1. Spring Framework Overview

Spring makes it easy to create Java enterprise applications. It provides everything you need to embrace the Java language in an enterprise environment, with support for **Groovy and Kotlin** as alternative languages on the **JVM**, and with the flexibility to create many kinds of architectures depending on an application's needs. **As of Spring Framework 6.0, Spring requires Java 17+.**

Spring supports a wide range of application scenarios. In a large enterprise, applications often exist for a long time and have to run on a JDK and application server whose upgrade cycle is beyond developer control.

Others may run as a single jar with the server embedded, possibly in a cloud environment. Yet others may be standalone applications (such as batch or integration workloads) that do not need a server.

Spring is open source. It has a large and active community that provides continuous feedback based on a diverse range of real-world use cases. This has helped Spring to successfully evolve over a very long time.

What We Mean by "Spring"

The term "Spring" means different things in different contexts. It can be used to refer to the Spring Framework project itself, which is where it all started.
Over time, other Spring projects have been built on top of the Spring Framework. Most often, when people say "Spring", they mean the entire family of projects.

The Spring Framework is divided into modules. Applications can choose which modules they need. At the heart are the modules of the core

container, including a configuration model and a dependency injection mechanism. Beyond that, the Spring Framework provides foundational support for different application architectures, including messaging, transactional data and persistence, and web. It also includes the Servlet-based Spring MVC web framework and, in parallel, the Spring WebFlux reactive web framework.

History of Spring and the Spring Framework

Spring came into being in **2003** as a response to the complexity of the early J2EE specifications. While some consider Java EE and its modern-day successor Jakarta EE to be in competition with Spring, they are in fact complementary. The Spring programming model does not embrace the Jakarta EE platform specification; rather, it integrates with carefully selected individual specifications from the traditional EE umbrella:

- **Servlet API (JSR 340)**
- **WebSocket API (JSR 356)**
- **Concurrency Utilities (JSR 236)**
- **JSON Binding API (JSR 367)**
- **Bean Validation (JSR 303)**
- **JPA (JSR 338)**
- **JMS (JSR 914)**

- as well as **JTA/JCA** setups for transaction coordination, if necessary.
  The Spring Framework also supports the Dependency Injection (JSR 330) and Common Annotations (JSR 250) specifications, which application developers may choose to use instead of the Spring-specific mechanisms provided by the Spring Framework. Originally, those were based on common javax packages.

As of **Spring Framework 6.0,** Spring has been upgraded to the **Jakarta EE 9 level (e.g. Servlet 5.0+, JPA 3.0+),** based on the **jakarta** namespace instead of the traditional **javax** packages.

With **EE 9** as the minimum and **EE 10** supported already, Spring is prepared to provide out-of-the-box support for the further evolution of the **Jakarta EE APIs**. **Spring Framework 6.0** is fully compatible with **Tomcat 10.1, Jetty 11 and Undertow 2.3 as web servers**, and also with **Hibernate ORM 6.1**.

Over time, the role of Java/Jakarta EE in application development has evolved. In the early days of J2EE and Spring, applications were created to be deployed to an application server.

Today, with the help of Spring Boot, applications are created in a devops- and cloud-friendly way, with the Servlet container embedded and trivial to change.

As of Spring Framework 5, a WebFlux application does not even use the Servlet API directly and can run on servers (such as Netty) that are not Servlet containers.

Spring continues to innovate and to evolve. Beyond the Spring Framework, there are other projects, such as Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, among others. It's important to remember that each project has its own source code repository, issue tracker, and release cadence.

2. Introduction to the Spring Framework

The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

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 benefit from the Spring platform:

- Make a Java method execute in a database transaction without having to deal with transaction APIs.
- Make a local Java method an HTTP endpoint without having to deal with the Servlet API.
- Make a local Java method a message handler without having to deal with the JMS API.
- Make a local Java method a management operation without having to deal with the JMX API.

2.1 Dependency Injection and Inversion of Control

A Java application — a loose term that runs the gamut from constrained, embedded applications to n-tier, server-side enterprise applications — typically consists of objects that collaborate to form the application proper. Thus the objects in an application have *dependencies* on each other.

Although the Java platform provides a wealth of application development functionality, it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers. Although you can use design patterns such as *Factory*, *Abstract Factory*, *Builder*, *Decorator*, and *Service Locator* to compose the various classes and object instances that make up an application, these patterns are simply that: best practices given a name, with a description of what the pattern does, where to apply it, the problems it addresses, and so forth. Patterns are formalized best practices that *you must implement yourself* in your application.

The Spring Framework *Inversion of Control* (IoC) component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own application(s). Numerous organizations and institutions use the Spring Framework in this manner to engineer robust, *maintainable* applications.

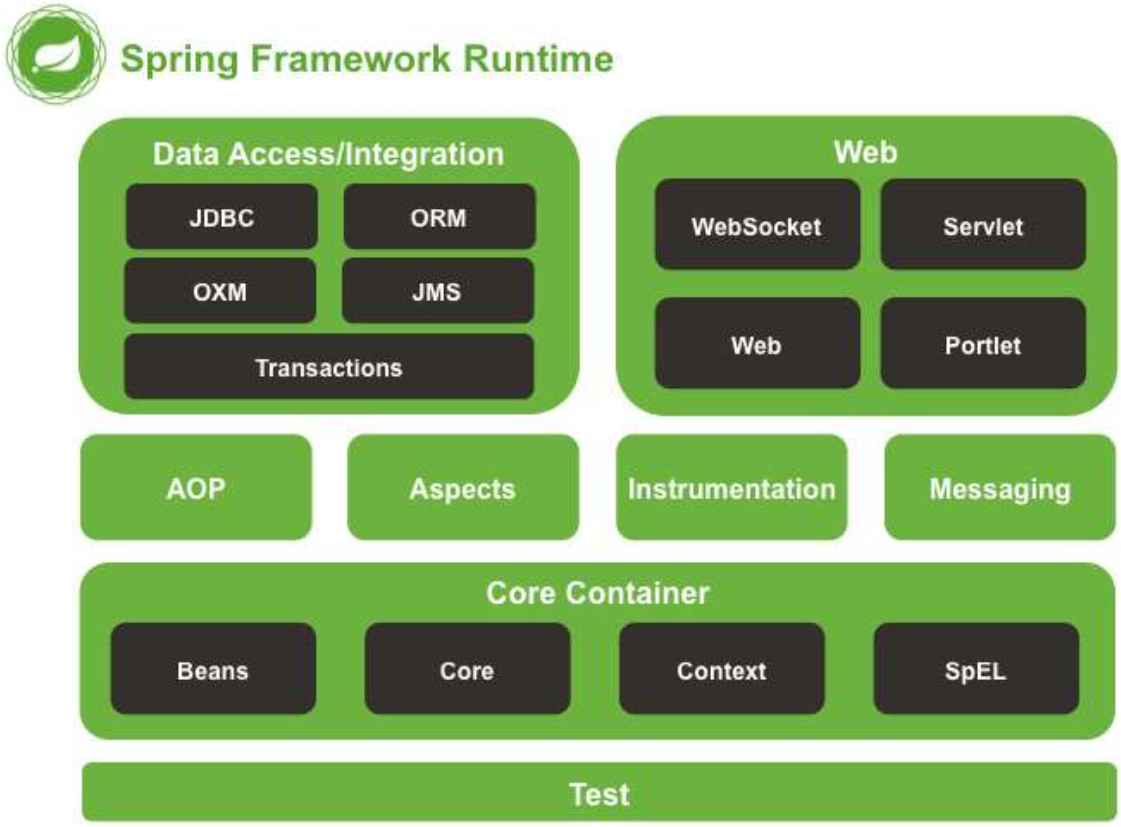
Background

"***The question is, what aspect of control are [they] inverting?***" **Martin Fowler** posed this question about **Inversion of Control (IoC**) on his site (https://martinfowler.com/articles/injection.html) in 2004. Fowler suggested renaming the principle to make it more self-explanatory and came up with *Dependency Injection*.

2.2 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, Messaging, and Test,** as shown in the following diagram.

Figure 2.1. Overview of the Spring Framework



The following sections list the available modules for each feature along with their artifact names and the topics they cover. Artifact names correlate to *artifact IDs* used in Dependency Management tools.

2.2.1 Core Container

The *Core Container* consists of the **spring-core**, **spring-beans**, **spring-context**, **spring-context-support**, and **spring-expression** (Spring Expression Language) modules.

The **spring-core** and **spring-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*** (**spring-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. **spring-context-support** provides support for integrating common third-party libraries into a Spring application context for caching (**EhCache, Guava, JCache**), mailing (**JavaMail**), scheduling (**CommonJ, Quartz**) and template engines (**FreeMarker, JasperReports, Velocity**).

The **spring-expression** 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 content 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.

2.2.2 AOP and Instrumentation

The spring-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.

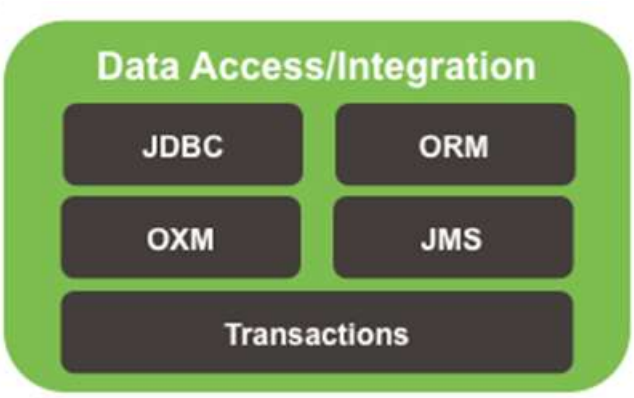
The separate **spring-aspects** module provides integration with **AspectJ.**

The **spring-instrument** module provides class instrumentation support and classloader implementations to be used in certain application servers. The **spring-instrument-tomcat** module contains Spring's instrumentation agent for Tomcat.

2.2.3 Messaging

Spring Framework 4 includes a **spring-messaging** module with key abstractions from the *Spring Integration* project such as **Message**, **MessageChannel**, **MessageHandler**, and others to serve as a foundation for messaging-based applications. The module also includes a set of annotations for mapping messages to methods, similar to the Spring MVC annotation based programming model.

2.2.4 Data Access/Integration



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

The **spring-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 **spring-tx** module supports programmatic and declarative transaction management for classes that implement special interfaces and for *all your POJOs (Plain Old Java Objects)*.

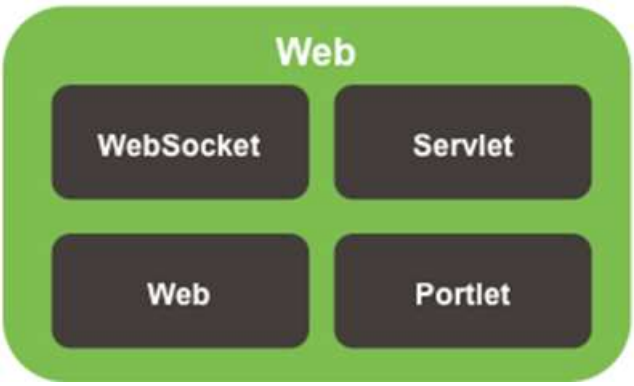
The **spring-orm** module provides integration layers for popular object-relational mapping APIs, including **JPA, JDO**, and **Hibernate**. Using the **spring-orm** module you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple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

The **spring-oxm** module provides an abstraction layer that supports **Object/XML mapping** implementations such as **JAXB, Castor, XMLBeans, JiBX and XStream.**

The **spring-jms** module (**Java Messaging Service**) contains features for producing and consuming messages. Since Spring Framework 4.1, it provides integration with the **spring-messaging** module.

2.2.5 Web

The *Web* layer consists of the **spring-web**, **spring-webmvc**, **spring-websocket**, **and** **spring-webmvc-portlet** modules.



The **spring-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 an HTTP client and the web-related parts of Spring's remoting support.

The **spring-webmvc** module (also known as the ***Web-Servlet* module**) contains Spring's model-view-controller (***MVC***) and REST Web Services implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms and integrates with all of the other features of the Spring Framework.

The **spring-webmvc-portlet** module (also known as the *Web-Portlet* module) provides the MVC implementation to be used in a Portlet environment and mirrors the functionality of the Servlet-based **spring-webmvc** module.

2.2.6 Test

The **spring-test** module supports the unit **testing and integration** testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring **ApplicationContext**s and **caching** of those

contexts. It also provides **mock objects** that you can use to test your code in isolation.

Maven Dependency Management

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.3.15.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Introduction to the Spring IoC Container and Beans

This chapter covers the Spring Framework implementation of the **Inversion of Control** (**IoC**) principle. **Dependency injection (DI)** is a specialized form of **IoC**, 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 **IoC** 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** interface provides an advanced configuration mechanism capable of managing any type of object. **ApplicationContext** 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. The **ApplicationContext** is a complete superset of the **BeanFactory** and is used exclusively in this chapter in descriptions of Spring's IoC container.

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 managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

Configuration Metadata

XML as an External Configuration DSL

XML-based configuration metadata configures these beans as <bean/> elements inside a top-level <beans/> element. The following example shows the basic structure of XML-based configuration metadata:

```
<?xml version="1.0" encoding="UTF-8"?>
<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

        https://www.springframework.org/schema/beans/spring-beans.xsd">

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

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

        <!-- more bean definitions go here -->

</beans>
```

The **id** attribute is a string that identifies the individual bean definition.

The **class** attribute defines the type of the bean and uses the fully qualified class name.

The value of the id attribute can be used to refer to collaborating objects. The XML for referring to collaborating objects is not shown in this example.

For instantiating a container, the location path or paths to the XML resource files need to be supplied
a **ClassPathXmlApplicationContext** constructor 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.