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.

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.

Dependency Injection and Inversion of Control

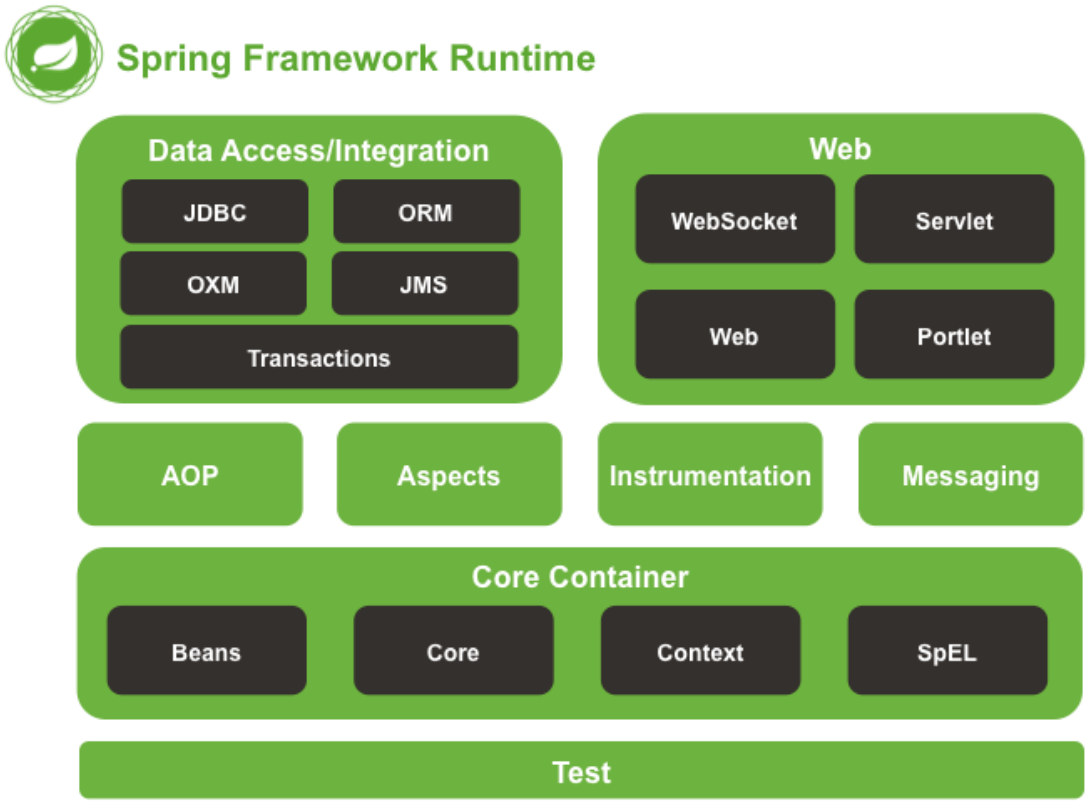
A Java application — a loose term that runs the gamut from constrained, embedded applications to n-tier, server-side enterprise applications — typically consists of objects that collaborate to form the application proper. Thus the objects in an application have *dependencies* on each other.

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.

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.

Overview of the Spring Framework



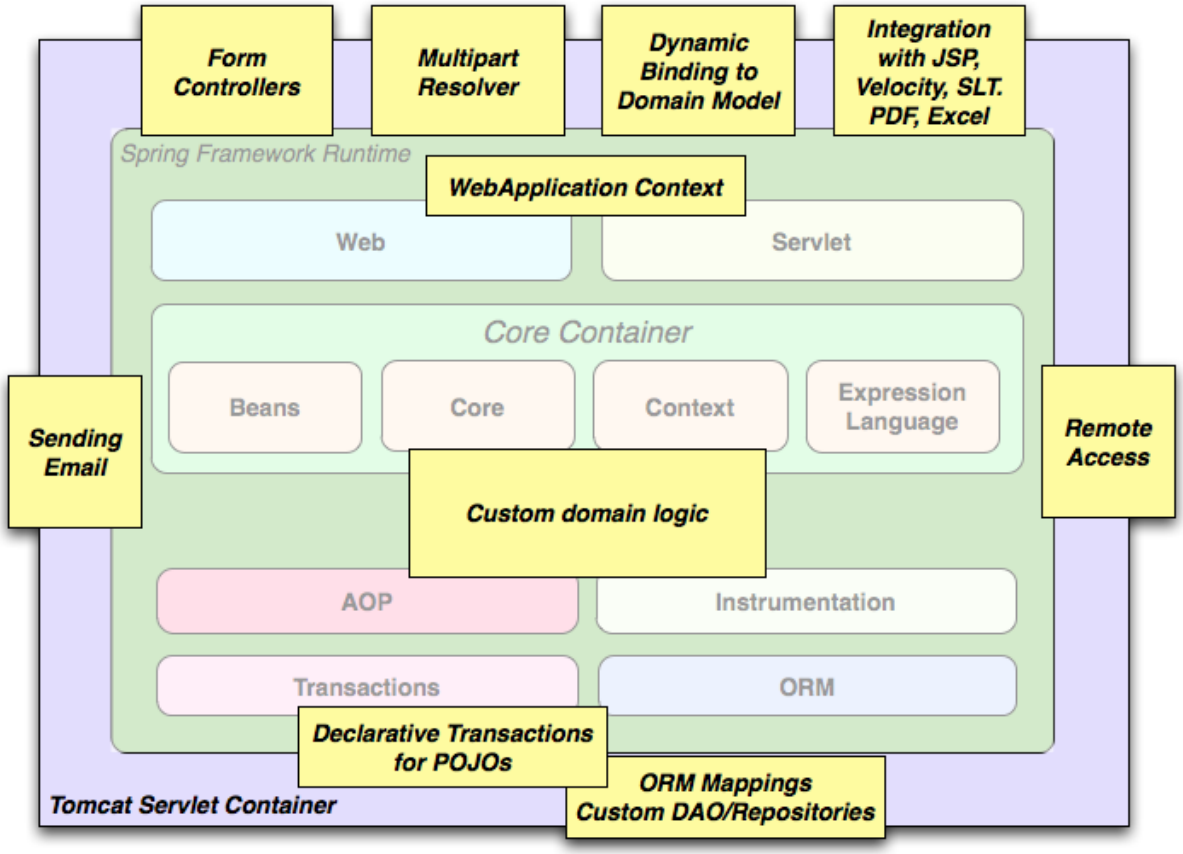
The spring-core and spring-beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration 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 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.

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.

Typical full-fledged Spring web application



Configuration metadata

Configuration metadata is traditionally supplied in a simple and intuitive XML format. 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>

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

</beans>
```

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.

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

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

You use `getBean()` to retrieve instances of your beans.

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.

The bean definition

Property	Means
class	specify the type (or class) of object that is to be instantiated in the `class` attribute
name	The id and/or name attributes to specify the bean identifier(s)
scope	Defines Scope – discussed later
constructor arguments	Constructor DI
properties	Setter DI
autowiring mode	Autowiring discussed later
lazy-initialization mode	discussed later
initialization method	Init method for Bean
destruction method	Destroy method for Bean

Instantiation with a static factory 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;
    }
```

```
}
```

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.

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`.
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`.

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.

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.

```
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.

```
ackage x.y;

public class Foo {

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

```
<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>
```

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>
```

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.

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 serviceis 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-method="close">
    <!-- 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. 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

```
<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>
```

References to other beans

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.

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

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 bean inline -->
    <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.

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>
```

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

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

Lazy-initialized beans

By default, `ApplicationContext` implementations eagerly create and configure all singleton beans as part of the initialization process.

When this behaviour 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"/>
```

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>
```

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`

When using XML-based configuration metadata you specify autowire mode for a bean definition with the `autowire` attribute of the `<bean/>` element. The autowiring functionality has four modes.

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.

Lifecycle callbacks

To interact with the container's management of the bean lifecycle, you can implement the Spring InitializingBean and `Disposable Bean` 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.

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

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...

    }
```

Bean definition inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on. A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed. Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.

```
<bean id="inheritedTestBean" abstract="true"
        class="org.springframework.beans.TestBean">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
        class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBean" init-method="initialize">
    <property name="name" value="override"/>
    <!-- the age property value of 1 will be inherited from parent -->
</bean>
```

The preceding example explicitly marks the parent bean definition as abstract by using the `abstract` attribute. If the parent definition does not specify a class, explicitly marking the parent bean definition as `abstract` is required, as follows:

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass"
class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBeanWithoutClass" init-
method="initialize">
    <property name="name" value="override"/>
    <!-- age will inherit the value of 1 from the parent bean
definition-->
</bean>
```

Customizing beans using a BeanPostProcessor

The `BeanPostProcessor` interface defines *callback methods* that you can implement to provide your own (or override the container's default) instantiation logic, dependency-resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more `BeanPostProcessor` implementations.

Find below the custom `BeanPostProcessor` implementation class definition:

```
package scripting;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InstantiationTracingBeanPostProcessor implements
BeanPostProcessor {

    // simply return the instantiated bean as-is
    public Object postProcessBeforeInitialization(Object bean,
            String beanName) throws BeansException {
        return bean; // we could potentially return any object
reference here...
    }

    public Object postProcessAfterInitialization(Object bean,
            String beanName) throws BeansException {
        System.out.println("Bean '" + beanName + "' created : " +
bean.toString());
        return bean;
    }

}
```

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

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

    <!--
    when the above bean (messenger) is instantiated, this custom
    BeanPostProcessor implementation will output the fact to the system
console
    -->
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

Notice how the `InstantiationTracingBeanPostProcessor` is simply defined. It does not even have a name, and because it is a bean it can be dependency-injected just like any other bean.

Annotation-based container configuration

Are annotations better than XML for configuring Spring?

The introduction of annotation-based configurations raised the question of whether this approach is 'better' than XML. The short answer is *it depends*. The long answer is that each approach has its pros and cons, and usually it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

Annotation injection is performed *before* XML injection, thus the latter configuration will override the former for properties wired through both approaches.

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

    <context:annotation-config/>

</beans>
```

@Required

The `@Required` annotation applies to bean property setter methods, as in the following example:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Required
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

@Autowired

You can apply the `@Autowired` annotation to constructors. you can also apply the `@Autowired` annotation to "traditional" setter methods. You can apply `@Autowired` to fields as well and even mix it with constructors:

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    private MovieCatalog movieCatalog;


    @Autowired
    public MovieRecommender(CustomerPreferenceDao
customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }

       @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

}
```

@Resource

`@Resource` takes a name attribute, and by default Spring interprets that value as the bean name to be injected.

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Resource(name="myMovieFinder")
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

}
```

If no name is specified explicitly, the default name is derived from the field name or setter method. In case of a field, it takes the field name; in case of a setter method, it takes the bean property name.

@PostConstruct and @PreDestroy

Methods analogous to init-method and destroy-method configuration.

```
public class CachingMovieLister {

    @PostConstruct
    public void populateMovieCache() {
        // populates the movie cache upon initialization...
    }

    @PreDestroy
    public void clearMovieCache() {
        // clears the movie cache upon destruction...
    }

}
```

@Component and further stereotype annotations

The `@Repository` annotation is a marker for any class that fulfills the role or *stereotype* of a repository (also known as Data Access Object or DAO). Spring provides further stereotype

annotations: `@Component`, `@Service`, and `@Controller`. `@Component` is a generic stereotype for any Spring-managed component.