Spring Framework Reference Documentation

Authors

Rod Johnson , Juergen Hoeller , Keith Donald , Colin Sampaleanu , Rob Harrop , Thomas Risberg , Alef Arendsen , Darren Davison , Dmitriy Kopylenko , Mark Pollack , Thierry Templier , Erwin Vervaet , Portia Tung , Ben Hale , Adrian Colyer , John Lewis , Costin Leau , Mark Fisher , Sam Brannen , Ramnivas Laddad , Arjen Poutsma , Chris Beams , Tareq Abedrabbo , Andy Clement , Dave Syer , Oliver Gierke , Rossen Stoyanchev , Phillip Webb , Rob Winch , Brian Clozel , Stephane Nicoll , Sebastien Deleuze

4.3.4.RELEASE

Table of Contents

11. Aspect Oriented Programming with Spring

17. Transaction Management

28. Remoting and web services using Spring

Part I. Overview of Spring Framework

The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications. However, Spring is modular, allowing you to use only those parts that you need, without having to bring in the rest. You can use the IoC container, with any web framework on top, but you can also use only the Hibernate integration code or the JDBC abstraction layer. The Spring Framework supports declarative transaction management, remote access to your logic through RMI or web services, and various options for persisting your data. It offers a full-featured MVC framework, and enables you to integrate AOP transparently into your software.

Spring is designed to be non-intrusive, meaning that your domain logic code generally has no dependencies on the framework itself. In your integration layer (such as the data access layer), some dependencies on the data access technology and the Spring libraries will exist. However, it should be easy to isolate these dependencies from the rest of your code base.

This document is a reference guide to Spring Framework features. If you have any requests, comments, or questions on this document, please post them on the user mailing list. Questions on the Framework itself should be asked on StackOverflow (see https://spring.io/questions).

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 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 to generate a basic project or follow one of the "Getting Started" guides like the Getting Started Building a RESTful Web 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 a remote procedure without having to deal with remote APIs.
- Make a local Java method a management operation without having to deal with JMX APIs.
- Make a local Java method a message handler without having to deal with JMS APIs.

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.

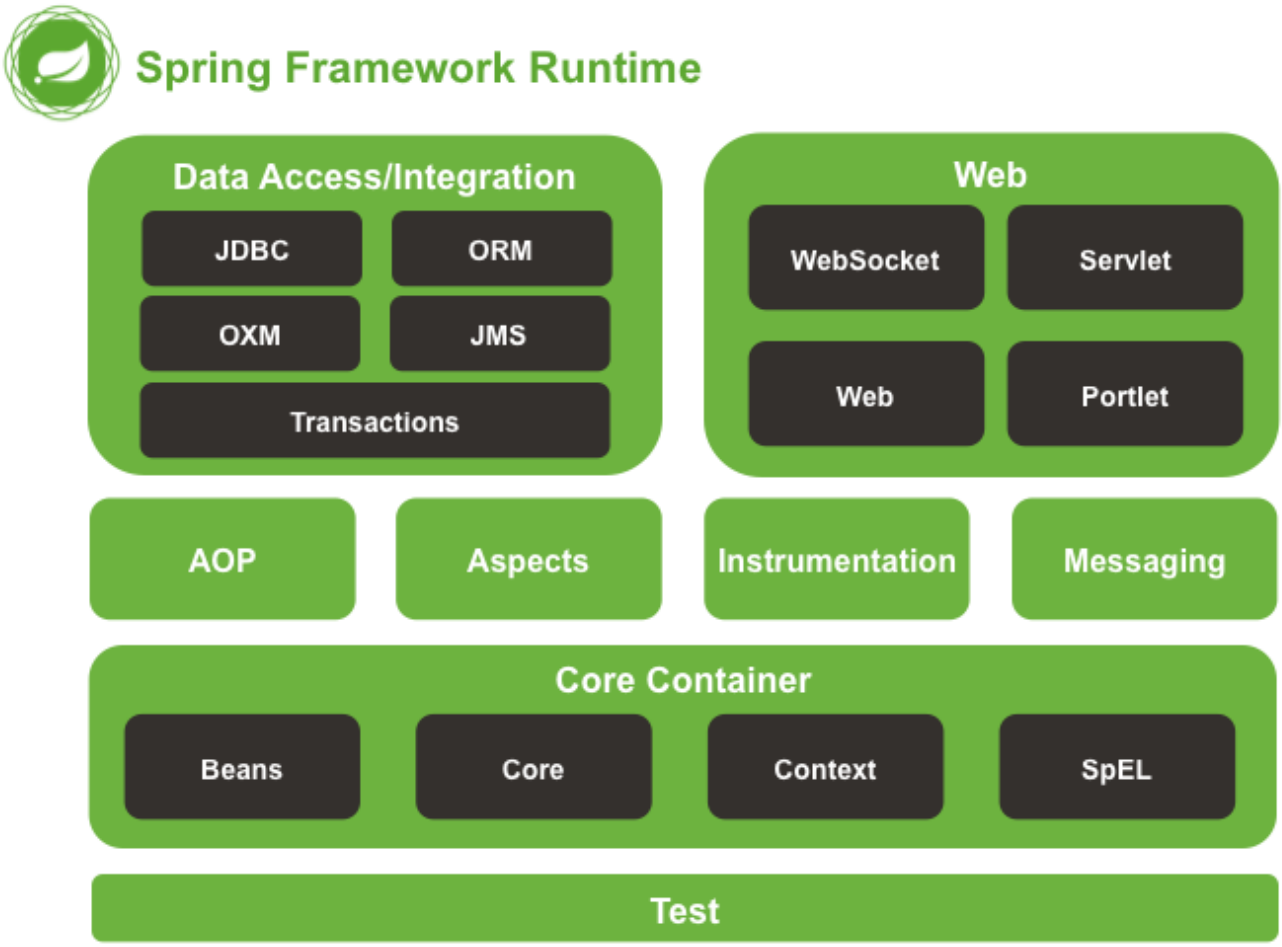
The Spring Framework *Inversion of Control* (IoC) component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into 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 in 2004. Fowler suggested renaming the principle to make it more self-explanatory and came up with *Dependency Injection*.

2.2 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 `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.

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 2.2. Typical full-fledged Spring web application

Spring's declarative transaction management features 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, Hibernate and JDO; 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.

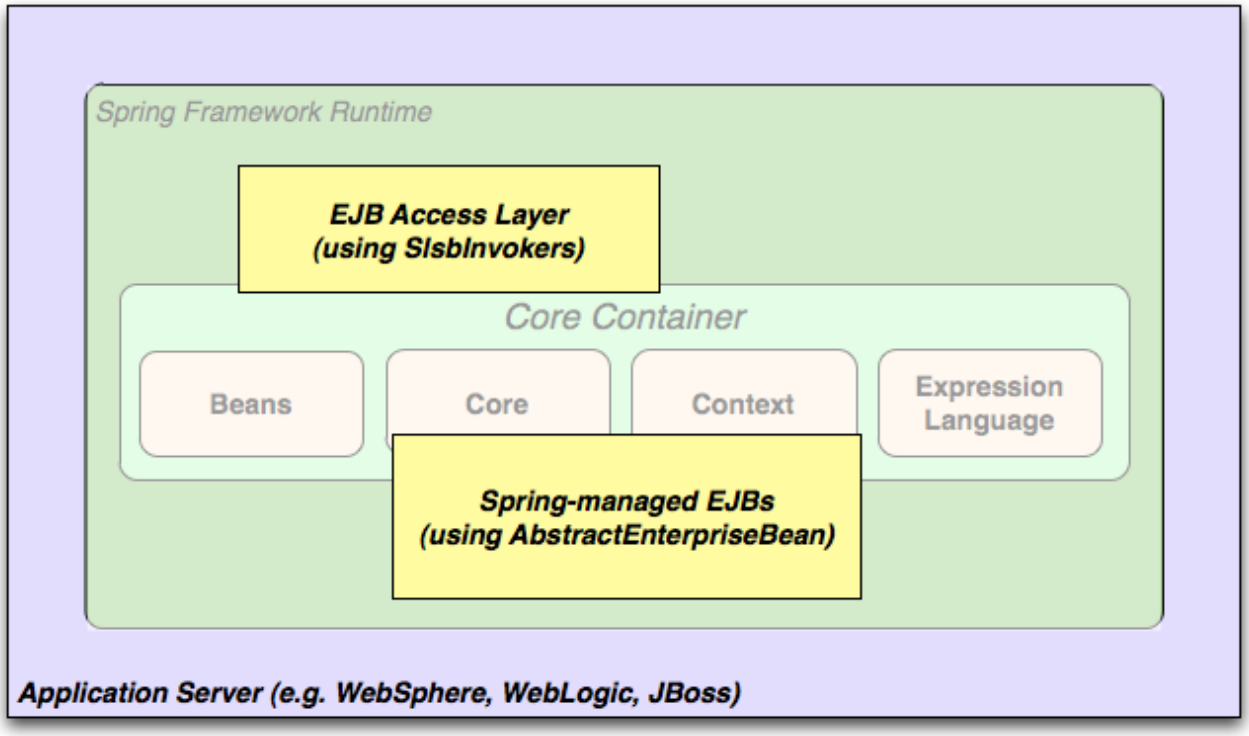
Figure 2.3. Spring middle-tier using a third-party web framework

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 2.4. Remoting usage scenario

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

Figure 2.5. EJBs - Wrapping existing POJOs



The Spring Framework also provides an access and abstraction layer 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-4.3.4.RELEASE.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.

You will find bellow the list of Spring artifacts. For a more complete description of each modules, see Section 2.2, "Modules".

Table 2.1. Spring Framework Artifacts

GroupId	ArtifactId	Description
org.springframework	spring-aop	Proxy-based AOP support
org.springframework	spring-aspects	AspectJ based aspects

GroupId	ArtifactId	Description
org.springframework	spring-beans	Beans support, including Groovy
org.springframework	spring-context	Application context runtime, including scheduling and remoting abstractions
org.springframework	spring-context-support	Support classes for integrating common third-party libraries into a Spring application context
org.springframework	spring-core	Core utilities, used by many other Spring modules
org.springframework	spring-expression	Spring Expression Language (SpEL)
org.springframework	spring-instrument	Instrumentation agent for JVM bootstrapping
org.springframework	spring-instrument-tomcat	Instrumentation agent for Tomcat
org.springframework	spring-jdbc	JDBC support package, including DataSource setup and JDBC access support
org.springframework	spring-jms	JMS support package, including helper classes to send and receive JMS messages
org.springframework	spring-messaging	Support for messaging architectures and protocols
org.springframework	spring-orm	Object/Relational Mapping, including JPA and Hibernate support
org.springframework	spring-oxm	Object/XML Mapping
org.springframework	spring-test	Support for unit testing and integration testing Spring components
org.springframework	spring-tx	Transaction infrastructure, including DAO support and JCA integration
org.springframework	spring-web	Web support packages, including client and web remoting

GroupId	ArtifactId	Description
org.springframework	spring-webmvc	REST Web Services and model-view-controller implementation for web applications
org.springframework	spring-webmvc-portlet	MVC implementation to be used in a Portlet environment
org.springframework	spring-websocket	WebSocket and SockJS implementations, including STOMP support

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 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>4.3.4.RELEASE</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>
```

For milestones:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.milestone</id>
        <url>http://repo.spring.io/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

And for snapshots:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.snapshot</id>
        <url>http://repo.spring.io/snapshot/</url>
        <snapshots><enabled>true</enabled></snapshots>
    </repository>
</repositories>
```

Maven "Bill Of Materials" Dependency

It is possible to accidentally mix different versions of Spring JARs when using Maven. For example, you may find that a third-party library, or another Spring project, pulls in a transitive dependency to an older release. If you forget to explicitly declare a direct dependency yourself, all sorts of unexpected issues can arise.

To overcome such problems Maven supports the concept of a "bill of materials" (BOM) dependency. You can import the `spring-framework-bom` in your `dependencyManagement` section to ensure that all spring dependencies (both direct and transitive) are at the same version.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>4.3.4.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

An added benefit of using the BOM is that you no longer need to specify the `<version>` attribute when depending on Spring Framework artifacts:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
    </dependency>
<dependencies>
```

Gradle Dependency Management

To use the Spring repository with the Gradle 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:4.3.4.RELEASE")
    testCompile("org.springframework:spring-test:4.3.4.RELEASE")
}
```

Ivy Dependency Management

If you prefer to use 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>
```

You can change the `root` URL from `/release/` to `/milestone/` or `/snapshot/` as appropriate.

Once configured, you can add dependencies in the usual way. For example (in `ivy.xml`):

```
<dependency org="org.springframework"
    name="spring-core" rev="4.3.4.RELEASE" conf="compile->runtime"/>
```

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 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 and snapshots.

2.3.2 Logging

Logging is a very important dependency for Spring because *a)* it is the only mandatory external dependency, *b)* everyone likes to see some output from the tools they are using, and *c)* Spring integrates with lots of other tools all of which have also made a choice of logging dependency. One of the goals of an application developer is often to have unified logging configured in a central place for the whole application, including all external components. This is more difficult than it might have been since there are so many choices of logging framework.

The mandatory logging dependency in Spring is the Jakarta Commons Logging API (JCL). We compile against JCL and we also make JCL `Log` objects visible for classes that extend the Spring Framework. It's important to users that all versions of Spring use the same logging library: migration is easy because backwards compatibility is preserved even with applications that extend Spring. The way we do this is to make one of the modules in Spring depend explicitly on `commons-logging` (the canonical implementation of JCL), and then make all the other modules depend on that at compile time. If you are using Maven for example, and wondering where you picked up the dependency on `commons-logging`, then it is from Spring and specifically from the central module called `spring-core`.

The nice thing about `commons-logging` is that you don't need anything else to make your application work. It has a runtime discovery algorithm that looks for other logging frameworks in well known places on the classpath and uses one that it thinks is appropriate (or you can tell it which one if you need to). If nothing else is available you get pretty nice looking logs just from the JDK (java.util.logging or JUL for short). You should find that your Spring application works and logs happily to the console out of the box in most situations, and that's important.

Not Using Commons Logging

Unfortunately, the runtime discovery algorithm in `commons-logging`, while convenient for the end-user, is problematic. If we could turn back the clock and start Spring now as a new project it would use a different logging dependency. The first choice would probably be the Simple Logging Facade for Java (SLF4J), which is also used by a lot of other tools that people use with Spring inside their applications.

There are basically two ways to switch off `commons-logging`:

1. Exclude the dependency from the `spring-core` module (as it is the only module that explicitly depends on `commons-logging`)
2. Depend on a special `commons-logging` dependency that replaces the library with an empty jar (more details can be found in the SLF4J FAQ)

To exclude commons-logging, add the following to your `dependencyManagement` section:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.3.4.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

Now this application is probably broken because there is no implementation of the JCL API on the classpath, so to fix it a new one has to be provided. In the next section we show you how to provide an alternative implementation of JCL using SLF4J as an example.

Using SLF4J

SLF4J is a cleaner dependency and more efficient at runtime than `commons-logging` because it uses compile-time bindings instead of runtime discovery of the other logging frameworks it integrates. This also means that you have to be more explicit about what you want to happen at runtime, and declare it or configure it accordingly. SLF4J provides bindings to many common logging frameworks, so you can usually choose one that you already use, and bind to that for configuration and management.

SLF4J provides bindings to many common logging frameworks, including JCL, and it also does the reverse: bridges between other logging frameworks and itself. So to use SLF4J with Spring you need to replace the `commons-logging` dependency with the SLF4J-JCL bridge. Once you have done that then logging calls from within Spring will be translated into logging calls to the SLF4J API, so if other libraries in your application use that API, then you have a single place to configure and manage logging.

A common choice might be to bridge Spring to SLF4J, and then provide explicit binding from SLF4J to Log4J. You need to supply 4 dependencies (and exclude the existing `commons-logging`): the bridge, the SLF4J API, the binding to Log4J, and the Log4J implementation itself. In Maven you would do that like this

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
```

```
        <version>4.3.4.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
```

That might seem like a lot of dependencies just to get some logging. Well it is, but it *is* optional, and it should behave better than the vanilla `commons-logging` with respect to classloader issues, notably if you are in a strict container like an OSGi platform. Allegedly there is also a performance benefit because the bindings are at compile-time not runtime.

A more common choice amongst SLF4J users, which uses fewer steps and generates fewer dependencies, is to bind directly to Logback. This removes the extra binding step because Logback implements SLF4J directly, so you only need to depend on two libraries not four (`jcl-over-slf4j` and `logback`). If you do that you might also need to exclude the slf4j-api dependency from other external dependencies (not Spring), because you only want one version of that API on the classpath.

Using Log4J

Many people use Log4j as a logging framework for configuration and management purposes. It's efficient and well-established, and in fact it's what we use at runtime when we build and test Spring. Spring also provides some utilities for configuring and initializing Log4j, so it has an optional compile-time dependency on Log4j in some modules.

To make Log4j work with the default JCL dependency (`commons-logging`) all you need to do is put Log4j on the classpath, and provide it with a configuration file (`log4j.properties` or `log4j.xml` in the root of the classpath). So for Maven users this is your dependency declaration:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.3.4.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
```

And here's a sample log4j.properties for logging to the console:

```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n

log4j.category.org.springframework.beans.factory=DEBUG
```

Runtime Containers with Native JCL

Many people run their Spring applications in a container that itself provides an implementation of JCL. IBM Websphere Application Server (WAS) is the archetype. This often causes problems, and unfortunately there is no silver bullet solution; simply excluding `commons-logging` from your application is not enough in most situations.

To be clear about this: the problems reported are usually not with JCL per se, or even with `commons-logging`: rather they are to do with binding `commons-logging` to another framework (often Log4J). This can fail because `commons-logging` changed the way they do the runtime discovery in between the older versions (1.0) found in some containers and the modern versions that most people use now (1.1). Spring does not use any unusual parts of the JCL API, so

nothing breaks there, but as soon as Spring or your application tries to do any logging you can find that the bindings to Log4J are not working.

In such cases with WAS the easiest thing to do is to invert the class loader hierarchy (IBM calls it "parent last") so that the application controls the JCL dependency, not the container. That option isn't always open, but there are plenty of other suggestions in the public domain for alternative approaches, and your mileage may vary depending on the exact version and feature set of the container.

Part II. What's New in Spring Framework 4.x

3. New Features and Enhancements in Spring Framework 4.0

The Spring Framework was first released in 2004; since then there have been significant major revisions: Spring 2.0 provided XML namespaces and AspectJ support; Spring 2.5 embraced annotation-driven configuration; Spring 3.0 introduced a strong Java 5+ foundation across the framework codebase, and features such as the Java-based `@Configuration` model.

Version 4.0 is the latest major release of the Spring Framework and the first to fully support Java 8 features. You can still use Spring with older versions of Java, however, the minimum requirement has now been raised to Java SE 6. We have also taken the opportunity of a major release to remove many deprecated classes and methods.

A migration guide for upgrading to Spring 4.0 is available on the Spring Framework GitHub Wiki.

3.1 Improved Getting Started Experience

The new spring.io website provides a whole series of "Getting Started" guides to help you learn Spring. You can read more about the guides in the Chapter 1, *Getting Started with Spring* section in this document. The new website also provides a comprehensive overview of the many additional projects that are released under the Spring umbrella.

If you are a Maven user you may also be interested in the helpful bill of materials POM file that is now published with each Spring Framework release.

3.2 Removed Deprecated Packages and Methods

All deprecated packages, and many deprecated classes and methods have been removed with version 4.0. If you are upgrading from a previous release of Spring, you should ensure that you have fixed any deprecated calls that you were making to outdated APIs.

For a complete set of changes, check out the API Differences Report.

Note that optional third-party dependencies have been raised to a 2010/2011 minimum (i.e. Spring 4 generally only supports versions released in late 2010 or later now): notably, Hibernate 3.6+, EhCache 2.1+, Quartz 1.8+, Groovy 1.8+, and Joda-Time 2.0+. As an exception to the rule, Spring 4 requires the recent Hibernate Validator 4.3+, and support for Jackson has been focused on 2.0+ now (with Jackson 1.8/1.9 support retained for the time being where Spring 3.2 had it; now just in deprecated form).

3.3 Java 8 (as well as 6 and 7)

Spring Framework 4.0 provides support for several Java 8 features. You can make use of *lambda expressions* and *method references* with Spring's callback interfaces. There is first-class support for `java.time` (JSR-310), and several existing annotations have been retrofitted as `@Repeatable`. You can also use Java 8's parameter name discovery (based on the `-parameters` compiler flag) as an alternative to compiling your code with debug information enabled.

Spring remains compatible with older versions of Java and the JDK: concretely, Java SE 6 (specifically, a minimum level equivalent to JDK 6 update 18, as released in January 2010) and above are still fully supported. However, for newly started development projects based on Spring 4, we recommend the use of Java 7 or 8.

3.4 Java EE 6 and 7

Java EE version 6 or above is now considered the baseline for Spring Framework 4, with the JPA 2.0 and Servlet 3.0 specifications being of particular relevance. In order to remain compatible with Google App Engine and older application servers, it is possible to deploy a Spring 4 application into a Servlet 2.5 environment. However, Servlet 3.0+ is strongly recommended and a prerequisite in Spring's test and mock packages for test setups in development environments.

If you are a WebSphere 7 user, be sure to install the JPA 2.0 feature pack. On WebLogic 10.3.4 or higher, install the JPA 2.0 patch that comes with it. This turns

> both of those server generations into Spring 4 compatible deployment environments.

On a more forward-looking note, Spring Framework 4.0 supports the Java EE 7 level of applicable specifications now: in particular, JMS 2.0, JTA 1.2, JPA 2.1, Bean Validation 1.1, and JSR-236 Concurrency Utilities. As usual, this support focuses on individual use of those specifications, e.g. on Tomcat or in standalone environments. However, it works equally well when a Spring application is deployed to a Java EE 7 server.

Note that Hibernate 4.3 is a JPA 2.1 provider and therefore only supported as of Spring Framework 4.0. The same applies to Hibernate Validator 5.0 as a Bean Validation 1.1 provider. Neither of the two are officially supported with Spring Framework 3.2.

3.5 Groovy Bean Definition DSL

Beginning with Spring Framework 4.0, it is possible to define external bean configuration using a Groovy DSL. This is similar in concept to using XML bean definitions but allows for a more concise syntax. Using Groovy also allows you to easily embed bean definitions directly in your bootstrap code. For example:

```
def reader = new GroovyBeanDefinitionReader(myApplicationContext)
reader.beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
        settings = [mynew:"setting"]
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

For more information consult the `GroovyBeanDefinitionReader` javadocs.

3.6 Core Container Improvements

There have been several general improvements to the core container:

- Spring now treats *generic types* as a form of *qualifier* when injecting Beans. For example, if you are using a Spring Data `Repository` you can now easily inject a specific implementation: `@Autowired Repository<Customer> customerRepository`.
- If you use Spring's meta-annotation support, you can now develop custom annotations that expose specific attributes from the source annotation.
- Beans can now be *ordered* when they are autowired into lists and arrays. Both the `@Order` annotation and `Ordered` interface are supported.
- The `@Lazy` annotation can now be used on injection points, as well as on `@Bean` definitions.
- The `@Description` annotation has been introduced for developers using Java-based configuration.
- A generalized model for conditionally filtering beans has been added via the `@Conditional` annotation. This is similar to `@Profile` support but allows for user-defined strategies to be developed programmatically.
- CGLIB-based proxy classes no longer require a default constructor. Support is provided via the objenesis library which is repackaged *inline* and distributed as part of the Spring Framework. With this strategy, no constructor at all is being invoked for proxy instances anymore.
- There is managed time zone support across the framework now, e.g. on `LocaleContext`.

3.7 General Web Improvements

Deployment to Servlet 2.5 servers remains an option, but Spring Framework 4.0 is now focused primarily on Servlet 3.0+ environments. If you are using the Spring MVC Test Framework you will need to ensure that a Servlet 3.0 compatible JAR is in your *test classpath*.

In addition to the WebSocket support mentioned later, the following general improvements have been made to Spring's Web modules:

- You can use the new `@RestController` annotation with Spring MVC applications, removing the need to add `@ResponseBody` to each of your `@RequestMapping` methods.
- The `AsyncRestTemplate` class has been added, allowing non-blocking asynchronous support when developing REST clients.
- Spring now offers comprehensive timezone support when developing Spring MVC applications.

3.8 WebSocket, SockJS, and STOMP Messaging

A new `spring-websocket` module provides comprehensive support for WebSocket-based, two-way communication between client and server in web applications. It is compatible with JSR-356, the Java WebSocket API, and in addition provides SockJS-based fallback options (i.e. WebSocket emulation) for use in browsers that don't yet support the WebSocket protocol (e.g. Internet Explorer < 10).

A new `spring-messaging` module adds support for STOMP as the WebSocket sub-protocol to use in applications along with an annotation programming model for routing and processing STOMP messages from WebSocket clients. As a result an `@Controller` can now contain both `@RequestMapping` and `@MessageMapping` methods for handling HTTP requests and messages from WebSocket-connected clients. The new `spring-messaging` module also contains key abstractions formerly from the Spring Integration project such as `Message`, `MessageChannel`, `MessageHandler`, and others to serve as a foundation for messaging-based applications.

For further details, including a more thorough introduction, see the Chapter 26, *WebSocket Support* section.

3.9 Testing Improvements

In addition to pruning of deprecated code within the `spring-test` module, Spring Framework 4.0 introduces several new features for use in unit and integration testing.

- Almost all annotations in the `spring-test` module (e.g., `@ContextConfiguration`, `@WebAppConfiguration`, `@ContextHierarchy`, `@ActiveProfiles`, etc.) can now be used as meta-annotations to create custom *composed annotations* and reduce configuration duplication across a test suite.
- Active bean definition profiles can now be resolved programmatically, simply by implementing a custom `ActiveProfilesResolver` and registering it via the `resolver` attribute of `@ActiveProfiles`.
- A new `SocketUtils` class has been introduced in the `spring-core` module which enables you to scan for free TCP and UDP server ports on localhost. This functionality is not specific to testing but can prove very useful when writing integration tests that require the use of sockets, for example tests that start an in-memory SMTP server, FTP server, Servlet container, etc.
- As of Spring 4.0, the set of mocks in the `org.springframework.mock.web` package is now based on the Servlet 3.0 API. Furthermore, several of the Servlet API mocks (e.g., `MockHttpServletRequest`, `MockServletContext`, etc.) have been updated with minor enhancements and improved configurability.

4. New Features and Enhancements in Spring Framework 4.1

4.1 JMS Improvements

Spring 4.1 introduces a much simpler infrastructure to register JMS listener endpoints by annotating bean methods with `@JmsListener`. The XML namespace has been enhanced to support this new style (`jms:annotation-driven`), and it is also possible to fully configure the infrastructure using Java config (`@EnableJms`, `JmsListenerContainerFactory`). It is also possible to register listener endpoints programmatically using `JmsListenerConfigurer`.

Spring 4.1 also aligns its JMS support to allow you to benefit from the `spring-messaging` abstraction introduced in 4.0, that is:

- Message listener endpoints can have a more flexible signature and benefit from standard messaging annotations such as `@Payload`, `@Header`, `@Headers`, and `@SendTo`. It is also possible to use a standard `Message` in lieu of `javax.jms.Message` as method argument.
- A new `JmsMessageOperations` interface is available and permits `JmsTemplate` like operations using the `Message` abstraction.

Finally, Spring 4.1 provides additional miscellaneous improvements:

- Synchronous request-reply operations support in `JmsTemplate`
- Listener priority can be specified per `<jms:listener/>` element
- Recovery options for the message listener container are configurable using a `BackOff` implementation
- JMS 2.0 shared consumers are supported

4.2 Caching Improvements

Spring 4.1 supports JCache (JSR-107) annotations using Spring's existing cache configuration and infrastructure abstraction; no changes are required to use the standard annotations.

Spring 4.1 also improves its own caching abstraction significantly:

- Caches can be resolved at runtime using a `CacheResolver`. As a result the `value` argument defining the cache name(s) to use is no longer mandatory.
- More operation-level customizations: cache resolver, cache manager, key generator

- A new `@CacheConfig` class-level annotation allows common settings to be shared at the class level **without** enabling any cache operation.
- Better exception handling of cached methods using `CacheErrorHandler`

Spring 4.1 also has a breaking change in the `Cache` interface as a new `putIfAbsent` method has been added.

4.3 Web Improvements

- The existing support for resource handling based on the `ResourceHttpRequestHandler` has been expanded with new abstractions `ResourceResolver`, `ResourceTransformer`, and `ResourceUrlProvider`. A number of built-in implementations provide support for versioned resource URLs (for effective HTTP caching), locating gzipped resources, generating an HTML 5 AppCache manifests, and more. See Section 22.16.9, "Serving of Resources".
- JDK 1.8's `java.util.Optional` is now supported for `@RequestParam`, `@RequestHeader`, and `@MatrixVariable` controller method arguments.
- `ListenableFuture` is supported as a return value alternative to `DeferredResult` where an underlying service (or perhaps a call to `AsyncRestTemplate`) already returns `ListenableFuture`.
- `@ModelAttribute` methods are now invoked in an order that respects inter-dependencies. See SPR-6299.
- Jackson's `@JsonView` is supported directly on `@ResponseBody` and `ResponseEntity` controller methods for serializing different amounts of detail for the same POJO (e.g. summary vs. detail page). This is also supported with View-based rendering by adding the serialization view type as a model attribute under a special key. See the section called "Jackson Serialization View Support" for details.
- JSONP is now supported with Jackson. See the section called "Jackson JSONP Support".
- A new lifecycle option is available for intercepting `@ResponseBody` and `ResponseEntity` methods just after the controller method returns and before the response is written. To take advantage declare an `@ControllerAdvice` bean that implements `ResponseBodyAdvice`. The built-in support for `@JsonView` and JSONP take advantage of this. See Section 22.4.1, "Intercepting requests with a HandlerInterceptor".
- There are three new `HttpMessageConverter` options:
  - Gson — lighter footprint than Jackson; has already been in use in Spring Android.
  - Google Protocol Buffers — efficient and effective as an inter-service communication data protocol within an enterprise but can also be exposed as JSON and XML for browsers.
  - Jackson based XML serialization is now supported through the jackson-dataformat-xml extension. When using `@EnableWebMvc` or `<mvc:annotation-driven/>`, this is used

by default instead of JAXB2 if `jackson-dataformat-xml` is in the classpath.

- Views such as JSPs can now build links to controllers by referring to controller mappings by name. A default name is assigned to every `@RequestMapping`. For example `FooController` with method `handleFoo` is named "FC#handleFoo". The naming strategy is pluggable. It is also possible to name an `@RequestMapping` explicitly through its name attribute. A new `mvcUrl` function in the Spring JSP tag library makes this easy to use in JSP pages. See Section 22.7.2, “Building URIs to Controllers and methods from views”.
- `ResponseEntity` provides a builder-style API to guide controller methods towards the preparation of server-side responses, e.g. `ResponseEntity.ok()`.
- `RequestEntity` is a new type that provides a builder-style API to guide client-side REST code towards the preparation of HTTP requests.
- MVC Java config and XML namespace:
  - View resolvers can now be configured including support for content negotiation, see Section 22.16.8, “View Resolvers”.
  - View controllers now have built-in support for redirects and for setting the response status. An application can use this to configure redirect URLs, render 404 responses with a view, send "no content" responses, etc. Some use cases are listed here.
  - Path matching customizations are frequently used and now built-in. See Section 22.16.11, “Path Matching”.
- Groovy markup template support (based on Groovy 2.3). See the `GroovyMarkupConfigurer` and respecitve `ViewResolver` and `View' implementations.

4.4 WebSocket Messaging Improvements

- SockJS (Java) client-side support. See `SockJsClient` and classes in same package.
- New application context events `SessionSubscribeEvent` and `SessionUnsubscribeEvent` published when STOMP clients subscribe and unsubscribe.
- New "websocket" scope. See Section 26.4.14, “WebSocket Scope”.
- `@SendToUser` can target only a single session and does not require an authenticated user.
- `@MessageMapping` methods can use dot "." instead of slash "/" as path separator. See SPR-11660.
- STOMP/WebSocket monitoring info collected and logged. See Section 26.4.16, “Runtime Monitoring”.
- Significantly optimized and improved logging that should remain very readable and compact even at DEBUG level.
- Optimized message creation including support for temporary message mutability and avoiding automatic message id and timestamp creation. See Javadoc of `MessageHeaderAccessor`.

- Close STOMP/WebSocket connections that have no activity within 60 seconds after the WebSocket session is established. See SPR-11884.

4.5 Testing Improvements

- Groovy scripts can now be used to configure the `ApplicationContext` loaded for integration tests in the TestContext framework.
  - See the section called "Context configuration with Groovy scripts" for details.
- Test-managed transactions can now be programmatically started and ended within transactional test methods via the new `TestTransaction` API.
  - See the section called "Programmatic transaction management" for details.
- SQL script execution can now be configured declaratively via the new `@Sql` and `@SqlConfig` annotations on a per-class or per-method basis.
  - See Section 15.5.8, "Executing SQL scripts" for details.
- Test property sources which automatically override system and application property sources can be configured via the new `@TestPropertySource` annotation.
  - See the section called "Context configuration with test property sources" for details.
- Default `TestExecutionListener`s can now be automatically discovered.
  - See the section called "Automatic discovery of default TestExecutionListeners" for details.
- Custom `TestExecutionListener`s can now be automatically merged with the default listeners.
  - See the section called "Merging TestExecutionListeners" for details.
- The documentation for transactional testing support in the TestContext framework has been improved with more thorough explanations and additional examples.
  - See Section 15.5.7, "Transaction management" for details.
- Various improvements to `MockServletContext`, `MockHttpServletRequest`, and other Servlet API mocks.
- `AssertThrows` has been refactored to support `Throwable` instead of `Exception`.
- In Spring MVC Test, JSON responses can be asserted with JSON Assert as an extra option to using JSONPath much like it has been possible to do for XML with XMLUnit.
- `MockMvcBuilder` *recipes* can now be created with the help of `MockMvcConfigurer`. This was added to make it easy to apply Spring Security setup but can be used to encapsulate common setup for any 3rd party framework or within a project.
- `MockRestServiceServer` now supports the `AsyncRestTemplate` for client-side testing.

5. New Features and Enhancements in Spring Framework 4.2

5.1 Core Container Improvements

- Annotations such as `@Bean` get detected and processed on Java 8 default methods as well, allowing for composing a configuration class from interfaces with default `@Bean` methods.
- Configuration classes may declare `@Import` with regular component classes now, allowing for a mix of imported configuration classes and component classes.
- Configuration classes may declare an `@Order` value, getting processed in a corresponding order (e.g. for overriding beans by name) even when detected through classpath scanning.
- `@Resource` injection points support an `@Lazy` declaration, analogous to `@Autowired`, receiving a lazy-initializing proxy for the requested target bean.
- The application event infrastructure now offers an annotation-based model as well as the ability to publish any arbitrary event.
  - Any public method in a managed bean can be annotated with `@EventListener` to consume events.
  - `@TransactionalEventListener` provides transaction-bound event support.
- Spring Framework 4.2 introduces first-class support for declaring and looking up aliases for annotation attributes. The new `@AliasFor` annotation can be used to declare a pair of aliased attributes within a single annotation or to declare an alias from one attribute in a custom composed annotation to an attribute in a meta-annotation.
  - The following annotations have been retrofitted with `@AliasFor` support in order to provide meaningful aliases for their `value` attributes: `@Cacheable`, `@CacheEvict`, `@CachePut`, `@ComponentScan`, `@ComponentScan.Filter`, `@ImportResource`, `@Scope`, `@ManagedResource`, `@Header`, `@Payload`, `@SendToUser`, `@ActiveProfiles`, `@ContextConfiguration`, `@Sql`, `@TestExecutionListeners`, `@TestPropertySource`, `@Transactional`, `@ControllerAdvice`, `@CookieValue`, `@CrossOrigin`, `@MatrixVariable`, `@RequestHeader`, `@RequestMapping`, `@RequestParam`, `@RequestPart`, `@ResponseStatus`, `@SessionAttributes`, `@ActionMapping`, `@RenderMapping`, `@EventListener`, `@TransactionalEventListener`.
  - For example, `@ContextConfiguration` from the `spring-test` module is now declared as follows:

```
public @interface ContextConfiguration {

    @AliasFor("locations")
```

```
    String[] value() default {};

    @AliasFor("value")
    String[] locations() default {};

    // ...
}
```

- Similarly, *composed annotations* that override attributes from meta-annotations can now use `@AliasFor` for fine-grained control over exactly which attributes are overridden within an annotation hierarchy. In fact, it is now possible to declare an alias for the `value` attribute of a meta-annotation.
- For example, one can now develop a composed annotation with a custom attribute override as follows.

```
@ContextConfiguration
public @interface MyTestConfig {

    @AliasFor(annotation = ContextConfiguration.class, attribute = "value")
    String[] xmlFiles();

    // ...
}
```

- See Spring Annotation Programming Model.

- Numerous improvements to Spring's search algorithms used for finding meta-annotations. For example, locally declared *composed annotations* are now favored over inherited annotations.
- *Composed annotations* that override attributes from meta-annotations can now be discovered on interfaces and on abstract, bridge, & interface methods as well as on classes, standard methods, constructors, and fields.
- Maps representing annotation attributes (and `AnnotationAttributes` instances) can be *synthesized* (i.e., converted) into an annotation.
- The features of field-based data binding (`DirectFieldAccessor`) have been aligned with the current property-based data binding (`BeanWrapper`). In particular, field-based binding now supports navigation for Collections, Arrays, and Maps.
- `DefaultConversionService` now provides out-of-the-box converters for `Stream`, `Charset`, `Currency`, and `TimeZone`. Such converters can be added individually to any arbitrary `ConversionService` as well.
- `DefaultFormattingConversionService` comes with out-of-the-box support for the value types in JSR-354 Money & Currency (if the 'javax.money' API is present on the classpath):

namely, `MonetaryAmount` and `CurrencyUnit`. This includes support for applying `@NumberFormat`.
- `@NumberFormat` can now be used as a meta-annotation.
- `JavaMailSenderImpl` has a new `testConnection()` method for checking connectivity to the server.
- `ScheduledTaskRegistrar` exposes scheduled tasks.
- Apache `commons-pool2` is now supported for a pooling AOP `CommonsPool2TargetSource`.
- Introduced `StandardScriptFactory` as a JSR-223 based mechanism for scripted beans, exposed through the `lang:std` element in XML. Supports e.g. JavaScript and JRuby. (Note: JRubyScriptFactory and `lang:jruby` are deprecated now, in favor of using JSR-223.)

5.2 Data Access Improvements

- `javax.transaction.Transactional` is now supported via AspectJ.
- `SimpleJdbcCallOperations` now supports named binding.
- Full support for Hibernate ORM 5.0: as a JPA provider (automatically adapted) as well as through its native API (covered by the new `org.springframework.orm.hibernate5` package).
- Embedded databases can now be automatically assigned unique names, and `<jdbc:embedded-database>` supports a new `database-name` attribute. See "Testing Improvements" below for further details.

5.3 JMS Improvements

- The `autoStartup` attribute can be controlled via `JmsListenerContainerFactory`.
- The type of the reply `Destination` can now be configured per listener container.
- The value of the `@SendTo` annotation can now use a SpEL expression.
- The response destination can be computed at runtime using `JmsResponse`
- `@JmsListener` is now a repeatable annotation to declare several JMS containers on the same method (use the newly introduced `@JmsListeners` if you're not using Java8 yet).

5.4 Web Improvements

- HTTP Streaming and Server-Sent Events support, see the section called "HTTP Streaming".
- Built-in support for CORS including global (MVC Java config and XML namespace) and local (e.g. `@CrossOrigin`) configuration. See Chapter 27, *CORS Support* for details.

- HTTP caching updates:
  - new `CacheControl` builder; plugged into `ResponseEntity`, `WebContentGenerator`, `ResourceHttpRequestHandler`.
  - improved ETag/Last-Modified support in `WebRequest`.
- Custom mapping annotations, using `@RequestMapping` as a meta-annotation.
- Public methods in `AbstractHandlerMethodMapping` to register and unregister request mappings at runtime.
- Protected `createDispatcherServlet` method in `AbstractDispatcherServletInitializer` to further customize the `DispatcherServlet` instance to use.
- `HandlerMethod` as a method argument on `@ExceptionHandler` methods, especially handy in `@ControllerAdvice` components.
- `java.util.concurrent.CompletableFuture` as an `@Controller` method return value type.
- Byte-range request support in `HttpHeaders` and for serving static resources.
- `@ResponseStatus` detected on nested exceptions.
- `UriTemplateHandler` extension point in the `RestTemplate`.
  - `DefaultUriTemplateHandler` exposes `baseUrl` property and path segment encoding options.
  - the extension point can also be used to plug in any URI template library.
- OkHTTP integration with the `RestTemplate`.
- Custom `baseUrl` alternative for methods in `MvcUriComponentsBuilder`.
- Serialization/deserialization exception messages are now logged at WARN level.
- Default JSON prefix has been changed from "{} && " to the safer ")]}', " one.
- New `RequestBodyAdvice` extension point and built-in implementation to support Jackson's `@JsonView` on `@RequestBody` method arguments.
- When using GSON or Jackson 2.6+, the handler method return type is used to improve serialization of parameterized types like `List<Foo>`.
- Introduced `ScriptTemplateView` as a JSR-223 based mechanism for scripted web views, with a focus on JavaScript view templating on Nashorn (JDK 8).

5.5 WebSocket Messaging Improvements

- Expose presence information about connected users and subscriptions:
  - new `SimpUserRegistry` exposed as a bean named "userRegistry".
  - sharing of presence information across cluster of servers (see broker relay config options).
- Resolve user destinations across cluster of servers (see broker relay config options).

- `StompSubProtocolErrorHandler` extension point to customize and control STOMP ERROR frames to clients.
- Global `@MessageExceptionHandler` methods via `@ControllerAdvice` components.
- Heart-beats and a SpEL expression 'selector' header for subscriptions with `SimpleBrokerMessageHandler`.
- STOMP client for use over TCP and WebSocket; see Section 26.4.13, “STOMP Client”.
- `@SendTo` and `@SendToUser` can contain destination variable placeholders.
- Jackson’s `@JsonView` supported for return values on `@MessageMapping` and `@SubscribeMapping` methods.
- `ListenableFuture` and `CompletableFuture` as return value types from `@MessageMapping` and `@SubscribeMapping` methods.
- `MarshallingMessageConverter` for XML payloads.

5.6 Testing Improvements

- JUnit-based integration tests can now be executed with JUnit rules instead of the `SpringJUnit4ClassRunner`. This allows Spring-based integration tests to be run with alternative runners like JUnit’s `Parameterized` or third-party runners such as the `MockitoJUnitRunner`.
  - See the section called “Spring JUnit 4 Rules” for details.
- The Spring MVC Test framework now provides first-class support for HtmlUnit, including integration with Selenium’s WebDriver, allowing for page-based web application testing without the need to deploy to a Servlet container.
  - See Section 15.6.2, “HtmlUnit Integration” for details.
- `AopTestUtils` is a new testing utility that allows developers to obtain a reference to the underlying target object hidden behind one or more Spring proxies.
  - See Section 14.2.1, “General testing utilities” for details.
- `ReflectionTestUtils` now supports setting and getting `static` fields, including constants.
- The original ordering of bean definition profiles declared via `@ActiveProfiles` is now retained in order to support use cases such as Spring Boot’s `ConfigFileApplicationListener` which loads configuration files based on the names of active profiles.
- `@DirtiesContext` supports new `BEFORE_METHOD`, `BEFORE_CLASS`, and `BEFORE_EACH_TEST_METHOD` modes for closing the `ApplicationContext` *before* a test — for example, if some rogue (i.e., yet to be determined) test within a large test suite has corrupted the original configuration for the `ApplicationContext`.

- `@Commit` is a new annotation that may be used as a direct replacement for `@Rollback(false)`.
- `@Rollback` may now be used to configure class-level *default rollback* semantics.
  - Consequently, `@TransactionConfiguration` is now deprecated and will be removed in a subsequent release.
- `@Sql` now supports execution of *inlined SQL statements* via a new `statements` attribute.
- The `ContextCache` that is used for caching `ApplicationContext`s between tests is now a public API with a default implementation that can be replaced for custom caching needs.
- `DefaultTestContext`, `DefaultBootstrapContext`, and `DefaultCacheAwareContextLoaderDelegate` are now public classes in the `support` subpackage, allowing for custom extensions.
- `TestContextBootstrapper`s are now responsible for building the `TestContext`.
- In the Spring MVC Test framework, `MvcResult` details can now be logged at `DEBUG` level or written to a custom `OutputStream` or `Writer`. See the new `log()`, `print(OutputStream)`, and `print(Writer)` methods in `MockMvcResultHandlers` for details.
- The JDBC XML namespace supports a new `database-name` attribute in `<jdbc:embedded-database>`, allowing developers to set unique names for embedded databases –- for example, via a SpEL expression or a property placeholder that is influenced by the current active bean definition profiles.
- Embedded databases can now be automatically assigned a unique name, allowing common test database configuration to be reused in different `ApplicationContext`s within a test suite.
  - See Section 19.8.6, “Generating unique names for embedded databases” for details.
- `MockHttpServletRequest` and `MockHttpServletResponse` now provide better support for date header formatting via the `getDateHeader` and `setDateHeader` methods.

6. New Features and Enhancements in Spring Framework 4.3

6.1 Core Container Improvements

- Core container exceptions provide richer metadata to evaluate programmatically.
- Java 8 default methods get detected as bean property getters/setters.
- Lazy candidate beans are not being created in case of injecting a primary bean.
- It is no longer necessary to specify the `@Autowired` annotation if the target bean only defines one constructor.

- `@Configuration` classes support constructor injection.
- Any SpEL expression used to specify the `condition` of an `@EventListener` can now refer to beans (e.g. `@beanName.method()`).
- *Composed annotations* can now override array attributes in meta-annotations with a single element of the component type of the array. For example, the `String[] path` attribute of `@RequestMapping` can be overridden with `String path` in a composed annotation.
- `@PersistenceContext`/`@PersistenceUnit` selects a primary `EntityManagerFactory` bean if declared as such.
- `@Scheduled` and `@Schedules` may now be used as *meta-annotations* to create custom *composed annotations* with attribute overrides.
- `@Scheduled` is properly supported on beans of any scope.

6.2 Data Access Improvements

- `jdbc:initialize-database` and `jdbc:embedded-database` support a configurable separator to be applied to each script.

6.3 Caching Improvements

Spring 4.3 allows concurrent calls on a given key to be synchronized so that the value is only computed once. This is an opt-in feature that should be enabled via the new `sync` attribute on `@Cacheable`. This features introduces a breaking change in the `Cache` interface as a `get(Object key, Callable<T> valueLoader)` method has been added.

Spring 4.3 also improves the caching abstraction as follows:

- SpEL expressions in caches-related annotations can now refer to beans (i.e. `@beanName.method()`).
- `ConcurrentMapCacheManager` and `ConcurrentMapCache` now support the serialization of cache entries via a new `storeByValue` attribute.
- `@Cacheable`, `@CacheEvict`, `@CachePut`, and `@Caching` may now be used as *meta-annotations* to create custom *composed annotations* with attribute overrides.

6.4 JMS Improvements

- `@SendTo` can now be specified at the class level to share a common reply destination.
- `@JmsListener` and `@JmsListeners` may now be used as *meta-annotations* to create custom *composed annotations* with attribute overrides.

6.5 Web Improvements

- Built-in support for HTTP HEAD and HTTP OPTIONS.
- New `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, and `@PatchMapping` *composed annotations* for `@RequestMapping`.
  - See Composed @RequestMapping Variants for details.
- New `@RequestScope`, `@SessionScope`, and `@ApplicationScope` *composed annotations* for web scopes.
  - See Request scope, Session scope, and Application scope for details.
- New `@RestControllerAdvice` annotation with combined `@ControllerAdvice` with `@ResponseBody` semantics.
- `@ResponseStatus` is now supported at the class level and inherited by all methods.
- New `@SessionAttribute` annotation for access to session attributes (see example).
- New `@RequestAttribute` annotation for access to request attributes (see example).
- `@ModelAttribute` allows preventing data binding via `binding=false` attribute (see reference).
- `@PathVariable` may be declared as optional (for use on `@ModelAttribute` methods).
- Consistent exposure of Errors and custom Throwables to MVC exception handlers.
- Consistent charset handling in HTTP message converters, including a UTF-8 default for multipart text content.
- Static resource handling uses the configured `ContentNegotiationManager` for media type determination.
- `RestTemplate` and `AsyncRestTemplate` support strict URI variable encoding via `DefaultUriTemplateHandler`.
- `AsyncRestTemplate` supports request interception.

6.6 WebSocket Messaging Improvements

- `@SendTo` and `@SendToUser` can now be specified at class-level to share a common destination.

6.7 Testing Improvements

- The JUnit support in the *Spring TestContext Framework* now requires JUnit 4.12 or higher.
- New `SpringRunner` *alias* for the `SpringJUnit4ClassRunner`.
- Test related annotations may now be declared on interfaces — for example, for use with *test interfaces* that make use of Java 8 based interface default methods.

- An empty declaration of `@ContextConfiguration` can now be completely omitted if default XML files, Groovy scripts, or `@Configuration` classes are detected.
- `@Transactional` test methods are no longer required to be `public` (e.g., in TestNG and JUnit 5).
- `@BeforeTransaction` and `@AfterTransaction` methods are no longer required to be `public` and may now be declared on Java 8 based interface default methods.
- The `ApplicationContext` cache in the *Spring TestContext Framework* is now bounded with a default maximum size of 32 and a *least recently used* eviction policy. The maximum size can be configured by setting a JVM system property or Spring property called `spring.test.context.cache.maxSize`.
- New `ContextCustomizer` API for customizing a test `ApplicationContext` *after* bean definitions have been loaded into the context but *before* the context has been refreshed. Customizers can be registered globally by third parties, foregoing the need to implement a custom `ContextLoader`.
- `@Sql` and `@SqlGroup` may now be used as *meta-annotations* to create custom *composed annotations* with attribute overrides.
- `ReflectionTestUtils` now automatically unwraps proxies when setting or getting a field.
- Server-side Spring MVC Test supports expectations on response headers with multiple values.
- Server-side Spring MVC Test parses form data request content and populates request parameters.
- Server-side Spring MVC Test supports mock-like assertions for invoked handler methods.
- Client-side REST test support allows indicating how many times a request is expected and whether the order of declaration for expectations should be ignored (see Section 15.6.3, “Client-Side REST Tests”).
- Client-side REST Test supports expectations for form data in the request body.

6.8 Support for new library and server generations

- Hibernate ORM 5.2 (still supporting 4.2/4.3 and 5.0/5.1 as well, with 3.6 deprecated now)
- Hibernate Validator 5.3 (minimum remains at 4.3)
- Jackson 2.8 (minimum raised to Jackson 2.6+ as of Spring 4.3)
- OkHttp 3.x (still supporting OkHttp 2.x side by side)
- Tomcat 8.5 as well as 9.0 milestones
- Netty 4.1
- Undertow 1.4
- WildFly 10.1

Furthermore, Spring Framework 4.3 embeds the updated ASM 5.1, CGLIB 3.2.4, and Objenesis 2.4 in `spring-core.jar`.

Part III. 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.

7. The IoC container

7.1 Introduction to the Spring IoC container and beans

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) [1] 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` 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; 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 Section 7.16, "The 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.

7.2 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` or `FileSystemXmlApplicationContext`. 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 Section 7.15.4, “Convenient ApplicationContext instantiation for web applications”). If you are using the Spring Tool Suite 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.

Figure 7.1. The Spring IoC container

7.2.1 Configuration metadata

As the preceding diagram shows, 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 is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.

> XML-based metadata is *not* the only allowed form of configuration metadata. The Spring IoC container itself is *totally* decoupled from the format in which this configuration metadata is actually written. These days many developers choose Java-based configuration for their Spring applications.

For information about using other forms of metadata with the Spring container, see:

- Annotation-based configuration: Spring 2.5 introduced support for annotation-based configuration metadata.
- Java-based configuration: Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework. Thus you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the `@Configuration`, `@Bean`, `@Import` and `@DependsOn` annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata shows these beans configured as `<bean/>` elements inside a top-level `<beans/>` element. Java configuration typically uses `@Bean` annotated methods within a `@Configuration` class.

These bean definitions correspond to the actual objects that make up your application. 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. However, you can use Spring's integration with AspectJ to configure objects that have been created outside the control of an IoC container. See Using AspectJ to dependency-inject domain objects with Spring.

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
        http://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 you use to identify the individual bean definition. The `class` attribute defines the type of the bean and uses the fully qualified classname. The value of the id attribute refers to collaborating objects. The XML for referring to collaborating objects is not shown in this example; see Dependencies for more information.

7.2.2 Instantiating a container

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an `ApplicationContext` constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java `CLASSPATH`, and so on.

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});
```

After you learn about Spring's IoC container, you may want to know more about Spring's `Resource` abstraction, as described in Chapter 8, *Resources*, which provides a convenient mechanism for reading an InputStream from locations defined in a URI syntax. In particular, `Resource` paths are used to construct applications contexts as described in Section 8.7, "Application contexts and Resource paths".

The following example shows the service layer objects `(services.xml)` configuration file:

```
<?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
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- services -->

    <bean id="petStore" class="org.springframework.samples.jpetstore.services.PetS
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

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

</beans>
```

The following example shows the data access objects `daos.xml` file:

```
<?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
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="accountDao"
        class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaIte
```

```
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for data access objects go here -->

</beans>
```

In the preceding example, the service layer consists of the class `PetStoreServiceImpl`, and two data access objects of the type `JpaAccountDao` and `JpaItemDao` (based on the JPA Object/Relational mapping standard). The `property name` element refers to the name of the JavaBean property, and the `ref` element refers to the name of another bean definition. This linkage between `id` and `ref` elements expresses the dependency between collaborating objects. For details of configuring an object's dependencies, see Dependencies.

Composing XML-based configuration metadata

It can be useful to have bean definitions span multiple XML files. Often each individual XML configuration file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple `Resource` locations, as was shown in the previous section. Alternatively, use one or more occurrences of the `<import/>` element to load bean definitions from another file or files. For example:

```
<beans>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

In the preceding example, external bean definitions are loaded from three files: `services.xml`, `messageSource.xml`, and `themeSource.xml`. All location paths are relative to the definition file doing the importing, so `services.xml` must be in the same directory or classpath location as the file doing the importing, while `messageSource.xml` and `themeSource.xml` must be in a `resources` location below the location of the importing file. As you can see, a leading slash is ignored, but given that these paths are relative, it is better form not to use the slash at all. The contents of the files being imported, including the top level `<beans/>` element, must be valid XML bean definitions according to the Spring Schema.

It is possible, but not recommended, to reference files in parent directories using a relative "../" path. Doing so creates a dependency on a file that is outside the current application. In particular, this reference is not recommended for "classpath:" URLs (for example, "classpath:../services.xml"), where the runtime resolution process chooses the "nearest" classpath root and then looks into its parent directory. Classpath configuration changes may lead to the choice of a different, incorrect directory.
You can always use fully qualified resource locations instead of relative paths: for example, "file:C:/config/services.xml" or "classpath:/config/services.xml". However, be aware that you are coupling your application's configuration to specific absolute locations. It is generally preferable to keep an indirection for such absolute locations, for example, through "${…}" placeholders that are resolved against JVM system properties at runtime.

7.2.3 Using the container

The `ApplicationContext` is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. Using the method `T getBean(String name, Class<T> requiredType)` you can retrieve instances of your beans.

The `ApplicationContext` enables you to read bean definitions and access them as follows:

```
// create and configure beans
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

// retrieve configured instance
PetStoreService service = context.getBean("petStore", PetStoreService.class);

// use configured instance
List<String> userList = service.getUsernameList();
```

You use `getBean()` to retrieve instances of your beans. The `ApplicationContext` interface has a few other methods for retrieving beans, but ideally your application code should never use them. Indeed, your application code should have no calls to the `getBean()` method at all, and thus no dependency on Spring APIs at all. For example, Spring's integration with web frameworks provides for dependency injection for various web framework classes such as controllers and JSF-managed beans.

7.3 Bean overview

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 class name:* typically the actual implementation class of the bean being defined.
- Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
- References to other beans that are needed for the bean to do its work; these references are also called *collaborators* or *dependencies*.
- Other configuration settings to set in the newly created object, for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool.

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

Table 7.1. The bean definition

Property	Explained in…
class	Section 7.3.2, “Instantiating beans”
name	Section 7.3.1, “Naming beans”
scope	Section 7.5, “Bean scopes”
constructor arguments	Section 7.4.1, “Dependency Injection”
properties	Section 7.4.1, “Dependency Injection”
autowiring mode	Section 7.4.5, “Autowiring collaborators”
lazy-initialization mode	Section 7.4.4, “Lazy-initialized beans”
initialization method	the section called “Initialization callbacks”
destruction method	the section called “Destruction callbacks”

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 via the method `getBeanFactory()` which returns the BeanFactory implementation `DefaultListableBeanFactory`. `DefaultListableBeanFactory` supports this registration through the methods `registerSingleton(..)` and `registerBeanDefinition(..)`. However, typical applications work solely with beans defined through metadata bean definitions.

> 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. While overriding of existing metadata and existing singleton instances is supported to some degree, the registration of new beans at runtime (concurrently with live access to factory) is not officially supported and may lead to concurrent access exceptions and/or inconsistent state in the bean container.

7.3.1 Naming beans

Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier, but if it requires more than one, the extra ones can be considered aliases.

In XML-based configuration metadata, you use the `id` and/or `name` attributes to specify the bean identifier(s). The `id` attribute allows you to specify exactly one id. Conventionally these names are alphanumeric ('myBean', 'fooService', etc.), but may contain special characters as well. If you want to introduce other aliases to the bean, you can also specify them in the `name` attribute, separated by a comma (`,`), semicolon (`;`), or white space. As a historical note, in versions prior to Spring 3.1, the `id` attribute was defined as an `xsd:ID` type, which constrained possible characters. As of 3.1, it is defined as an `xsd:string` type. Note that bean `id` uniqueness is still enforced by the container, though no longer by XML parsers.

You are not required to supply a name or id for a bean. If no name or id is supplied explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name, through the use of the `ref` element or Service Locator style lookup, you must provide a name. Motivations for not supplying a name are related to using inner beans and autowiring collaborators.

Bean Naming Conventions

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter, and are camel-cased from then on. Examples of such names would be (without quotes) `'accountManager'`, `'accountService'`, `'userDao'`, `'loginController'`, and so forth.

Naming beans consistently makes your configuration easier to read and understand, and if you are using Spring AOP it helps a lot when applying advice to a set of beans related by name.

With component scanning in the classpath, Spring generates bean names for unnamed components, following the rules above: essentially, taking the simple class name and turning its initial character to lower-case. However, in the (unusual) special case when there is more than one character and both the first and second characters are upper case, the original casing gets preserved. These are the same rules as defined by `java.beans.Introspector.decapitalize` (which Spring is using here).

Aliasing a bean outside the bean definition

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. These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.

Specifying all aliases where the bean is actually defined is not always adequate, however. It is sometimes desirable to introduce an alias for a bean that is defined elsewhere. This is commonly the case in large systems where configuration is split amongst each subsystem, each subsystem having its own set of object definitions. In XML-based configuration metadata, you can use the `<alias/>` element to accomplish this.

```
<alias name="fromName" alias="toName"/>
```

In this case, a bean in the same container which is named `fromName`, may also, after the use of this alias definition, be referred to as `toName`.

For example, the configuration metadata for subsystem A may refer to a DataSource via the name `subsystemA-dataSource`. The configuration metadata for subsystem B may refer to a DataSource via the name `subsystemB-dataSource`. When composing the main application that uses both these subsystems the main application refers to the DataSource via the name

`myApp-dataSource`. To have all three names refer to the same object you add to the MyApp configuration metadata the following aliases definitions:

```
<alias name="subsystemA-dataSource" alias="subsystemB-dataSource"/>
<alias name="subsystemA-dataSource" alias="myApp-dataSource" />
```

Now each component and the main application can refer to the dataSource through a name that is unique and guaranteed not to clash with any other definition (effectively creating a namespace), yet they refer to the same bean.

Java-configuration

If you are using Java-configuration, the `@Bean` annotation can be used to provide aliases see Section 7.12.3, "Using the @Bean annotation" for details.

7.3.2 Instantiating beans

A bean definition essentially is 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 use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the `class` attribute of the `<bean/>` element. This `class` attribute, which internally is a `Class` property on a `BeanDefinition` instance, is usually mandatory. (For exceptions, see the section called "Instantiation using an instance factory method" and Section 7.7, "Bean definition inheritance".) You use the `Class` property in one of two ways:

- Typically, to specify the bean class to be constructed in the case where the container itself directly creates the bean by calling its constructor reflectively, somewhat equivalent to Java code using the `new` operator.
- To specify the actual class containing the `static` factory method that will be invoked to create the object, in the less common case where the container invokes a `static` *factory* method on a class to create the bean. The object type returned from the invocation of the `static` factory method may be the same class or another class entirely.

Inner class names. 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 `Foo` in the `com.example` package, and this `Foo` class has a `static` nested class called `Bar`, the value of the `'class'` attribute on a bean definition would be…

`com.example.Foo$Bar`

Notice the use of the `$` character in the name to separate the nested class name from the outer class name.

Instantiation with a constructor

When you create a bean by the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being developed does not need to implement any specific interfaces or to be coded in a specific fashion. Simply specifying the bean class should suffice. However, depending on what type of IoC you use for that specific bean, you may need a default (empty) constructor.

The Spring IoC container can manage virtually *any* class you want it to manage; it is not limited to managing true JavaBeans. Most Spring users prefer actual JavaBeans with only a default (no-argument) constructor and appropriate setters and getters modeled after the properties in the container. You can also have more exotic non-bean-style classes in your container. If, for example, you need to use a legacy connection pool that absolutely does not adhere to the JavaBean specification, Spring can manage it as well.

With XML-based configuration metadata you can specify your bean class as follows:

```
<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

For details about the mechanism for supplying arguments to the constructor (if required) and setting object instance properties after the object is constructed, see Injecting Dependencies.

Instantiation with a static factory method

When defining a bean that you create with a static factory method, you use the `class` attribute to specify the class containing the `static` factory method and an attribute named `factory-method` to specify the name of the factory method itself. You should be able to call this method (with optional arguments as described later) and return a live object, which subsequently is treated as if it had been created through a constructor. One use for such a bean definition is to call `static` factories in legacy code.

The following bean definition specifies that the bean will be created by calling a factory-method. The definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, the `createInstance()` method must be a *static* method.

```
<bean id="clientService"
    class="examples.ClientService"
    factory-method="createInstance"/>
```

```
public class ClientService {
    private static ClientService clientService = new ClientService();
    private ClientService() {}

    public static ClientService createInstance() {
        return clientService;
    }
}
```

For details about the mechanism for supplying (optional) arguments to the factory method and setting object instance properties after the object is returned from the factory, see Dependencies and configuration in detail.

Instantiation using an instance factory method

Similar to instantiation through a static factory method, instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean. To use this mechanism, leave the `class` attribute empty, and in the `factory-bean` attribute, specify the name of a bean in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the `factory-method` attribute.

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
```

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();
    private DefaultServiceLocator() {}

    public ClientService createClientServiceInstance() {
        return clientService;
```

```
    }
}
```

One factory class can also hold more than one factory method as shown here:

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>

<bean id="accountService"
    factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
```

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();
    private static AccountService accountService = new AccountServiceImpl();

    private DefaultServiceLocator() {}

    public ClientService createClientServiceInstance() {
        return clientService;
    }

    public AccountService createAccountServiceInstance() {
        return accountService;
    }

}
```

This approach shows that the factory bean itself can be managed and configured through dependency injection (DI). See Dependencies and configuration in detail.

In Spring documentation, *factory bean* refers to a bean that is configured in the Spring container that will create objects through an instance or static factory method. By contrast, `FactoryBean` (notice the capitalization) refers to a Spring-specific `FactoryBean`.

7.4 Dependencies

A typical enterprise application does not consist of a single object (or bean in the Spring parlance). Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application. This next section explains how you go from defining a number of bean definitions that stand alone to a fully realized application where objects collaborate to achieve a goal.

7.4.1 Dependency 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.

Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies. As such, your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

DI exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

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...

}
```

Constructor argument resolution

Constructor argument resolution matching occurs using the argument's type. If no potential ambiguity exists in the constructor arguments of a bean definition, then the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated. Consider the following class:

```
package x.y;

public class Foo {

    public Foo(Bar bar, Baz baz) {
        // ...
    }

}
```

No potential ambiguity exists, assuming that `Bar` and `Baz` classes are not related by inheritance. Thus the following configuration works fine, and you do not need to specify the constructor argument indexes and/or types explicitly in the `<constructor-arg/>` element.

```
<beans>
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
    </bean>

    <bean id="bar" class="x.y.Bar"/>

    <bean id="baz" class="x.y.Baz"/>
</beans>
```

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as `<value>true</value>`, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class:

```
package examples;

public class ExampleBean {

    // Number of years to calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }

}
```

In the preceding scenario, the container *can* use type matching with simple types if you explicitly specify the type of the constructor argument using the `type` attribute. For example:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

Use the `index` attribute to specify explicitly the index of constructor arguments. For example:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type. Note that the *index is 0 based*.

You can also use the constructor parameter name for value disambiguation:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
```

```
    <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

Keep in mind that to make this work out of the box your code must be compiled with the debug flag enabled so that Spring can look up the parameter name from the constructor. If you can't compile your code with debug flag (or don't want to) you can use @ConstructorProperties JDK annotation to explicitly name your constructor arguments. The sample class would then have to look as follows:

```
package examples;

public class ExampleBean {

    // Fields omitted

    @ConstructorProperties({"years", "ultimateAnswer"})
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }

}
```

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 `PropertyEditor` instances 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.

Constructor-based or setter-based DI?

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*. Note that use of the @Required annotation on a setter method can be used to make the property a required dependency.

The Spring team generally advocates constructor injection as it enables one to implement application components as *immutable objects* and to ensure that required dependencies are not `null`. Furthermore constructor-injected components are always returned to client (calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a *bad code smell*, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

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 is therefore a compelling use case for setter injection.

Use the DI style that makes the most sense for a particular class. Sometimes, when dealing with third-party classes for which you do not have the source, the choice is made for you. For example, if a third-party class does not expose any setter methods, then constructor injection may be the only available form of DI.

Dependency resolution process

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 via 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 are using 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 which 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`, etc.

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*. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in Section 7.5, “Bean scopes”. Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean’s dependencies and its dependencies' dependencies (and so on) are created and assigned. Note that resolution mismatches among those dependencies may show up late, i.e. on first creation of the affected bean.

Circular dependencies

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the 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.

Unlike the *typical* case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

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 which 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. For example, the bean throws an exception as a result of a missing or invalid property. This potentially delayed visibility of some configuration issues is why `ApplicationContext` implementations by default pre-instantiate singleton beans. At the cost of some upfront time and memory to create these beans before they are actually needed, you discover configuration issues when the `ApplicationContext` is created, not later. You can still override this default behavior so that singleton beans will lazy-initialize, rather than be pre-instantiated.

If no circular dependencies exist, when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is *totally* configured prior to being injected into the dependent bean. This means that if bean A has a dependency on bean B, the Spring IoC container completely configures bean B prior to invoking the setter method on bean A. In other words, the bean is instantiated (if not a pre-instantiated singleton), its dependencies are set, and the relevant lifecycle methods (such as a configured init method or the InitializingBean callback method) are invoked.

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;
    }
```

```
}
```

The constructor arguments specified in the bean definition will be used as arguments to the constructor of the `ExampleBean`.

Now consider a variant of this example, where instead of using a constructor, Spring is told to call a `static` factory method to return an instance of the object:

```
<bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance
    <constructor-arg ref="anotherExampleBean"/>
    <constructor-arg ref="yetAnotherBean"/>
    <constructor-arg value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

    // a private constructor
    private ExampleBean(...) {
        ...
    }

    // a static factory method; the arguments to this method can be
    // considered the dependencies of the bean that is returned,
    // regardless of how those arguments are actually used.
    public static ExampleBean createInstance (
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

        ExampleBean eb = new ExampleBean (...);
        // some other operations...
        return eb;
    }

}
```

Arguments to the `static` factory method are supplied via `<constructor-arg/>` elements, exactly the same as if a constructor had actually been used. The type of the class being returned by the factory method does not have to be of the same type as the class that contains the `static` factory method, although in this example it is. An instance (non-static) factory method would be used in an essentially identical fashion (aside from the use of the `factory-bean` attribute instead of the `class` attribute), so details will not be discussed here.

7.4.2 Dependencies and configuration in detail

As mentioned in the previous section, you can define bean properties and constructor arguments as references to other managed beans (collaborators), or as values defined inline. Spring's XML-based configuration metadata supports sub-element types within its `<property/>` and `<constructor-arg/>` elements for this purpose.

Straight values (primitives, Strings, and so on)

The `value` attribute of the `<property/>` element specifies a property or constructor argument as a human-readable string representation. Spring's conversion service is used to convert these values from a `String` to the actual type of the property or argument.

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-me
    <!-- results in a setDriverClassName(String) call -->
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="masterkaoli"/>
</bean>
```

The following example uses the p-namespace for even more succinct XML configuration.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="com.mysql.jdbc.Driver"
        p:url="jdbc:mysql://localhost:3306/mydb"
        p:username="root"
        p:password="masterkaoli"/>

</beans>
```

The preceding XML is more succinct; however, typos are discovered at runtime rather than design time, unless you use an IDE such as IntelliJ IDEA or the Spring Tool Suite (STS) that support automatic property completion when you create bean definitions. Such IDE assistance is highly recommended.

You can also configure a `java.util.Properties` instance as:

```
<bean id="mappings"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"

    <!-- typed as a java.util.Properties -->
    <property name="properties">
        <value>
            jdbc.driver.className=com.mysql.jdbc.Driver
            jdbc.url=jdbc:mysql://localhost:3306/mydb
        </value>
    </property>
</bean>
```

The Spring container converts the text inside the `<value/>` element into a `java.util.Properties` instance by using the JavaBeans `PropertyEditor` mechanism. This is a nice shortcut, and is one of a few places where the Spring team do favor the use of the nested `<value/>` element over the `value` attribute style.

The idref element

The `idref` element is simply an error-proof way to pass the *id* (string value - not a reference) of another bean in the container to a `<constructor-arg/>` or `<property/>` element.

```
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean" />
    </property>
</bean>
```

The above bean definition snippet is *exactly* equivalent (at runtime) to the following snippet:

```
<bean id="theTargetBean" class="..." />

<bean id="client" class="...">
    <property name="targetName" value="theTargetBean" />
</bean>
```

The first form is preferable to the second, because using the `idref` tag allows the container to validate *at deployment time* that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the `targetName` property of

the `client` bean. Typos are only discovered (with most likely fatal results) when the `client` bean is actually instantiated. If the `client` bean is a prototype bean, this typo and the resulting exception may only be discovered long after the container is deployed.

> The `local` attribute on the `idref` element is no longer supported in the 4.0 beans xsd since it does not provide value over a regular `bean` reference anymore. Simply change your existing `idref local` references to `idref bean` when upgrading to the 4.0 schema.

A common place (at least in versions earlier than Spring 2.0) where the `<idref/>` element brings value is in the configuration of AOP interceptors in a `ProxyFactoryBean` bean definition. Using `<idref/>` elements when you specify the interceptor names prevents you from misspelling an interceptor id.

References to other beans (collaborators)

The `ref` element is the final element inside a `<constructor-arg/>` or `<property/>` definition element. Here you set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container. The referenced bean is a dependency of the bean whose property will be set, and it is initialized on demand as needed before the property is set. (If the collaborator is a singleton bean, it may be initialized already by the container.) All references are ultimately a reference to another object. Scoping and validation depend on whether you specify the id/name of the other object through the `bean`, `local,` or `parent` attributes.

Specifying the target bean through the `bean` attribute of the `<ref/>` tag is the most general form, and allows creation of a reference to any bean in the same container or parent container, regardless of whether it is in the same XML file. The value of the `bean` attribute may be the same as the `id` attribute of the target bean, or as one of the values in the `name` attribute of the target bean.

```
<ref bean="someBean"/>
```

Specifying the target bean through the `parent` attribute creates a reference to a bean that is in a parent container of the current container. The value of the `parent` attribute may be the same as either the `id` attribute of the target bean, or one of the values in the `name` attribute of the target bean, and the target bean must be in a parent container of the current one. You use this bean reference variant mainly when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with a proxy that will have the same name as the parent bean.

```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService">
    <!-- insert dependencies as required as here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="accountService" <!-- bean name is the same as the parent bean -->
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/> <!-- notice how we refer to the parent bean
    </property>
    <!-- insert other configuration and dependencies as required here -->
</bean>
```

> The `local` attribute on the `ref` element is no longer supported in the 4.0 beans xsd since it does not provide value over a regular `bean` reference anymore. Simply change your existing `ref local` references to `ref bean` when upgrading to the 4.0 schema.

Inner beans

A `<bean/>` element inside the `<property/>` or `<constructor-arg/>` elements defines a so-called *inner bean*.

```
<bean id="outer" class="...">
    <!-- instead of using a reference to a target bean, simply define the target be
    <property name="target">
        <bean class="com.example.Person"> <!-- this is the inner bean -->
            <property name="name" value="Fiona Apple"/>
            <property name="age" value="25"/>
        </bean>
    </property>
</bean>
```

An inner bean definition does not require a defined id or name; if specified, the container does not use such a value as an identifier. The container also ignores the `scope` flag on creation: Inner beans are *always* anonymous and they are *always* created with the outer bean. It is *not* possible to inject inner beans into collaborating beans other than into the enclosing bean or to access them independently.

As a corner case, it is possible to receive destruction callbacks from a custom scope, e.g. for a request-scoped inner bean contained within a singleton bean: The creation of the inner bean instance will be tied to its containing bean, but destruction callbacks allow it to participate in the request scope's lifecycle. This is not a common scenario; inner beans typically simply share their containing bean's scope.

Collections

In the `<list/>`, `<set/>`, `<map/>`, and `<props/>` elements, you set the properties and arguments of the Java `Collection` types `List`, `Set`, `Map`, and `Properties`, respectively.

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</prop>
        </props>
    </property>
    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource" />
        </list>
    </property>
    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key ="a ref" value-ref="myDataSource"/>
        </map>
    </property>
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>
```

The value of a map key or value, or a set value, can also again be any of the following elements:

```
bean | ref | idref | list | set | map | props | value | null
```

Collection merging

The Spring container also supports the *merging* of collections. An application developer can define a parent-style `<list/>`, `<map/>`, `<set/>` or `<props/>` element, and have child-style `<list/>`, `<map/>`, `<set/>` or `<props/>` elements inherit and override values from the parent collection. That is, the child collection's values are the result of merging the elements of the parent and child collections, with the child's collection elements overriding values specified in the parent collection.

This section on merging discusses the parent-child bean mechanism. Readers unfamiliar with parent and child bean definitions may wish to read the relevant section before continuing.

The following example demonstrates collection merging:

```
<beans>
    <bean id="parent" abstract="true" class="example.ComplexObject">
        <property name="adminEmails">
            <props>
                <prop key="administrator">administrator@example.com</prop>
                <prop key="support">support@example.com</prop>
            </props>
        </property>
    </bean>
    <bean id="child" parent="parent">
        <property name="adminEmails">
            <!-- the merge is specified on the child collection definition -->
            <props merge="true">
                <prop key="sales">sales@example.com</prop>
                <prop key="support">support@example.co.uk</prop>
            </props>
        </property>
    </bean>
<beans>
```

Notice the use of the `merge=true` attribute on the `<props/>` element of the `adminEmails` property of the `child` bean definition. When the `child` bean is resolved and instantiated by the container, the resulting instance has an `adminEmails` `Properties` collection that contains the result of the merging of the child's `adminEmails` collection with the parent's `adminEmails` collection.

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

The child `Properties` collection's value set inherits all property elements from the parent `<props/>`, and the child's value for the `support` value overrides the value in the parent collection.

This merging behavior applies similarly to the `<list/>`, `<map/>`, and `<set/>` collection types. In the specific case of the `<list/>` element, the semantics associated with the `List` collection type, that is, the notion of an `ordered` collection of values, is maintained; the parent's values precede all of the child list's values. In the case of the `Map`, `Set`, and `Properties` collection types, no ordering exists. Hence no ordering semantics are in effect for the collection types that underlie the associated `Map`, `Set`, and `Properties` implementation types that the container uses internally.

Limitations of collection merging

You cannot merge different collection types (such as a `Map` and a `List`), and if you do attempt to do so an appropriate `Exception` is thrown. The `merge` attribute must be specified on the lower, inherited, child definition; specifying the `merge` attribute on a parent collection definition is redundant and will not result in the desired merging.

Strongly-typed collection

With the introduction of generic types in Java 5, you can use strongly typed collections. That is, it is possible to declare a `Collection` type such that it can only contain `String` elements (for example). If you are using Spring to dependency-inject a strongly-typed `Collection` into a bean, you can take advantage of Spring's type-conversion support such that the elements of your strongly-typed `Collection` instances are converted to the appropriate type prior to being added to the `Collection`.

```
public class Foo {

    private Map<String, Float> accounts;

    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
}
```

```
<beans>
    <bean id="foo" class="x.y.Foo">
        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
```

```
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
    </bean>
</beans>
```

When the `accounts` property of the `foo` bean is prepared for injection, the generics information about the element type of the strongly-typed `Map<String, Float>` is available by reflection. Thus Spring's type conversion infrastructure recognizes the various value elements as being of type `Float`, and the string values `9.99, 2.75`, and `3.99` are converted into an actual `Float` type.

Null and empty string values

Spring treats empty arguments for properties and the like as empty `Strings`. The following XML-based configuration metadata snippet sets the email property to the empty `String` value ("").

```
<bean class="ExampleBean">
    <property name="email" value=""/>
</bean>
```

The preceding example is equivalent to the following Java code:

```
exampleBean.setEmail("")
```

The `<null/>` element handles `null` values. For example:

```
<bean class="ExampleBean">
    <property name="email">
        <null/>
    </property>
</bean>
```

The above configuration is equivalent to the following Java code:

```
exampleBean.setEmail(null)
```

XML shortcut with the p-namespace

The p-namespace enables you to use the `bean` element's attributes, instead of nested `<property/>` elements, to describe your property values and/or collaborating beans.

Spring supports extensible configuration formats with namespaces, which are based on an XML Schema definition. The `beans` configuration format discussed in this chapter is defined in an XML Schema document. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

The following example shows two XML snippets that resolve to the same result: The first uses standard XML format and the second uses the p-namespace.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value="foo@bar.com"/>
    </bean>

    <bean name="p-namespace" class="com.example.ExampleBean"
        p:email="foo@bar.com"/>
</beans>
```

The example shows an attribute in the p-namespace called email in the bean definition. This tells Spring to include a property declaration. As previously mentioned, the p-namespace does not have a schema definition, so you can set the name of the attribute to the property name.

This next example includes two more bean definitions that both have a reference to another bean:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>

    <bean name="john-modern"
        class="com.example.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>
```

```
    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

As you can see, this example includes not only a property value using the p-namespace, but also uses a special format to declare property references. Whereas the first bean definition uses `<property name="spouse" ref="jane"/>` to create a reference from bean `john` to bean `jane`, the second bean definition uses `p:spouse-ref="jane"` as an attribute to do the exact same thing. In this case `spouse` is the property name, whereas the `-ref` part indicates that this is not a straight value but rather a reference to another bean.

> The p-namespace is not as flexible as the standard XML format. For example, the format for declaring property references clashes with properties that end in `Ref`, whereas the standard XML format does not. We recommend that you choose your approach carefully and communicate this to your team members, to avoid producing XML documents that use all three approaches at the same time.

XML shortcut with the c-namespace

Similar to the the section called “XML shortcut with the p-namespace”, the *c-namespace*, newly introduced in Spring 3.1, allows usage of inlined attributes for configuring the constructor arguments rather then nested `constructor-arg` elements.

Let’s review the examples from the section called “Constructor-based dependency injection” with the `c:` namespace:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="bar" class="x.y.Bar"/>
    <bean id="baz" class="x.y.Baz"/>

    <!-- traditional declaration -->
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
        <constructor-arg value="foo@bar.com"/>
    </bean>
```

```
    <!-- c-namespace declaration -->
    <bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@ba

</beans>
```

The `c:` namespace uses the same conventions as the `p:` one (trailing `-ref` for bean references) for setting the constructor arguments by their names. And just as well, it needs to be declared even though it is not defined in an XSD schema (but it exists inside the Spring core).

For the rare cases where the constructor argument names are not available (usually if the bytecode was compiled without debugging information), one can use fallback to the argument indexes:

```
<!-- c-namespace index declaration -->
<bean id="foo" class="x.y.Foo" c:_0-ref="bar" c:_1-ref="baz"/>
```

> Due to the XML grammar, the index notation requires the presence of the leading `_` as XML attribute names cannot start with a number (even though some IDE allow it).

In practice, the constructor resolution mechanism is quite efficient in matching arguments so unless one really needs to, we recommend using the name notation through-out your configuration.

Compound property names

You can use compound or nested property names when you set bean properties, as long as all components of the path except the final property name are not `null`. Consider the following bean definition.

```
<bean id="foo" class="foo.Bar">
    <property name="fred.bob.sammy" value="123" />
</bean>
```

The `foo` bean has a `fred` property, which has a `bob` property, which has a `sammy` property, and that final `sammy` property is being set to the value `123`. In order for this to work, the `fred` property of `foo`, and the `bob` property of `fred` must not be `null` after the bean is constructed, or a `NullPointerException` is thrown.

7.4.3 Using depends-on

If a bean is a dependency of another that usually means that one bean is set as a property of another. Typically you accomplish this with the `<ref/>` element in XML-based configuration metadata. However, sometimes dependencies between beans are less direct; for example, a static initializer in a class needs to be triggered, such as database driver registration. The `depends-on` attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized. The following example uses the `depends-on` attribute to express a dependency on a single bean:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>
<bean id="manager" class="ManagerBean" />
```

To express a dependency on multiple beans, supply a list of bean names as the value of the `depends-on` attribute, with commas, whitespace and semicolons, used as valid delimiters:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
    <property name="manager" ref="manager" />
</bean>

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

The `depends-on` attribute in the bean definition can specify both an initialization time dependency and, in the case of singleton beans only, a corresponding destroy time dependency. Dependent beans that define a `depends-on` relationship with a given bean are destroyed first, prior to the given bean itself being destroyed. Thus `depends-on` can also control shutdown order.

7.4.4 Lazy-initialized beans

By default, `ApplicationContext` implementations eagerly create and configure all 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 lazy-initialized. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

In XML, this behavior is controlled by the `lazy-init` attribute on the `<bean/>` element; for example:

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

When the preceding configuration is consumed by an `ApplicationContext`, the bean named `lazy` is not eagerly pre-instantiated when the `ApplicationContext` is starting up, whereas the `not.lazy` bean is eagerly pre-instantiated.

However, when a lazy-initialized bean is a dependency of a singleton bean that is *not* lazy-initialized, the `ApplicationContext` creates the lazy-initialized bean at startup, because it must satisfy the singleton's dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is not lazy-initialized.

You can also control lazy-initialization at the container level by using the `default-lazy-init` attribute on the `<beans/>` element; for example:

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

7.4.5 Autowiring collaborators

The Spring container can *autowire* relationships between collaborating beans. You can allow Spring to 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. (Other mechanisms such as a bean template discussed elsewhere in this chapter are also valuable in this regard.)
- 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.

When using XML-based configuration metadata [2], you specify autowire mode for a bean definition with the `autowire` attribute of the `<bean/>` element. The autowiring functionality has four modes. You specify autowiring *per* bean and thus can choose which ones to autowire.

Table 7.2. Autowiring modes

Mode	Explanation
no	(Default) No autowiring. Bean references must be defined via a `ref` element. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system.
byName	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name, and it contains a *master* property (that is, it has a *setMaster(..)* method), Spring looks for a bean definition named `master`, and uses it to set the property.
byType	Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use *byType* autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.
constructor	Analogous to *byType*, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

With *byType* or *constructor* autowiring mode, you can wire arrays and typed-collections. In such cases *all* autowire candidates within the container that match the expected type are provided to satisfy the dependency. You can autowire strongly-typed Maps if the expected key type is `String`. An autowired Maps values will consist of all bean instances that match the expected type, and the Maps keys will contain the corresponding bean names.

You can combine autowire behavior with dependency checking, which is performed after autowiring completes.

Limitations and disadvantages of autowiring

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.

Consider the limitations and disadvantages of autowiring:

- Explicit dependencies in `property` and `constructor-arg` settings always override autowiring. You cannot autowire so-called *simple* properties such as primitives, `Strings`, and `Classes` (and arrays of such simple properties). This limitation is by-design.

- Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.
- Wiring information may not be available to tools that may generate documentation from a Spring container.
- Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily a problem. However for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

In the latter scenario, you have several options:

- Abandon autowiring in favor of explicit wiring.
- Avoid autowiring for a bean definition by setting its `autowire-candidate` attributes to `false` as described in the next section.
- Designate a single bean definition as the *primary* candidate by setting the `primary` attribute of its `<bean/>` element to `true`.
- Implement the more fine-grained control available with annotation-based configuration, as described in Section 7.9, “Annotation-based container configuration”.

Excluding a bean from autowiring

On a per-bean basis, you can exclude a bean from autowiring. In Spring’s XML format, set the `autowire-candidate` attribute of the `<bean/>` element to `false`; the container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations such as `@Autowired`).

You can also limit autowire candidates based on pattern-matching against bean names. The top-level `<beans/>` element accepts one or more patterns within its `default-autowire-candidates` attribute. For example, to limit autowire candidate status to any bean whose name ends with *Repository,* provide a value of *Repository. To provide multiple patterns, define them in a comma-separated list. An explicit value of `true` or `false` for a bean definitions `autowire-candidate` attribute always takes precedence, and for such beans, the pattern matching rules do not apply.

These techniques are useful for beans that you never want to be injected into other beans by autowiring. It does not mean that an excluded bean cannot itself be configured using autowiring. Rather, the bean itself is not a candidate for autowiring other beans.

7.4.6 Method injection

In most application scenarios, most beans in the container are singletons. When a singleton bean needs to collaborate with another singleton bean, or a non-singleton bean needs to collaborate with another non-singleton bean, you typically handle the dependency by defining one bean as a property of the other. A problem arises when the bean lifecycles are different. Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A. The container only creates the singleton bean A once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

A solution is to forego some inversion of control. You can make bean A aware of the container by implementing the `ApplicationContextAware` interface, and by making a getBean("B") call to the container ask for (a typically new) bean B instance every time bean A needs it. The following is an example of this approach:

```
// a class that uses a stateful Command-style class to perform some processing
package fiona.apple;

// Spring-API imports
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class CommandManager implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public Object process(Map commandState) {
        // grab a new instance of the appropriate Command
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    protected Command createCommand() {
        // notice the Spring API dependency!
        return this.applicationContext.getBean("command", Command.class);
    }

    public void setApplicationContext(
            ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

The preceding is not desirable, because the business code is aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, allows this use case to be handled in a clean fashion.

> You can read more about the motivation for Method Injection in this blog entry.

Lookup method injection

Lookup method injection is the ability of the container to override methods on *container managed beans*, to return the lookup result for another named bean in the container. The lookup typically involves a prototype bean as in the scenario described in the preceding section. The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to generate dynamically a subclass that overrides the method.

- For this dynamic subclassing to work, the class that the Spring bean container will subclass cannot be `final`, and the method to be overridden cannot be `final` either.
- Unit-testing a class that has an `abstract` method requires you to subclass the class yourself and to supply a stub implementation of the `abstract` method.
- Concrete methods are also necessary for component scanning which requires concrete classes to pick up.
- A further key limitation is that lookup methods won't work with factory methods and in particular not with `@Bean` methods in configuration classes, since the container is not in charge of creating the instance in that case and therefore cannot create a runtime-generated subclass on the fly.

Looking at the `CommandManager` class in the previous code snippet, you see that the Spring container will dynamically override the implementation of the `createCommand()` method. Your `CommandManager` class will not have any Spring dependencies, as can be seen in the reworked example:

```
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
```

```
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

In the client class containing the method to be injected (the `CommandManager` in this case), the method to be injected requires a signature of the following form:

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

If the method is `abstract`, the dynamically-generated subclass implements the method. Otherwise, the dynamically-generated subclass overrides the concrete method defined in the original class. For example:

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">
    <!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
    <lookup-method name="createCommand" bean="myCommand"/>
</bean>
```

The bean identified as *commandManager* calls its own method `createCommand()` whenever it needs a new instance of the *myCommand* bean. You must be careful to deploy the `myCommand` bean as a prototype, if that is actually what is needed. If it is as a singleton, the same instance of the `myCommand` bean is returned each time.

Alternatively, within the annotation-based component model, you may declare a lookup method through the `@Lookup` annotation:

```
public abstract class CommandManager {

    public Object process(Object commandState) {
        Command command = createCommand();
        command.setState(commandState);
        return command.execute();
    }
```

```
    @Lookup("myCommand")
    protected abstract Command createCommand();
}
```

Or, more idiomatically, you may rely on the target bean getting resolved against the declared return type of the lookup method:

```
public abstract class CommandManager {

    public Object process(Object commandState) {
        MyCommand command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup
    protected abstract MyCommand createCommand();
}
```

Note that you will typically declare such annotated lookup methods with a concrete stub implementation, in order for them to be compatible with Spring's component scanning rules where abstract classes get ignored by default. This limitation does not apply in case of explicitly registered or explicitly imported bean classes.

> Another way of accessing differently scoped target beans is an `ObjectFactory`/ `Provider` injection point. Check out the section called "Scoped beans as dependencies".
> The interested reader may also find the `ServiceLocatorFactoryBean` (in the `org.springframework.beans.factory.config` package) to be of use.

Arbitrary method replacement

A less useful form of method injection than lookup method injection is the ability to replace arbitrary methods in a managed bean with another method implementation. Users may safely skip the rest of this section until the functionality is actually needed.

With XML-based configuration metadata, you can use the `replaced-method` element to replace an existing method implementation with another, for a deployed bean. Consider the following class, with a method computeValue, which we want to override:

```
public class MyValueCalculator {
```

```
    public String computeValue(String input) {
        // some real code...
    }

    // some other methods...

}
```

A class implementing the `org.springframework.beans.factory.support.MethodReplacer` interface provides the new method definition.

```
/**
 * meant to be used to override the existing computeValue(String)
 * implementation in MyValueCalculator
 */
public class ReplacementComputeValue implements MethodReplacer {

    public Object reimplement(Object o, Method m, Object[] args) throws Throwable
        // get the input value, work with it, and return a computed result
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

The bean definition to deploy the original class and specify the method override would look like this:

```
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">
    <!-- arbitrary method replacement -->
    <replaced-method name="computeValue" replacer="replacementComputeValue">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

You can use one or more contained `<arg-type/>` elements within the `<replaced-method/>` element to indicate the method signature of the method being overridden. The signature for the arguments is necessary only if the method is overloaded and multiple variants exist within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, the following all match `java.lang.String`:

```
java.lang.String
String
Str
```

Because the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by allowing you to type only the shortest string that will match an argument type.

7.5 Bean scopes

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 the *scope* of the objects created from a particular bean definition. This approach is powerful and flexible in that you can *choose* the scope of the objects you create through configuration instead of having to bake in the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports seven scopes, five of which are available only if you use a web-aware `ApplicationContext`.

The following scopes are supported out of the box. You can also create a custom scope.

Table 7.3. Bean scopes

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.

Scope	Description
globalSession	Scopes a single bean definition to the lifecycle of a global HTTP `Session`. Typically only valid when used in a Portlet context. Only valid in the context of a web-aware Spring `ApplicationContext`.
application	Scopes a single bean definition to the lifecycle of a `ServletContext`. Only valid in the context of a web-aware Spring `ApplicationContext`.
websocket	Scopes a single bean definition to the lifecycle of a `WebSocket`. Only valid in the context of a web-aware Spring `ApplicationContext`.

As of Spring 3.0, a *thread scope* is available, but is not registered by default. For more information, see the documentation for `SimpleThreadScope`. For instructions on how to register this or any other custom scope, see the section called “Using a custom scope”.

7.5.1 The singleton scope

Only one *shared* instance of a singleton bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, the Spring IoC container creates *exactly one* instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and *all subsequent requests and references* for that named bean return the cached object.

Spring’s concept of a singleton bean differs from the Singleton pattern as defined in the Gang of Four (GoF) patterns book. The GoF Singleton hard-codes the scope of an object such that one *and only one* instance of a particular class is created *per ClassLoader*. The scope of the Spring singleton is best described as *per container and per bean*. This means that if you define one bean for a particular class in a single Spring container, then the Spring container creates one *and only one* instance of the class defined by that bean definition. *The singleton scope is the default scope in Spring*. To define a bean as a singleton in XML, you would write, for example:

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>
```

```
<!-- the following is equivalent, though redundant (singleton scope is the default)
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

7.5.2 The prototype scope

The non-singleton, prototype scope of bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a `getBean()` method call on the container. As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.

The following diagram illustrates the Spring prototype scope. *A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram.*

The following example defines a bean as a prototype in XML:

```
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>
```

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client, with no further record of that prototype instance. Thus, although *initialization* lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured *destruction* lifecycle callbacks are *not* called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding. To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.

In some respects, the Spring container's role in regard to a prototype-scoped bean is a replacement for the Java `new` operator. All lifecycle management past that point must be handled by the client. (For details on the lifecycle of a bean in the Spring container, see Section 7.6.1, "Lifecycle callbacks".)

7.5.3 Singleton beans with prototype-bean dependencies

When you use singleton-scoped beans with dependencies on prototype beans, be aware that *dependencies are resolved at instantiation time*. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then

dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only *once*, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies. If you need a new instance of a prototype bean at runtime more than once, see Section 7.4.6, “Method injection”

7.5.4 Request, session, global session, application, and WebSocket scopes

The `request`, `session`, `globalSession`, `application`, and `websocket` scopes are *only* available if you use a web-aware Spring `ApplicationContext` implementation (such as `XmlWebApplicationContext`). If you use these scopes with regular Spring IoC containers such as the `ClassPathXmlApplicationContext`, an `IllegalStateException` will be thrown complaining about an unknown bean scope.

Initial web configuration

To support the scoping of beans at the `request`, `session`, `globalSession`, `application`, and `websocket` levels (web-scoped beans), some minor initial configuration is required before you define your beans. (This initial setup is *not* required for the standard scopes, `singleton` and `prototype`.)

How you accomplish this initial setup depends on your particular Servlet environment.

If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring `DispatcherServlet` or `DispatcherPortlet`, then no special setup is necessary: `DispatcherServlet` and `DispatcherPortlet` already expose all relevant state.

If you use a Servlet 2.5 web container, with requests processed outside of Spring’s `DispatcherServlet` (for example, when using JSF or Struts), you need to register the `org.springframework.web.context.request.RequestContextListener` `ServletRequestListener`. For Servlet 3.0+, this can be done programmatically via the `WebApplicationInitializer` interface. Alternatively, or for older containers, add the following declaration to your web application’s `web.xml` file:

```
<web-app>
    ...
```

```
    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    ...
</web-app>
```

Alternatively, if there are issues with your listener setup, consider using Spring's `RequestContextFilter`. The filter mapping depends on the surrounding web application configuration, so you have to change it as appropriate.

```
<web-app>
    ...
    <filter>
        <filter-name>requestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestContextFilter</filter-
    </filter>
    <filter-mapping>
        <filter-name>requestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

`DispatcherServlet`, `RequestContextListener`, and `RequestContextFilter` all do exactly the same thing, namely bind the HTTP request object to the `Thread` that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

Request scope

Consider the following XML configuration for a bean definition:

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

The Spring container creates a new instance of the `LoginAction` bean by using the `loginAction` bean definition for each and every HTTP request. That is, the `loginAction` bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want, because other instances created from the same `loginAction` bean definition will not see these changes in state; they are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

When using annotation-driven components or Java Config, the `@RequestScope` annotation can be used to assign a component to the `request` scope.

```
@RequestScope
@Component
public class LoginAction {
    // ...
}
```

Session scope

Consider the following XML configuration for a bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

The Spring container creates a new instance of the `UserPreferences` bean by using the `userPreferences` bean definition for the lifetime of a single HTTP `Session`. In other words, the `userPreferences` bean is effectively scoped at the HTTP `Session` level. As with `request-scoped` beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP `Session` instances that are also using instances created from the same `userPreferences` bean definition do not see these changes in state, because they are particular to an individual HTTP `Session`. When the HTTP `Session` is eventually discarded, the bean that is scoped to that particular HTTP `Session` is also discarded.

When using annotation-driven components or Java Config, the `@SessionScope` annotation can be used to assign a component to the `session` scope.

```
@SessionScope
@Component
public class UserPreferences {
    // ...
}
```

Global session scope

Consider the following bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

The `globalSession` scope is similar to the standard HTTP `Session` scope (described above), and applies only in the context of portlet-based web applications. The portlet specification defines the notion of a global `Session` that is shared among all portlets that make up a single portlet

web application. Beans defined at the `globalSession` scope are scoped (or bound) to the lifetime of the global portlet `Session`.

If you write a standard Servlet-based web application and you define one or more beans as having `globalSession` scope, the standard HTTP `Session` scope is used, and no error is raised.

Application scope

Consider the following XML configuration for a bean definition:

```
<bean id="appPreferences" class="com.foo.AppPreferences" scope="application"/>
```

The Spring container creates a new instance of the `AppPreferences` bean by using the `appPreferences` bean definition once for the entire web application. That is, the `appPreferences` bean is scoped at the `ServletContext` level, stored as a regular `ServletContext` attribute. This is somewhat similar to a Spring singleton bean but differs in two important ways: It is a singleton per `ServletContext`, not per Spring 'ApplicationContext' (for which there may be several in any given web application), and it is actually exposed and therefore visible as a `ServletContext` attribute.

When using annotation-driven components or Java Config, the `@ApplicationScope` annotation can be used to assign a component to the `application` scope.

```
@ApplicationScope
@Component
public class AppPreferences {
    // ...
}
```

Scoped beans as dependencies

The Spring IoC container manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies). If you want to inject (for example) an HTTP request scoped bean into another bean of a longer-lived scope, you may choose to inject an AOP proxy in place of the scoped bean. That is, you need to inject a proxy object that exposes the same public interface as the scoped object but that can also retrieve the real target object from the relevant scope (such as an HTTP request) and delegate method calls onto the real object.

You may also use `<aop:scoped-proxy/>` between beans that are scoped as `singleton`, with the reference then going through an intermediate proxy that is

> serializable and therefore able to re-obtain the target singleton bean on deserialization.
> When declaring `<aop:scoped-proxy/>` against a bean of scope `prototype`, every method call on the shared proxy will lead to the creation of a new target instance which the call is then being forwarded to.
> Also, scoped proxies are not the only way to access beans from shorter scopes in a lifecycle-safe fashion. You may also simply declare your injection point (i.e. the constructor/setter argument or autowired field) as `ObjectFactory<MyTargetBean>`, allowing for a `getObject()` call to retrieve the current instance on demand every time it is needed - without holding on to the instance or storing it separately.
> The JSR-330 variant of this is called `Provider`, used with a `Provider<MyTargetBean>` declaration and a corresponding `get()` call for every retrieval attempt. See here for more details on JSR-330 overall.

The configuration in the following example is only one line, but it is important to understand the "why" as well as the "how" behind it.

```
<?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:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- an HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
        <!-- instructs the container to proxy the surrounding bean -->
        <aop:scoped-proxy/>
    </bean>

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.foo.SimpleUserService">
        <!-- a reference to the proxied userPreferences bean -->
        <property name="userPreferences" ref="userPreferences"/>
    </bean>
</beans>
```

To create such a proxy, you insert a child `<aop:scoped-proxy/>` element into a scoped bean definition (see the section called “Choosing the type of proxy to create” and Chapter 41, *XML*

Schema-based configuration). Why do definitions of beans scoped at the `request`, `session`, `globalSession` and custom-scope levels require the `<aop:scoped-proxy/>` element? Let's examine the following singleton bean definition and contrast it with what you need to define for the aforementioned scopes (note that the following `userPreferences` bean definition as it stands is *incomplete*).

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

In the preceding example, the singleton bean `userManager` is injected with a reference to the HTTP `Session`-scoped bean `userPreferences`. The salient point here is that the `userManager` bean is a singleton: it will be instantiated *exactly once* per container, and its dependencies (in this case only one, the `userPreferences` bean) are also injected only once. This means that the `userManager` bean will only operate on the exact same `userPreferences` object, that is, the one that it was originally injected with.

This is *not* the behavior you want when injecting a shorter-lived scoped bean into a longer-lived scoped bean, for example injecting an HTTP `Session`-scoped collaborating bean as a dependency into singleton bean. Rather, you need a single `userManager` object, and for the lifetime of an HTTP `Session`, you need a `userPreferences` object that is specific to said HTTP `Session`. Thus the container creates an object that exposes the exact same public interface as the `UserPreferences` class (ideally an object that *is a* `UserPreferences` instance) which can fetch the real `UserPreferences` object from the scoping mechanism (HTTP request, `Session`, etc.). The container injects this proxy object into the `userManager` bean, which is unaware that this `UserPreferences` reference is a proxy. In this example, when a `UserManager` instance invokes a method on the dependency-injected `UserPreferences` object, it actually is invoking a method on the proxy. The proxy then fetches the real `UserPreferences` object from (in this case) the HTTP `Session`, and delegates the method invocation onto the retrieved real `UserPreferences` object.

Thus you need the following, correct and complete, configuration when injecting `request-`, `session-`, and `globalSession-scoped` beans into collaborating objects:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
```

```
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

Choosing the type of proxy to create

By default, when the Spring container creates a proxy for a bean that is marked up with the `<aop:scoped-proxy/>` element, *a CGLIB-based class proxy is created*.

CGLIB proxies only intercept public method calls! Do not call non-public methods on such a proxy; they will not be delegated to the actual scoped target object.

Alternatively, you can configure the Spring container to create standard JDK interface-based proxies for such scoped beans, by specifying `false` for the value of the `proxy-target-class` attribute of the `<aop:scoped-proxy/>` element. Using JDK interface-based proxies means that you do not need additional libraries in your application classpath to effect such proxying. However, it also means that the class of the scoped bean must implement at least one interface, and *that all* collaborators into which the scoped bean is injected must reference the bean through one of its interfaces.

```
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

For more detailed information about choosing class-based or interface-based proxying, see Section 11.6, “Proxying mechanisms”.

7.5.5 Custom scopes

The bean scoping mechanism is extensible; You can define your own scopes, or even redefine existing scopes, although the latter is considered bad practice and you *cannot* override the built-in `singleton` and `prototype` scopes.

Creating a custom scope

To integrate your custom scope(s) into the Spring container, you need to implement the `org.springframework.beans.factory.config.Scope` interface, which is described in this section. For an idea of how to implement your own scopes, see the `Scope` implementations that are supplied with the Spring Framework itself and the `Scope` javadocs, which explains the methods you need to implement in more detail.

The `Scope` interface has four methods to get objects from the scope, remove them from the scope, and allow them to be destroyed.

The following method returns the object from the underlying scope. The session scope implementation, for example, returns the session-scoped bean (and if it does not exist, the method returns a new instance of the bean, after having bound it to the session for future reference).

```
Object get(String name, ObjectFactory objectFactory)
```

The following method removes the object from the underlying scope. The session scope implementation for example, removes the session-scoped bean from the underlying session. The object should be returned, but you can return null if the object with the specified name is not found.

```
Object remove(String name)
```

The following method registers the callbacks the scope should execute when it is destroyed or when the specified object in the scope is destroyed. Refer to the javadocs or a Spring scope implementation for more information on destruction callbacks.

```
void registerDestructionCallback(String name, Runnable destructionCallback)
```

The following method obtains the conversation identifier for the underlying scope. This identifier is different for each scope. For a session scoped implementation, this identifier can be the session identifier.

```
String getConversationId()
```

Using a custom scope

After you write and test one or more custom `Scope` implementations, you need to make the Spring container aware of your new scope(s). The following method is the central method to register a new `Scope` with the Spring container:

```
void registerScope(String scopeName, Scope scope);
```

This method is declared on the `ConfigurableBeanFactory` interface, which is available on most of the concrete `ApplicationContext` implementations that ship with Spring via the BeanFactory property.

The first argument to the `registerScope(..)` method is the unique name associated with a scope; examples of such names in the Spring container itself are `singleton` and `prototype`. The second argument to the `registerScope(..)` method is an actual instance of the custom `Scope` implementation that you wish to register and use.

Suppose that you write your custom `Scope` implementation, and then register it as below.

> The example below uses `SimpleThreadScope` which is included with Spring, but not registered by default. The instructions would be the same for your own custom `Scope` implementations.

```
Scope threadScope = new SimpleThreadScope();
beanFactory.registerScope("thread", threadScope);
```

You then create bean definitions that adhere to the scoping rules of your custom `Scope`:

```
<bean id="..." class="..." scope="thread">
```

With a custom `Scope` implementation, you are not limited to programmatic registration of the scope. You can also do the `Scope` registration declaratively, using the `CustomScopeConfigurer` class:

```
<?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:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="thread">
                    <bean class="org.springframework.context.support.SimpleThreadS
                </entry>
            </map>
```

```
        </property>
    </bean>

    <bean id="bar" class="x.y.Bar" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="foo" class="x.y.Foo">
        <property name="bar" ref="bar"/>
    </bean>

</beans>
```

When you place `<aop:scoped-proxy/>` in a `FactoryBean` implementation, it is the factory bean itself that is scoped, not the object returned from `getObject()`.

7.6 Customizing the nature of a bean

7.6.1 Lifecycle callbacks

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 allow the bean to 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. For details see Section 7.9.8, "@PostConstruct and @PreDestroy".
> If you don't want to use the JSR-250 annotations but you are still looking to remove coupling consider the use of init-method and destroy-method object definition metadata.

Internally, the Spring Framework uses `BeanPostProcessor` implementations to process any callback interfaces it can find and call the appropriate methods. If you need custom features or

other lifecycle behavior Spring does not offer out-of-the-box, you can implement a `BeanPostProcessor` yourself. For more information, see Section 7.8, “Container Extension Points”.

In addition to the initialization and destruction callbacks, Spring-managed objects may also implement the `Lifecycle` interface so that those objects can participate in the startup and shutdown process as driven by the container’s own lifecycle.

The lifecycle callback interfaces are described in this section.

Initialization callbacks

The `org.springframework.beans.factory.InitializingBean` interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container. The `InitializingBean` interface specifies a single method:

```
void afterPropertiesSet() throws Exception;
```

It is recommended that you do not use the `InitializingBean` interface because it unnecessarily couples the code to Spring. Alternatively, use the `@PostConstruct` annotation or specify a POJO initialization method. In the case of XML-based configuration metadata, you use the `init-method` attribute to specify the name of the method that has a void no-argument signature. With Java config, you use the `initMethod` attribute of `@Bean`, see the section called “Receiving lifecycle callbacks”. For example, the following:

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

```
public class ExampleBean {

    public void init() {
        // do some initialization work
    }

}
```

…is exactly the same as…

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements InitializingBean {

    public void afterPropertiesSet() {
        // do some initialization work
```

```
    }

}
```

but does not couple the code to Spring.

Destruction callbacks

Implementing the `org.springframework.beans.factory.DisposableBean` interface allows a bean to get a callback when the container containing it is destroyed. The `DisposableBean` interface specifies a single method:

```
void destroy() throws Exception;
```

It is recommended that you do not use the `DisposableBean` callback interface because it unnecessarily couples the code to Spring. Alternatively, use the `@PreDestroy` annotation or specify a generic method that is supported by bean definitions. With XML-based configuration metadata, you use the `destroy-method` attribute on the `<bean/>`. With Java config, you use the `destroyMethod` attribute of `@Bean`, see the section called “Receiving lifecycle callbacks”. For example, the following definition:

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

```
public class ExampleBean {

    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
    }

}
```

is exactly the same as:

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements DisposableBean {

    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }

}
```

but does not couple the code to Spring.

> The `destroy-method` attribute of a `<bean>` element can be assigned a special `(inferred)` value which instructs Spring to automatically detect a public `close` or `shutdown` method on the specific bean class (any class that implements `java.lang.AutoCloseable` or `java.io.Closeable` would therefore match). This special `(inferred)` value can also be set on the `default-destroy-method` attribute of a `<beans>` element to apply this behavior to an entire set of beans (see the section called “Default initialization and destroy methods”). Note that this is the default behavior with Java config.

Default initialization and destroy methods

When you write initialization and destroy method callbacks that do not use the Spring-specific `InitializingBean` and `DisposableBean` callback interfaces, you typically write methods with names such as `init()`, `initialize()`, `dispose()`, and so on. Ideally, the names of such lifecycle callback methods are standardized across a project so that all developers use the same method names and ensure consistency.

You can configure the Spring container to `look` for named initialization and destroy callback method names on *every* bean. This means that you, as an application developer, can write your application classes and use an initialization callback called `init()`, without having to configure an `init-method="init"` attribute with each bean definition. The Spring IoC container calls that method when the bean is created (and in accordance with the standard lifecycle callback contract described previously). This feature also enforces a consistent naming convention for initialization and destroy method callbacks.

Suppose that your initialization callback methods are named `init()` and destroy callback methods are named `destroy()`. Your class will resemble the class in the following example.

```
public class DefaultBlogService implements BlogService {

    private BlogDao blogDao;

    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }

    // this is (unsurprisingly) the initialization callback method
    public void init() {
        if (this.blogDao == null) {
```

```
            throw new IllegalStateException("The [blogDao] property must be set.");
        }
    }

}
```

```
<beans default-init-method="init">

    <bean id="blogService" class="com.foo.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

The presence of the `default-init-method` attribute on the top-level `<beans/>` element attribute causes the Spring IoC container to recognize a method called `init` on beans as the initialization method callback. When a bean is created and assembled, if the bean class has such a method, it is invoked at the appropriate time.

You configure destroy method callbacks similarly (in XML, that is) by using the `default-destroy-method` attribute on the top-level `<beans/>` element.

Where existing bean classes already have callback methods that are named at variance with the convention, you can override the default by specifying (in XML, that is) the method name using the `init-method` and `destroy-method` attributes of the `<bean/>` itself.

The Spring container guarantees that a configured initialization callback is called immediately after a bean is supplied with all dependencies. Thus the initialization callback is called on the raw bean reference, which means that AOP interceptors and so forth are not yet applied to the bean. A target bean is fully created *first*, *then* an AOP proxy (for example) with its interceptor chain is applied. If the target bean and the proxy are defined separately, your code can even interact with the raw target bean, bypassing the proxy. Hence, it would be inconsistent to apply the interceptors to the init method, because doing so would couple the lifecycle of the target bean with its proxy/interceptors and leave strange semantics when your code interacts directly to the raw target bean.

Combining lifecycle mechanisms

As of Spring 2.5, you have three options for controlling bean lifecycle behavior: the `InitializingBean` and `DisposableBean` callback interfaces; custom `init()` and `destroy()` methods; and the `@PostConstruct` and `@PreDestroy` annotations. You can combine these mechanisms to control a given bean.

> If multiple lifecycle mechanisms are configured for a bean, and each mechanism is configured with a different method name, then each configured method is executed in the order listed below. However, if the same method name is configured - for example, `init()` for an initialization method - for more than one of these lifecycle mechanisms, that method is executed once, as explained in the preceding section.

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

- Methods annotated with `@PostConstruct`
- `afterPropertiesSet()` as defined by the `InitializingBean` callback interface
- A custom configured `init()` method

Destroy methods are called in the same order:

- Methods annotated with `@PreDestroy`
- `destroy()` as defined by the `DisposableBean` callback interface
- A custom configured `destroy()` method

Startup and shutdown callbacks

The `Lifecycle` interface defines the essential methods for any object that has its own lifecycle requirements (e.g. starts and stops some background process):

```
public interface Lifecycle {

    void start();

    void stop();

    boolean isRunning();

}
```

Any Spring-managed object may implement that interface. Then, when the `ApplicationContext` itself receives start and stop signals, e.g. for a stop/restart scenario at runtime, it will cascade those calls to all `Lifecycle` implementations defined within that context. It does this by delegating to a `LifecycleProcessor`:

```
public interface LifecycleProcessor extends Lifecycle {

    void onRefresh();
```

```
    void onClose();

}
```

Notice that the `LifecycleProcessor` is itself an extension of the `Lifecycle` interface. It also adds two other methods for reacting to the context being refreshed and closed.

> Note that the regular `org.springframework.context.Lifecycle` interface is just a plain contract for explicit start/stop notifications and does NOT imply auto-startup at context refresh time. Consider implementing `org.springframework.context.SmartLifecycle` instead for fine-grained control over auto-startup of a specific bean (including startup phases). Also, please note that stop notifications are not guaranteed to come before destruction: On regular shutdown, all `Lifecycle` beans will first receive a stop notification before the general destruction callbacks are being propagated; however, on hot refresh during a context's lifetime or on aborted refresh attempts, only destroy methods will be called.

The order of startup and shutdown invocations can be important. If a "depends-on" relationship exists between any two objects, the dependent side will start *after* its dependency, and it will stop *before* its dependency. However, at times the direct dependencies are unknown. You may only know that objects of a certain type should start prior to objects of another type. In those cases, the `SmartLifecycle` interface defines another option, namely the `getPhase()` method as defined on its super-interface, `Phased`.

```
public interface Phased {

    int getPhase();

}
```

```
public interface SmartLifecycle extends Lifecycle, Phased {

    boolean isAutoStartup();

    void stop(Runnable callback);

}
```

When starting, the objects with the lowest phase start first, and when stopping, the reverse order is followed. Therefore, an object that implements `SmartLifecycle` and whose `getPhase()` method returns `Integer.MIN_VALUE` would be among the first to start and the last to stop. At the other end of the spectrum, a phase value of `Integer.MAX_VALUE` would indicate that the object should be started last and stopped first (likely because it depends on other processes to be running). When considering the phase value, it's also important to know that the default phase for any "normal" `Lifecycle` object that does not implement `SmartLifecycle` would be 0. Therefore, any negative phase value would indicate that an object should start before those standard components (and stop after them), and vice versa for any positive phase value.

As you can see the stop method defined by `SmartLifecycle` accepts a callback. Any implementation *must* invoke that callback's `run()` method after that implementation's shutdown process is complete. That enables asynchronous shutdown where necessary since the default implementation of the `LifecycleProcessor` interface, `DefaultLifecycleProcessor`, will wait up to its timeout value for the group of objects within each phase to invoke that callback. The default per-phase timeout is 30 seconds. You can override the default lifecycle processor instance by defining a bean named "lifecycleProcessor" within the context. If you only want to modify the timeout, then defining the following would be sufficient:

```
<bean id="lifecycleProcessor" class="org.springframework.context.support.DefaultLi
    <!-- timeout value in milliseconds -->
    <property name="timeoutPerShutdownPhase" value="10000"/>
</bean>
```

As mentioned, the `LifecycleProcessor` interface defines callback methods for the refreshing and closing of the context as well. The latter will simply drive the shutdown process as if `stop()` had been called explicitly, but it will happen when the context is closing. The 'refresh' callback on the other hand enables another feature of `SmartLifecycle` beans. When the context is refreshed (after all objects have been instantiated and initialized), that callback will be invoked, and at that point the default lifecycle processor will check the boolean value returned by each `SmartLifecycle` object's `isAutoStartup()` method. If "true", then that object will be started at that point rather than waiting for an explicit invocation of the context's or its own `start()` method (unlike the context refresh, the context start does not happen automatically for a standard context implementation). The "phase" value as well as any "depends-on" relationships will determine the startup order in the same way as described above.

Shutting down the Spring IoC container gracefully in non-web applications

> This section applies only to non-web applications. Spring's web-based `ApplicationContext` implementations already have code in place to shut down the Spring IoC container gracefully when the relevant web application is shut down.

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you 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. Of course, you must still configure and implement these destroy callbacks correctly.

To register a shutdown hook, you call the `registerShutdownHook()` method that is declared on the `ConfigurableApplicationContext` interface:

```
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Boot {

    public static void main(final String[] args) throws Exception {

        ConfigurableApplicationContext ctx = new ClassPathXmlApplicationContext(
                new String []{"beans.xml"});

        // add a shutdown hook for the above context...
        ctx.registerShutdownHook();

        // app runs here...

        // main method exits, hook is called prior to the app shutting down...

    }
}
```

7.6.2 ApplicationContextAware and BeanNameAware

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`.

```
public interface ApplicationContextAware {

    void setApplicationContext(ApplicationContext applicationContext) throws Beans
```

```
}
```

Thus beans can manipulate programmatically 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. Sometimes this capability is useful; however, in general you should avoid it, because it couples the code to Spring and does not follow the Inversion of Control style, where collaborators are provided to beans as properties. Other methods of the `ApplicationContext` provide access to file resources, publishing application events, and accessing a `MessageSource`. These additional features are described in Section 7.15, “Additional Capabilities of the ApplicationContext”

As of Spring 2.5, autowiring is another alternative to obtain reference to the `ApplicationContext`. The "traditional" `constructor` and `byType` autowiring modes (as described in Section 7.4.5, “Autowiring collaborators”) can provide a dependency of type `ApplicationContext` for a constructor argument or setter method parameter, respectively. For more flexibility, including the ability to autowire fields and multiple parameter methods, use the new annotation-based autowiring features. If you do, the `ApplicationContext` is autowired into a field, constructor argument, or method parameter that is expecting the `ApplicationContext` type if the field, constructor, or method in question carries the `@Autowired` annotation. For more information, see Section 7.9.2, “@Autowired”.

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.

```
public interface BeanNameAware {

    void setBeanName(String name) throws BeansException;

}
```

The callback is invoked after population of normal bean properties but before an initialization callback such as `InitializingBean` *afterPropertiesSet* or a custom init-method.

7.6.3 Other Aware interfaces

Besides `ApplicationContextAware` and `BeanNameAware` discussed above, Spring offers a range of `Aware` interfaces that allow beans to indicate to the container that they require a certain

infrastructure dependency. The most important `Aware` interfaces are summarized below - as a general rule, the name is a good indication of the dependency type:

Table 7.4. Aware interfaces

Name	Injected Dependency	Explained in…
`ApplicationContextAware`	Declaring `ApplicationContext`	Section 7.6.2, “ApplicationContextAware and BeanNameAware”
`ApplicationEventPublisherAware`	Event publisher of the enclosing `ApplicationContext`	Section 7.15, “Additional Capabilities of the ApplicationContext”
`BeanClassLoaderAware`	Class loader used to load the bean classes.	Section 7.3.2, “Instantiating beans”
`BeanFactoryAware`	Declaring `BeanFactory`	Section 7.6.2, “ApplicationContextAware and BeanNameAware”
`BeanNameAware`	Name of the declaring bean	Section 7.6.2, “ApplicationContextAware and BeanNameAware”
`BootstrapContextAware`	Resource adapter `BootstrapContext` the container runs in. Typically available only in JCA aware `ApplicationContext`s	Chapter 32, *JCA CCI*
`LoadTimeWeaverAware`	Defined *weaver* for processing class definition at load time	Section 11.8.4, “Load-time weaving with AspectJ in the Spring Framework”

Name	Injected Dependency	Explained in…
`MessageSourceAware`	Configured strategy for resolving messages (with support for parametrization and internationalization)	Section 7.15, “Additional Capabilities of the ApplicationContext”
`NotificationPublisherAware`	Spring JMX notification publisher	Section 31.7, “Notifications”
`PortletConfigAware`	Current `PortletConfig` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 25, *Portlet MVC Framework*
`PortletContextAware`	Current `PortletContext` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 25, *Portlet MVC Framework*
`ResourceLoaderAware`	Configured loader for low-level access to resources	Chapter 8, *Resources*
`ServletConfigAware`	Current `ServletConfig` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 22, *Web MVC framework*
`ServletContextAware`	Current `ServletContext` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 22, *Web MVC framework*

Note again that usage of these interfaces ties your code to the Spring API and does not follow the Inversion of Control style. A