

UNIT I

Spring 5 Basics : Why Spring, What is Spring Framework, Spring Framework - Modules, Configuring IoC container using Java-based configuration, Introduction To Dependency Injection, Constructor Injection, Setter Injection, What is AutoScanning

Spring 5 Basics

1: Why Spring:

Spring makes programming Java quicker, easier, and safer for everybody. Spring's focus on speed, simplicity, and productivity has made it the [world's most popular](#) Java framework.

Spring is everywhere

Spring's flexible libraries are trusted by developers all over the world. Spring delivers delightful experiences to millions of end-users every day—whether that's [streaming TV](#), [online shopping](#), or countless other innovative solutions. Spring also has contributions from all the big names in tech, including Alibaba, Amazon, Google, Microsoft, and more.

Spring is flexible

Spring's flexible and comprehensive set of extensions and third-party libraries let developers build almost any application imaginable. At its core, Spring Framework's [Inversion of Control \(IoC\)](#) and [Dependency Injection \(DI\)](#) features provide the foundation for a wide-ranging set of features and functionality. Whether you're building secure, reactive, cloud-based microservices for the web, or complex streaming data flows for the enterprise, Spring has the tools to help.

Spring is productive

[Spring Boot](#) transforms how you approach Java programming tasks, radically streamlining your experience. Spring Boot combines necessities such as an application context and an auto-configured, embedded web server to make [microservice](#) development a cinch. To go even faster, you can combine Spring Boot with Spring Cloud's rich set of supporting libraries, servers, patterns, and templates, to safely deploy entire microservices-based architectures into the [cloud](#), in record time.

Spring is fast

Our engineers care deeply about performance. With Spring, you'll notice fast startup, fast shutdown, and optimized execution, by default. Increasingly, Spring projects also support the [reactive](#) (nonblocking) programming model for even greater efficiency. Developer productivity

2 SIR C R REDDY COLLEGE OF ENGINEERING, ELURU

API & MICROSERVICES Unit-1 Dept. of CSE, **J S V G K R I S H N A**

is Spring's superpower. Spring Boot helps developers build applications with ease and with far less toil than other competing paradigms. Embedded web servers, auto-configuration, and “fat jars” help you get started quickly, and innovations like [LiveReload in Spring DevTools](#) mean developers can iterate faster than ever before. You can even start a new Spring project in seconds, with the Spring Initializr at [start.spring.io](#).

Spring is secure

Spring has a proven track record of dealing with security issues quickly and responsibly. The Spring committers work with security professionals to patch and test any reported vulnerabilities. Third-party dependencies are also monitored closely, and regular updates are issued to help keep your data and applications as safe as possible. In addition, [Spring Security](#) makes it easier for you to integrate with industry-standard security schemes and deliver trustworthy solutions that are secure by default.

Spring is supportive

The [Spring community](#) is enormous, global, diverse, and spans folks of all ages and capabilities, from complete beginners to seasoned pros. No matter where you are on your journey, you can find the support and resources you need to get you to the next level: [quickstarts](#), [videos](#), [meetups](#), [support](#), or even formal [training and certification](#).

What Spring can do?

Microservices

Quickly deliver production-grade features with independently evolvable microservices.

Reactive

Spring's asynchronous, nonblocking architecture means you can get more from your computing resources.

Cloud

Your code, any cloud—we've got you covered. Connect and scale your services, whatever your platform.

Web apps

Frameworks for fast, secure, and responsive web applications connected to any data store.

Serverless

The ultimate flexibility. Scale up on demand and scale to zero when there's no demand.

Event Driven

Integrate with your enterprise. React to business events. Act on your streaming data in realtime.

Batch

Automated tasks. Offline processing of data at a time to suit you.

2. What is Spring Framework:

The Spring Framework is a popular and widely used framework for building enterprise-level Java applications. It provides comprehensive infrastructure support for developing robust and maintainable applications. The framework is modular and provides various modules that cater to different aspects of application development.

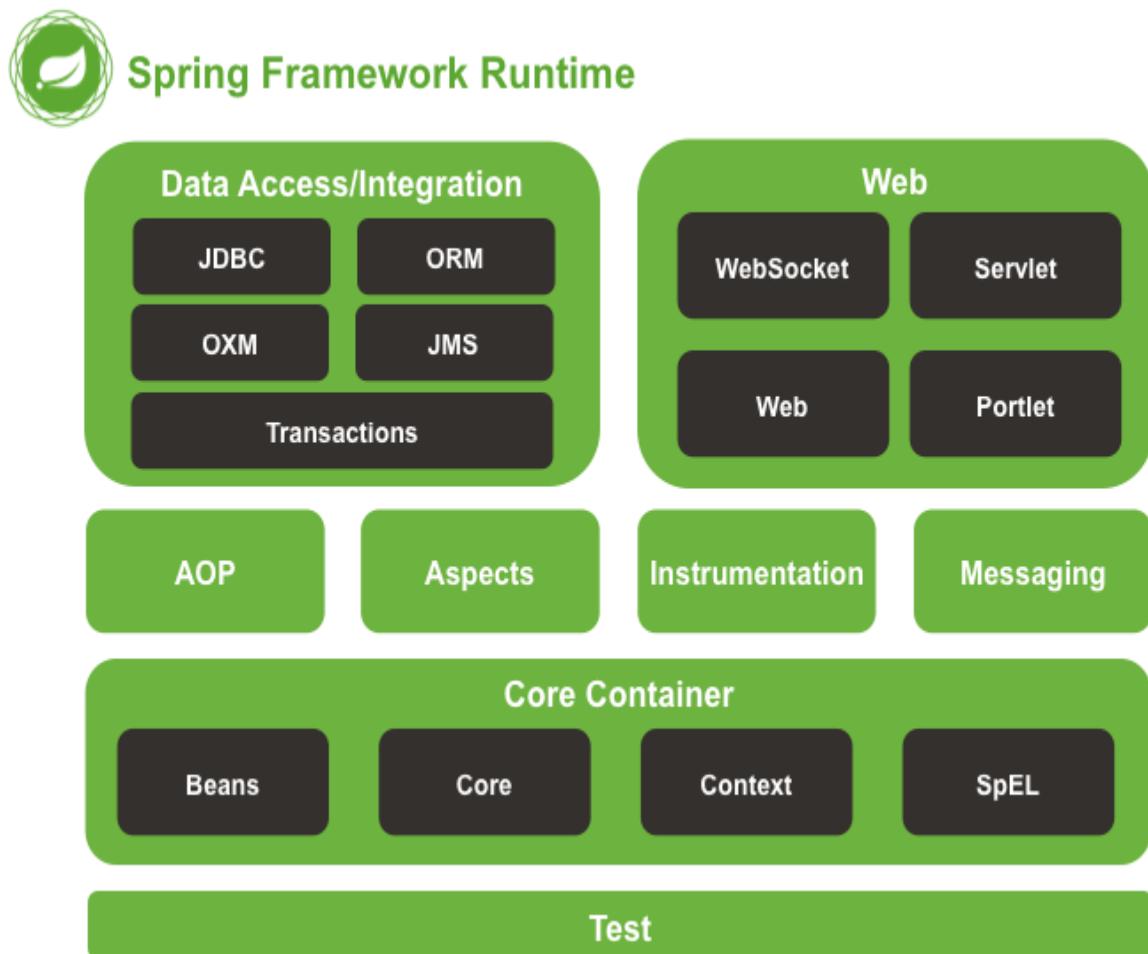
Spring enables you to build applications from "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.

Examples of how you, as an application developer, can use the Spring platform advantage:

- Make a Java method execute in a database transaction without having to deal with transaction APIs.
- Make a local Java method a remote procedure without having to deal with remote APIs.
- Make a local Java method a management operation without having to deal with JMX

3: Spring Framework - Modules:

Spring Framework Modules: 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**.



1.Core Container:

The *Core Container* consists of the Core, Beans, Context, and Expression Language modules.

The *Core and Beans* modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The *Context* module builds on the solid base provided by the *Core and Beans* modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event-propagation, resource-loading, and the transparent creation of contexts by, for example, a servlet container. The Context module also supports Java EE features such as EJB, JMX, and basic remoting. The ApplicationContext interface is the focal point of the Context module.

The *Expression Language* module provides a powerful expression language for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection as well as common list aggregations.

2.Data Access/Integration:

The *Data Access/Integration* layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

The *JDBC* module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

The *ORM* module provides integration layers for popular object-relational mapping APIs, including *JPA*, *JDO*, and *Hibernate*. Using the ORM package you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.

6 SIR C R REDDY COLLEGE OF ENGINEERING, ELURU

API & MICROSERVICES Unit-1 Dept. of CSE, **J S V G K R I S H N A**

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 (plain old Java objects)*.

3. Web:

The *Web* layer consists of the Web, Web-Servlet, WebSocket and Web-Portlet modules.

Spring's *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. It also contains the web-related parts of Spring's remoting support.

The *Web-Servlet* module contains Spring's model-view-controller ([MVC](#)) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.

The *Web-Portlet* module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

4. AOP and Instrumentation:

Spring's [AOP](#) module provides an *AOP Alliance*-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, you can also incorporate behavioral information into your code, in a manner similar to that of

.NET attributes.

The separate *Aspects* module provides integration with AspectJ.

The *Instrumentation* module provides class instrumentation support and classloader implementations to be used in certain application servers.

5.Test:

The *Test* module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

4:Configuring IoC Container Using Java-based Configuration:

1. Basic Concepts: @Bean and @Configuration:

The central artifacts in Spring's new Java-configuration support are `@Configuration`-annotated classes and `@Bean`-annotated methods.

The `@Bean` annotation is used to indicate that a method instantiates, configures and initializes a new object to be managed by the Spring IoC container. For those familiar with Spring's `<beans>` XML configuration the `@Bean` annotation plays the same role as the `<bean>` element. You can use `@Bean` annotated methods with any Spring `@Component`, however, they are most often used with `@Configuration` beans.

Annotating a class with `@Configuration` indicates that its primary purpose is as a source of bean definitions. Furthermore, `@Configuration` classes allow inter-bean dependencies to be defined by simply calling other `@Bean` methods in the same class. The simplest possible `@Configuration` class would read as follows:

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

The `AppConfig` class above would be equivalent to the following Spring `<beans>` XML:

```
<beans>
<bean id="myService" class="com.acme.services.MyServiceImpl"/>
```

</beans>

The @Bean and @Configuration annotations will be discussed in depth in the sections below. First, however, we'll cover the various ways of creating a spring container using Java-based configuration.

2. Instantiating the Spring container using AnnotationConfigApplicationContext

The sections below document

Spring's AnnotationConfigApplicationContext, new in Spring 3.0. This versatile ApplicationContext implementation is capable of accepting not only @Configuration classes as input, but also plain @Component classes and classes annotated with JSR-330 metadata.

When @Configuration classes are provided as input, the @Configuration class itself is registered as a bean definition, and all declared @Bean methods within the class are also registered as bean definitions.

When @Component and JSR-330 classes are provided, they are registered as bean definitions, and it is assumed that DI metadata such as @Autowired or @Inject are used within those classes where necessary.

Simple construction

In much the same way that Spring XML files are used as input when instantiating a ClassPathXmlApplicationContext, @Configuration classes may be used as input when instantiating an AnnotationConfigApplicationContext. This allows for completely XML-free usage of the Spring container:

```
public static void main(String[] args) {  
    ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(AppConfig.class);  
    MyService myService = ctx.getBean(MyService.class);  
    myService.doStuff();  
}
```

As mentioned above, AnnotationConfigApplicationContext is not limited to working only with @Configuration classes. Any @Component or JSR-330 annotated class may be supplied as input to the constructor. For example:

```
public static void main(String[] args) {  
    ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class,  
    Dependency2.class);  
    MyService myService = ctx.getBean(MyService.class);  
    myService.doStuff();  
}
```

The above assumes

that MyServiceImpl, Dependency1 and Dependency2 use Spring dependency injection annotations such as @Autowired.

3. Using the @Bean annotation:

@Bean is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports some of the attributes offered by <bean/>, such as: [init-method](#), [destroy-method](#), [autowiring](#) and name.

You can use the @Bean annotation in a @Configuration-annotated or in a @Component-annotated class.

Declaring a bean

To declare a bean, simply annotate a method with the @Bean annotation. You use this method to register a bean definition within an ApplicationContext of the type specified as the method's return value. By default, the bean name will be the same as the method name. The following is a simple example of a @Bean method declaration:

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public TransferService transferService() {  
        return new TransferServiceImpl();  
    }  
}
```

```
}
```

```
}
```

The preceding configuration is exactly equivalent to the following Spring XML:

```
<beans>
  <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

Both declarations make a bean named transferService available in the ApplicationContext, bound to an object instance of type TransferServiceImpl:

transferService -> com.acme.TransferServiceImpl

4. Using the @Configuration annotation:

@Configuration is a class-level annotation indicating that an object is a source of bean definitions. @Configuration classes declare beans via public @Bean annotated methods. Calls to @Bean methods on @Configuration classes can also be used to define inter-bean dependencies..

Injecting inter-bean dependencies

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another:

```
@Configuration
public class AppConfig {

  @Bean
  public Foo foo() {
    return new Foo(bar());
  }

  @Bean
  public Bar bar() {
    return new Bar();
}
```

In the example above, the foobean receives a reference to barvia constructor injection.

5:Dependency Injection:Constructor Injection and Setter Injection:

Dependency injection (DI) 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* (IoC), 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.

Constructor-based dependency injection:

Constructor-based DI is 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. The following example shows a class that can only be dependency-injected with constructor injection. Notice that there is nothing *special* about this class, it is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

```
public class SimpleMovieLister {  
  
    // the SimpleMovieLister has a dependency on a MovieFinder  
    private MovieFinder movieFinder;
```

```
// a constructor so that the Spring container can inject a MovieFinder  
public SimpleMovieLister(MovieFinder movieFinder) {  
    this.movieFinder = movieFinder;  
}  
  
// business logic that actually uses the injected MovieFinder is omitted...  
}
```

Setter-based dependency injection:

Setter-based DI is 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.

The following example shows a class that can only be dependency-injected using pure setter

injection. This class is conventional Java. It is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

```
public class SimpleMovieLister {  
  
    // the SimpleMovieLister has a dependency on the MovieFinder  
    private MovieFinder movieFinder;  
  
    // a setter method so that the Spring container can inject a MovieFinder  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // business logic that actually uses the injected MovieFinder is omitted...  
}
```

The ApplicationContext supports constructor-based and setter-based DI for the beans it manages. It also supports setter-based DI after some dependencies have already been injected through the constructor approach. You configure the dependencies in the form of a BeanDefinition, which you use in conjunction with PropertyEditorinstances to convert properties from one format to another. However, most Spring users do not work with these classes directly (i.e., programmatically) but rather with XML bean definitions, annotated components (i.e., classes annotated with @Component, @Controller, etc.), or @Bean methods in Java-based @Configuration classes. These sources are then converted internally into instances of BeanDefinition and used to load an entire Spring IoC container instance.

Examples of dependency injection:

The following example uses XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifies some bean definitions:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- setter injection using the nested ref element -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>

    <!-- setter injection using the neater ref attribute -->
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }

    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }
}
```

```
public void setIntegerProperty(int i) {
    this.i = i;
}

}
```

In the preceding example, setters are declared to match against the properties specified in the XML file. The following example uses constructor-based DI:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- constructor injection using the nested ref element -->
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>

    <!-- constructor injection using the neater ref attribute -->
    <constructor-arg ref="yetAnotherBean"/>

    <constructor-arg type="int" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public ExampleBean(
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
        this.beanOne = anotherBean;
        this.beanTwo = yetAnotherBean;
        this.i = i;
    }
}
```

6:AutoScanning in Spring.

AutoScanning in Spring refers to the process of automatically detecting and registering Spring components, such as beans, within the application context. This eliminates the need for explicit configuration and allows for a more streamlined and flexible development process.

AutoScanning is a key feature of Spring that simplifies the management of components and promotes modularity and reusability.

Automatically Detecting Classes and Registering Bean Definitions

Spring can automatically detect stereotyped classes and register corresponding BeanDefinition instances with the ApplicationContext. For example, the following two classes are

15 SIR C R REDDY COLLEGE OF ENGINEERING, ELURU

API & MICROSERVICES Unit-1 Dept. of CSE, JSVGKRISHNA

eligible for such autodetection:

```
@Service
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

}

```
@Repository
public class JpaMovieFinder implements MovieFinder {
    // implementation elided for clarity
}
```

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.)

```
@Configuration
@ComponentScan(basePackages = "org.example") s
public class AppConfig {
    // ...
}
```

The following alternative uses XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           https://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           https://www.springframework.org/schema/context/spring-
                           context.xsd">

    <context:component-scan base-package="org.example"/>

</beans>
```

API & MICROSERVICES

IMPORTANT QUESTIONS - UNIT-1

1. Why Spring, What is Spring Framework?
2. Explain Spring Framework - Modules?
3. Describe about Configuring IoC container using Java-based configuration?
4. Discuss Dependency Injection and Constructor Injection?
5. Describe Setter Injection?
6. Explain AutoScanning?