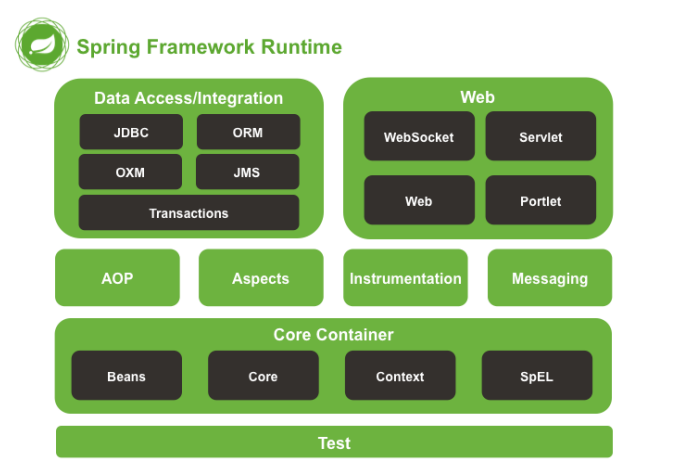
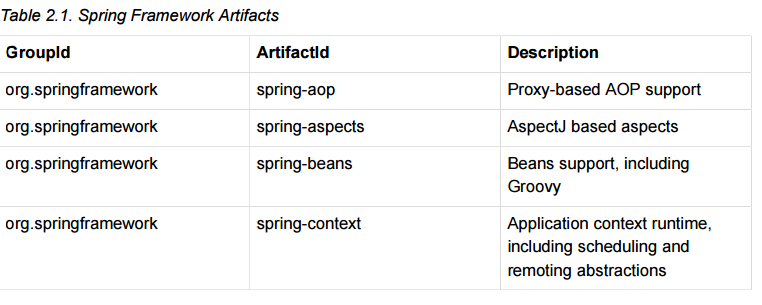
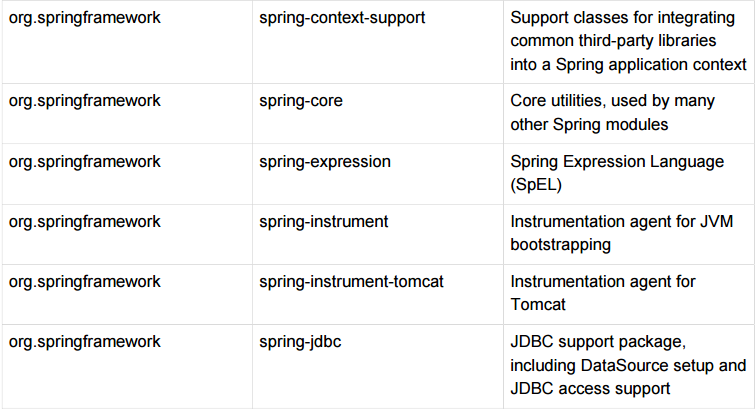
Spring framework (4.2.0)

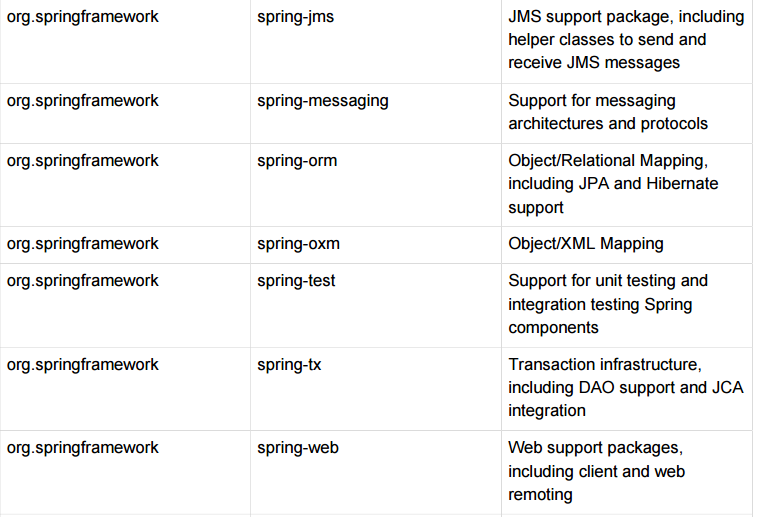
The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications.

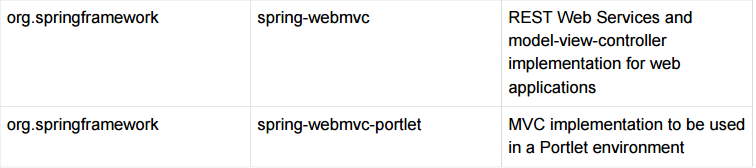


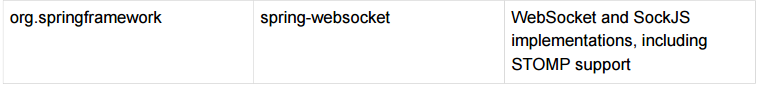
Core Container The Core Container consists of the spring-core, spring-beans, spring-context, springcontext-support, and spring-expression (Spring Expression Language) modules.











## Inversion of Control (IoC) 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

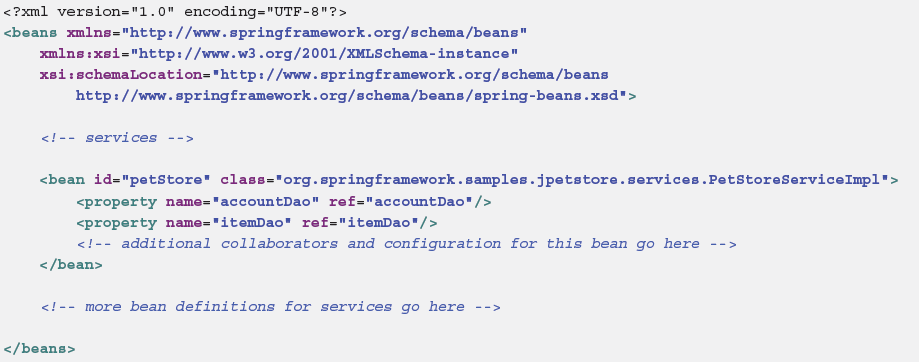
controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the *Service Locator* pattern.

The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of object. ApplicationContext is a subinterface 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,

## 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-thebox 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

The following example shows the basic structure of XML-based configuration metadata: 

**Instantiating a container**

ApplicationContext context =

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

**Composing XML-based configuration metadata**

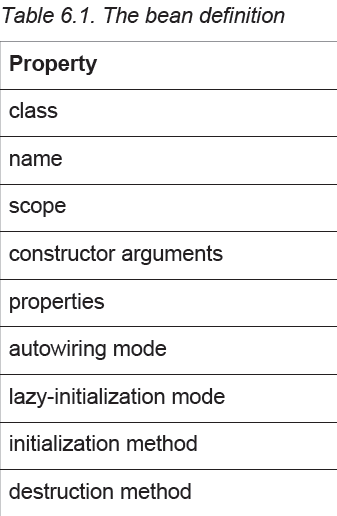


**Using the container**

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



**Bean overview**



**Aliasing a bean outside the bean definition**

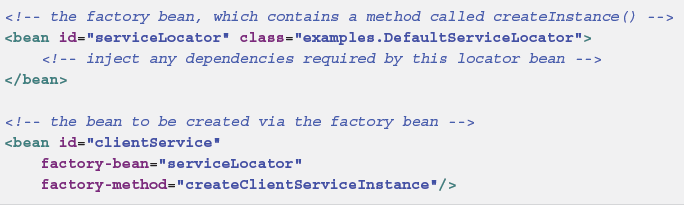
In a bean definition itself, you can supply more than one name for the bean, by using a combination of up to one name specified by the id attribute, and any number of other names in the name attribute. These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.

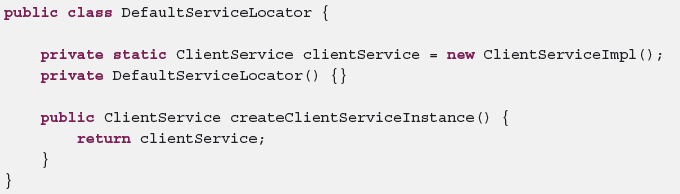
**<alias name**=**"fromName" alias**=**"toName"/>**

**Instantiation with a static factory method**



**Instantiation using an instance factory method**





**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.

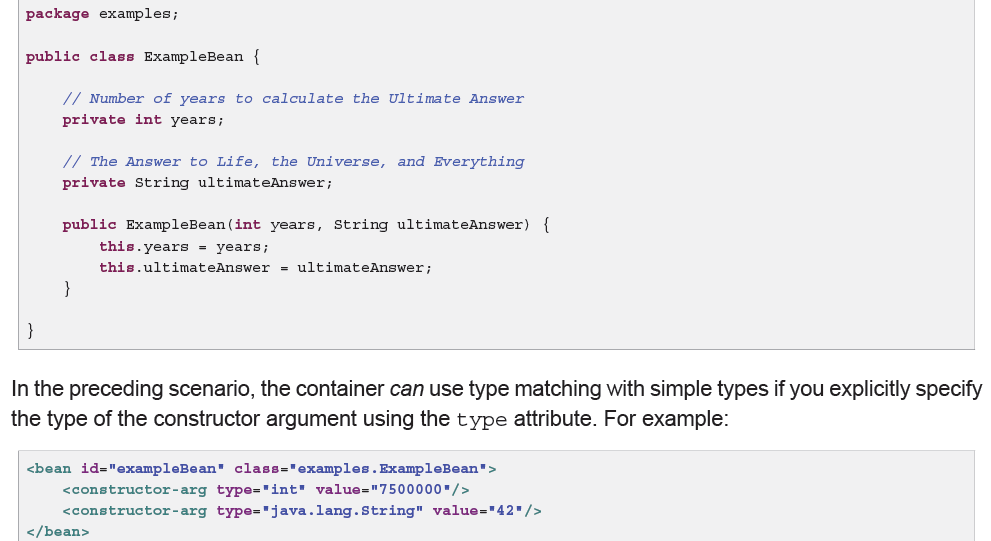
DI exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

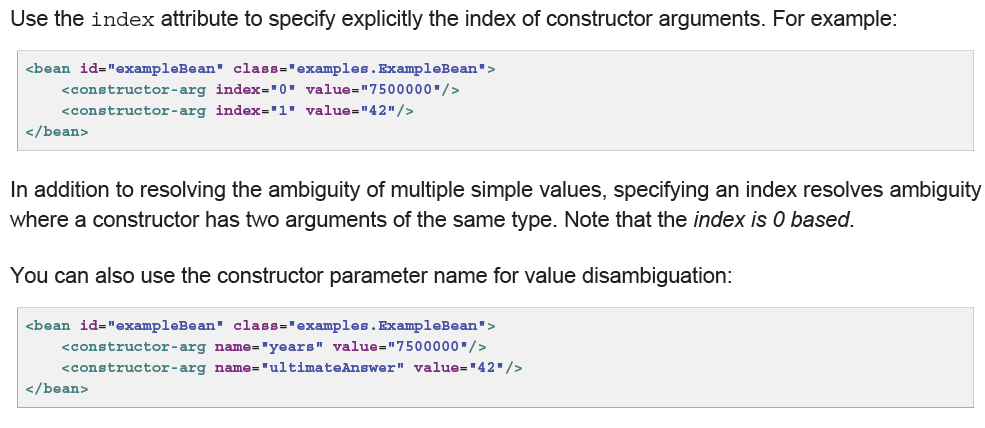
**Constructor-based dependency injection**

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.

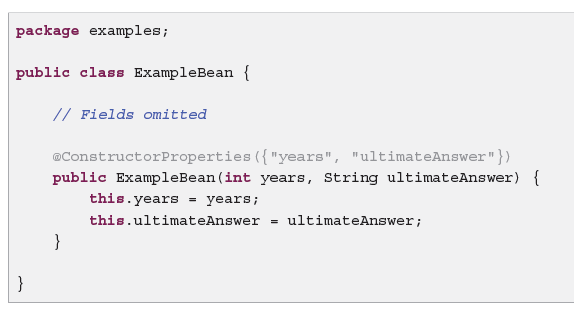
**Constructor argument resolution**

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



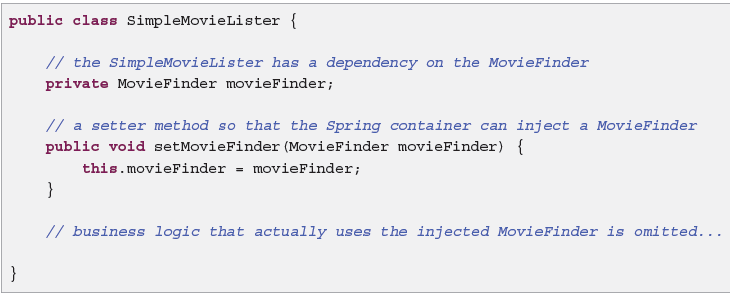


Keep in mind that to make this work out of the box your code must be compiled with the debug flag enabled so that Spring can look up the parameter name from the constructor. If you can’t compile your code with debug flag (or don’t want to) you can use @ConstructorProperties JDK annotation to explicitly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



**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.



**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.

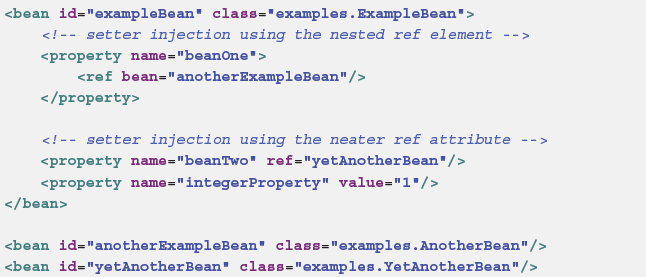
**Circular dependencies**

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.

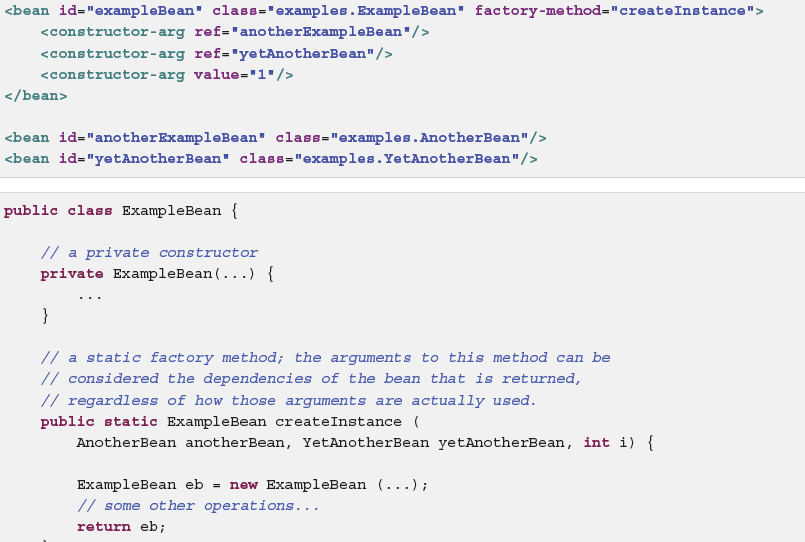
Example:



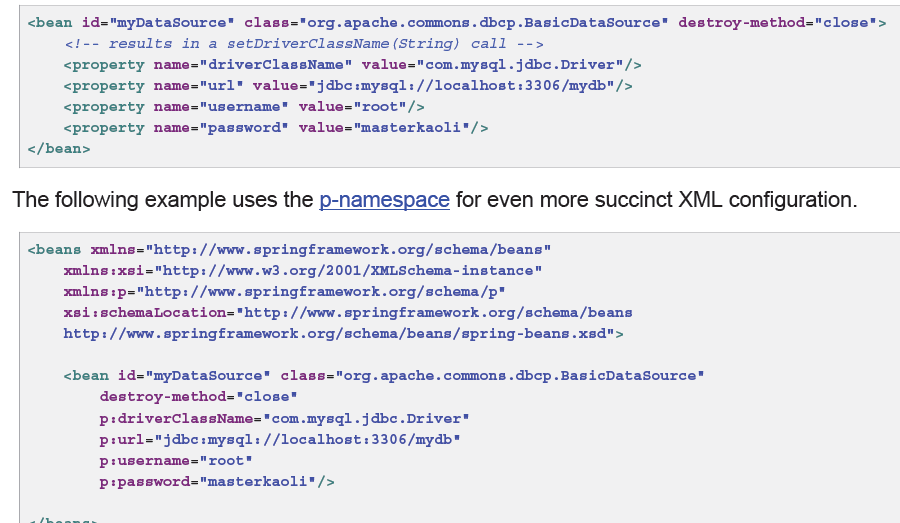


Now consider a variant of this example, where instead of using a constructor, Spring is told to call a

static factory method to return an instance of the object:



**Dependencies and configuration in detail**



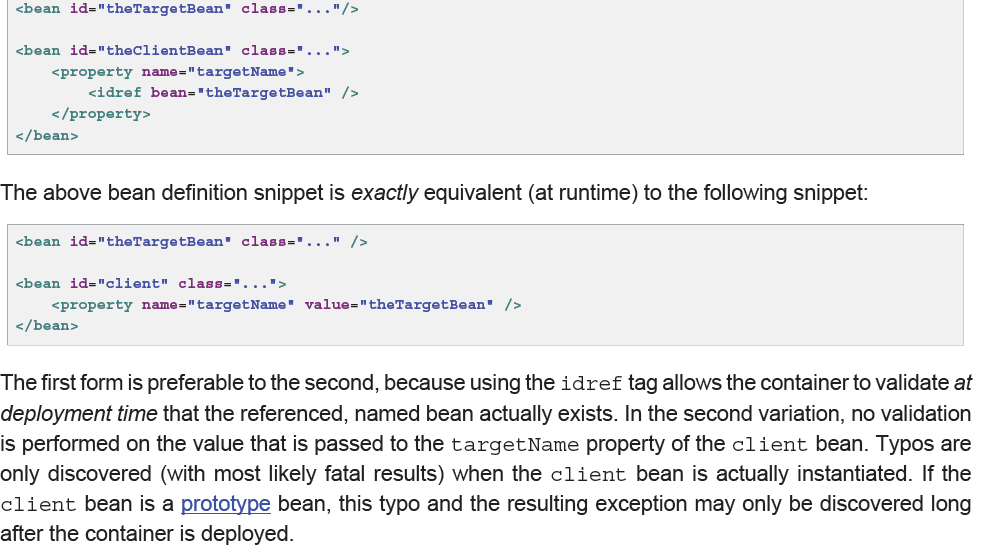
You can also configure a java.util.Properties instance as:



**The idref element**

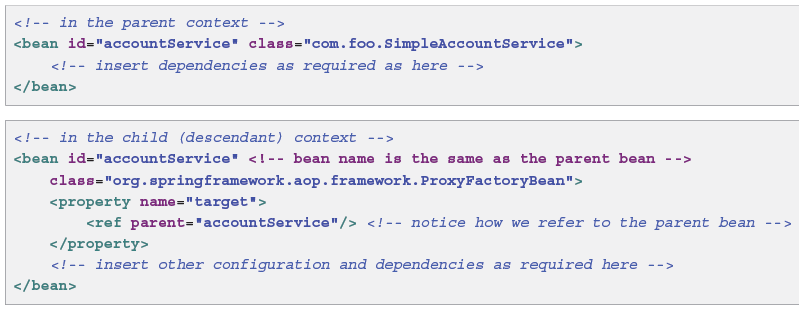
The idref element is simply an error-proof way to pass the *id* (string value - not a reference) of another

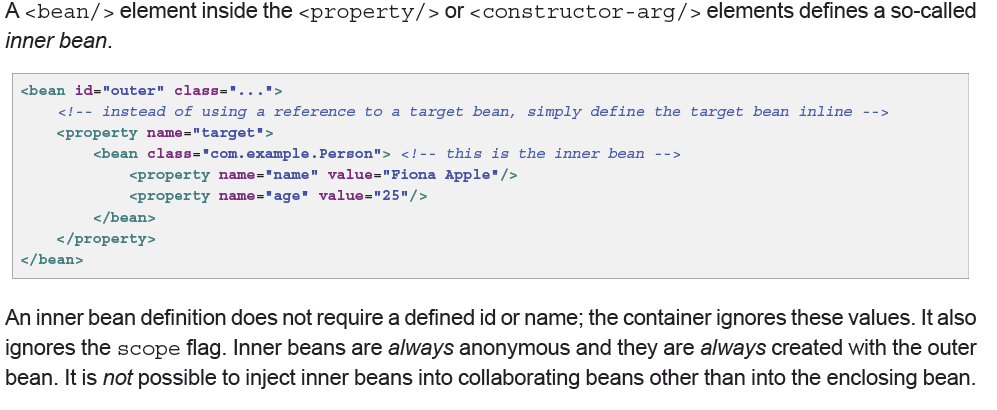
bean in the container to a <constructor-arg/> or <property/> element.



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.

****

****

**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.

****

**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.

****

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

**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 ("").

**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.



**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.



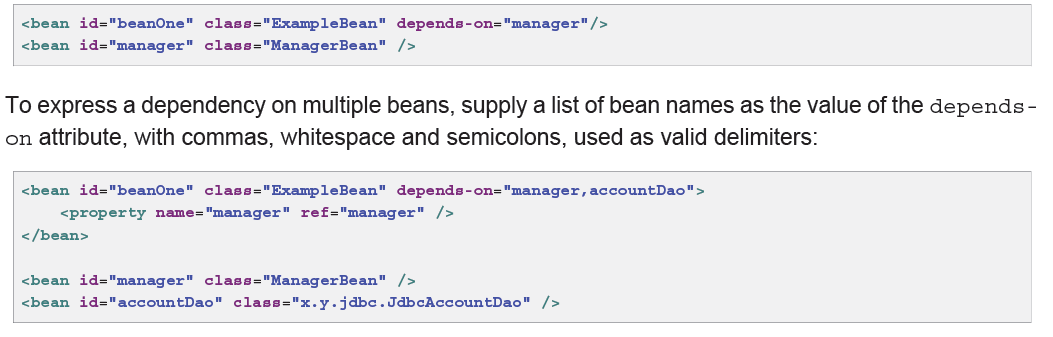
****

**Using depends-on**

The depends-on attribute can explicitly force one

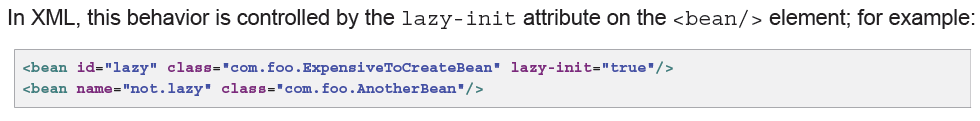
or more beans to be initialized before the bean using this element is initialized. The following example

uses the depends-on attribute to express a dependency on a single bean:

****

**Lazy-initialized beans**

A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

****

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:

****

**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

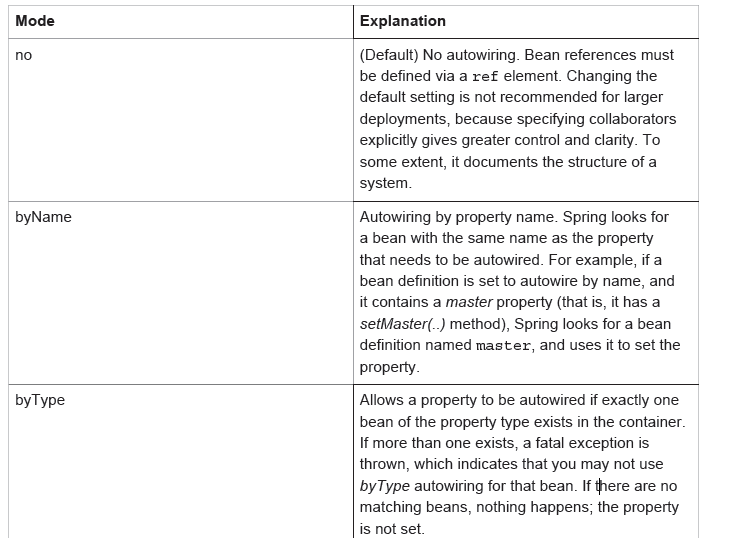
regard.)

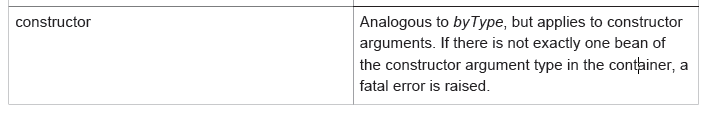
• Autowiring can update a configuration as your objects evolve. For example, if you need to add a

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.

****

****

**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.

**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).

**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

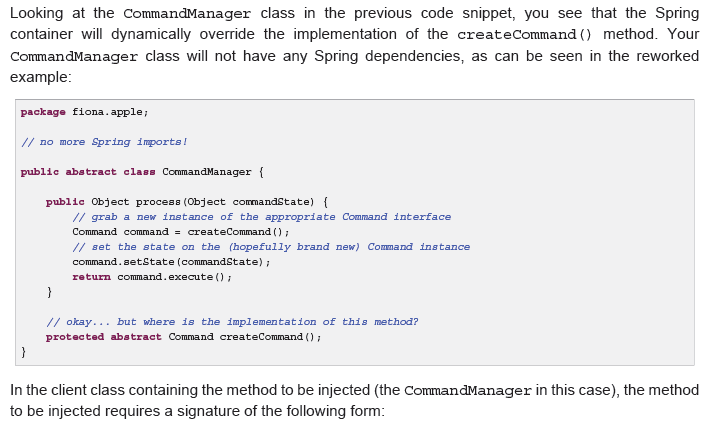
container ask for (a typically new) bean B instance every time bean A needs it. The following is an

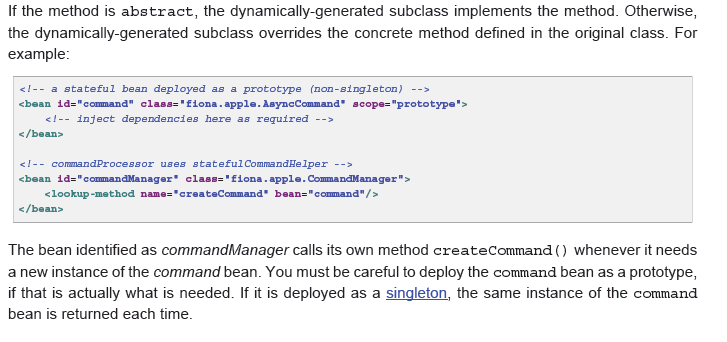
example of this approach:

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.

****

****

**6.5 Bean scopes**

****

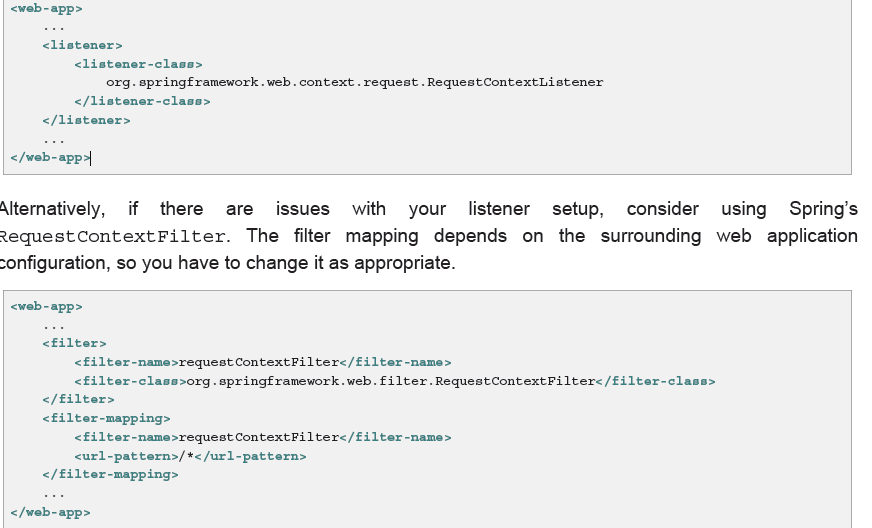
**Request, session, and global session scopes**

The request, session, and global session 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,

you get an IllegalStateException complaining about an unknown bean scope.

****

**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.

**Customizing the nature of a bean**

**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.

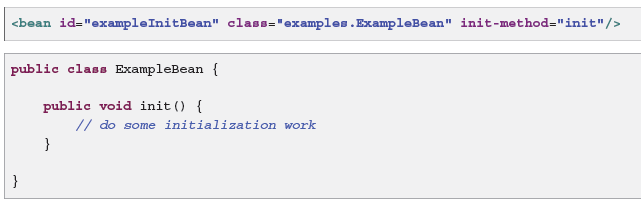
The JSR-250 @PostConstruct and @PreDestroy annotations are generally considered best

practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations

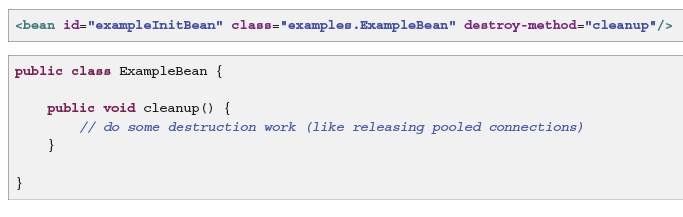
means that your beans are not coupled to Spring specific interfaces. For details see the section

called “@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.

**Initializing a bean**



**Destroying a Bean**



**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 initmethod="

init" attribute with each bean definition. The Spring IoC container calls that method

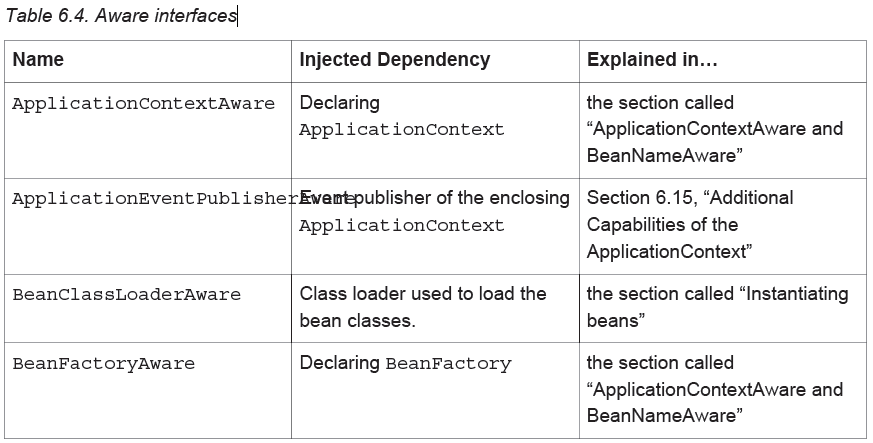
when the bean is created (and in accordance with the standard lifecycle callback contract described previously). This feature also enforces a consistent naming convention for initialization and destroy

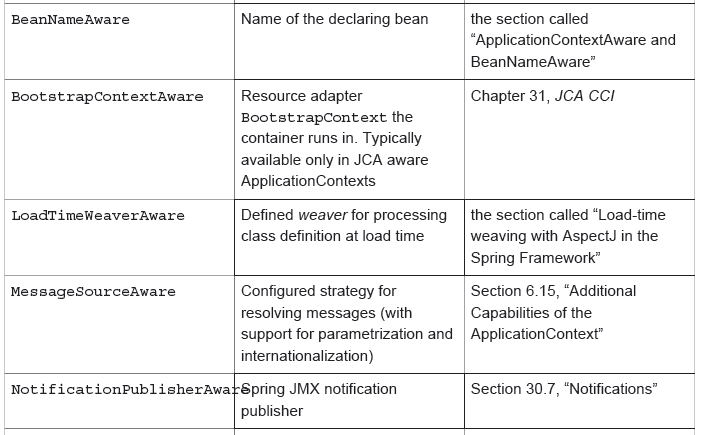
method callbacks.

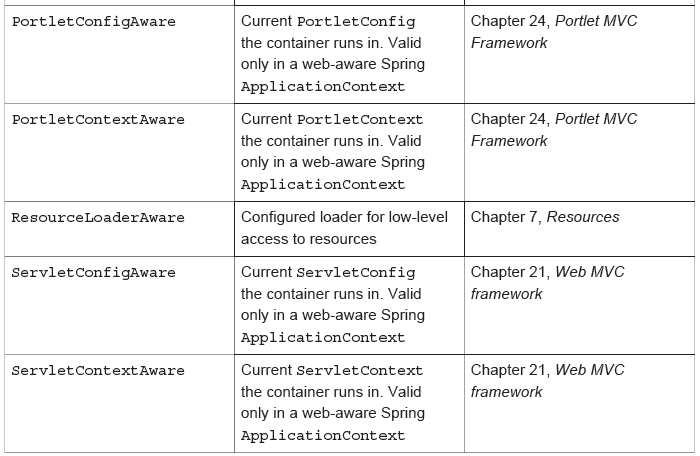
Suppose that your initialization callback methods are named init() and destroy callback methods are

named destroy(). Your class will resemble the class in the following example.

****

****

****

****

**6.8 Container Extension Points**

**Customizing beans using a BeanPostProcessor**

****

****

**Reading a values in the properties file.**

****

With the context namespace introduced in Spring 2.5, it is possible to configure property placeholders

with a dedicated configuration element. One or more locations can be provided as a comma-separated

list in the location attribute

**<context:property-placeholder location**=**"classpath:com/foo/jdbc.properties"/>**

**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.

**@Required -** This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring.

**@Autowired –** setter methods, method arguments and constructor specified as @Autowired will be automatically taken care by the container.

@Resource - The Resource annotation marks a resource that is needed by the application

**@PostConstruct and @PreDestroy –** Applied on methods which will be called for initializing and before destroying beans.

**@Component and further stereotype annotations**

Spring provides further stereotype annotations: @Component, @Service, and @Controller.

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service,

and @Controller are specializations of @Component for more specific use cases, for example,

in the persistence, service, and presentation layers, respectively. Therefore, you can annotate your

component classes with @Component, but by annotating them with @Repository, @Service, or

@Controller instead, your classes are more properly suited for processing by tools or associating

with aspects. For example, these stereotype annotations make ideal targets for pointcuts. It is also

possible that @Repository, @Service, and @Controller may carry additional semantics in future

releases of the Spring Framework.

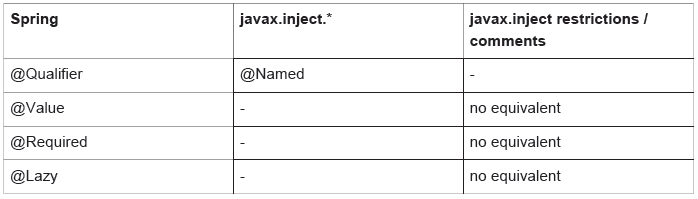
@Qualifier – To refer to the correct instance of the bean

Instead of @Autowired, @javax.inject.Inject may be used.

**@Named: a standard equivalent to the @Component annotation**

*Table 6.6. Spring annotations vs. standard annotations*





**7. Resources**

Java’s standard java.net.URL class and standard handlers for various URL prefixes unfortunately are

not quite adequate enough for all access to low-level resources. For example, there is no standardized

URL implementation that may be used to access a resource that needs to be obtained from the classpath, or relative to a ServletContext.

**Built-in Resource implementations**

**UrlResource**

The UrlResource wraps a java.net.URL, and may be used to access any object that is normally

accessible via a URL, such as files, an HTTP target, an FTP target, etc.

Ex:

Resource template = ctx.getResource(***"http://myhost.com/resource/path/myTemplate.txt"***);

**ClassPathResource**

This class represents a resource which should be obtained from the classpath.

Ex: Resource template = ctx.getResource(***"classpath:some/resource/path/myTemplate.txt"***);

**FileSystemResource**

This is a Resource implementation for java.io.File handles. It obviously supports resolution as a

File, and as a URL.

**ServletContextResource**

This is a Resource implementation for ServletContext resources, interpreting relative paths within

the relevant web application’s root directory.

**InputStreamResource**

A Resource implementation for a given InputStream. This should only be used if no specific

Resource implementation is applicable. In particular, prefer ByteArrayResource or any of the filebased

Resource implementations where possible.

**ByteArrayResource**

This is a Resource implementation for a given byte array. It creates a ByteArrayInputStream for

the given byte array.

**7.5 The ResourceLoaderAware interface**

The ResourceLoaderAware interface is a special marker interface, identifying objects that expect to

be provided with a ResourceLoader reference.

**7.7 Application contexts and Resource paths**

**Constructing application contexts**

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(***"conf/appContext.xml"***);

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext(***"conf/appContext.xml"***);

Note that the use of the special classpath prefix or a standard URL prefix on the location

path will override the default type of Resource created to load the definition. So this FileSystemXmlApplicationContext…

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext(***"classpath:conf/appContext.xml"***);

will actually load its bean definitions from the classpath. However, it is still a

FileSystemXmlApplicationContext. If it is subsequently used as a ResourceLoader, any

unprefixed paths will still be treated as filesystem paths.

**Validation, Data Binding, and Type Conversion**

There are pros and cons for considering validation as business logic, and Spring offers a design for

validation (and data binding) that does not exclude either one of them. Specifically validation should

not be tied to the web tier, should be easy to localize and it should be possible to plug in any validator

available. Considering the above, Spring has come up with a Validator interface that is both basic

and eminently usable in every layer of an application.

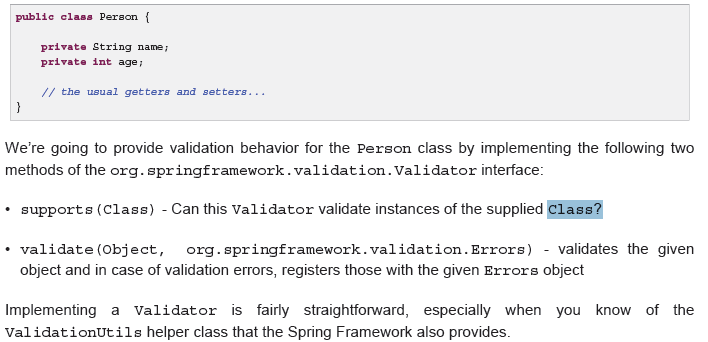
Data binding is useful for allowing user input to be dynamically bound to the domain model of

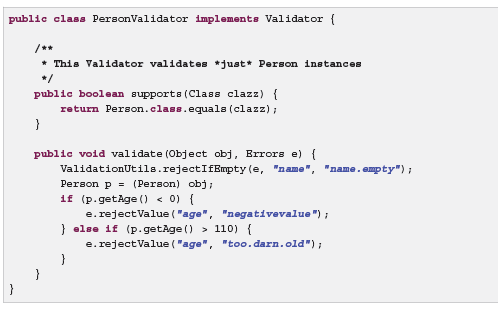
an application (or whatever objects you use to process user input). Spring provides the so-called

DataBinder to do exactly that. The Validator and the DataBinder make up the validation

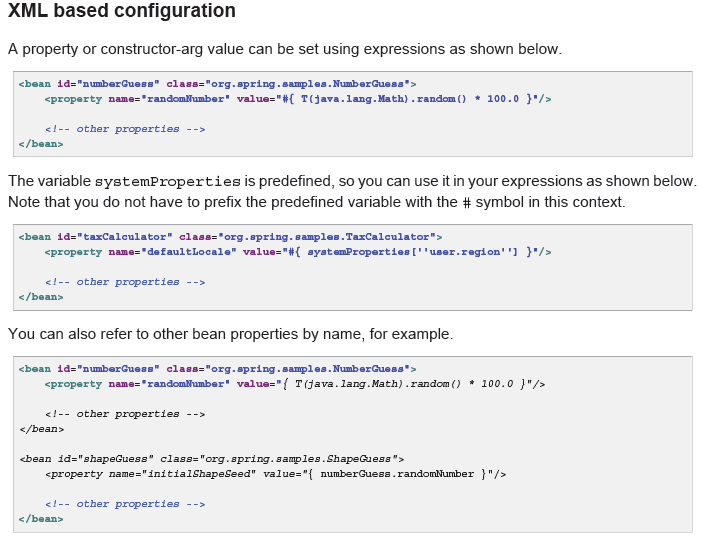
package, which is primarily used in but not limited to the MVC framework.

Ex:





**9. Spring Expression Language (SpEL)**

****

**Aspect Oriented Programming with Spring**

The key unit of modularity in OOP is the class, whereas

in AOP the unit of modularity is the *aspect*. Aspects enable the modularization of concerns such as

transaction management that cut across multiple types and objects.

AOP complements Spring IoC to provide a very capable middleware solution. AOP is used in the Spring Framework to provide declarative enterprise services.

**AOP concepts**

* *Aspect*: a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation (the @AspectJ style).
* *Join point*: a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always* represents a method execution.
* *Advice*: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors *around* the join point.

• *Pointcut*: a predicate that matches join points. Advice is associated with a pointcut expression and

runs at any join point matched by the pointcut (for example, the execution of a method with a certain

name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring

uses the AspectJ pointcut expression language by default.

• *Introduction*: declaring additional methods or fields on behalf of a type. Spring AOP allows you to

introduce new interfaces (and a corresponding implementation) to any advised object. For example,

you could use an introduction to make a bean implement an IsModified interface, to simplify

caching. (An introduction is known as an inter-type declaration in the AspectJ community.)

• *Target object*: object being advised by one or more aspects. Also referred to as the *advised* object.

Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object.

• *AOP proxy*: an object created by the AOP framework in order to implement the aspect contracts

(advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic

proxy or a CGLIB proxy.

• *Weaving*: linking aspects with other application types or objects to create an advised object. This can

be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring

AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

• *Before advice*: Advice that executes before a join point, but which does not have the ability to prevent

execution flow proceeding to the join point (unless it throws an exception).

• *After returning advice*: Advice to be executed after a join point completes normally: for example, if a

method returns without throwing an exception.

• *After throwing advice*: Advice to be executed if a method exits by throwing an exception.

• *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal

or exceptional return).

• *Around advice*: Advice that surrounds a join point such as a method invocation. This is the most

powerful kind of advice. Around advice can perform custom behavior before and after the method

invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the

advised method execution by returning its own return value or throwing an exception.

**Part IV. Testing**

**13.1 Mock Objects**

**Environment**

MockEnvironment and MockPropertySource are

useful for developing *out-of-container* tests for code that depends on environment-specific properties

**JNDI**

The org.springframework.mock.jndi package contains an implementation of the JNDI SPI,

which you can use to set up a simple JNDI environment for test suites or stand-alone applications.

**Servlet API**

The org.springframework.mock.web package contains a comprehensive set of Servlet API mock

objects, which are useful for testing web contexts, controllers, and filters.

**Context management and caching**

The Spring TestContext Framework provides consistent loading of Spring ApplicationContexts and

WebApplicationContexts as well as caching of those contexts.

By default, once loaded, the configured ApplicationContext is reused for each test. Thus the

setup cost is incurred only once per test suite, and subsequent test execution is much faster.

**Transaction management**

The TestContext framework addresses this issue. By default, the framework will create and roll back a

transaction for each test. You simply write code that can assume the existence of a transaction. If you

call transactionally proxied objects in your tests, they will behave correctly, according to their configured

transactional semantics. In addition, if a test method deletes the contents of selected tables while running

within the transaction managed for the test, the transaction will roll back by default, and the database

will return to its state prior to execution of the test. Transactional support is provided to a test via a

PlatformTransactionManager bean defined in the test’s application context.

**14.3 JDBC Testing Support**

The org.springframework.test.jdbc package contains JdbcTestUtils, which is a collection

of JDBC related utility functions intended to simplify standard database testing scenarios. Specifically,

JdbcTestUtils provides the following static utility methods.

• countRowsInTable(..): counts the number of rows in the given table

• countRowsInTableWhere(..): counts the number of rows in the given table, using the provided

WHERE clause

• deleteFromTables(..): deletes all rows from the specified tables

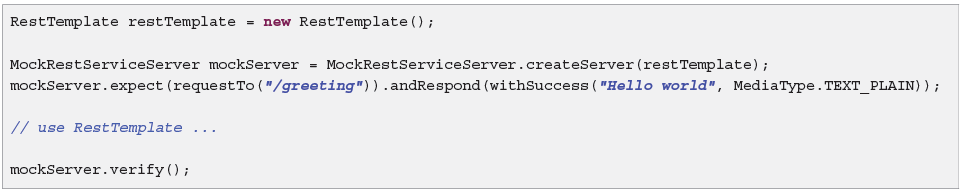
• deleteFromTableWhere(..): deletes rows from the given table, using the provided WHERE clause

• dropTables(..): drops the specified tables

**Client-Side REST Tests**

Client-side tests are for code using the RestTemplate. The goal is to define expected requests and

provide "stub" responses:



**Part V. Data Access**

**16. Transaction Management**

The Spring Framework provides a consistent abstraction for transaction management that

delivers the following benefits:

• Consistent programming model across different transaction APIs such as Java Transaction API (JTA),

JDBC, Hibernate, Java Persistence API (JPA), and Java Data Objects (JDO).

• Support for declarative transaction management.

• Simpler API for programmatic transaction management than complex transaction APIs such as JTA.

• Excellent integration with Spring’s data access abstractions.

**Global transactions**

Global transactions enable you to work with multiple transactional resources, typically relational

databases and message queues. The application server manages global transactions through the

JTA, which is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA

UserTransaction normally needs to be sourced from JNDI, meaning that you *also* need to use JNDI

in order to use JTA. Obviously the use of global transactions would limit any potential reuse of application

code, as JTA is normally only available in an application server environment.

**Local transactions**

Local transactions are resource-specific, such as a transaction associated with a JDBC connection.

Local transactions may be easier to use, but have significant disadvantages: they cannot work

across multiple transactional resources. For example, code that manages transactions using a JDBC

connection cannot run within a global JTA transaction. Because the application server is not involved in

transaction management, it cannot help ensure correctness across multiple resources. (It is worth noting

that most applications use a single transaction resource.) Another downside is that local transactions

are invasive to the programming model

**Spring Framework’s consistent programming model**

You write your code once, and it can benefit

from different transaction management strategies in different environments. The Spring Framework

provides both declarative and programmatic transaction management

The key to the Spring transaction abstraction is the notion of

a *transaction strategy*. A transaction strategy is defined by the

org.springframework.transaction.PlatformTransactionManager interface:

Again in keeping with Spring’s philosophy, the TransactionException that can be thrown by any

of the PlatformTransactionManager interface’s methods is *unchecked* (that is, it extends the

java.lang.RuntimeException class). Transaction infrastructure failures are almost invariably fatal.

In rare cases where application code can actually recover from a transaction failure, the application

developer can still choose to catch and handle TransactionException. The salient point is that

developers are not *forced* to do so.

The getTransaction(..) method returns a TransactionStatus object, depending on a

TransactionDefinition parameter. The returned TransactionStatus might represent a new

transaction, or can represent an existing transaction if a matching transaction exists in the current

call stack. The implication in this latter case is that, as with Java EE transaction contexts, a

TransactionStatus is associated with a *thread* of execution.

The TransactionDefinition interface specifies:

• *Isolation*: The degree to which this transaction is isolated from the work of other transactions. For

example, can this transaction see uncommitted writes from other transactions?

• *Propagation*: Typically, all code executed within a transaction scope will run in that transaction.

However, you have the option of specifying the behavior in the event that a transactional method

is executed when a transaction context already exists. For example, code can continue running in the existing transaction (the common case); or the existing transaction can be suspended and a new

transaction created. *Spring offers all of the transaction propagation options familiar from EJB CMT*.

To read about the semantics of transaction propagation in Spring, see the section called “Transaction

propagation”.

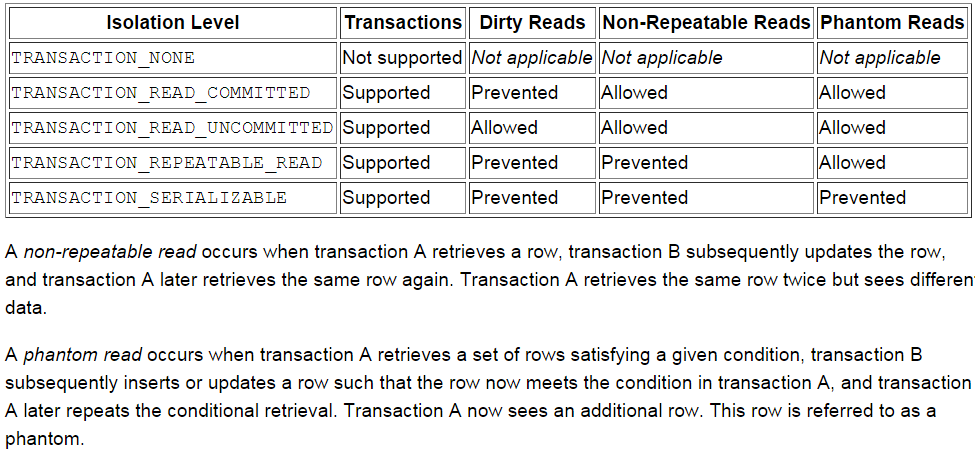
• *Timeout*: How long this transaction runs before timing out and being rolled back automatically by the

underlying transaction infrastructure.

• *Read-only status*: A read-only transaction can be used when your code reads but does not modify

data. Read-only transactions can be a useful optimization in some cases, such as when you are using

Hibernate.

****

Dirty Read: Accessing an updated value that has not been committed is considered a dirty read because it is possible for that value to be rolled back to its previous value. If you read a value that is later rolled back, you will have read an invalid value.

**Transaction propagation**

PROPAGATION\_REQUIRED

When the propagation setting is PROPAGATION\_REQUIRED, a *logical* transaction scope is created for

each method upon which the setting is applied. Each such logical transaction scope can determine

rollback-only status individually, with an outer transaction scope being logically independent from the

inner transaction scope. Of course, in case of standard PROPAGATION\_REQUIRED behavior, all these

scopes will be mapped to the same physical transaction. So a rollback-only marker set in the inner

transaction scope does affect the outer transaction’s chance to actually commit (as you would expect

it to).

PROPAGATION\_REQUIRES\_NEW, in contrast to PROPAGATION\_REQUIRED, uses a *completely*

independent transaction for each affected transaction scope

PROPAGATION\_NESTED uses a *single* physical transaction with multiple savepoints that it can roll back

to. Such partial rollbacks allow an inner transaction scope to trigger a rollback *for its scope*, with the

outer transaction being able to continue the physical transaction despite some operations having been

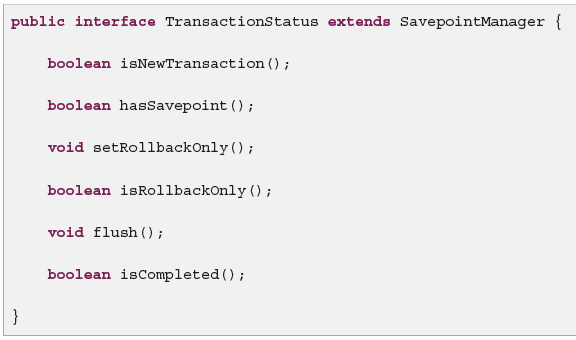
rolled back. This setting is typically mapped onto JDBC savepoints, so will only work with JDBC resource

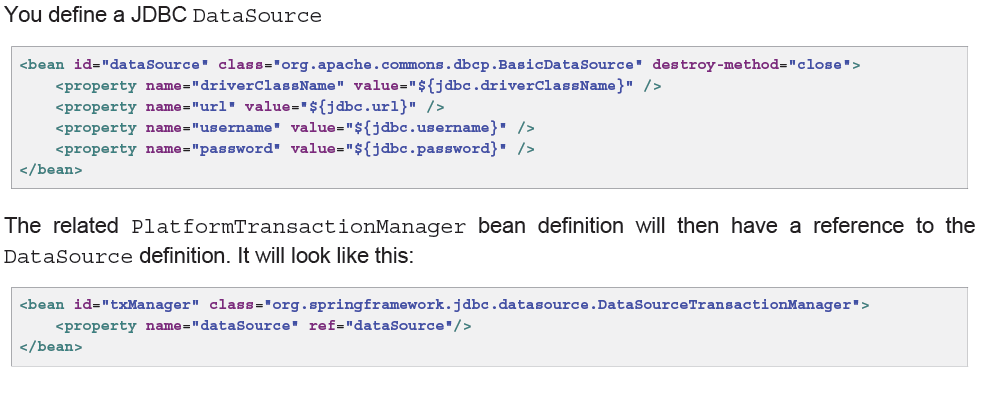
transactions. See Spring’s DataSourceTransactionManager.

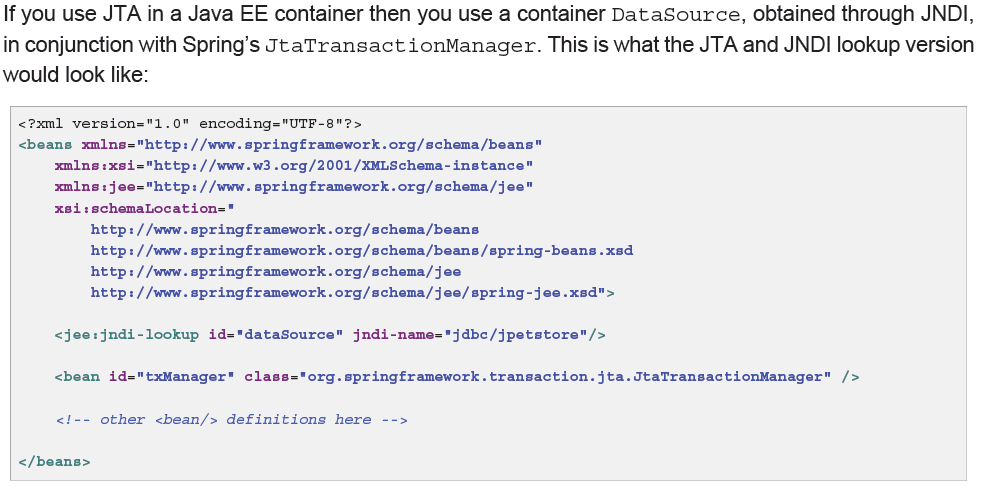
The TransactionStatus interface provides a simple way for transactional code to control transaction

execution and query transaction status. The concepts should be familiar, as they are common to all

transaction APIs:

****

****

****

**With Hibernate**

****

**High-level synchronization approach**

The preferred approach is to use Spring’s highest level template based persistence integration APIs or to

use native ORM APIs with transaction- aware factory beans or proxies for managing the native resource

factories. These transaction-aware solutions internally handle resource creation and reuse, cleanup,

optional transaction synchronization of the resources, and exception mapping. Thus user data access

code does not have to address these tasks, but can be focused purely on non-boilerplate persistence

logic. Generally, you use the native ORM API or take a *template* approach for JDBC access by using the

JdbcTemplate.

**Low-level synchronization approach**

Classes such as DataSourceUtils (for JDBC), EntityManagerFactoryUtils (for JPA),

SessionFactoryUtils (for Hibernate), PersistenceManagerFactoryUtils (for JDO), and so

on exist at a lower level. When you want the application code to deal directly with the resource types of

the native persistence APIs, you use these classes to ensure that proper Spring Framework-managed

instances are obtained, transactions are (optionally) synchronized, and exceptions that occur in the

process are properly mapped to a consistent API.

**TransactionAwareDataSourceProxy**

At the very lowest level exists the TransactionAwareDataSourceProxy class. This is a proxy for

a target DataSource, which wraps the target DataSource to add awareness of Spring-managed

transactions. In this respect, it is similar to a transactional JNDI DataSource as provided by a Java

EE server.

**16.5 Declarative transaction management**

The Spring Framework’s declarative transaction management is made possible with Spring aspectoriented programming (AOP), although, as the transactional aspects code comes with the Spring

Framework distribution and may be used in a boilerplate fashion, AOP concepts do not generally have

to be understood to make effective use of this code.

The differences between EJB CMT and sping declarative transaction management are,

• Unlike EJB CMT, which is tied to JTA, the Spring Framework’s declarative transaction management

works in any environment. It can work with JTA transactions or local transactions using JDBC, JPA,

Hibernate or JDO by simply adjusting the configuration files.

• You can apply the Spring Framework declarative transaction management to any class, not merely

special classes such as EJBs.

• The Spring Framework offers declarative *rollback rules*,a feature with no EJB equivalent. Both

programmatic and declarative support for rollback rules is provided.

• The Spring Framework enables you to customize transactional behavior, by using AOP. For example,

you can insert custom behavior in the case of transaction rollback. You can also add arbitrary advice,

along with the transactional advice. With EJB CMT, you cannot influence the container’s transaction

management except with setRollbackOnly().

• The Spring Framework does not support propagation of transaction contexts across remote calls, as

do high-end application servers. If you need this feature, we recommend that you use EJB. However,

consider carefully before using such a feature, because normally, one does not want transactions to

span remote calls.

Although EJB container default behavior automatically rolls back the transaction on a *system exception*

(usually a runtime exception), EJB CMT does not roll back the transaction automatically on an*application*

*exception* (that is, a checked exception other than java.rmi.RemoteException). While the Spring

default behavior for declarative transaction management follows EJB convention (roll back is automatic

only on unchecked exceptions), it is often useful to customize this behavior.

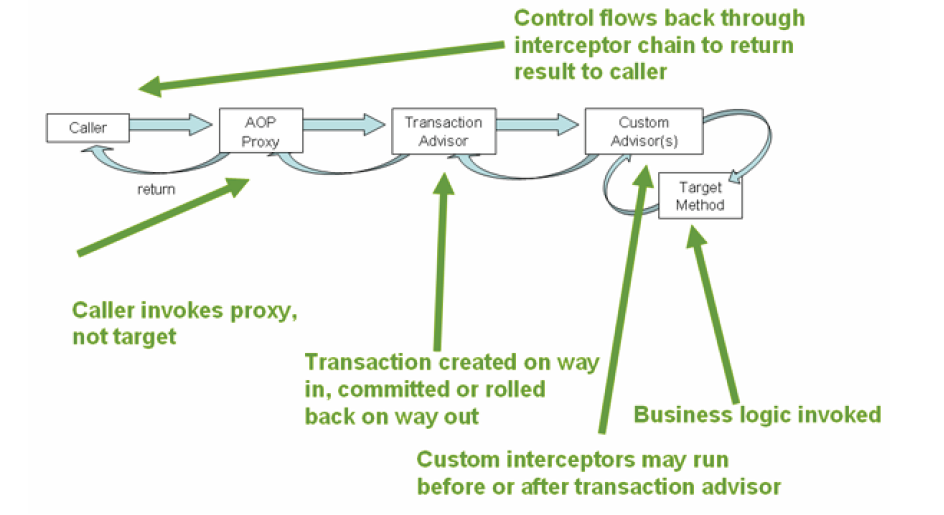
The most important concepts to grasp with regard to the Spring Framework’s declarative transaction

support are that this support is enabled *via AOP proxies*, and that the transactional advice is driven by

*metadata* (currently XML- or annotation-based). The combination of AOP with transactional metadata

yields an AOP proxy that uses a TransactionInterceptor in conjunction with an appropriate

PlatformTransactionManager implementation to drive transactions *around method invocations*.

****

**Example of declarative transaction implementation**

Consider the following interface, and its attendant implementation. This example uses Foo and Bar

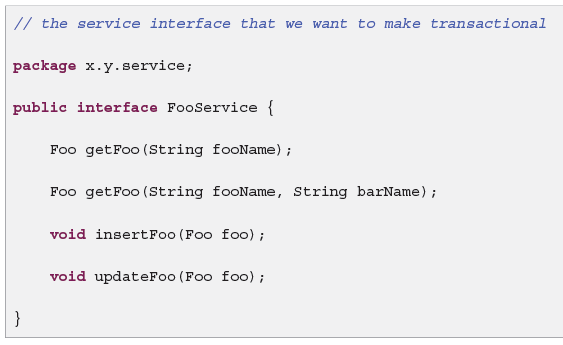
classes as placeholders so that you can concentrate on the transaction usage without focusing on a

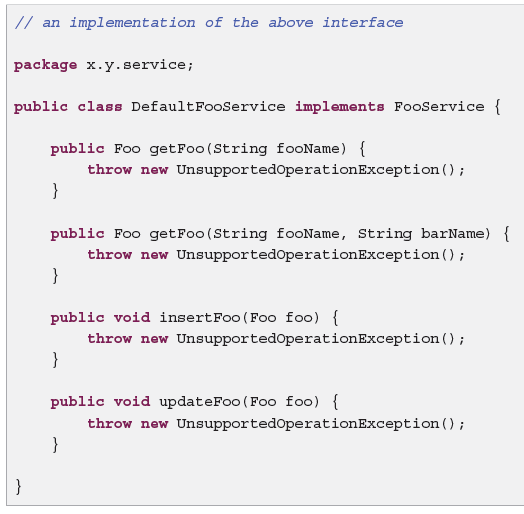
particular domain model. For the purposes of this example, the fact that the DefaultFooService

class throws UnsupportedOperationException instances in the body of each implemented

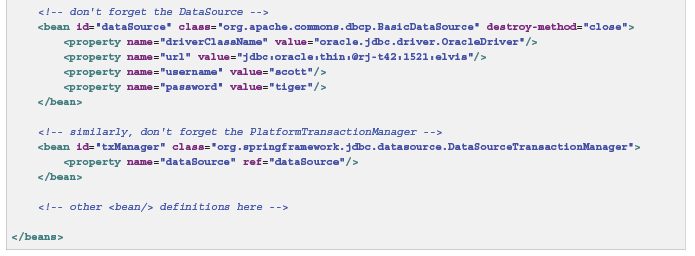
method is good; it allows you to see transactions created and then rolled back in response to the

UnsupportedOperationException instance.

****

****

****

****

**Rolling back a declarative transaction**

The recommended way to indicate to the Spring Framework’s transaction infrastructure that a

transaction’s work is to be rolled back is to throw an Exception from code that is currently executing

in the context of a transaction. The Spring Framework’s transaction infrastructure code will catch any

unhandled Exception as it bubbles up the call stack, and make a determination whether to mark the

transaction for rollback.

In its default configuration, the Spring Framework’s transaction infrastructure code *only* marks a

transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception

is an instance or subclass of RuntimeException. ( Errors will also - by default - result in a rollback).

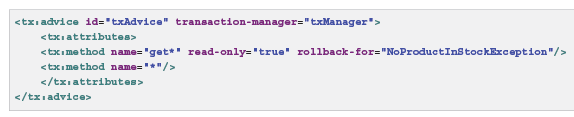
Checked exceptions that are thrown from a transactional method do *not* result in rollback in the default

configuration.

You can configure exactly which Exception types mark a transaction for rollback, including checked

exceptions. The following XML snippet demonstrates how you configure rollback for a checked,

application-specific Exception type.

****

You can also specify 'no rollback rules', if you do *not* want a transaction rolled back when an exception

is thrown. The following example tells the Spring Framework’s transaction infrastructure to commit the

attendant transaction even in the face of an unhandled InstrumentNotFoundException.

****

**16.6 Programmatic transaction management**

The Spring Framework provides two means of programmatic transaction management:

• Using the TransactionTemplate.

• Using a PlatformTransactionManager implementation directly.

The Spring team generally recommends the TransactionTemplate for programmatic transaction

management. The second approach is similar to using the JTA UserTransaction API, although

exception handling is less cumbersome.

**Using the TransactionTemplate**

The TransactionTemplate adopts the same approach as other Spring *templates* such as the

JdbcTemplate. It uses a callback approach, to free application code from having to do the boilerplate

acquisition and release of transactional resources, and results in code that is intention driven, in that

the code that is written focuses solely on what the developer wants to do.

****

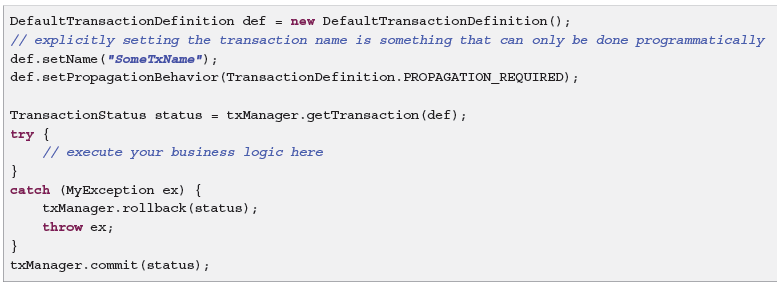
You can also use the org.springframework.transaction.PlatformTransactionManager

directly to manage your transaction. Simply pass the implementation of the

PlatformTransactionManager you are using to your bean through a bean reference. Then, using

the TransactionDefinition and TransactionStatus objects you can initiate transactions, roll

back, and commit.

****

**16.7 Choosing between programmatic and declarative**

**transaction management**

Programmatic transaction management is usually a good idea only if you have a small number of

transactional operations. For example, if you have a web application that require transactions only for

certain update operations, you may not want to set up transactional proxies using Spring or any other

technology. In this case, using the TransactionTemplate *may* be a good approach. Being able to

set the transaction name explicitly is also something that can only be done using the programmatic

approach to transaction management.

On the other hand, if your application has numerous transactional operations, declarative transaction

management is usually worthwhile. It keeps transaction management out of business logic, and is not

difficult to configure. When using the Spring Framework, rather than EJB CMT, the configuration cost

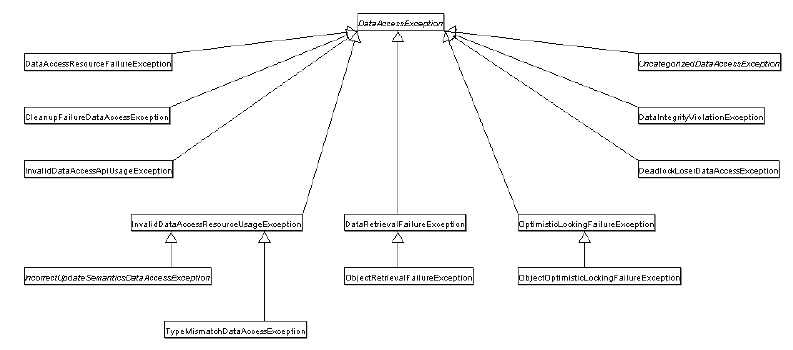
of declarative transaction management is greatly reduced

**17. DAO support**

**17.2 Consistent exception hierarchy**

Spring provides a convenient translation from technology-specific exceptions like SQLException to

its own exception class hierarchy with the DataAccessException as the root exception.

****

**17.3 Annotations used for configuring DAO or Repository**

**Classes**

The best way to guarantee that your Data Access Objects (DAOs) or repositories provide exception

translation is to use the @Repository annotation. This annotation also allows the component scanning

support to find and configure your DAOs and repositories without having to provide XML configuration

entries for them.

**18. Data access with JDBC**

**18.1 Introduction to Spring Framework JDBC**

**Choosing an approach for JDBC database access**

• *JdbcTemplate* is the classic Spring JDBC approach and the most popular. This "lowest level"

approach and all others use a JdbcTemplate under the covers.

• *NamedParameterJdbcTemplate* wraps a JdbcTemplate to provide named parameters instead of

the traditional JDBC "?" placeholders. This approach provides better documentation and ease of use

when you have multiple parameters for an SQL statement.

• *SimpleJdbcInsert and SimpleJdbcCall* optimize database metadata to limit the amount of necessary

configuration. This approach simplifies coding so that you only need to provide the name of the table

or procedure and provide a map of parameters matching the column names. This only works if the

database provides adequate metadata. If the database doesn’t provide this metadata, you will have

to provide explicit configuration of the parameters.

• *RDBMS Objects including MappingSqlQuery, SqlUpdate and StoredProcedure* requires you to create

reusable and thread-safe objects during initialization of your data access layer. This approach is

modeled after JDO Query wherein you define your query string, declare parameters, and compile

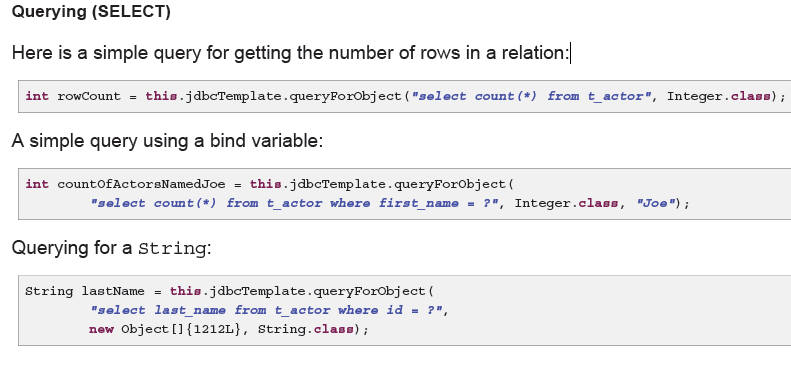
the query. Once you do that, execute methods can be called multiple times with various parameter

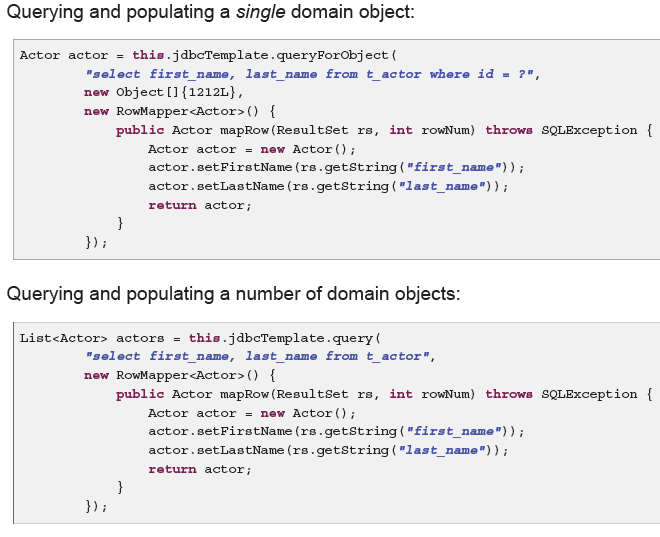
values passed in.

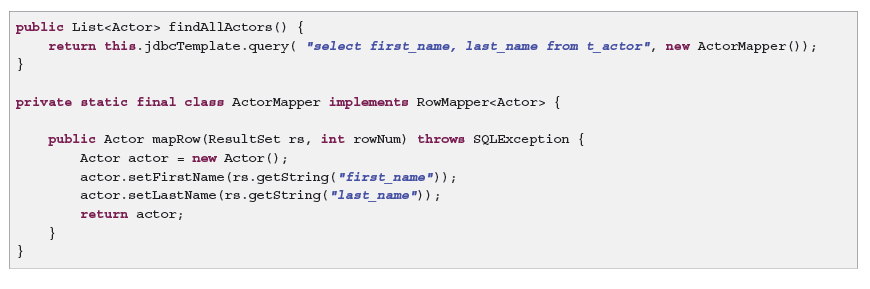
**18.2 Using the JDBC core classes to control basic JDBC**

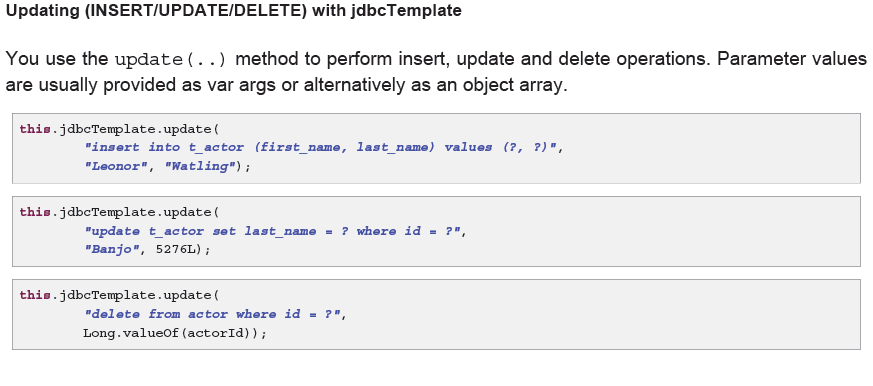
**processing and error handling**

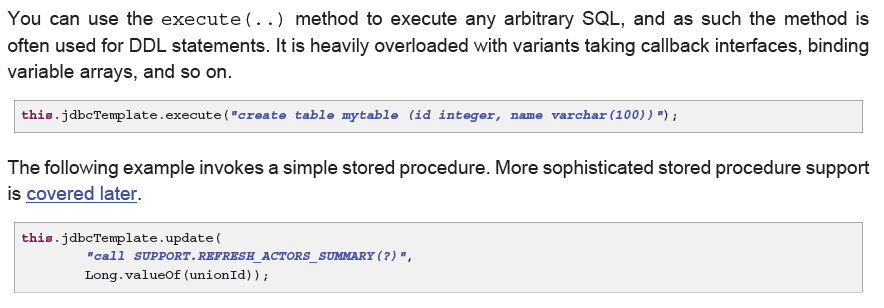
**Ex:**

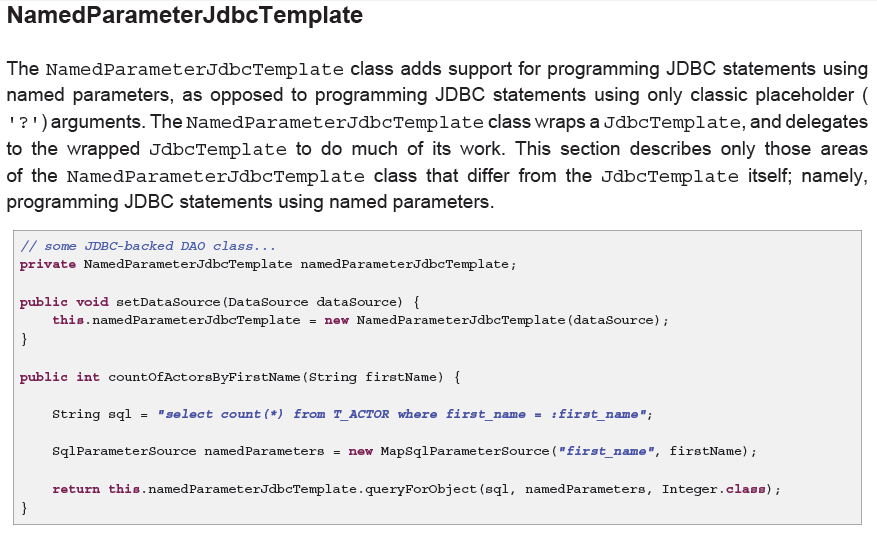
****

****

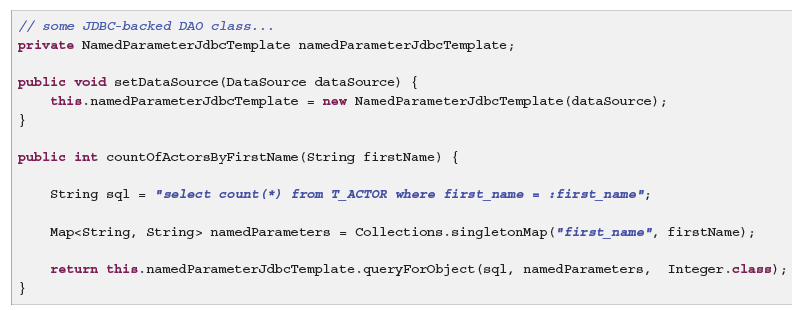
****

****

****

****

The following example shows the use of the Map-based style.

****

**Inserting data using SimpleJdbcInsert**

****

**19. Object Relational Mapping (ORM) Data Access**

The Spring Framework supports integration with Hibernate, Java Persistence API (JPA) and Java Data

Objects (JDO) for resource management, data access object (DAO) implementations, and transaction

strategies.

Benefits of using the Spring Framework to create your ORM DAOs include:

• *Easier testing.* Spring’s IoC approach makes it easy to swap the implementations and configuration

locations of Hibernate SessionFactory instances, JDBC DataSource instances, transaction

managers, and mapped object implementations (if needed). This in turn makes it much easier to test

each piece of persistence-related code in isolation.

• *Common data access exceptions.* Spring can wrap exceptions from your ORM tool, converting

them from proprietary (potentially checked) exceptions to a common runtime DataAccessException

hierarchy. This feature allows you to handle most persistence exceptions, which are non-recoverable,

only in the appropriate layers, without annoying boilerplate catches, throws, and exception

declarations. You can still trap and handle exceptions as necessary. Remember that JDBC exceptions

(including DB-specific dialects) are also converted to the same hierarchy, meaning that you can

perform some operations with JDBC within a consistent programming model.

• *General resource management.* Spring application contexts can handle the location and configuration

of Hibernate SessionFactory instances, JPA EntityManagerFactory instances, JDBC

DataSource instances, and other related resources. This makes these values easy to manage and

change. Spring offers efficient, easy, and safe handling of persistence resources. For example, related

code that uses Hibernate generally needs to use the same Hibernate Session to ensure efficiency

and proper transaction handling. Spring makes it easy to create and bind a Session to the current

thread transparently, by exposing a current Session through the Hibernate SessionFactory. Thus

Spring solves many chronic problems of typical Hibernate usage, for any local or JTA transaction

environment.

• *Integrated transaction management.* You can wrap your ORM code with a declarative, aspect-oriented

programming (AOP) style method interceptor either through the @Transactional annotation or

by explicitly configuring the transaction AOP advice in an XML configuration file. In both cases,

transaction semantics and exception handling (rollback, and so on) are handled for you. As discussed below, in Resource and transaction management, you can also swap various transaction managers,without affecting your ORM-related code. For example, you can swap between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios.

Additionally, JDBC-related code can fully integrate transactionally with the code you use to do ORM.

This is useful for data access that is not suitable for ORM, such as batch processing and BLOB

streaming, which still need to share common transactions with ORM operations.

**Exception translation**

When you use Hibernate, JPA, or JDO in a DAO, you must decide how to handle the persistence

technology’s native exception classes. The DAO throws a subclass of a HibernateException,

PersistenceException or JDOException depending on the technology. These exceptions are

all run-time exceptions and do not have to be declared or caught. You may also have to deal with

IllegalArgumentException and IllegalStateException. This means that callers can only

treat exceptions as generally fatal, unless they want to depend on the persistence technology’s own

exception structure. Catching specific causes such as an optimistic locking failure is not possible without

tying the caller to the implementation strategy. This trade off might be acceptable to applications that

are strongly ORM-based and/or do not need any special exception treatment. However, Spring enables

exception translation to be applied transparently through the @Repository annotation:

@Repository

**public class** ProductDaoImpl **implements** ProductDao {

*// class body here...*

}

**<beans>**

*<!-- Exception translation bean post processor -->*

**<bean class**=**"org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>**

**<bean id**=**"myProductDao" class**=**"product.ProductDaoImpl"/>**

**</beans>**

The postprocessor automatically looks for all exception translators (implementations of the

PersistenceExceptionTranslator interface) and advises all beans marked with the

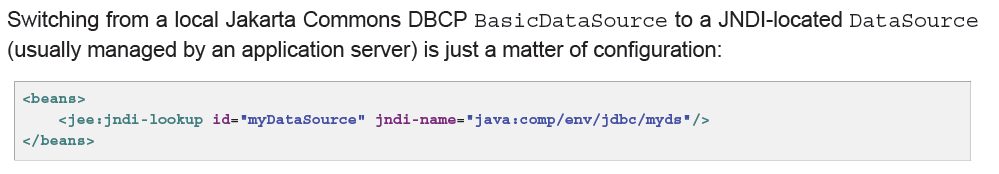
@Repository annotation so that the discovered translators can intercept and apply the appropriate

translation on the thrown exceptions.

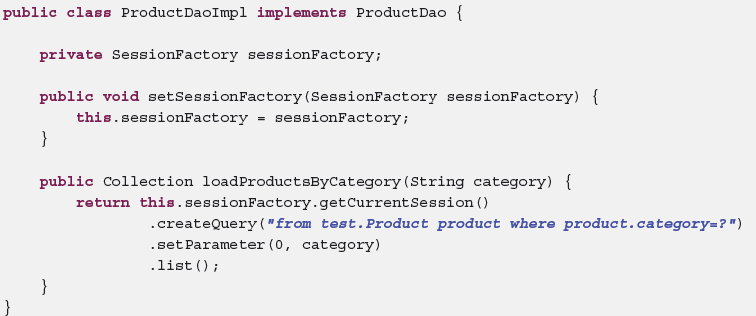
**19.3 Hibernate**

**SessionFactory setup in a Spring container**

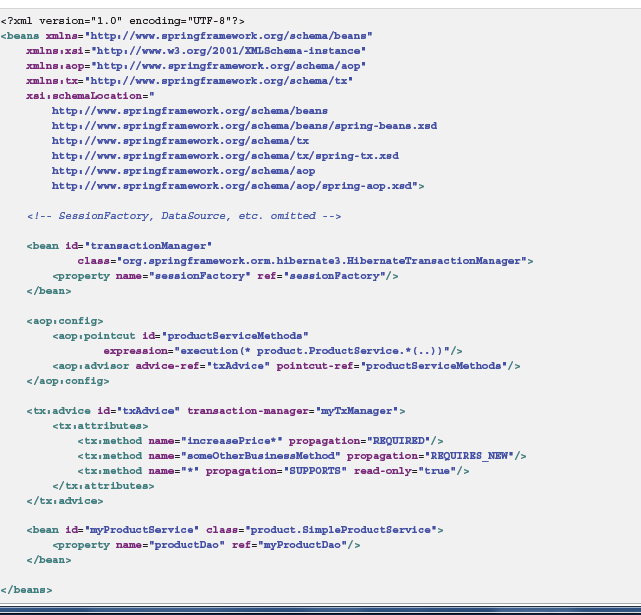


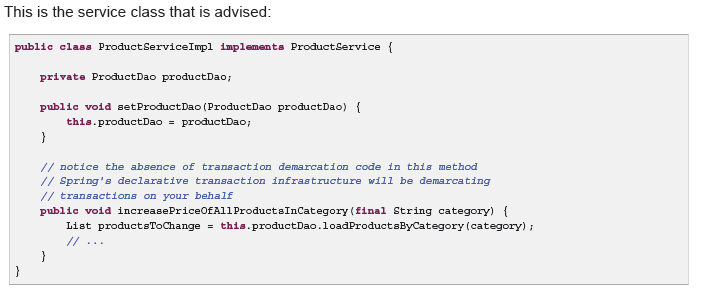


**Implementing DAOs based on plain Hibernate 3 API**



**Declarative transaction demarcation**



****

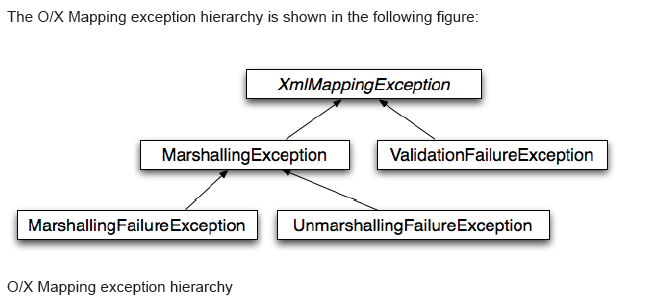
**Programmatic transaction demarcation**

****

**20. Marshalling XML using O/X Mappers**

**20.2 Marshaller and Unmarshaller**

A *marshaller* serializes an object to XML, and an *unmarshaller* deserializes XML stream to an object.

****

**Ex:**

****

****

****

****

Currently, the following tags are available:

• jaxb2-marshaller

• xmlbeans-marshaller

• castor-marshaller

• jibx-marshaller

**25. WebSocket**

The WebSocket protocol RFC 6455 defines an important new capability for web applications: full-duplex,

two-way communication between client and server. It is an exciting new capability on the heels of a

long history of techniques to make the web more interactive including Java Applets, XMLHttpRequest,

Adobe Flash, ActiveXObject, various Comet techniques, server-sent events, and others.

**Different types of advice examples**

[**http://www.mkyong.com/spring/spring-aop-examples-advice/**](http://www.mkyong.com/spring/spring-aop-examples-advice/)

**References:**

**http://docs.spring.io/spring-framework/docs/4.2.0.RELEASE/spring-framework-reference/pdf/spring-framework-reference.pdf**