient.
2. **Data Access** – transactions, DAO Support, JDBC, ORM, Marshalling XML
3. **Spring MVC** and **Spring WebFlux** web frameworks.
4. **Integration –** remoting, JMS, JCA, JMX, email, tasks, scheduling, cache.
5. **Languages –** Kotlin, Groovy, dynamic languages.

**Core Technologies**

**================**

* All the technologies are absolutely integral to the Spring Framework.
* Inversion of Control container, Aspect Oriented Programming (AOP).

The IOC Container

--------------------------

* Inversion Of Control (IOC) principle also known as Dependency Injection (DI).
* It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.
* The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes or a mechanism such as the Service Locator pattern.
* The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework’s IoC container.
* The [BeanFactory](https://docs.spring.io/spring-framework/docs/5.2.1.RELEASE/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface provides an advanced configuration mechanism capable of managing any type of object.
* [ApplicationContext](https://docs.spring.io/spring-framework/docs/5.2.1.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html) is a sub-interface of BeanFactory. It adds:
* Easier integration with Spring’s AOP features
* Message resource handling (for use in internationalization)
* Event publication
* Application-layer specific contexts such as the WebApplicationContext for use in web applications.
* In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality.
* In Spring, 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.
* The org.springframework.context.ApplicationContext interface represents the **Spring IoC container** and is responsible for instantiating, configuring, and assembling the beans.
* The container gets its instructions on what objects to instantiate, configure, and assemble by reading **configuration metadata**.
* The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects.
* Several implementations of the ApplicationContext interface are supplied with Spring.
* In stand-alone applications, it is common to create an instance of [ClassPathXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.2.1.RELEASE/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) or [FileSystemXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.2.1.RELEASE/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html).
* The Spring IoC container consumes a form of configuration metadata.
* This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.
* Configuration metadata can be in the form of XML, Annotation-Based or Java-Based.
* Typically, you define service layer objects, data access objects (DAOs), presentation objects such as Struts Action instances, infrastructure objects such as Hibernate SessionFactories, JMS Queues, and so forth. Typically, one does not configure fine-grained domain objects in the container, because it is usually the responsibility of DAOs and business logic to create and load domain objects.
* The location path or paths supplied to an ApplicationContext constructor are resource strings that let the container load configuration metadata from a variety of external resources, such as the local file system, the Java CLASSPATH, and so on.
* The ApplicationContext is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies.
* By using the method T getBean(String name, Class<T> requiredType), you can retrieve instances of your beans.
* your application code should have no calls to the getBean() method at all and thus have no dependency on Spring APIs at all.
* Spring’s integration with web frameworks provides dependency injection for various web framework components such as controllers and JSF-managed beans, letting you declare a dependency on a specific bean through metadata (such as an autowiring annotation).
* A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container (for example, in the form of XML <bean/> definitions).
* Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata:
* A package-qualified classname
* Bean behavioral configuration elements (scope, lifecycle, callback)
* References to other beans (collaborators or dependencies)
* Other configuration settings

This metadata translates to a set of properties that make up each bean definition. These are :

Class, Name, Scope, Constructor arguments, Properties, Autowiring mode, Lazy Initialization mode, Initialization method, Destruction method.

* In addition to bean definitions that contain information on how to create a specific bean, the ApplicationContext implementations also permit the registration of existing objects that are created outside the container (by users). This is done by accessing the ApplicationContext’s BeanFactory through the getBeanFactory() method, which returns the BeanFactory DefaultListableBeanFactory implementation. DefaultListableBeanFactory supports this registration through the registerSingleton(..) and registerBeanDefinition(..) methods. However, typical applications work solely with beans defined through regular bean definition metadata.
* Bean metadata and manually supplied singleton instances need to be registered as early as possible, in order for the container to properly reason about them during autowiring and other introspection steps.
* The registration of new beans at runtime (concurrently with live access to the factory) is not officially supported and may lead to concurrent access exceptions, inconsistent state in the bean container, or both.
* Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean.
* In XML-based configuration metadata, you use the id attribute, the name attribute, or both to specify the bean identifiers.
* You are not required to supply a name or an id for a bean. If you do not supply a name or id explicitly, the container generates a unique name for that bean.
* In a bean definition itself, 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.
* A bean definition is essentially a recipe for creating one or more objects. The container looks at the recipe for a named bean when asked and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.
* If you want to configure a bean definition for a static nested class, you have to use the binary name of the nested class.
* For example, if you have a class called SomeThing in the com.example package, and this SomeThing class has a static nested class called OtherThing, the value of the class attribute on a bean definition would be com.example.SomeThing$OtherThing.
* **Dependency injection (DI)** is a process whereby objects define their dependencies (that is, the other objects with which they work) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.
* The container then injects those dependencies when it creates the bean.
* This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes or the Service Locator pattern.
* Code is cleaner with the DI principle, and decoupling is more effective when objects are provided with their dependencies. Classes become easier to test.
* DI exists in two major variants: [Constructor-based dependency injection](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-constructor-injection) and [Setter-based dependency injection](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-setter-injection).
* **Constructor-based DI** is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.
* **Setter-based DI** is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or a no-argument static factory method to instantiate your bean.
* Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies.
* The Spring team generally advocates constructor injection, as it lets you implement application components as immutable objects and ensures that required dependencies are not null.  constructor-injected components are always returned to the client (calling) code in a fully initialized state.
* Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency.
* One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later. Management through [JMX MBeans](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/integration.html#jmx) is therefore a compelling use case for setter injection.
* The container performs bean dependency resolution as follows:
* The ApplicationContext is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified by XML, Java code, or annotations.
* For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method (if you use that instead of a normal constructor). These dependencies are provided to the bean, when the bean is actually created.
* Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
* Each property or constructor argument that is a value is converted from its specified format to the actual type of that property or constructor argument. By default, Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, and so forth.
* The Spring container validates the configuration of each bean as the container is created. However, the bean properties themselves are not set until the bean is actually created.
* he Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.
* One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.
* You can generally trust Spring to do the right thing. It detects configuration problems, such as references to non-existent beans and circular dependencies, at container load-time. Spring sets properties and resolves dependencies as late as possible, when the bean is actually created. This means that a Spring container that has loaded correctly can later generate an exception when you request an object if there is a problem creating that object or one of its dependencies.
* By default, ApplicationContext implementations eagerly create and configure all [singleton](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-factory-scopes-singleton) beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later.
* When this behavior is not desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as being lazy-initialized. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.
* The Spring container can autowire relationships between collaborating beans. You can let Spring resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext. Autowiring has the following advantages:
* Autowiring can significantly reduce the need to specify properties or constructor arguments.
* Autowiring can update a configuration as your objects evolve. For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration. Thus autowiring can be especially useful during development, without negating the option of switching to explicit wiring when the code base becomes more stable.
* The autowiring functionality has four modes. You specify autowiring per bean and can thus choose which ones to autowire. The following table describes the four autowiring modes:

No, byname, byType, constructor

* Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions.
* When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.
* You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition but also control the scope of the objects created from a particular bean definition.
* Beans can be defined to be deployed in one of a number of scopes. The Spring Framework supports six scopes, four of which are available only if you use a **web-aware ApplicationContext**. You can also create [a custom scope.](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-factory-scopes-custom)
* The following are the supported **bean scopes**:

1. **Singleton –** (Default) Scopes a single bean definition to a single object instance for each Spring IOC container.
2. **Prototype –** Scopes a single bean definition to any number of object instances.
3. **Request –** Scopes a single bean definition to the lifecycle of a single HTTP request.
4. **Session –** Scopes a single bean definition to the lifecycle of an HTTP session.
5. **Application –** Scopes a single bean definition to the lifecycle of a ServletContext.
6. **WebSocket –** Scopes a single bean definition to the lifecycle of a WebSocket.

* The Spring Framework provides a number of interfaces you can use to customize the nature of a bean. This section groups them as follows:

1. Lifecycle Callbacks
2. ApplicationContextAware and BeanNameAware
3. Other Aware Interfaces.

* To interact with the container’s management of the bean lifecycle, you can implement the Spring InitializingBean and DisposableBean interfaces. The container calls afterPropertiesSet() for the former and destroy() for the latter to let the bean perform certain actions upon initialization and destruction of your beans.
* The JSR-250 **@PostConstruct** and **@PreDestroy** annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring-specific interfaces.
* As of Spring 2.5, you have three options for controlling bean lifecycle behavior:

1. The InitializingBean and DisposableBean callback interfaces.
2. Custom init() and destroy() methods
3. The @PostConstruct and @PreDestroy annotations

You can combine these mechanisms to control of a given bean.

* Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

@PostContruct then afterPropertiesSet() of InitializingBean then init()

* Destroy methods are called in the same order:

@PreDestroy then destroy() of DisposableBean then custom configured destroy()

* The Lifecycle interface defines the essential methods for any object that has its own lifecycle requirements (such as starting and stopping some background process):

start() , stop(), isRunning()

* If you use Spring’s IoC container in a non-web application environment (for example, in a rich client desktop environment), register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. You must still configure and implement these destroy callbacks correctly.
* To register a shutdown hook, call the registerShutdownHook() method that is declared on the ConfigurableApplicationContext interface, as the following example shows:
* Spring’s web-based ApplicationContext implementations already have code in place to gracefully shut down the Spring IoC container when the relevant web application is shut down.
* When an ApplicationContext creates an object instance that implements the org.springframework.context.ApplicationContextAware interface, the instance is provided with a reference to that ApplicationContext.
* Thus, beans can programmatically manipulate the ApplicationContext that created them, through the ApplicationContext interface or by casting the reference to a known subclass of this interface (such as ConfigurableApplicationContext, which exposes additional functionality).
* One use would be the programmatic retrieval of other beans.
* Autowiring is another alternative to obtain a reference to the ApplicationContext.
* When an ApplicationContext creates a class that implements the org.springframework.beans.factory.BeanNameAware interface, the class is provided with a reference to the name defined in its associated object definition.
* Besides ApplicationContextAware and BeanNameAware (discussed [earlier](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-factory-aware)), Spring offers a wide range of Aware callback interfaces that let beans indicate to the container that they require a certain infrastructure dependency. As a general rule, the name indicates the dependency type. The following table summarizes the most important Aware interfaces:

1. ApplicationContextAware
2. ApplicationEventPublisherAware
3. BeanClassLoaderAware
4. BeanFactoryAware
5. BeanNameAware
6. BootStrapContextAware
7. LoadTimeWeaverAware
8. MessageSourceAware
9. NotificationPublisherAware
10. ResourceLoaderAware
11. ServletConfigAware
12. ServletContextAware

* Note again that using these interfaces ties your code to the Spring API and does not follow the Inversion of Control style. As a result, we recommend them for infrastructure beans that require programmatic access to the container.
* A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information, such as the initialization method, a static factory method name, and so on.
* If you work with an ApplicationContext interface programmatically, child bean definitions are represented by the ChildBeanDefinition class.
* An application developer does not need to subclass ApplicationContext implementation classes. Instead, the Spring IoC container can be extended by plugging in implementations of special integration interfaces. The next few sections describe these integration interfaces.
* An ApplicationContext automatically detects any beans that are defined in the configuration metadata that implements the BeanPostProcessor interface. The ApplicationContext registers these beans as post-processors so that they can be called later, upon bean creation. Bean post-processors can be deployed in the container in the same fashion as any other beans.
* The semantics of this interface are similar to those of the BeanPostProcessor, with one major difference: BeanFactoryPostProcessor operates on the bean configuration metadata. That is, the Spring IoC container lets a BeanFactoryPostProcessor read the configuration metadata and potentially change it before the container instantiates any beans other than BeanFactoryPostProcessor instances.
* You can use the PropertySourcesPlaceholderConfigurer to externalize property values from a bean definition in a separate file by using the standard Java Properties format.
* The PropertyOverrideConfigurer, another bean factory post-processor, resembles the PropertySourcesPlaceholderConfigurer, but unlike the latter, the original definitions can have default values or no values at all for bean properties.
* You can implement the **org.springframework.beans.factory.FactoryBean** interface for objects that are themselves factories.
* The **FactoryBean** interface is a point of pluggability into the Spring IoC container’s instantiation logic. If you have complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your own FactoryBean, write the complex initialization inside that class, and then plug your custom FactoryBean into the container.
* The FactoryBean concept and interface is used in a number of places within the Spring Framework. More than 50 implementations of the FactoryBeaninterface ship with Spring itself.
* An alternative to XML setup is provided by annotation-based configuration, which relies on the bytecode metadata for wiring up components instead of angle-bracket declarations.
* Are annotations better than XML for configuring Spring?
* Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them.
* Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.
* Annotation injection is performed before XML injection. Thus, the XML configuration overrides the annotations for properties wired through both approaches.
* Spring also supports injection by using the JSR-250 @Resource annotation (javax.annotation.Resource) on fields or bean property setter methods. This is a common pattern in Java EE: for example, in JSF-managed beans and JAX-WS endpoints. Spring supports this pattern for Spring-managed objects as well.
* Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration.
* Spring can automatically detect stereotyped classes and register corresponding BeanDefinition instances with the ApplicationContext.
* To autodetect these classes and register the corresponding beans, you need to add @ComponentScan to your @Configuration class, where the basePackages attribute is a common parent package for the two classes. (Alternatively, you can specify a comma- or semicolon- or space-separated list that includes the parent package of each class.)
* By default, classes annotated with @Component, @Repository, @Service, @Controller, @Configuration, or a custom annotation that itself is annotated with @Component are the only detected candidate components.
* However, you can modify and extend this behavior by applying custom filters. Add them as includeFilters or excludeFilters attributes of the @ComponentScan annotation (or as <context:include-filter /> or <context:exclude-filter /> child elements of the <context:component-scan> element in XML configuration). Each filter element requires the type and expression attributes.
* Starting with Spring 3.0, Spring offers support for JSR-330 standard annotations (Dependency Injection). Those annotations are scanned in the same way as the Spring annotations. To use them, you need to have the relevant jars in your classpath. (javax.inject / @Inject / @Named)