# FORM:

# https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html

# Spring Framework

Version 5.0.0.RC2

Table of Contents

* [1. Getting Started with Spring](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html" \l "overview-getting-started-with-spring)
* [2. Introduction to the Spring Framework](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html" \l "overview)
  + [2.1. Dependency Injection and Inversion of Control](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html" \l "overview-dependency-injection)
  + [2.2. Framework Modules](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html" \l "overview-modules)
  + [2.3. Usage scenarios](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html" \l "overview-usagescenarios)

## 1. Getting Started with Spring

This reference guide provides detailed information about the Spring Framework. It provides comprehensive documentation for all features, as well as some background about the underlying concepts (such as *"Dependency Injection"*) that Spring has embraced.

If you are just getting started with Spring, you may want to begin using the Spring Framework by creating a [Spring Boot](https://projects.spring.io/spring-boot/) based application. Spring Boot provides a quick (and opinionated) way to create a production-ready Spring based application. It is based on the Spring Framework, favors convention over configuration, and is designed to get you up and running as quickly as possible.

You can use [start.spring.io](https://start.spring.io/) to generate a basic project or follow one of the ["Getting Started" guides](https://spring.io/guides) like the [Getting Started Building a RESTful Web Service](https://spring.io/guides/gs/rest-service/) one. As well as being easier to digest, these guides are very *task focused*, and most of them are based on Spring Boot. They also cover other projects from the Spring portfolio that you might want to consider when solving a particular problem.

## 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.

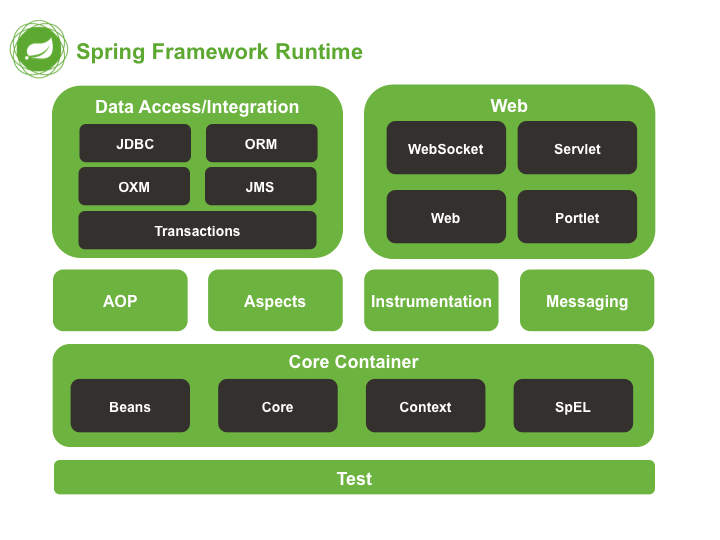
The Spring Framework *Inversion of Control* (IoC) component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into 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](http://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 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](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html" \l "dependency-management).

#### 2.2.1. Core Container

The *[Core Container](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/core.html" \l "beans-introduction)* 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](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/core.html" \l "beans-introduction), 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](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/core.html" \l "context-introduction)* (spring-context) module builds on the solid base provided by the *[Core and Beans](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/core.html" \l "beans-introduction)* 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, in particular for caching (EhCache, JCache) and scheduling (CommonJ, Quartz).

The spring-expression module provides a powerful *[Expression Language](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/core.html" \l "expressions)* 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](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/core.html" \l "aop-introduction)* 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](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/data-access.html" \l "jdbc-introduction)-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](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/data-access.html" \l "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](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/data-access.html" \l "orm-introduction) APIs, including [JPA](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/data-access.html" \l "orm-jpa) and [Hibernate](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/data-access.html" \l "orm-hibernate). Using the spring-orm module you can use 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](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/data-access.html" \l "oxm) implementations such as JAXB, Castor, JiBX and XStream.

The spring-jms module ([Java Messaging Service](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/integration.html" \l "jms)) 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 and spring-websocket 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.

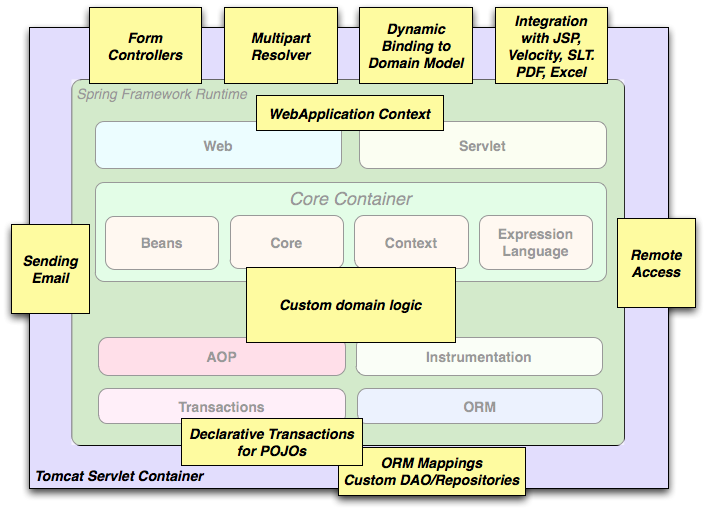
The spring-webmvc module (also known as the *Web-Servlet* module) contains Spring’s model-view-controller (*[MVC](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/web.html" \l "mvc-introduction)*) and REST Web Services implementation for web applications. Spring’s MVC framework provides a clean separation between domain model code and web forms and integrates with all of the other features of the Spring Framework.

#### 2.2.6. Test

The spring-test module supports the [unit testing](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/testing.html" \l "unit-testing) and [integration testing](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/testing.html" \l "integration-testing) of Spring components with JUnit or TestNG. It provides consistent [loading](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/testing.html" \l "testcontext-ctx-management) of Spring ApplicationContexts and [caching](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/testing.html" \l "testcontext-ctx-management-caching) of those contexts. It also provides [mock objects](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/testing.html" \l "mock-objects) that you can use to test your code in isolation.

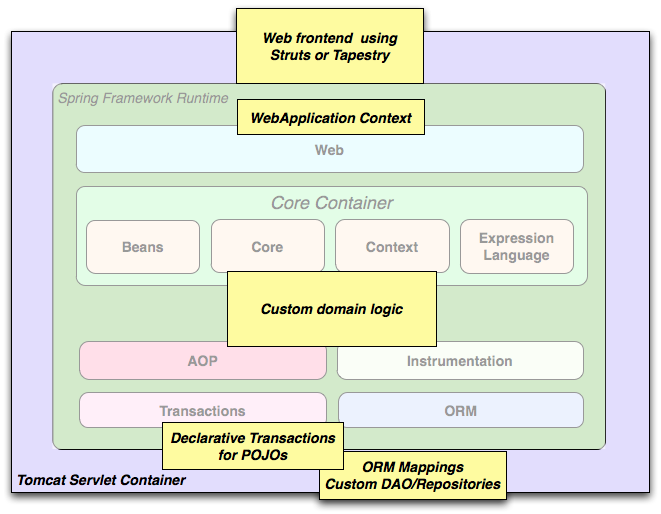
### 2.3. Usage scenarios

The building blocks described previously make Spring a logical choice in many scenarios, from embedded applications that run on resource-constrained devices to full-fledged enterprise applications that use Spring’s transaction management functionality and web framework integration.



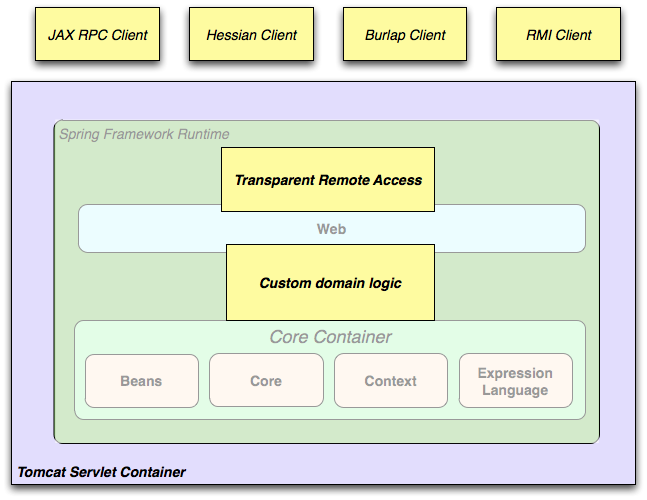
*Figure 2. Typical full-fledged Spring web application*

Spring’s [declarative transaction management features](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/data-access.html" \l "transaction-declarative) make the web application fully transactional, just as it would be if you used EJB container-managed transactions. All your custom business logic can be implemented with simple POJOs and managed by Spring’s IoC container. Additional services include support for sending email and validation that is independent of the web layer, which lets you choose where to execute validation rules. Spring’s ORM support is integrated with JPA and Hibernate; for example, when using Hibernate, you can continue to use your existing mapping files and standard Hibernate SessionFactory configuration. Form controllers seamlessly integrate the web-layer with the domain model, removing the need for ActionForms or other classes that transform HTTP parameters to values for your domain model.



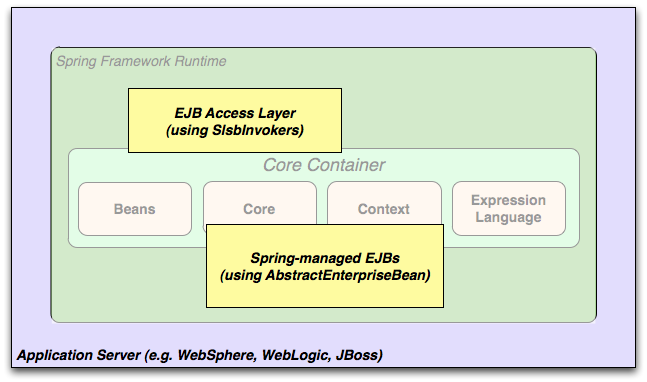
*Figure 3. Spring middle-tier using a third-party web framework*

Sometimes circumstances do not allow you to completely switch to a different framework. The Spring Framework does *not* force you to use everything within it; it is not an *all-or-nothing* solution. Existing front-ends built with Struts, Tapestry, JSF or other UI frameworks can be integrated with a Spring-based middle-tier, which allows you to use Spring transaction features. You simply need to wire up your business logic using an ApplicationContext and use a WebApplicationContext to integrate your web layer.



*Figure 4. Remoting usage scenario*

When you need to access existing code through web services, you can use Spring’s Hessian-, Rmi- or HttpInvokerProxyFactoryBean classes. Enabling remote access to existing applications is not difficult.



*Figure 5. EJBs - Wrapping existing POJOs*

The Spring Framework also provides an [access and abstraction layer](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/integration.html" \l "ejb) for Enterprise JavaBeans, enabling you to reuse your existing POJOs and wrap them in stateless session beans for use in scalable, fail-safe web applications that might need declarative security.

#### 2.3.1. Dependency Management and Naming Conventions

Dependency management and dependency injection are different things. To get those nice features of Spring into your application (like dependency injection) you need to assemble all the libraries needed (jar files) and get them onto your classpath at runtime, and possibly at compile time. These dependencies are not virtual components that are injected, but physical resources in a file system (typically). The process of dependency management involves locating those resources, storing them and adding them to classpaths. Dependencies can be direct (e.g. my application depends on Spring at runtime), or indirect (e.g. my application depends on commons-dbcp which depends on commons-pool). The indirect dependencies are also known as "transitive" and it is those dependencies that are hardest to identify and manage.

If you are going to use Spring you need to get a copy of the jar libraries that comprise the pieces of Spring that you need. To make this easier Spring is packaged as a set of modules that separate the dependencies as much as possible, so for example if you don’t want to write a web application you don’t need the spring-web modules. To refer to Spring library modules in this guide we use a shorthand naming convention spring-\* or spring-\*.jar, where \* represents the short name for the module (e.g. spring-core, spring-webmvc, spring-jms, etc.). The actual jar file name that you use is normally the module name concatenated with the version number (e.g. *spring-core-5.0.0.RC2.jar*).

Each release of the Spring Framework will publish artifacts to the following places:

Maven Central, which is the default repository that Maven queries, and does not require any special configuration to use. Many of the common libraries that Spring depends on also are available from Maven Central and a large section of the Spring community uses Maven for dependency management, so this is convenient for them. The names of the jars here are in the form spring-\*-<version>.jar and the Maven groupId is org.springframework.

In a public Maven repository hosted specifically for Spring. In addition to the final GA releases, this repository also hosts development snapshots and milestones. The jar file names are in the same form as Maven Central, so this is a useful place to get development versions of Spring to use with other libraries deployed in Maven Central. This repository also contains a bundle distribution zip file that contains all Spring jars bundled together for easy download.

So the first thing you need to decide is how to manage your dependencies: we generally recommend the use of an automated system like Maven, Gradle or Ivy, but you can also do it manually by downloading all the jars yourself.

##### Spring Dependencies and Depending on Spring

Although Spring provides integration and support for a huge range of enterprise and other external tools, it intentionally keeps its mandatory dependencies to an absolute minimum: you shouldn’t have to locate and download (even automatically) a large number of jar libraries in order to use Spring for simple use cases. For basic dependency injection there is only one mandatory external dependency, and that is for logging (see below for a more detailed description of logging options).

Next we outline the basic steps needed to configure an application that depends on Spring, first with Maven and then with Gradle and finally using Ivy. In all cases, if anything is unclear, refer to the documentation of your dependency management system, or look at some sample code - Spring itself uses Gradle to manage dependencies when it is building, and our samples mostly use Gradle or Maven.

##### Maven Dependency Management

If you are using [Maven](https://maven.apache.org/) for dependency management you don’t even need to supply the logging dependency explicitly. For example, to create an application context and use dependency injection to configure an application, your Maven dependencies will look like this:

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.0.0.RC2</version>

<scope>runtime</scope>

</dependency></dependencies>

That’s it. Note the scope can be declared as runtime if you don’t need to compile against Spring APIs, which is typically the case for basic dependency injection use cases.

The example above works with the Maven Central repository. To use the Spring Maven repository (e.g. for milestones or developer snapshots), you need to specify the repository location in your Maven configuration. For full releases:

<repositories>

<repository>

<id>io.spring.repo.maven.release</id>

<url>http://repo.spring.io/release/</url>

<snapshots><enabled>false</enabled></snapshots>

</repository></repositories>

##### Gradle Dependency Management

To use the Spring repository with the [Gradle](http://www.gradle.org/) build system, include the appropriate URL in the repositories section:

repositories {

mavenCentral()

*// and optionally...*

maven { url "http://repo.spring.io/release" }

}

You can change the repositories URL from /release to /milestone or /snapshot as appropriate. Once a repository has been configured, you can declare dependencies in the usual Gradle way:

dependencies {

compile("org.springframework:spring-context:5.0.0.RC2")

testCompile("org.springframework:spring-test:5.0.0.RC2")

}

##### Ivy Dependency Management

If you prefer to use [Ivy](https://ant.apache.org/ivy) to manage dependencies then there are similar configuration options.

To configure Ivy to point to the Spring repository add the following resolver to your ivysettings.xml:

<resolvers>

<ibiblio name="io.spring.repo.maven.release"

m2compatible="true"

root="http://repo.spring.io/release/"/></resolvers>

##### Distribution Zip Files

Although using a build system that supports dependency management is the recommended way to obtain the Spring Framework, it is still possible to download a distribution zip file.

Distribution zips are published to the Spring Maven Repository (this is just for our convenience, you don’t need Maven or any other build system in order to download them).

To download a distribution zip open a web browser to [http://repo.spring.io/release/org/springframework/spring](https://repo.spring.io/release/org/springframework/spring) and select the appropriate subfolder for the version that you want. Distribution files end -dist.zip, for example spring-framework-{spring-version}-RELEASE-dist.zip. Distributions are also published for [milestones](https://repo.spring.io/milestone/org/springframework/spring) and [snapshots](https://repo.spring.io/snapshot/org/springframework/spring).

#### 2.3.2. Logging

Spring’s logging setup has been revised for Spring 5: It is still based on the Apache Commons Logging API, also known as Jakarta Commons Logging (JCL). However, spring-core refers to a custom Commons Logging bridge in the spring-jcl module now, with a Spring-specific LogFactory implementation which automatically bridges to [Log4j 2](https://logging.apache.org/log4j/2.x/), [SLF4J](http://www.slf4j.org/), or the JDK’s own java.util.logging (JUL). This implementation acts like the JCL-over-SLF4J bridge but with a range of dynamically detected providers, analogous to JBoss Logging’s common targets (as supported by e.g. Hibernate and Undertow).

As a benefit, there is no need for external bridges like JCL-over-SLF4J anymore, and correspondingly no need for a manual exclude of the standard Commons Logging jar from spring-core dependencies. Instead, it all just works in Spring’s autodetection style at runtime: Simply put Log4j 2.x or SLF4J on your classpath, without any extra bridge jars, or rely on default logging through JUL (with a customizable JUL setup). And nicely aligned, default Hibernate setup will choose the same common log targets.

If both Log4j and SLF4J are present, the Log4j API will be used preferably (since it directly matches JCL’s signatures and natively supports a 'fatal' log level as well as lazily resolved message objects), analogous to JBoss Logging’s provider preferences. Log4j may nevertheless be configured to delegate to SLF4J, or SLF4J may be configured to delegate to Log4j: Please check the instructions on their websites on how to arrive at a consistent outcome in such a mixed scenario.

##### Using Log4j 2.x

[Log4j 2](https://logging.apache.org/log4j) established itself as a fresh rewrite of the original Log4j project (1.x is EOL now). As of Spring 5, the embedded logging bridge will automatically delegate to Log4j 2.x when available on the classpath.

So to use Log4j with Spring, all you need to do is put Log4j on the classpath and provide it with a configuration file (log4j2.xml, log4j2.properties, or other [supported configuration formats](https://logging.apache.org/log4j/2.x/manual/configuration.html)). For Maven users, the minimal dependency needed is:

<dependencies>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.8.2</version>

</dependency></dependencies>

If you also wish to enable SLF4J to delegate to Log4j, e.g. for other libraries which use SLF4J by default, the following dependency is also needed:

<dependencies>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-slf4j-impl</artifactId>

<version>2.8.2</version>

</dependency></dependencies>

##### Using SLF4J with Logback

The Simple Logging Facade for Java ([SLF4J](http://www.slf4j.org/)) is a popular API used by other libraries commonly used with Spring. It is typically used with [Logback](https://logback.qos.ch/) which is a native implementation of the SLF4J API and therefore autodetected by Spring when added to the application classpath:

<dependencies>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.2</version>

</dependency></dependencies>

Alternatively, you may also configure SLF4J to delegate to Log4j (see above) or to JUL, in particular for other libraries which use SLF4J by default. Note that it is not important for all libraries to go through the same logging facade; it only matters that they eventually delegate to the same log provider. So while Spring may go to Log4j directly, other libraries may go through the SLF4J binding for Log4j, or analogously for JUL.

##### Using JUL (java.util.logging)

Spring will delegate to java.util.logging by default, provided that no Log4j or SLF4J API is detected on the classpath. So there is no special dependency to set up: just use Spring with no external dependency for log output to java.util.logging, either in a standalone application (with a custom or default JUL setup at the JDK level) or with an application server’s log system (and its system-wide JUL setup).

Note that the java.logging module is NOT present by default on JDK 9, since it is not included in java.base. This works fine when using Spring with Log4j or SLF4J since the JUL API is not referenced in such a scenario. However, when choosing to use JUL as a default log provider, remember to activate the java.logging module.

##### Commons Logging on WebSphere

Spring applications may run on a container that itself provides an implementation of JCL, e.g. IBM’s WebSphere Application Server (WAS). This does not cause issues per se but leads to two different scenarios that need to be understood:

In a "parent first" ClassLoader delegation model (the default on WAS), applications will always pick up the server-provided version of Commons Logging, delegating to the WAS logging subsystem (which is actually based on JUL). An application-provided variant of JCL, whether Spring 5’s or the JCL-over-SLF4J bridge, will effectively be ignored, along with any locally included log provider.

With a "parent last" delegation model (the default in a regular Servlet container but an explicit configuration option on WAS), an application-provided Commons Logging variant will be picked up, enabling you to set up a locally included log provider, e.g. Log4j or Logback, within your application. In case of no local log provider, Spring (like regular Commons Logging) will delegate to JUL by default, effectively logging to WebSphere’s logging subsystem like in the "parent first" scenario.

All in all, we recommend deploying Spring applications in the "parent last" model since it naturally allows for local providers as well as the server’s log subsystem.

# Core Technologies

This part of the reference documentation covers all of those technologies that are absolutely integral to the Spring Framework.

Foremost amongst these is the Spring Framework’s Inversion of Control (IoC) container. A thorough treatment of the Spring Framework’s IoC container is closely followed by comprehensive coverage of Spring’s Aspect-Oriented Programming (AOP) technologies. The Spring Framework has its own AOP framework, which is conceptually easy to understand, and which successfully addresses the 80% sweet spot of AOP requirements in Java enterprise programming.

Coverage of Spring’s integration with AspectJ (currently the richest - in terms of features - and certainly most mature AOP implementation in the Java enterprise space) is also provided.

## 1. The IoC container

### 1.1. Introduction to the Spring IoC container and beans

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) [[1](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/core.html" \l "_footnote_1" \o "View footnote.)] principle. IoC is 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* (IoC), 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/5.0.0.RC2/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/5.0.0.RC2/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; and 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. For more information on using the BeanFactory instead of the ApplicationContext, refer to [The BeanFactory](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/core.html" \l "beans-beanfactory).

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. 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.

### 1.2. Container overview

The interface org.springframework.context.ApplicationContext represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned 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 allows you to express the objects that compose your application and the rich interdependencies between such objects.

Several implementations of the ApplicationContext interface are supplied out-of-the-box with Spring. In standalone applications it is common to create an instance of [ClassPathXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.0.RC2/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) or [FileSystemXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.0.RC2/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html). While XML has been the traditional format for defining configuration metadata you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate web descriptor XML in the web.xml file of the application will typically suffice (see [Convenient ApplicationContext instantiation for web applications](https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/core.html" \l "context-create)). If you are using the[Spring Tool Suite](https://spring.io/tools/sts) Eclipse-powered development environment this boilerplate configuration can be easily created with few mouse clicks or keystrokes.

The following diagram is a high-level view of how Spring works. Your application classes are combined with configuration metadata so that after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.

