Appendices

Version 5.0.0.RELEASE

Table of Contents

Chapter 1. What’s New in the Spring Framework

"What’s New" guides for releases of the Spring Framework are now provided as a Wiki page.

Chapter 2. Migrating to Spring Framework 4.3 / 5.0

Migration guides for upgrading from previous releases of the Spring Framework are now provided as a Wiki page.

Chapter 3. Spring Annotation Programming Model

Spring's annotation programming model is documented in the Spring Framework Wiki.

Chapter 4. Classic Spring Usage

This appendix discusses some classic Spring usage patterns as a reference for developers maintaining legacy Spring applications. These usage patterns no longer reflect the recommended way of using these features, and the current recommended usage is covered in the respective sections of the reference manual.

4.1. Classic ORM usage

This section documents the classic usage patterns that you might encounter in a legacy Spring application. For the currently recommended usage patterns, please refer to the ORM chapter.

4.1.1. Hibernate

For the currently recommended usage patterns for Hibernate see the Hibernate section.

The HibernateTemplate

The basic programming model for templating looks as follows, for methods that can be part of any custom data access object or business service. There are no restrictions on the implementation of the surrounding object at all, it just needs to provide a Hibernate `SessionFactory`. It can get the latter from anywhere, but preferably as bean reference from a Spring IoC container - via a simple `setSessionFactory(..)` bean property setter. The following snippets show a DAO definition in a Spring container, referencing the above defined `SessionFactory`, and an example for a DAO method implementation.

```
<beans>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private HibernateTemplate hibernateTemplate;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }

    public Collection loadProductsByCategory(String category) throws
DataAccessException {
        return this.hibernateTemplate.find("from test.Product product where
product.category=?", category);
    }
}
```

The `HibernateTemplate` class provides many methods that mirror the methods exposed on the Hibernate `Session` interface, in addition to a number of convenience methods such as the one shown above. If you need access to the `Session` to invoke methods that are not exposed on the `HibernateTemplate`, you can always drop down to a callback-based approach like so.

```
public class ProductDaoImpl implements ProductDao {

    private HibernateTemplate hibernateTemplate;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }

    public Collection loadProductsByCategory(final String category) throws
DataAccessException {
        return this.hibernateTemplate.execute(new HibernateCallback() {
            public Object doInHibernate(Session session) {
                Criteria criteria = session.createCriteria(Product.class);
                criteria.add(Expression.eq("category", category));
                criteria.setMaxResults(6);
                return criteria.list();
            }
        };
    }

}
```

A callback implementation effectively can be used for any Hibernate data access. `HibernateTemplate` will ensure that `Session` instances are properly opened and closed, and automatically participate in transactions. The template instances are thread-safe and reusable, they can thus be kept as instance variables of the surrounding class. For simple single step actions like a single find, load, saveOrUpdate, or delete call, `HibernateTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient

`HibernateDaoSupport` base class that provides a `setSessionFactory(..)` method for receiving a `SessionFactory`, and `getSessionFactory()` and `getHibernateTemplate()` for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws
DataAccessException {
        return this.getHibernateTemplate().find(
            "from test.Product product where product.category=?", category);
    }

}
```

Implementing Spring-based DAOs without callbacks

As alternative to using Spring's `HibernateTemplate` to implement DAOs, data access code can also be written in a more traditional fashion, without wrapping the Hibernate access code in a callback, while still respecting and participating in Spring's generic `DataAccessException` hierarchy. The `HibernateDaoSupport` base class offers methods to access the current transactional `Session` and to convert exceptions in such a scenario; similar methods are also available as static helpers on the `SessionFactoryUtils` class. Note that such code will usually pass `false` as the value of the `getSession(..)` methods `allowCreate` argument, to enforce running within a transaction (which avoids the need to close the returned `Session`, as its lifecycle is managed by the transaction).

```
public class HibernateProductDao extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws
DataAccessException, MyException {
        Session session = getSession(false);
        try {
            Query query = session.createQuery("from test.Product product where
product.category=?");
            query.setString(0, category);
            List result = query.list();
            if (result == null) {
                throw new MyException("No search results.");
            }
            return result;
        }
        catch (HibernateException ex) {
            throw convertHibernateAccessException(ex);
        }
    }
}
```

The advantage of such direct Hibernate access code is that it allows *any* checked application exception to be thrown within the data access code; contrast this to the `HibernateTemplate` class

which is restricted to throwing only unchecked exceptions within the callback. Note that you can often defer the corresponding checks and the throwing of application exceptions to after the callback, which still allows working with `HibernateTemplate`. In general, the `HibernateTemplate` class' convenience methods are simpler and more convenient for many scenarios.

4.2. JMS Usage

One of the benefits of Spring's JMS support is to shield the user from differences between the JMS 1.0.2 and 1.1 APIs. (For a description of the differences between the two APIs see sidebar on Domain Unification). Since it is now common to encounter only the JMS 1.1 API the use of classes that are based on the JMS 1.0.2 API has been deprecated in Spring 3.0. This section describes Spring JMS support for the JMS 1.0.2 deprecated classes.

Domain Unification

There are two major releases of the JMS specification, 1.0.2 and 1.1.

JMS 1.0.2 defined two types of messaging domains, point-to-point (Queues) and publish/subscribe (Topics). The 1.0.2 API reflected these two messaging domains by providing a parallel class hierarchy for each domain. As a result, a client application became domain specific in its use of the JMS API. JMS 1.1 introduced the concept of domain unification that minimized both the functional differences and client API differences between the two domains. As an example of a functional difference that was removed, if you use a JMS 1.1 provider you can transactionally consume a message from one domain and produce a message on the other using the same `Session`.

The JMS 1.1 specification was released in April 2002 and incorporated as part of J2EE 1.4 in November 2003. As a result, common J2EE 1.3 application servers which are still in widespread use (such as BEA WebLogic 8.1 and IBM WebSphere 5.1) are based on JMS 1.0.2.

4.2.1. JmsTemplate

Located in the package `org.springframework.jms.core` the class `JmsTemplate102` provides all of the features of the `JmsTemplate` described the JMS chapter, but is based on the JMS 1.0.2 API instead of the JMS 1.1 API. As a consequence, if you are using JmsTemplate102 you need to set the boolean property `pubSubDomain` to configure the `JmsTemplate` with knowledge of what JMS domain is being used. By default the value of this property is false, indicating that the point-to-point domain, Queues, will be used.

4.2.2. Asynchronous Message Reception

MessageListenerAdapter's are used in conjunction with Spring's message listener containers to support asynchronous message reception by exposing almost any class as a Message-driven POJO. If you are using the JMS 1.0.2 API, you will want to use the 1.0.2 specific classes such as `MessageListenerAdapter102`, `SimpleMessageListenerContainer102`, and

`DefaultMessageListenerContainer102`. These classes provide the same functionality as the JMS 1.1 based counterparts but rely only on the JMS 1.0.2 API.

4.2.3. Connections

The `ConnectionFactory` interface is part of the JMS specification and serves as the entry point for working with JMS. Spring provides an implementation of the `ConnectionFactory` interface, `SingleConnectionFactory102`, based on the JMS 1.0.2 API that will return the same `Connection` on all `createConnection()` calls and ignore calls to `close()`. You will need to set the boolean property `pubSubDomain` to indicate which messaging domain is used as `SingleConnectionFactory102` will always explicitly differentiate between a `javax.jms.QueueConnection` and a `javax.jmsTopicConnection`.

4.2.4. Transaction Management

In a JMS 1.0.2 environment the class `JmsTransactionManager102` provides support for managing JMS transactions for a single Connection Factory. Please refer to the reference documentation on JMS Transaction Management for more information on this functionality.

Chapter 5. Classic Spring AOP Usage

In this appendix we discuss the lower-level Spring AOP APIs and the AOP support used in Spring 1.2 applications. For new applications, we recommend the use of the Spring 2.0 AOP support described in the AOP chapter, but when working with existing applications, or when reading books and articles, you may come across Spring 1.2 style examples. Spring 2.0 is fully backwards compatible with Spring 1.2 and everything described in this appendix is fully supported in Spring 2.0.

5.1. Pointcut API in Spring

Let's look at how Spring handles the crucial pointcut concept.

5.1.1. Concepts

Spring's pointcut model enables pointcut reuse independent of advice types. It's possible to target different advice using the same pointcut.

The `org.springframework.aop.Pointcut` interface is the central interface, used to target advices to particular classes and methods. The complete interface is shown below:

```
public interface Pointcut {

    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();

}
```

Splitting the `Pointcut` interface into two parts allows reuse of class and method matching parts, and fine-grained composition operations (such as performing a "union" with another method matcher).

The `ClassFilter` interface is used to restrict the pointcut to a given set of target classes. If the `matches()` method always returns true, all target classes will be matched:

```
public interface ClassFilter {

    boolean matches(Class clazz);

}
```

The `MethodMatcher` interface is normally more important. The complete interface is shown below:

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

    boolean matches(Method m, Class targetClass, Object[] args);

}
```

The `matches(Method, Class)` method is used to test whether this pointcut will ever match a given method on a target class. This evaluation can be performed when an AOP proxy is created, to avoid the need for a test on every method invocation. If the 2-argument matches method returns true for a given method, and the `isRuntime()` method for the MethodMatcher returns true, the 3-argument matches method will be invoked on every method invocation. This enables a pointcut to look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most MethodMatchers are static, meaning that their `isRuntime()` method returns false. In this case, the 3-argument matches method will never be invoked.

If possible, try to make pointcuts static, allowing the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

5.1.2. Operations on pointcuts

Spring supports operations on pointcuts: notably, *union* and *intersection*.

- Union means the methods that either pointcut matches.
- Intersection means the methods that both pointcuts match.
- Union is usually more useful.
- Pointcuts can be composed using the static methods in the *org.springframework.aop.support.Pointcuts* class, or using the *ComposablePointcut* class in the same package. However, using AspectJ pointcut expressions is usually a simpler approach.

5.1.3. AspectJ expression pointcuts

Since 2.0, the most important type of pointcut used by Spring is `org.springframework.aop.aspectj.AspectJExpressionPointcut`. This is a pointcut that uses an AspectJ supplied library to parse an AspectJ pointcut expression string.

See the previous chapter for a discussion of supported AspectJ pointcut primitives.

5.1.4. Convenience pointcut implementations

Spring provides several convenient pointcut implementations. Some can be used out of the box; others are intended to be subclassed in application-specific pointcuts.

Static pointcuts

Static pointcuts are based on method and target class, and cannot take into account the method's arguments. Static pointcuts are sufficient - *and best* - for most usages. It's possible for Spring to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Let's consider some static pointcut implementations included with Spring.

Regular expression pointcuts

One obvious way to specify static pointcuts is regular expressions. Several AOP frameworks besides Spring make this possible. `org.springframework.aop.support.Perl5RegexpMethodPointcut` is a generic regular expression pointcut, using Perl 5 regular expression syntax. The `Perl5RegexpMethodPointcut` class depends on Jakarta ORO for regular expression matching. Spring also provides the `JdkRegexpMethodPointcut` class that uses the regular expression support in JDK 1.4+.

Using the `Perl5RegexpMethodPointcut` class, you can provide a list of pattern Strings. If any of these is a match, the pointcut will evaluate to true. (So the result is effectively the union of these pointcuts.)

The usage is shown below:

```
<bean id="settersAndAbsquatulatePointcut"
        class="org.springframework.aop.support.Perl5RegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.<strong>set.</strong></value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

Spring provides a convenience class, `RegexpMethodPointcutAdvisor`, that allows us to also reference an Advice (remember that an Advice can be an interceptor, before advice, throws advice etc.). Behind the scenes, Spring will use a `JdkRegexpMethodPointcut`. Using `RegexpMethodPointcutAdvisor` simplifies wiring, as the one bean encapsulates both pointcut and advice, as shown below:

```
<bean id="settersAndAbsquatulateAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref bean="beanNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.<strong>set.</strong></value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

RegexpMethodPointcutAdvisor can be used with any Advice type.

Attribute-driven pointcuts

An important type of static pointcut is a *metadata-driven* pointcut. This uses the values of metadata attributes: typically, source-level metadata.

Dynamic pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method*arguments*, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

The main example is the `control flow` pointcut.

Control flow pointcuts

Spring control flow pointcuts are conceptually similar to AspectJ *cflow* pointcuts, although less powerful. (There is currently no way to specify that a pointcut executes below a join point matched by another pointcut.) A control flow pointcut matches the current call stack. For example, it might fire if the join point was invoked by a method in the `com.mycompany.web` package, or by the `SomeCaller` class. Control flow pointcuts are specified using the `org.springframework.aop.support.ControlFlowPointcut` class.

Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts. In Java 1.4, the cost is about 5 times that of other dynamic pointcuts.

5.1.5. Pointcut superclasses

Spring provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are most useful, you'll probably subclass StaticMethodMatcherPointcut, as shown below. This requires implementing just one abstract method (although it's possible to override other methods to customize behavior):

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method m, Class targetClass) {
        // return true if custom criteria match
    }

}
```

There are also superclasses for dynamic pointcuts.

You can use custom pointcuts with any advice type in Spring 1.0 RC2 and above.

5.1.6. Custom pointcuts

Because pointcuts in Spring AOP are Java classes, rather than language features (as in AspectJ) it's possible to declare custom pointcuts, whether static or dynamic. Custom pointcuts in Spring can be arbitrarily complex. However, using the AspectJ pointcut expression language is recommended if possible.

Later versions of Spring may offer support for "semantic pointcuts" as offered by JAC: for example, "all methods that change instance variables in the target object."

5.2. Advice API in Spring

Let's now look at how Spring AOP handles advice.

5.2.1. Advice lifecycles

Each advice is a Spring bean. An advice instance can be shared across all advised objects, or unique to each advised object. This corresponds to *per-class* or *per-instance* advice.

Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

It's possible to use a mix of shared and per-instance advice in the same AOP proxy.

5.2.2. Advice types in Spring

Spring provides several advice types out of the box, and is extensible to support arbitrary advice types. Let us look at the basic concepts and standard advice types.

Interception around advice

The most fundamental advice type in Spring is *interception around advice*.

Spring is compliant with the AOP Alliance interface for around advice using method interception. MethodInterceptors implementing around advice should implement the following interface:

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;

}
```

The `MethodInvocation` argument to the `invoke()` method exposes the method being invoked; the target join point; the AOP proxy; and the arguments to the method. The `invoke()` method should return the invocation's result: the return value of the join point.

A simple `MethodInterceptor` implementation looks as follows:

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }

}
```

Note the call to the MethodInvocation's `proceed()` method. This proceeds down the interceptor chain towards the join point. Most interceptors will invoke this method, and return its return value. However, a MethodInterceptor, like any around advice, can return a different value or throw an exception rather than invoke the proceed method. However, you don't want to do this without good reason!

MethodInterceptors offer interoperability with other AOP Alliance-compliant AOP implementations. The other advice types discussed in the remainder of this section implement common AOP concepts, but in a Spring-specific way. While there is an advantage in using the most specific advice type, stick with MethodInterceptor around advice if you are likely to want to run the aspect in another AOP framework. Note that pointcuts are not currently interoperable between frameworks, and the AOP Alliance does not currently define pointcut interfaces.

Before advice

A simpler advice type is a *before advice*. This does not need a `MethodInvocation` object, since it will only be called before entering the method.

The main advantage of a before advice is that there is no need to invoke the `proceed()` method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

The `MethodBeforeAdvice` interface is shown below. (Spring's API design would allow for field before advice, although the usual objects apply to field interception and it's unlikely that Spring will ever implement it).

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;

}
```

Note the return type is `void`. Before advice can insert custom behavior before the join point executes, but cannot change the return value. If a before advice throws an exception, this will abort further execution of the interceptor chain. The exception will propagate back up the interceptor chain. If it is unchecked, or on the signature of the invoked method, it will be passed directly to the client; otherwise it will be wrapped in an unchecked exception by the AOP proxy.

An example of a before advice in Spring, which counts all method invocations:

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {

    private int count;

    public void before(Method m, Object[] args, Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

Before advice can be used with any pointcut.

Throws advice

Throws advice is invoked after the return of the join point if the join point threw an exception. Spring offers typed throws advice. Note that this means that the `org.springframework.aop.ThrowsAdvice` interface does not contain any methods: It is a tag interface identifying that the given object implements one or more typed throws advice methods. These should be in the form of:

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

Only the last argument is required. The method signatures may have either one or four arguments, depending on whether the advice method is interested in the method and arguments. The following classes are examples of throws advice.

The advice below is invoked if a `RemoteException` is thrown (including subclasses):

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }

}
```

The following advice is invoked if a `ServletException` is thrown. Unlike the above advice, it declares 4 arguments, so that it has access to the invoked method, method arguments and target object:

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException
ex) {
        // Do something with all arguments
    }

}
```

The final example illustrates how these two methods could be used in a single class, which handles both `RemoteException` and `ServletException`. Any number of throws advice methods can be combined in a single class.

```
public static class CombinedThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }

    public void afterThrowing(Method m, Object[] args, Object target, ServletException
ex) {
        // Do something with all arguments
    }
}
```

Note: If a throws-advice method throws an exception itself, it will override the original exception (i.e. change the exception thrown to the user). The overriding exception will typically be a RuntimeException; this is compatible with any method signature. However, if a throws-advice method throws a checked exception, it will have to match the declared exceptions of the target method and is hence to some degree coupled to specific target method signatures. *Do not throw an undeclared checked exception that is incompatible with the target method's signature!*

Throws advice can be used with any pointcut.

After Returning advice

An after returning advice in Spring must implement the *org.springframework.aop.AfterReturningAdvice* interface, shown below:

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args,
            Object target) throws Throwable;

}
```

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {

    private int count;

    public void afterReturning(Object returnValue, Method m, Object[] args,
            Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }

}
```

This advice doesn't change the execution path. If it throws an exception, this will be thrown up the interceptor chain instead of the return value.

After returning advice can be used with any pointcut.

Introduction advice

Spring treats introduction advice as a special kind of interception advice.

Introduction requires an `IntroductionAdvisor`, and an `IntroductionInterceptor`, implementing the following interface:

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);

}
```

The `invoke()` method inherited from the AOP Alliance `MethodInterceptor` interface must implement the introduction: that is, if the invoked method is on an introduced interface, the introduction interceptor is responsible for handling the method call - it cannot invoke `proceed()`.

Introduction advice cannot be used with any pointcut, as it applies only at class, rather than method, level. You can only use introduction advice with the `IntroductionAdvisor`, which has the following methods:

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

    ClassFilter getClassFilter();

    void validateInterfaces() throws IllegalArgumentException;

}

public interface IntroductionInfo {

    Class[] getInterfaces();

}
```

There is no `MethodMatcher`, and hence no `Pointcut`, associated with introduction advice. Only class filtering is logical.

The `getInterfaces()` method returns the interfaces introduced by this advisor.

The `validateInterfaces()` method is used internally to see whether or not the introduced interfaces can be implemented by the configured `IntroductionInterceptor`.

Let's look at a simple example from the Spring test suite. Let's suppose we want to introduce the following interface to one or more objects:

```
public interface Lockable {

    void lock();

    void unlock();

    boolean locked();

}
```

This illustrates a *mixin*. We want to be able to cast advised objects to Lockable, whatever their type, and call lock and unlock methods. If we call the lock() method, we want all setter methods to throw a `LockedException`. Thus we can add an aspect that provides the ability to make objects immutable, without them having any knowledge of it: a good example of AOP.

Firstly, we'll need an `IntroductionInterceptor` that does the heavy lifting. In this case, we extend the `org.springframework.aop.support.DelegatingIntroductionInterceptor` convenience class. We could implement IntroductionInterceptor directly, but using `DelegatingIntroductionInterceptor` is best for most cases.

The `DelegatingIntroductionInterceptor` is designed to delegate an introduction to an actual implementation of the introduced interface(s), concealing the use of interception to do so. The delegate can be set to any object using a constructor argument; the default delegate (when the no-arg constructor is used) is this. Thus in the example below, the delegate is the `LockMixin` subclass of `DelegatingIntroductionInterceptor`. Given a delegate (by default itself), a `DelegatingIntroductionInterceptor` instance looks for all interfaces implemented by the delegate (other than IntroductionInterceptor), and will support introductions against any of them. It's possible for subclasses such as `LockMixin` to call the `suppressInterface(Class intf)` method to suppress interfaces that should not be exposed. However, no matter how many interfaces an `IntroductionInterceptor` is prepared to support, the `IntroductionAdvisor` used will control which interfaces are actually exposed. An introduced interface will conceal any implementation of the same interface by the target.

Thus LockMixin subclasses `DelegatingIntroductionInterceptor` and implements Lockable itself. The superclass automatically picks up that Lockable can be supported for introduction, so we don't need to specify that. We could introduce any number of interfaces in this way.

Note the use of the `locked` instance variable. This effectively adds additional state to that held in the target object.

```
public class LockMixin extends DelegatingIntroductionInterceptor implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0) {
            throw new LockedException();
        }
        return super.invoke(invocation);
    }

}
```

Often it isn't necessary to override the `invoke()` method: the `DelegatingIntroductionInterceptor` implementation - which calls the delegate method if the method is introduced, otherwise proceeds towards the join point - is usually sufficient. In the present case, we need to add a check: no setter method can be invoked if in locked mode.

The introduction advisor required is simple. All it needs to do is hold a distinct `LockMixin` instance, and specify the introduced interfaces - in this case, just `Lockable`. A more complex example might take a reference to the introduction interceptor (which would be defined as a prototype): in this case, there's no configuration relevant for a `LockMixin`, so we simply create it using `new`.

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }

}
```

We can apply this advisor very simply: it requires no configuration. (However, it *is* necessary: It's impossible to use an `IntroductionInterceptor` without an *IntroductionAdvisor*.) As usual with introductions, the advisor must be per-instance, as it is stateful. We need a different instance of `LockMixinAdvisor`, and hence `LockMixin`, for each advised object. The advisor comprises part of the advised object's state.

We can apply this advisor programmatically, using the `Advised.addAdvisor()` method, or (the recommended way) in XML configuration, like any other advisor. All proxy creation choices discussed below, including "auto proxy creators," correctly handle introductions and stateful mixins.

5.3. Advisor API in Spring

In Spring, an Advisor is an aspect that contains just a single advice object associated with a pointcut expression.

Apart from the special case of introductions, any advisor can be used with any advice. `org.springframework.aop.support.DefaultPointcutAdvisor` is the most commonly used advisor class. For example, it can be used with a `MethodInterceptor`, `BeforeAdvice` or `ThrowsAdvice`.

It is possible to mix advisor and advice types in Spring in the same AOP proxy. For example, you could use a interception around advice, throws advice and before advice in one proxy configuration: Spring will automatically create the necessary interceptor chain.

5.4. Using the ProxyFactoryBean to create AOP proxies

If you're using the Spring IoC container (an ApplicationContext or BeanFactory) for your business objects - and you should be! - you will want to use one of Spring's AOP FactoryBeans. (Remember that a factory bean introduces a layer of indirection, enabling it to create objects of a different type.)

The Spring 2.0 AOP support also uses factory beans under the covers.

The basic way to create an AOP proxy in Spring is to use the *org.springframework.aop.framework.ProxyFactoryBean*. This gives complete control over the pointcuts and advice that will apply, and their ordering. However, there are simpler options that are preferable if you don't need such control.

5.4.1. Basics

The `ProxyFactoryBean`, like other Spring `FactoryBean` implementations, introduces a level of indirection. If you define a `ProxyFactoryBean` with name `foo`, what objects referencing `foo` see is not the `ProxyFactoryBean` instance itself, but an object created by the `ProxyFactoryBean's implementation of the 'getObject()` method. This method will create an AOP proxy wrapping a target object.

One of the most important benefits of using a `ProxyFactoryBean` or another IoC-aware class to create AOP proxies, is that it means that advices and pointcuts can also be managed by IoC. This is a powerful feature, enabling certain approaches that are hard to achieve with other AOP frameworks. For example, an advice may itself reference application objects (besides the target, which should be available in any AOP framework), benefiting from all the pluggability provided by Dependency Injection.

5.4.2. JavaBean properties

In common with most `FactoryBean` implementations provided with Spring, the `ProxyFactoryBean` class is itself a JavaBean. Its properties are used to:

- Specify the target you want to proxy.
- Specify whether to use CGLIB (see below and also JDK- and CGLIB-based proxies).

Some key properties are inherited from `org.springframework.aop.framework.ProxyConfig` (the superclass for all AOP proxy factories in Spring). These key properties include:

- `proxyTargetClass`: `true` if the target class is to be proxied, rather than the target class' interfaces. If this property value is set to `true`, then CGLIB proxies will be created (but see also below JDK- and CGLIB-based proxies).
- `optimize`: controls whether or not aggressive optimizations are applied to proxies *created via CGLIB*. One should not blithely use this setting unless one fully understands how the relevant AOP proxy handles optimization. This is currently used only for CGLIB proxies; it has no effect with JDK dynamic proxies.
- `frozen`: if a proxy configuration is `frozen`, then changes to the configuration are no longer allowed. This is useful both as a slight optimization and for those cases when you don't want callers to be able to manipulate the proxy (via the `Advised` interface) after the proxy has been created. The default value of this property is `false`, so changes such as adding additional advice are allowed.
- `exposeProxy`: determines whether or not the current proxy should be exposed in a `ThreadLocal` so that it can be accessed by the target. If a target needs to obtain the proxy and the `exposeProxy` property is set to `true`, the target can use the `AopContext.currentProxy()` method.
- `aopProxyFactory`: the implementation of `AopProxyFactory` to use. Offers a way of customizing whether to use dynamic proxies, CGLIB or any other proxy strategy. The default implementation will choose dynamic proxies or CGLIB appropriately. There should be no need to use this property; it is intended to allow the addition of new proxy types in Spring 1.1.

Other properties specific to `ProxyFactoryBean` include:

- `proxyInterfaces`: array of String interface names. If this isn't supplied, a CGLIB proxy for the target class will be used (but see also below JDK- and CGLIB-based proxies).
- `interceptorNames`: String array of `Advisor`, interceptor or other advice names to apply. Ordering is significant, on a first come-first served basis. That is to say that the first interceptor in the list will be the first to be able to intercept the invocation.

The names are bean names in the current factory, including bean names from ancestor factories. You can't mention bean references here since doing so would result in the `ProxyFactoryBean` ignoring the singleton setting of the advice.

You can append an interceptor name with an asterisk (`*`). This will result in the application of all advisor beans with names starting with the part before the asterisk to be applied. An example of using this feature can be found in Using 'global' advisors.

- singleton: whether or not the factory should return a single object, no matter how often the `getObject()` method is called. Several `FactoryBean` implementations offer such a method. The default value is `true`. If you want to use stateful advice - for example, for stateful mixins - use prototype advices along with a singleton value of `false`.

5.4.3. JDK- and CGLIB-based proxies

This section serves as the definitive documentation on how the `ProxyFactoryBean` chooses to create one of either a JDK- and CGLIB-based proxy for a particular target object (that is to be proxied).

The behavior of the `ProxyFactoryBean` with regard to creating JDK- or CGLIB-based proxies changed between versions 1.2.x and 2.0 of Spring. The `ProxyFactoryBean` now exhibits similar semantics with regard to auto-detecting interfaces as those of the `TransactionProxyFactoryBean` class.

If the class of a target object that is to be proxied (hereafter simply referred to as the target class) doesn't implement any interfaces, then a CGLIB-based proxy will be created. This is the easiest scenario, because JDK proxies are interface based, and no interfaces means JDK proxying isn't even possible. One simply plugs in the target bean, and specifies the list of interceptors via the `interceptorNames` property. Note that a CGLIB-based proxy will be created even if the `proxyTargetClass` property of the `ProxyFactoryBean` has been set to `false`. (Obviously this makes no sense, and is best removed from the bean definition because it is at best redundant, and at worst confusing.)

If the target class implements one (or more) interfaces, then the type of proxy that is created depends on the configuration of the `ProxyFactoryBean`.

If the `proxyTargetClass` property of the `ProxyFactoryBean` has been set to `true`, then a CGLIB-based proxy will be created. This makes sense, and is in keeping with the principle of least surprise. Even if the `proxyInterfaces` property of the `ProxyFactoryBean` has been set to one or more fully qualified interface names, the fact that the `proxyTargetClass` property is set to `true` *will* cause CGLIB-based proxying to be in effect.

If the `proxyInterfaces` property of the `ProxyFactoryBean` has been set to one or more fully qualified interface names, then a JDK-based proxy will be created. The created proxy will implement all of the interfaces that were specified in the `proxyInterfaces` property; if the target class happens to implement a whole lot more interfaces than those specified in the `proxyInterfaces` property, that is all well and good but those additional interfaces will not be implemented by the returned proxy.

If the `proxyInterfaces` property of the `ProxyFactoryBean` has *not* been set, but the target class *does implement one (or more)* interfaces, then the `ProxyFactoryBean` will auto-detect the fact that the target class does actually implement at least one interface, and a JDK-based proxy will be created. The interfaces that are actually proxied will be *all* of the interfaces that the target class implements; in effect, this is the same as simply supplying a list of each and every interface that the target class implements to the `proxyInterfaces` property. However, it is significantly less work, and less prone to typos.

5.4.4. Proxying interfaces

Let's look at a simple example of `ProxyFactoryBean` in action. This example involves:

- A *target bean* that will be proxied. This is the "personTarget" bean definition in the example below.
- An Advisor and an Interceptor used to provide advice.
- An AOP proxy bean definition specifying the target object (the personTarget bean) and the interfaces to proxy, along with the advices to apply.

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name"><value>Tony</value></property>
    <property name="age"><value>51</value></property>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty"><value>Custom string property value
</value></property>
</bean>

<bean id="debugInterceptor" class=
"org.springframework.aop.interceptor.DebugInterceptor">
</bean>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces"><value>com.mycompany.Person</value></property>
    <property name="target"><ref bean="personTarget"/></property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

Note that the `interceptorNames` property takes a list of String: the bean names of the interceptor or advisors in the current factory. Advisors, interceptors, before, after returning and throws advice objects can be used. The ordering of advisors is significant.

You might be wondering why the list doesn't hold bean references. The reason for this is that if the ProxyFactoryBean's singleton property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it's necessary to be able to obtain an instance of the prototype from the factory; holding a reference isn't sufficient.

The "person" bean definition above can be used in place of a Person implementation, as follows:

```
Person person = (Person) factory.getBean("person");
```

Other beans in the same IoC context can express a strongly typed dependency on it, as with an ordinary Java object:

```
<bean id="personUser" class="com.mycompany.PersonUser">
    <property name="person"><ref bean="person" /></property>
</bean>
```

The `PersonUser` class in this example would expose a property of type Person. As far as it's concerned, the AOP proxy can be used transparently in place of a "real" person implementation. However, its class would be a dynamic proxy class. It would be possible to cast it to the `Advised` interface (discussed below).

It's possible to conceal the distinction between target and proxy using an anonymous *inner bean*, as follows. Only the `ProxyFactoryBean` definition is different; the advice is included only for completeness:

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty"><value>Custom string property value
</value></property>
</bean>

<bean id="debugInterceptor" class=
"org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces"><value>com.mycompany.Person</value></property>
    <!-- Use inner bean, not local reference to target -->
    <property name="target">
        <bean class="com.mycompany.PersonImpl">
            <property name="name"><value>Tony</value></property>
            <property name="age"><value>51</value></property>
        </bean>
    </property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

This has the advantage that there's only one object of type `Person`: useful if we want to prevent users of the application context from obtaining a reference to the un-advised object, or need to avoid any ambiguity with Spring IoC *autowiring*. There's also arguably an advantage in that the ProxyFactoryBean definition is self-contained. However, there are times when being able to obtain

the un-advised target from the factory might actually be an *advantage*: for example, in certain test scenarios.

5.4.5. Proxying classes

What if you need to proxy a class, rather than one or more interfaces?

Imagine that in our example above, there was no `Person` interface: we needed to advise a class called `Person` that didn't implement any business interface. In this case, you can configure Spring to use CGLIB proxying, rather than dynamic proxies. Simply set the `proxyTargetClass` property on the ProxyFactoryBean above to true. While it's best to program to interfaces, rather than classes, the ability to advise classes that don't implement interfaces can be useful when working with legacy code. (In general, Spring isn't prescriptive. While it makes it easy to apply good practices, it avoids forcing a particular approach.)

If you want to, you can force the use of CGLIB in any case, even if you do have interfaces.

CGLIB proxying works by generating a subclass of the target class at runtime. Spring configures this generated subclass to delegate method calls to the original target: the subclass is used to implement the *Decorator* pattern, weaving in the advice.

CGLIB proxying should generally be transparent to users. However, there are some issues to consider:

- `Final` methods can't be advised, as they can't be overridden.
- As of Spring 3.2 it is no longer required to add CGLIB to your project classpath. CGLIB classes have been repackaged under org.springframework and included directly in the spring-core JAR. This is both for user convenience as well as to avoid potential conflicts with other projects that have dependence on a differing version of CGLIB.

There's little performance difference between CGLIB proxying and dynamic proxies. As of Spring 1.0, dynamic proxies are slightly faster. However, this may change in the future. Performance should not be a decisive consideration in this case.

5.4.6. Using 'global' advisors

By appending an asterisk to an interceptor name, all advisors with bean names matching the part before the asterisk, will be added to the advisor chain. This can come in handy if you need to add a standard set of 'global' advisors:

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="service"/>
    <property name="interceptorNames">
        <list>
            <value>global*</value>
        </list>
    </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class=
"org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

5.5. Concise proxy definitions

Especially when defining transactional proxies, you may end up with many similar proxy definitions. The use of parent and child bean definitions, along with inner bean definitions, can result in much cleaner and more concise proxy definitions.

First a parent, *template*, bean definition is created for the proxy:

```
<bean id="txProxyTemplate" abstract="true"
        class="
org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

This will never be instantiated itself, so may actually be incomplete. Then each proxy which needs to be created is just a child bean definition, which wraps the target of the proxy as an inner bean definition, since the target will never be used on its own anyway.

```
<bean id="myService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MyServiceImpl">
        </bean>
    </property>
</bean>
```

It is of course possible to override properties from the parent template, such as in this case, the transaction propagation settings:

```
<bean id="mySpecialService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MySpecialServiceImpl">
        </bean>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="store*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

Note that in the example above, we have explicitly marked the parent bean definition as *abstract* by using the *abstract* attribute, as described previously, so that it may not actually ever be instantiated. Application contexts (but not simple bean factories) will by default pre-instantiate all singletons. It is therefore important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the*abstract* attribute to *true*, otherwise the application context will actually try to pre-instantiate it.

5.6. Creating AOP proxies programmatically with the ProxyFactory

It's easy to create AOP proxies programmatically using Spring. This enables you to use Spring AOP without dependency on Spring IoC.

The following listing shows creation of a proxy for a target object, with one interceptor and one advisor. The interfaces implemented by the target object will automatically be proxied:

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addInterceptor(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

The first step is to construct an object of type `org.springframework.aop.framework.ProxyFactory`. You can create this with a target object, as in the above example, or specify the interfaces to be proxied in an alternate constructor.

You can add interceptors or advisors, and manipulate them for the life of the ProxyFactory. If you add an IntroductionInterceptionAroundAdvisor you can cause the proxy to implement additional interfaces.

There are also convenience methods on ProxyFactory (inherited from `AdvisedSupport`) which allow you to add other advice types such as before and throws advice. AdvisedSupport is the superclass of

both ProxyFactory and ProxyFactoryBean.

Integrating AOP proxy creation with the IoC framework is best practice in most applications. We recommend that you externalize configuration from Java code with AOP, as in general.

5.7. Manipulating advised objects

However you create AOP proxies, you can manipulate them using the `org.springframework.aop.framework.Advised` interface. Any AOP proxy can be cast to this interface, whichever other interfaces it implements. This interface includes the following methods:

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice) throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();
```

The `getAdvisors()` method will return an Advisor for every advisor, interceptor or other advice type that has been added to the factory. If you added an Advisor, the returned advisor at this index will be the object that you added. If you added an interceptor or other advice type, Spring will have wrapped this in an advisor with a pointcut that always returns true. Thus if you added a `MethodInterceptor`, the advisor returned for this index will be an `DefaultPointcutAdvisor` returning your `MethodInterceptor` and a pointcut that matches all classes and methods.

The `addAdvisor()` methods can be used to add any Advisor. Usually the advisor holding pointcut and advice will be the generic `DefaultPointcutAdvisor`, which can be used with any advice or pointcut (but not for introductions).

By default, it's possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it's impossible to add or remove an introduction advisor, as existing proxies from the factory will not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)

A simple example of casting an AOP proxy to the `Advised` interface and examining and manipulating its advice:

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.getAdvisors().length);
```

It's questionable whether it's advisable (no pun intended) to modify advice on a business object in production, although there are no doubt legitimate usage cases. However, it can be very useful in development: for example, in tests. I have sometimes found it very useful to be able to add test code in the form of an interceptor or other advice, getting inside a method invocation I want to test. (For example, the advice can get inside a transaction created for that method: for example, to run SQL to check that a database was correctly updated, before marking the transaction for roll back.)

Depending on how you created the proxy, you can usually set a `frozen` flag, in which case the `Advised isFrozen()` method will return true, and any attempts to modify advice through addition or removal will result in an `AopConfigException`. The ability to freeze the state of an advised object is useful in some cases, for example, to prevent calling code removing a security interceptor. It may also be used in Spring 1.1 to allow aggressive optimization if runtime advice modification is known not to be required.

5.8. Using the "autoproxy" facility

So far we've considered explicit creation of AOP proxies using a `ProxyFactoryBean` or similar factory bean.

Spring also allows us to use "autoproxy" bean definitions, which can automatically proxy selected bean definitions. This is built on Spring "bean post processor" infrastructure, which enables modification of any bean definition as the container loads.

In this model, you set up some special bean definitions in your XML bean definition file to configure the auto proxy infrastructure. This allows you just to declare the targets eligible for autoproxying: you don't need to use `ProxyFactoryBean`.

There are two ways to do this:

- Using an autoproxy creator that refers to specific beans in the current context.
- A special case of autoproxy creation that deserves to be considered separately; autoproxy creation driven by source-level metadata attributes.

5.8.1. Autoproxy bean definitions

The `org.springframework.aop.framework.autoproxy` package provides the following standard autoproxy creators.

BeanNameAutoProxyCreator

The `BeanNameAutoProxyCreator` class is a `BeanPostProcessor` that automatically creates AOP proxies for beans with names matching literal values or wildcards.

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames"><value>jdk*,onlyJdk</value></property>
    <property name="interceptorNames">
        <list>
            <value>myInterceptor</value>
        </list>
    </property>
</bean>
```

As with `ProxyFactoryBean`, there is an `interceptorNames` property rather than a list of interceptors, to allow correct behavior for prototype advisors. Named "interceptors" can be advisors or any advice type.

As with auto proxying in general, the main point of using `BeanNameAutoProxyCreator` is to apply the same configuration consistently to multiple objects, with minimal volume of configuration. It is a popular choice for applying declarative transactions to multiple objects.

Bean definitions whose names match, such as "jdkMyBean" and "onlyJdk" in the above example, are plain old bean definitions with the target class. An AOP proxy will be created automatically by the `BeanNameAutoProxyCreator`. The same advice will be applied to all matching beans. Note that if advisors are used (rather than the interceptor in the above example), the pointcuts may apply differently to different beans.

DefaultAdvisorAutoProxyCreator

A more general and extremely powerful auto proxy creator is `DefaultAdvisorAutoProxyCreator`. This will automagically apply eligible advisors in the current context, without the need to include specific bean names in the autoproxy advisor's bean definition. It offers the same merit of consistent configuration and avoidance of duplication as `BeanNameAutoProxyCreator`.

Using this mechanism involves:

- Specifying a `DefaultAdvisorAutoProxyCreator` bean definition.
- Specifying any number of Advisors in the same or related contexts. Note that these *must* be

Advisors, not just interceptors or other advices. This is necessary because there must be a pointcut to evaluate, to check the eligibility of each advice to candidate bean definitions.

The `DefaultAdvisorAutoProxyCreator` will automatically evaluate the pointcut contained in each advisor, to see what (if any) advice it should apply to each business object (such as "businessObject1" and "businessObject2" in the example).

This means that any number of advisors can be applied automatically to each business object. If no pointcut in any of the advisors matches any method in a business object, the object will not be proxied. As bean definitions are added for new business objects, they will automatically be proxied if necessary.

Autoproxying in general has the advantage of making it impossible for callers or dependencies to obtain an un-advised object. Calling getBean("businessObject1") on this ApplicationContext will return an AOP proxy, not the target business object. (The "inner bean" idiom shown earlier also offers this benefit.)

```
<bean class=
"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class=
"org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
    <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

The `DefaultAdvisorAutoProxyCreator` is very useful if you want to apply the same advice consistently to many business objects. Once the infrastructure definitions are in place, you can simply add new business objects without including specific proxy configuration. You can also drop in additional aspects very easily - for example, tracing or performance monitoring aspects - with minimal change to configuration.

The DefaultAdvisorAutoProxyCreator offers support for filtering (using a naming convention so that only certain advisors are evaluated, allowing use of multiple, differently configured, AdvisorAutoProxyCreators in the same factory) and ordering. Advisors can implement the `org.springframework.core.Ordered` interface to ensure correct ordering if this is an issue. The TransactionAttributeSourceAdvisor used in the above example has a configurable order value; the default setting is unordered.

AbstractAdvisorAutoProxyCreator

This is the superclass of DefaultAdvisorAutoProxyCreator. You can create your own autoproxy

creators by subclassing this class, in the unlikely event that advisor definitions offer insufficient customization to the behavior of the framework `DefaultAdvisorAutoProxyCreator`.

5.8.2. Using metadata-driven auto-proxying

A particularly important type of autoproxying is driven by metadata. This produces a similar programming model to .NET `ServicedComponents`. Instead of using XML deployment descriptors as in EJB, configuration for transaction management and other enterprise services is held in source-level attributes.

In this case, you use the `DefaultAdvisorAutoProxyCreator`, in combination with Advisors that understand metadata attributes. The metadata specifics are held in the pointcut part of the candidate advisors, rather than in the autoproxy creation class itself.

This is really a special case of the `DefaultAdvisorAutoProxyCreator`, but deserves consideration on its own. (The metadata-aware code is in the pointcuts contained in the advisors, not the AOP framework itself.)

The `/attributes` directory of the JPetStore sample application shows the use of attribute-driven autoproxying. In this case, there's no need to use the `TransactionProxyFactoryBean`. Simply defining transactional attributes on business objects is sufficient, because of the use of metadata-aware pointcuts. The bean definitions include the following code, in `/WEB-INF/declarativeServices.xml`. Note that this is generic, and can be used outside the JPetStore:

```
<bean class=
"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class=
"org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributeSource">
        <bean class=
"org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
            <property name="attributes" ref="attributes"/>
        </bean>
    </property>
</bean>

<bean id="attributes" class="org.springframework.metadata.commons.CommonsAttributes"/>
```

The `DefaultAdvisorAutoProxyCreator` bean definition (the name is not significant, hence it can even be omitted) will pick up all eligible pointcuts in the current application context. In this case, the "transactionAdvisor" bean definition, of type `TransactionAttributeSourceAdvisor`, will apply to classes or methods carrying a transaction attribute. The TransactionAttributeSourceAdvisor

depends on a TransactionInterceptor, via constructor dependency. The example resolves this via autowiring. The `AttributesTransactionAttributeSource` depends on an implementation of the `org.springframework.metadata.Attributes` interface. In this fragment, the "attributes" bean satisfies this, using the Jakarta Commons Attributes API to obtain attribute information. (The application code must have been compiled using the Commons Attributes compilation task.)

The `/annotation` directory of the JPetStore sample application contains an analogous example for auto-proxying driven by JDK 1.5+ annotations. The following configuration enables automatic detection of Spring's `Transactional` annotation, leading to implicit proxies for beans containing that annotation:

```
<bean class=
"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class=
"org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributeSource">
        <bean class=
"org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
    </property>
</bean>
```

The `TransactionInterceptor` defined here depends on a `PlatformTransactionManager` definition, which is not included in this generic file (although it could be) because it will be specific to the application's transaction requirements (typically JTA, as in this example, or Hibernate or JDBC):

```
<bean id="transactionManager"
        class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

If you require only declarative transaction management, using these generic XML definitions will result in Spring automatically proxying all classes or methods with transaction attributes. You won't need to work directly with AOP, and the programming model is similar to that of .NET ServicedComponents.

This mechanism is extensible. It's possible to do autoproxying based on custom attributes. You need to:

- Define your custom attribute.
- Specify an Advisor with the necessary advice, including a pointcut that is triggered by the presence of the custom attribute on a class or method. You may be able to use an existing advice, merely implementing a static pointcut that picks up the custom attribute.

It's possible for such advisors to be unique to each advised class (for example, mixins): they simply need to be defined as prototype, rather than singleton, bean definitions. For example, the `LockMixin` introduction interceptor from the Spring test suite, shown above, could be used in conjunction with an attribute-driven pointcut to target a mixin, as shown here. We use the generic `DefaultPointcutAdvisor`, configured using JavaBean properties:

```
<bean id="lockMixin" class="org.springframework.aop.LockMixin"
        scope="prototype"/>

<bean id="lockableAdvisor" class=
"org.springframework.aop.support.DefaultPointcutAdvisor"
        scope="prototype">
    <property name="pointcut" ref="myAttributeAwarePointcut"/>
    <property name="advice" ref="lockMixin"/>
</bean>

<bean id="anyBean" class="anyclass" ...
```

If the attribute aware pointcut matches any methods in the `anyBean` or other bean definitions, the mixin will be applied. Note that both `lockMixin` and `lockableAdvisor` definitions are prototypes. The `myAttributeAwarePointcut` pointcut can be a singleton definition, as it doesn't hold state for individual advised objects.

5.9. Using TargetSources

Spring offers the concept of a *TargetSource*, expressed in the `org.springframework.aop.TargetSource` interface. This interface is responsible for returning the "target object" implementing the join point. The `TargetSource` implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers using Spring AOP don't normally need to work directly with TargetSources, but this provides a powerful means of supporting pooling, hot swappable and other sophisticated targets. For example, a pooling TargetSource can return a different target instance for each invocation, using a pool to manage instances.

If you do not specify a TargetSource, a default implementation is used that wraps a local object. The same target is returned for each invocation (as you would expect).

Let's look at the standard target sources provided with Spring, and how you can use them.

When using a custom target source, your target will usually need to be a prototype rather than a singleton bean definition. This allows Spring to create a new target instance when required.

5.9.1. Hot swappable target sources

The `org.springframework.aop.target.HotSwappableTargetSource` exists to allow the target of an AOP proxy to be switched while allowing callers to keep their references to it.

Changing the target source's target takes effect immediately. The `HotSwappableTargetSource` is threadsafe.

You can change the target via the `swap()` method on HotSwappableTargetSource as follows:

```
HotSwappableTargetSource swapper = (HotSwappableTargetSource) beanFactory.getBean(
"swapper");
Object oldTarget = swapper.swap(newTarget);
```

The XML definitions required look as follows:

```
<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
    <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="swapper"/>
</bean>
```

The above `swap()` call changes the target of the swappable bean. Clients who hold a reference to that bean will be unaware of the change, but will immediately start hitting the new target.

Although this example doesn't add any advice - and it's not necessary to add advice to use a `TargetSource` - of course any `TargetSource` can be used in conjunction with arbitrary advice.

5.9.2. Pooling target sources

Using a pooling target source provides a similar programming model to stateless session EJBs, in which a pool of identical instances is maintained, with method invocations going to free objects in the pool.

A crucial difference between Spring pooling and SLSB pooling is that Spring pooling can be applied to any POJO. As with Spring in general, this service can be applied in a non-invasive way.

Spring provides out-of-the-box support for Commons Pool 2.2, which provides a fairly efficient pooling implementation. You'll need the commons-pool Jar on your application's classpath to use this feature. It's also possible to subclass `org.springframework.aop.target.AbstractPoolingTargetSource` to support any other pooling API.

Commons Pool 1.5+ is also supported but deprecated as of Spring Framework 4.2.

Sample configuration is shown below:

```
<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject" scope=
"prototype">
    ... properties omitted
</bean>

<bean id="poolTargetSource" class=
"org.springframework.aop.target.CommonsPool2TargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
    <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="poolTargetSource"/>
    <property name="interceptorNames" value="myInterceptor"/>
</bean>
```

Note that the target object - "businessObjectTarget" in the example - *must* be a prototype. This allows the `PoolingTargetSource` implementation to create new instances of the target to grow the pool as necessary. See the Javadoc for `AbstractPoolingTargetSource` and the concrete subclass you wish to use for information about its properties: "maxSize" is the most basic, and always guaranteed to be present.

In this case, "myInterceptor" is the name of an interceptor that would need to be defined in the same IoC context. However, it isn't necessary to specify interceptors to use pooling. If you want only pooling, and no other advice, don't set the interceptorNames property at all.

It's possible to configure Spring so as to be able to cast any pooled object to the `org.springframework.aop.target.PoolingConfig` interface, which exposes information about the configuration and current size of the pool through an introduction. You'll need to define an advisor like this:

```
<bean id="poolConfigAdvisor" class=
"org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="poolTargetSource"/>
    <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>
```

This advisor is obtained by calling a convenience method on the `AbstractPoolingTargetSource` class, hence the use of MethodInvokingFactoryBean. This advisor's name ("poolConfigAdvisor" here) must be in the list of interceptors names in the ProxyFactoryBean exposing the pooled object.

The cast will look as follows:

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

Pooling stateless service objects is not usually necessary. We don't believe it should be the default choice, as most stateless objects are naturally thread safe, and instance pooling is problematic if resources are cached.

Simpler pooling is available using autoproxying. It's possible to set the TargetSources used by any autoproxy creator.

5.9.3. Prototype target sources

Setting up a "prototype" target source is similar to a pooling TargetSource. In this case, a new instance of the target will be created on every method invocation. Although the cost of creating a new object isn't high in a modern JVM, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus you shouldn't use this approach without very good reason.

To do this, you could modify the `poolTargetSource` definition shown above as follows. (I've also changed the name, for clarity.)

```
<bean id="prototypeTargetSource" class=
"org.springframework.aop.target.PrototypeTargetSource">
    <property name="targetBeanName" ref="businessObjectTarget"/>
</bean>
```

There's only one property: the name of the target bean. Inheritance is used in the TargetSource implementations to ensure consistent naming. As with the pooling target source, the target bean must be a prototype bean definition.

5.9.4. ThreadLocal target sources

`ThreadLocal` target sources are useful if you need an object to be created for each incoming request (per thread that is). The concept of a `ThreadLocal` provide a JDK-wide facility to transparently store resource alongside a thread. Setting up a `ThreadLocalTargetSource` is pretty much the same as was explained for the other types of target source:

```
<bean id="threadlocalTargetSource" class=
"org.springframework.aop.target.ThreadLocalTargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

ThreadLocals come with serious issues (potentially resulting in memory leaks) when incorrectly using them in a multi-threaded and multi-classloader environments. One should always consider wrapping a threadlocal in some other class and never directly use the `ThreadLocal` itself (except of course in the wrapper class). Also, one should always remember to correctly set and unset (where the latter simply involved a call to `ThreadLocal.set(null)`) the resource local to the thread. Unsetting should be done in any case since not unsetting it might result in problematic behavior. Spring's ThreadLocal support does this for you and should always be considered in favor of using ThreadLocals without other proper handling code.

5.10. Defining new Advice types

Spring AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to the out-of-the-box interception around advice, before, throws advice and after returning advice.

The `org.springframework.aop.framework.adapter` package is an SPI package allowing support for new custom advice types to be added without changing the core framework. The only constraint on a custom `Advice` type is that it must implement the `org.aopalliance.aop.Advice` tag interface.

Please refer to the `org.springframework.aop.framework.adapter` package's Javadocs for further information.

5.11. Further resources

Please refer to the Spring sample applications for further examples of Spring AOP:

- The JPetStore's default configuration illustrates the use of the `TransactionProxyFactoryBean` for declarative transaction management.
- The `/attributes` directory of the JPetStore illustrates the use of attribute-driven declarative transaction management.

Chapter 6. XML Schema-based configuration

6.1. Introduction

This appendix details the XML Schema-based configuration introduced in Spring 2.0 and enhanced and extended in Spring 2.5 and 3.0.

> **DTD support?**
>
> Authoring Spring configuration files using the older DTD style is still fully supported.
>
> Nothing will break if you forego the use of the new XML Schema-based approach to authoring Spring XML configuration files. All that you lose out on is the opportunity to have more succinct and clearer configuration. Regardless of whether the XML configuration is DTD- or Schema-based, in the end it all boils down to the same object model in the container (namely one or more `BeanDefinition` instances).

The central motivation for moving to XML Schema based configuration files was to make Spring XML configuration easier. The *'classic'* `<bean/>`-based approach is good, but its generic-nature comes with a price in terms of configuration overhead.

From the Spring IoC containers point-of-view, *everything* is a bean. That's great news for the Spring IoC container, because if everything is a bean then everything can be treated in the exact same fashion. The same, however, is not true from a developer's point-of-view. The objects defined in a Spring XML configuration file are not all generic, vanilla beans. Usually, each bean requires some degree of specific configuration.

Spring 2.0's new XML Schema-based configuration addresses this issue. The `<bean/>` element is still present, and if you wanted to, you could continue to write the *exact same* style of Spring XML configuration using only `<bean/>` elements. The new XML Schema-based configuration does, however, make Spring XML configuration files substantially clearer to read. In addition, it allows you to express the intent of a bean definition.

The key thing to remember is that the new custom tags work best for infrastructure or integration beans: for example, AOP, collections, transactions, integration with 3rd-party frameworks such as Mule, etc., while the existing bean tags are best suited to application-specific beans, such as DAOs, service layer objects, validators, etc.

The examples included below will hopefully convince you that the inclusion of XML Schema support in Spring 2.0 was a good idea. The reception in the community has been encouraging; also, please note the fact that this new configuration mechanism is totally customisable and extensible. This means you can write your own domain-specific configuration tags that would better represent your application's domain; the process involved in doing so is covered in the appendix entitled Extensible XML authoring.

6.2. XML Schema-based configuration

6.2.1. Referencing the schemas

To switch over from the DTD-style to the new XML Schema-style, you need to make the following change.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<!-- bean definitions here -->

</beans>
```

The equivalent file in the XML Schema-style would be...

```
<?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 definitions here -->

</beans>
```

The `'xsi:schemaLocation'` fragment is not actually required, but can be included to reference a local copy of a schema (which can be useful during development).

The above Spring XML configuration fragment is boilerplate that you can copy and paste (!) and then plug `<bean/>` definitions into like you have always done. However, the entire point of switching over is to take advantage of the new Spring 2.0 XML tags since they make configuration easier. The section entitled the util schema demonstrates how you can start immediately by using some of the more common utility tags.

The rest of this chapter is devoted to showing examples of the new Spring XML Schema based configuration, with at least one example for every new tag. The format follows a before and after style, with a *before* snippet of XML showing the old (but still 100% legal and supported) style, followed immediately by an *after* example showing the equivalent in the new XML Schema-based style.

6.2.2. the util schema

First up is coverage of the `util` tags. As the name implies, the `util` tags deal with common, *utility* configuration issues, such as configuring collections, referencing constants, and suchlike.

To use the tags in the `util` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the snippet below references the correct schema so that the tags in the `util` namespace are available to you.

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

</beans>
```

<util:constant/>

Before...

```
<bean id="..." class="...">
    <property name="isolation">
        <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
                class=
"org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
    </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `FieldRetrievingFactoryBean`, to set the value of the `isolation` property on a bean to the value of the `java.sql.Connection.TRANSACTION_SERIALIZABLE` constant. This is all well and good, but it is a tad verbose and (unnecessarily) exposes Spring's internal plumbing to the end user.

The following XML Schema-based version is more concise and clearly expresses the developer's intent (*'inject this constant value'*), and it just reads better.

```
<bean id="..." class="...">
    <property name="isolation">
        <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
    </property>
</bean>
```

Setting a bean property or constructor arg from a field value

`FieldRetrievingFactoryBean` is a `FactoryBean` which retrieves a `static` or non-static field value. It is typically used for retrieving `public static final` constants, which may then be used to set a property value or constructor arg for another bean.

Find below an example which shows how a `static` field is exposed, by using the `staticField` property:

```
<bean id="myField"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
    <property name="staticField" value="java.sql.Connection.TRANSACTION_SERIALIZABLE
"/>
</bean>
```

There is also a convenience usage form where the `static` field is specified as the bean name:

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

This does mean that there is no longer any choice in what the bean id is (so any other bean that refers to it will also have to use this longer name), but this form is very concise to define, and very convenient to use as an inner bean since the id doesn't have to be specified for the bean reference:

```
<bean id="..." class="...">
    <property name="isolation">
        <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
                class=
"org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
    </property>
</bean>
```

It is also possible to access a non-static (instance) field of another bean, as described in the API documentation for the `FieldRetrievingFactoryBean` class.

Injecting enum values into beans as either property or constructor arguments is very easy to do in Spring, in that you don't actually have to *do* anything or know anything about the Spring internals (or even about classes such as the `FieldRetrievingFactoryBean`). Let's look at an example to see how easy injecting an enum value is; consider this JDK 5 enum:

```
package javax.persistence;

public enum PersistenceContextType {

    TRANSACTION,
    EXTENDED

}
```

Now consider a setter of type `PersistenceContextType`:

```
package example;

public class Client {

    private PersistenceContextType persistenceContextType;

    public void setPersistenceContextType(PersistenceContextType type) {
        this.persistenceContextType = type;
    }

}
```

a. and the corresponding bean definition:

```
<bean class="example.Client">
    <property name="persistenceContextType" value="TRANSACTION" />
</bean>
```

This works for classic type-safe emulated enums (on JDK 1.4 and JDK 1.3) as well; Spring will automatically attempt to match the string property value to a constant on the enum class.

<util:property-path/>

Before...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<bean id="testBean.age" class=
"org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

The above configuration uses a Spring `FactoryBean` implementation, the `PropertyPathFactoryBean`, to create a bean (of type `int`) called `testBean.age` that has a value equal to the `age` property of the `testBean` bean.

After...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<util:property-path id="name" path="testBean.age"/>
```

The value of the `path` attribute of the `<property-path/>` tag follows the form `beanName.beanProperty`.

Using <util:property-path/> to set a bean property or constructor-argument

`PropertyPathFactoryBean` is a `FactoryBean` that evaluates a property path on a given target object. The target object can be specified directly or via a bean name. This value may then be used in another bean definition as a property value or constructor argument.

Here's an example where a path is used against another bean, by name:

```
// target bean to be referenced by name
<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

// will result in 11, which is the value of property 'spouse.age' of bean 'person'
<bean id="theAge"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
    <property name="targetBeanName" value="person"/>
    <property name="propertyPath" value="spouse.age"/>
</bean>
```

In this example, a path is evaluated against an inner bean:

```
<!-- will result in 12, which is the value of property 'age' of the inner bean -->
<bean id="theAge"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
    <property name="targetObject">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="12"/>
        </bean>
    </property>
    <property name="propertyPath" value="age"/>
</bean>
```

There is also a shortcut form, where the bean name is the property path.

```
<!-- will result in 10, which is the value of property 'age' of bean 'person' -->
<bean id="person.age"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

This form does mean that there is no choice in the name of the bean. Any reference to it will also have to use the same id, which is the path. Of course, if used as an inner bean, there is no need to refer to it at all:

```
<bean id="..." class="...">
    <property name="age">
        <bean id="person.age"
                class=
"org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
    </property>
</bean>
```

The result type may be specifically set in the actual definition. This is not necessary for most use cases, but can be of use for some. Please see the Javadocs for more info on this feature.

<util:properties/>

Before...

```
<!-- creates a java.util.Properties instance with values loaded from the supplied
location -->
<bean id="jdbcConfiguration" class=
"org.springframework.beans.factory.config.PropertiesFactoryBean">
    <property name="location" value="classpath:com/foo/jdbc-production.properties"/>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `PropertiesFactoryBean`, to instantiate a `java.util.Properties` instance with values loaded from the supplied `Resource` location).

After...

```
<!-- creates a java.util.Properties instance with values loaded from the supplied
location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-
production.properties"/>
```

<util:list/>

Before...

```
<!-- creates a java.util.List instance with values loaded from the supplied
'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
    <property name="sourceList">
        <list>
            <value>pechorin@hero.org</value>
            <value>raskolnikov@slums.org</value>
            <value>stavrogin@gov.org</value>
            <value>porfiry@gov.org</value>
        </list>
    </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `ListFactoryBean`, to create a `java.util.List` instance initialized with values taken from the supplied `sourceList`.

After...

```
<!-- creates a java.util.List instance with the supplied values -->
<util:list id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:list>
```

You can also explicitly control the exact type of `List` that will be instantiated and populated via the use of the `list-class` attribute on the `<util:list/>` element. For example, if we really need a `java.util.LinkedList` to be instantiated, we could use the following configuration:

```
<util:list id="emails" list-class="java.util.LinkedList">
    <value>jackshaftoe@vagabond.org</value>
    <value>eliza@thinkingmanscrumpet.org</value>
    <value>vanhoek@pirate.org</value>
    <value>d'Arcachon@nemesis.org</value>
</util:list>
```

If no `list-class` attribute is supplied, a `List` implementation will be chosen by the container.

<util:map/>

Before...

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap'
-->
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">
    <property name="sourceMap">
        <map>
            <entry key="pechorin" value="pechorin@hero.org"/>
            <entry key="raskolnikov" value="raskolnikov@slums.org"/>
            <entry key="stavrogin" value="stavrogin@gov.org"/>
            <entry key="porfiry" value="porfiry@gov.org"/>
        </map>
    </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `MapFactoryBean`, to create a `java.util.Map` instance initialized with key-value pairs taken from the supplied `'sourceMap'`.

After...

```
<!-- creates a java.util.Map instance with the supplied key-value pairs -->
<util:map id="emails">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

You can also explicitly control the exact type of `Map` that will be instantiated and populated via the use of the `'map-class'` attribute on the `<util:map/>` element. For example, if we really need a `java.util.TreeMap` to be instantiated, we could use the following configuration:

```
<util:map id="emails" map-class="java.util.TreeMap">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

If no `'map-class'` attribute is supplied, a `Map` implementation will be chosen by the container.

<util:set/>

Before...

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet'
-->
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">
    <property name="sourceSet">
        <set>
            <value>pechorin@hero.org</value>
            <value>raskolnikov@slums.org</value>
            <value>stavrogin@gov.org</value>
            <value>porfiry@gov.org</value>
        </set>
    </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `SetFactoryBean`, to create a `java.util.Set` instance initialized with values taken from the supplied `'sourceSet'`.

After...

```
<!-- creates a java.util.Set instance with the supplied values -->
<util:set id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

You can also explicitly control the exact type of `Set` that will be instantiated and populated via the use of the `'set-class'` attribute on the `<util:set/>` element. For example, if we really need a `java.util.TreeSet` to be instantiated, we could use the following configuration:

```
<util:set id="emails" set-class="java.util.TreeSet">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

If no `'set-class'` attribute is supplied, a `Set` implementation will be chosen by the container.

6.2.3. the jee schema

The `jee` tags deal with Java EE (Java Enterprise Edition)-related configuration issues, such as looking up a JNDI object and defining EJB references.

To use the tags in the `jee` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `jee` namespace are available to you.

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

</beans>
```

<jee:jndi-lookup/> (simple)

Before...

```
<bean id="<strong>dataSource</strong>"
class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
</bean>
<bean id="userDao" class="com.foo.JdbcUserDao">
    <!-- Spring will do the cast automatically (as usual) -->
    <property name="dataSource" ref="<strong>dataSource</strong>"/>
</bean>
```

After...

```
<jee:jndi-lookup id="<strong>dataSource</strong>" jndi-name="jdbc/MyDataSource"/>

<bean id="userDao" class="com.foo.JdbcUserDao">
    <!-- Spring will do the cast automatically (as usual) -->
    <property name="dataSource" ref="<strong>dataSource</strong>"/>
</bean>
```

<jee:jndi-lookup/> (with single JNDI environment setting)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
        </props>
    </property>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <jee:environment>foo=bar</jee:environment>
</jee:jndi-lookup>
```

<jee:jndi-lookup/> (with multiple JNDI environment settings)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <!-- newline-separated, key-value pairs for the environment (standard Properties
format) -->
    <jee:environment>
        foo=bar
        ping=pong
    </jee:environment>
</jee:jndi-lookup>
```

<jee:jndi-lookup/> (complex)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="cache" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="lookupOnStartup" value="false"/>
    <property name="expectedType" value="com.myapp.DefaultFoo"/>
    <property name="proxyInterface" value="com.myapp.Foo"/>
</bean>
```

After...

```
<jee:jndi-lookup id="simple"
        jndi-name="jdbc/MyDataSource"
        cache="true"
        resource-ref="true"
        lookup-on-startup="false"
        expected-type="com.myapp.DefaultFoo"
        proxy-interface="com.myapp.Foo"/>
```

<jee:local-slsb/> (simple)

The `<jee:local-slsb/>` tag configures a reference to an EJB Stateless SessionBean.

Before...

```
<bean id="simple"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/RentalServiceBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
</bean>
```

After...

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"
        business-interface="com.foo.service.RentalService"/>
```

<jee:local-slsb/> (complex)

```
<bean id="complexLocalEjb"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/RentalServiceBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
    <property name="cacheHome" value="true"/>
    <property name="lookupHomeOnStartup" value="true"/>
    <property name="resourceRef" value="true"/>
</bean>
```

After...

```
<jee:local-slsb id="complexLocalEjb"
        jndi-name="ejb/RentalServiceBean"
        business-interface="com.foo.service.RentalService"
        cache-home="true"
        lookup-home-on-startup="true"
        resource-ref="true">
```

<jee:remote-slsb/>

The `<jee:remote-slsb/>` tag configures a reference to a `remote` EJB Stateless SessionBean.

Before...

```
<bean id="complexRemoteEjb"
        class=
"org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/MyRemoteBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
    <property name="cacheHome" value="true"/>
    <property name="lookupHomeOnStartup" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="homeInterface" value="com.foo.service.RentalService"/>
    <property name="refreshHomeOnConnectFailure" value="true"/>
</bean>
```

After...

```
<jee:remote-slsb id="complexRemoteEjb"
        jndi-name="ejb/MyRemoteBean"
        business-interface="com.foo.service.RentalService"
        cache-home="true"
        lookup-home-on-startup="true"
        resource-ref="true"
        home-interface="com.foo.service.RentalService"
        refresh-home-on-connect-failure="true">
```

6.2.4. the lang schema

The `lang` tags deal with exposing objects that have been written in a dynamic language such as JRuby or Groovy as beans in the Spring container.

These tags (and the dynamic language support) are comprehensively covered in the chapter entitled Dynamic language support. Please do consult that chapter for full details on this support and the `lang` tags themselves.

In the interest of completeness, to use the tags in the `lang` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `lang` namespace are available to you.

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

</beans>
```

6.2.5. the jms schema

The `jms` tags deal with configuring JMS-related beans such as Spring's MessageListenerContainers. These tags are detailed in the section of the JMS chapter entitled JMS namespace support. Please do consult that chapter for full details on this support and the `jms` tags themselves.

In the interest of completeness, to use the tags in the `jms` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `jms` namespace are available to you.

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

</beans>
```

6.2.6. the tx (transaction) schema

The `tx` tags deal with configuring all of those beans in Spring's comprehensive support for transactions. These tags are covered in the chapter entitled Transaction Management.

You are strongly encouraged to look at the `'spring-tx.xsd'` file that ships with the Spring distribution. This file is (of course), the XML Schema for Spring's transaction configuration, and covers all of the various tags in the `tx` namespace, including attribute defaults and suchlike. This file is documented inline, and thus the information is not repeated here in the interests of adhering to the DRY (Don't Repeat Yourself) principle.

In the interest of completeness, to use the tags in the `tx` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `tx` namespace are available to you.

```
<?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"
        <em>xmlns:tx="http://www.springframework.org/schema/tx"</em>
xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        <em>http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd</em>
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd"> <!-- bean definitions here
-->

</beans>
```

Often when using the tags in the `tx` namespace you will also be using the tags from the `aop` namespace (since the declarative transaction support in Spring is implemented using AOP). The above XML snippet contains the relevant lines needed to reference the `aop` schema so that the tags in the `aop` namespace are available to you.

6.2.7. the aop schema

The `aop` tags deal with configuring all things AOP in Spring: this includes Spring's own proxy-based AOP framework and Spring's integration with the AspectJ AOP framework. These tags are comprehensively covered in the chapter entitled Aspect Oriented Programming with Spring.

In the interest of completeness, to use the tags in the `aop` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `aop` namespace are available to you.

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

</beans>
```

6.2.8. the context schema

The `context` tags deal with `ApplicationContext` configuration that relates to plumbing - that is, not usually beans that are important to an end-user but rather beans that do a lot of grunt work in Spring, such as `BeanfactoryPostProcessors`. The following snippet references the correct schema so that the tags in the `context` namespace are available to you.

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

</beans>
```

The `context` schema was only introduced in Spring 2.5.

<property-placeholder/>

This element activates the replacement of `${…}` placeholders, resolved against the specified properties file (as a Spring resource location). This element is a convenience mechanism that sets up a`PropertyPlaceholderConfigurer` for you; if you need more control over the `PropertyPlaceholderConfigurer`, just define one yourself explicitly.

<annotation-config/>

Activates the Spring infrastructure for various annotations to be detected in bean classes: Spring's `@Required` and `@Autowired`, as well as JSR 250's `@PostConstruct`, `@PreDestroy` and `@Resource` (if available), and JPA's `@PersistenceContext` and `@PersistenceUnit` (if available). Alternatively, you can

choose to activate the individual `BeanPostProcessors` for those annotations explicitly.

This element does *not* activate processing of Spring's `@Transactional` annotation. Use the `<tx:annotation-driven/>` element for that purpose.

<component-scan/>

This element is detailed in Annotation-based container configuration.

<load-time-weaver/>

This element is detailed in Load-time weaving with AspectJ in the Spring Framework.

<spring-configured/>

This element is detailed in Using AspectJ to dependency inject domain objects with Spring.

<mbean-export/>

This element is detailed in Configuring annotation based MBean export.

6.2.9. the tool schema

The `tool` tags are for use when you want to add tooling-specific metadata to your custom configuration elements. This metadata can then be consumed by tools that are aware of this metadata, and the tools can then do pretty much whatever they want with it (validation, etc.).

The `tool` tags are not documented in this release of Spring as they are currently undergoing review. If you are a third party tool vendor and you would like to contribute to this review process, then do mail the Spring mailing list. The currently supported `tool` tags can be found in the file `'spring-tool.xsd'` in the `'src/org/springframework/beans/factory/xml'` directory of the Spring source distribution.

6.2.10. the jdbc schema

The `jdbc` tags allow you to quickly configure an embedded database or initialize an existing data source. These tags are documented in Embedded database support and Initializing a DataSource respectively.

To use the tags in the `jdbc` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `jdbc` namespace are available to you.

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

</beans>
```

6.2.11. the cache schema

The `cache` tags can be used to enable support for Spring's `@CacheEvict`, `@CachePut` and `@Caching` annotations. It it also supports declarative XML-based caching. See Enable caching annotations and Declarative XML-based caching for details.

To use the tags in the `cache` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `cache` namespace are available to you.

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

</beans>
```

6.2.12. the beans schema

Last but not least we have the tags in the `beans` schema. These are the same tags that have been in Spring since the very dawn of the framework. Examples of the various tags in the `beans` schema are not shown here because they are quite comprehensively covered in Dependencies and configuration in detail (and indeed in that entire chapter).

One thing that is new to the beans tags themselves in Spring 2.0 is the idea of arbitrary bean metadata. In Spring 2.0 it is now possible to add zero or more key / value pairs to `<bean/>` XML definitions. What, if anything, is done with this extra metadata is totally up to your own custom logic (and so is typically only of use if you are writing your own custom tags as described in the

appendix entitled Extensible XML authoring).

Find below an example of the `<meta/>` tag in the context of a surrounding `<bean/>` (please note that without any logic to interpret it the metadata is effectively useless as-is).

```
<?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="foo" class="x.y.Foo">
        <em><meta key="cacheName" value="foo"/></em>
        <property name="name" value="Rick"/>
    </bean>

</beans>
```

In the case of the above example, you would assume that there is some logic that will consume the bean definition and set up some caching infrastructure using the supplied metadata.

Chapter 7. Extensible XML authoring

7.1. Introduction

Since version 2.0, Spring has featured a mechanism for schema-based extensions to the basic Spring XML format for defining and configuring beans. This section is devoted to detailing how you would go about writing your own custom XML bean definition parsers and integrating such parsers into the Spring IoC container.

To facilitate the authoring of configuration files using a schema-aware XML editor, Spring's extensible XML configuration mechanism is based on XML Schema. If you are not familiar with Spring's current XML configuration extensions that come with the standard Spring distribution, please first read the appendix entitled[xsd-config].

Creating new XML configuration extensions can be done by following these (relatively) simple steps:

- Authoring an XML schema to describe your custom element(s).
- Coding a custom `NamespaceHandler` implementation (this is an easy step, don't worry).
- Coding one or more `BeanDefinitionParser` implementations (this is where the real work is done).
- Registering the above artifacts with Spring (this too is an easy step).

What follows is a description of each of these steps. For the example, we will create an XML extension (a custom XML element) that allows us to configure objects of the type `SimpleDateFormat` (from the `java.text` package) in an easy manner. When we are done, we will be able to define bean definitions of type `SimpleDateFormat` like this:

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

(Don't worry about the fact that this example is very simple; much more detailed examples follow afterwards. The intent in this first simple example is to walk you through the basic steps involved.)

7.2. Authoring the schema

Creating an XML configuration extension for use with Spring's IoC container starts with authoring an XML Schema to describe the extension. What follows is the schema we'll use to configure `SimpleDateFormat` objects.

```
<!-- myns.xsd (inside package org/springframework/samples/xml) -->

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.mycompany.com/schema/myns"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:beans="http://www.springframework.org/schema/beans"
        targetNamespace="http://www.mycompany.com/schema/myns"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

    <xsd:import namespace="http://www.springframework.org/schema/beans"/>

    <xsd:element name="dateformat">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="beans:identifiedType">
                    <xsd:attribute name="lenient" type="xsd:boolean"/>
                    <xsd:attribute name="pattern" type="xsd:string" use="required"/>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

(The emphasized line contains an extension base for all tags that will be identifiable (meaning they have an `id` attribute that will be used as the bean identifier in the container). We are able to use this attribute because we imported the Spring-provided `'beans'` namespace.)

The above schema will be used to configure `SimpleDateFormat` objects, directly in an XML application context file using the `<myns:dateformat/>` element.

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

Note that after we've created the infrastructure classes, the above snippet of XML will essentially be exactly the same as the following XML snippet. In other words, we're just creating a bean in the container, identified by the name `'dateFormat'` of type `SimpleDateFormat`, with a couple of properties set.

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

The schema-based approach to creating configuration format allows for tight integration with an IDE that has a schema-aware XML editor. Using a properly authored schema, you can use autocompletion to have a user choose between several configuration options defined in the enumeration.

7.3. Coding a NamespaceHandler

In addition to the schema, we need a `NamespaceHandler` that will parse all elements of this specific namespace Spring encounters while parsing configuration files. The `NamespaceHandler` should in our case take care of the parsing of the `myns:dateformat` element.

The `NamespaceHandler` interface is pretty simple in that it features just three methods:

- `init()` - allows for initialization of the `NamespaceHandler` and will be called by Spring before the handler is used
- `BeanDefinition parse(Element, ParserContext)` - called when Spring encounters a top-level element (not nested inside a bean definition or a different namespace). This method can register bean definitions itself and/or return a bean definition.
- `BeanDefinitionHolder decorate(Node, BeanDefinitionHolder, ParserContext)` - called when Spring encounters an attribute or nested element of a different namespace. The decoration of one or more bean definitions is used for example with theout-of-the-box scopes Spring 2.0 supports. We'll start by highlighting a simple example, without using decoration, after which we will show decoration in a somewhat more advanced example.

Although it is perfectly possible to code your own `NamespaceHandler` for the entire namespace (and hence provide code that parses each and every element in the namespace), it is often the case that each top-level XML element in a Spring XML configuration file results in a single bean definition (as in our case, where a single `<myns:dateformat/>` element results in a single `SimpleDateFormat` bean definition). Spring features a number of convenience classes that support this scenario. In this example, we'll make use the `NamespaceHandlerSupport` class:

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        <strong>registerBeanDefinitionParser("dateformat", new
SimpleDateFormatBeanDefinitionParser());</strong>
    }

}
```

The observant reader will notice that there isn't actually a whole lot of parsing logic in this class. Indeed... the `NamespaceHandlerSupport` class has a built in notion of delegation. It supports the registration of any number of `BeanDefinitionParser` instances, to which it will delegate to when it

needs to parse an element in its namespace. This clean separation of concerns allows a `NamespaceHandler` to handle the orchestration of the parsing of *all* of the custom elements in its namespace, while delegating to `BeanDefinitionParsers` to do the grunt work of the XML parsing; this means that each `BeanDefinitionParser` will contain just the logic for parsing a single custom element, as we can see in the next step

7.4. BeanDefinitionParser

A `BeanDefinitionParser` will be used if the `NamespaceHandler` encounters an XML element of the type that has been mapped to the specific bean definition parser (which is `'dateformat'` in this case). In other words, the `BeanDefinitionParser` is responsible for parsing *one* distinct top-level XML element defined in the schema. In the parser, we'll have access to the XML element (and thus its subelements too) so that we can parse our custom XML content, as can be seen in the following example:

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;

import java.text.SimpleDateFormat;

public class SimpleDateFormatBeanDefinitionParser extends
AbstractSingleBeanDefinitionParser { ①

    protected Class getBeanClass(Element element) {
        return SimpleDateFormat.class; ②
    }

    protected void doParse(Element element, BeanDefinitionBuilder bean) {
        // this will never be null since the schema explicitly requires that a value
be supplied
        String pattern = element.getAttribute("pattern");
        bean.addConstructorArg(pattern);

        // this however is an optional property
        String lenient = element.getAttribute("lenient");
        if (StringUtils.hasText(lenient)) {
            bean.addPropertyValue("lenient", Boolean.valueOf(lenient));
        }
    }

}
```

① We use the Spring-provided `AbstractSingleBeanDefinitionParser` to handle a lot of the basic grunt work of creating a *single* `BeanDefinition`.

② We supply the `AbstractSingleBeanDefinitionParser` superclass with the type that our single `BeanDefinition` will represent.

In this simple case, this is all that we need to do. The creation of our single `BeanDefinition` is handled by the `AbstractSingleBeanDefinitionParser` superclass, as is the extraction and setting of the bean definition's unique identifier.

7.5. Registering the handler and the schema

The coding is finished! All that remains to be done is to somehow make the Spring XML parsing infrastructure aware of our custom element; we do this by registering our custom `namespaceHandler` and custom XSD file in two special purpose properties files. These properties files are both placed in a `'META-INF'` directory in your application, and can, for example, be distributed alongside your binary classes in a JAR file. The Spring XML parsing infrastructure will automatically pick up your new extension by consuming these special properties files, the formats of which are detailed below.

7.5.1. 'META-INF/spring.handlers'

The properties file called `'spring.handlers'` contains a mapping of XML Schema URIs to namespace handler classes. So for our example, we need to write the following:

```
http\://www.mycompany.com/schema/myns=org.springframework.samples.xml.MyNamespaceHandl
er
```

(The `':'` character is a valid delimiter in the Java properties format, and so the `':'` character in the URI needs to be escaped with a backslash.)

The first part (the key) of the key-value pair is the URI associated with your custom namespace extension, and needs to *match exactly* the value of the `'targetNamespace'` attribute as specified in your custom XSD schema.

7.5.2. 'META-INF/spring.schemas'

The properties file called `'spring.schemas'` contains a mapping of XML Schema locations (referred to along with the schema declaration in XML files that use the schema as part of the `'xsi:schemaLocation'` attribute) to *classpath* resources. This file is needed to prevent Spring from absolutely having to use a default `EntityResolver` that requires Internet access to retrieve the schema file. If you specify the mapping in this properties file, Spring will search for the schema on the classpath (in this case `'myns.xsd'` in the `'org.springframework.samples.xml'` package):

```
http\://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xs
d
```

The upshot of this is that you are encouraged to deploy your XSD file(s) right alongside the `NamespaceHandler` and `BeanDefinitionParser` classes on the classpath.

7.6. Using a custom extension in your Spring XML configuration

Using a custom extension that you yourself have implemented is no different from using one of the 'custom' extensions that Spring provides straight out of the box. Find below an example of using the custom `<dateformat/>` element developed in the previous steps in a Spring 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"
    xmlns:myns="http://www.mycompany.com/schema/myns"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.mycompany.com/schema/myns
http://www.mycompany.com/schema/myns/myns.xsd">

    <!-- as a top-level bean -->
    <myns:dateformat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm" lenient="true
"/>

    <bean id="jobDetailTemplate" abstract="true">
        <property name="dateFormat">
            <!-- as an inner bean -->
            <myns:dateformat pattern="HH:mm MM-dd-yyyy"/>
        </property>
    </bean>

</beans>
```

7.7. Meatier examples

Find below some much meatier examples of custom XML extensions.

7.7.1. Nesting custom tags within custom tags

This example illustrates how you might go about writing the various artifacts required to satisfy a target of the following configuration:

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

    <foo:component id="bionic-family" name="Bionic-1">
        <foo:component name="Mother-1">
            <foo:component name="Karate-1"/>
            <foo:component name="Sport-1"/>
        </foo:component>
        <foo:component name="Rock-1"/>
    </foo:component>

</beans>
```

The above configuration actually nests custom extensions within each other. The class that is actually configured by the above `<foo:component/>` element is the `Component` class (shown directly below). Notice how the `Component` class does *not* expose a setter method for the `'components'` property; this makes it hard (or rather impossible) to configure a bean definition for the `Component` class using setter injection.

```
package com.foo;

import java.util.ArrayList;
import java.util.List;

public class Component {

    private String name;
    private List<Component> components = new ArrayList<Component> ();

    // mmm, there is no setter method for the 'components'
    public void addComponent(Component component) {
        this.components.add(component);
    }

    public List<Component> getComponents() {
        return components;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

The typical solution to this issue is to create a custom `FactoryBean` that exposes a setter property for the `'components'` property.

```
package com.foo;

import org.springframework.beans.factory.FactoryBean;

import java.util.List;

public class ComponentFactoryBean implements FactoryBean<Component> {

    private Component parent;
    private List<Component> children;

    public void setParent(Component parent) {
        this.parent = parent;
    }

    public void setChildren(List<Component> children) {
        this.children = children;
    }

    public Component getObject() throws Exception {
        if (this.children != null && this.children.size() > 0) {
            for (Component child : children) {
                this.parent.addComponent(child);
            }
        }
        return this.parent;
    }

    public Class<Component> getObjectType() {
        return Component.class;
    }

    public boolean isSingleton() {
        return true;
    }

}
```

This is all very well, and does work nicely, but exposes a lot of Spring plumbing to the end user. What we are going to do is write a custom extension that hides away all of this Spring plumbing. If we stick to the steps described previously, we'll start off by creating the XSD schema to define the structure of our custom tag.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/component"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.foo.com/schema/component"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

    <xsd:element name="component">
        <xsd:complexType>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="component"/>
            </xsd:choice>
            <xsd:attribute name="id" type="xsd:ID"/>
            <xsd:attribute name="name" use="required" type="xsd:string"/>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

We'll then create a custom `NamespaceHandler`.

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class ComponentNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("component", new ComponentBeanDefinitionParser()
);
    }

}
```

Next up is the custom `BeanDefinitionParser`. Remember that what we are creating is a `BeanDefinition` describing a `ComponentFactoryBean`.

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.ManagedList;
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.util.xml.DomUtils;
import org.w3c.dom.Element;
```

```
import java.util.List;

public class ComponentBeanDefinitionParser extends AbstractBeanDefinitionParser {

    protected AbstractBeanDefinition parseInternal(Element element, ParserContext
parserContext) {
        return parseComponentElement(element);
    }

    private static AbstractBeanDefinition parseComponentElement(Element element) {
        BeanDefinitionBuilder factory = BeanDefinitionBuilder.rootBeanDefinition
(ComponentFactoryBean.class);
        factory.addPropertyValue("parent", parseComponent(element));

        List<Element> childElements = DomUtils.getChildElementsByTagName(element,
"component");
        if (childElements != null && childElements.size() > 0) {
            parseChildComponents(childElements, factory);
        }

        return factory.getBeanDefinition();
    }

    private static BeanDefinition parseComponent(Element element) {
        BeanDefinitionBuilder component = BeanDefinitionBuilder.rootBeanDefinition
(Component.class);
        component.addPropertyValue("name", element.getAttribute("name"));
        return component.getBeanDefinition();
    }

    private static void parseChildComponents(List<Element> childElements,
BeanDefinitionBuilder factory) {
        ManagedList<BeanDefinition> children = new ManagedList<BeanDefinition>
(childElements.size());
        for (Element element : childElements) {
            children.add(parseComponentElement(element));
        }
        factory.addPropertyValue("children", children);
    }

}
```

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

```
# in 'META-INF/spring.handlers'
http\://www.foo.com/schema/component=com.foo.ComponentNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/component/component.xsd=com/foo/component.xsd
```

7.7.2. Custom attributes on 'normal' elements

Writing your own custom parser and the associated artifacts isn't hard, but sometimes it is not the right thing to do. Consider the scenario where you need to add metadata to already existing bean definitions. In this case you certainly don't want to have to go off and write your own entire custom extension; rather you just want to add an additional attribute to the existing bean definition element.

By way of another example, let's say that the service class that you are defining a bean definition for a service object that will (unknown to it) be accessing a clustered JCache, and you want to ensure that the named JCache instance is eagerly started within the surrounding cluster:

```
<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"
        jcache:cache-name="checking.account">
    <!-- other dependencies here... -->
</bean>
```

What we are going to do here is create another `BeanDefinition` when the `'jcache:cache-name'` attribute is parsed; this `BeanDefinition` will then initialize the named JCache for us. We will also modify the existing `BeanDefinition` for the `'checkingAccountService'` so that it will have a dependency on this new JCache-initializing `BeanDefinition`.

```
package com.foo;

public class JCacheInitializer {

    private String name;

    public JCacheInitializer(String name) {
        this.name = name;
    }

    public void initialize() {
        // lots of JCache API calls to initialize the named cache...
    }

}
```

Now onto the custom extension. Firstly, the authoring of the XSD schema describing the custom attribute (quite easy in this case).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/jcache"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.foo.com/schema/jcache"
        elementFormDefault="qualified">

    <xsd:attribute name="cache-name" type="xsd:string"/>

</xsd:schema>
```

Next, the associated `NamespaceHandler`.

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class JCacheNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        super.registerBeanDefinitionDecoratorForAttribute("cache-name",
            new JCacheInitializingBeanDefinitionDecorator());
    }

}
```

Next, the parser. Note that in this case, because we are going to be parsing an XML attribute, we write a `BeanDefinitionDecorator` rather than a `BeanDefinitionParser`.

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.BeanDefinitionDecorator;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class JCacheInitializingBeanDefinitionDecorator implements
BeanDefinitionDecorator {

    private static final String[] EMPTY_STRING_ARRAY = new String[0];
```

```
    public BeanDefinitionHolder decorate(Node source, BeanDefinitionHolder holder,
            ParserContext ctx) {
        String initializerBeanName = registerJCacheInitializer(source, ctx);
        createDependencyOnJCacheInitializer(holder, initializerBeanName);
        return holder;
    }

    private void createDependencyOnJCacheInitializer(BeanDefinitionHolder holder,
            String initializerBeanName) {
        AbstractBeanDefinition definition = ((AbstractBeanDefinition) holder
.getBeanDefinition());
        String[] dependsOn = definition.getDependsOn();
        if (dependsOn == null) {
            dependsOn = new String[]{initializerBeanName};
        } else {
            List dependencies = new ArrayList(Arrays.asList(dependsOn));
            dependencies.add(initializerBeanName);
            dependsOn = (String[]) dependencies.toArray(EMPTY_STRING_ARRAY);
        }
        definition.setDependsOn(dependsOn);
    }

    private String registerJCacheInitializer(Node source, ParserContext ctx) {
        String cacheName = ((Attr) source).getValue();
        String beanName = cacheName + "-initializer";
        if (!ctx.getRegistry().containsBeanDefinition(beanName)) {
            BeanDefinitionBuilder initializer = BeanDefinitionBuilder
.rootBeanDefinition(JCacheInitializer.class);
            initializer.addConstructorArg(cacheName);
            ctx.getRegistry().registerBeanDefinition(beanName, initializer
.getBeanDefinition());
        }
        return beanName;
    }

}
```

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

```
# in 'META-INF/spring.handlers'
http\://www.foo.com/schema/jcache=com.foo.JCacheNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/jcache/jcache.xsd=com/foo/jcache.xsd
```

7.8. Further Resources

Find below links to further resources concerning XML Schema and the extensible XML support described in this chapter.

- The XML Schema Part 1: Structures Second Edition
- The XML Schema Part 2: Datatypes Second Edition

Chapter 8. spring JSP Tag Library

8.1. Introduction

One of the view technologies you can use with the Spring Framework is Java Server Pages (JSPs). To help you implement views using Java Server Pages the Spring Framework provides you with some tags for evaluating errors, setting themes and outputting internationalized messages.

Please note that the various tags generated by this form tag library are compliant with the XHTML-1.0-Strict specification and attendant DTD.

This appendix describes the `spring.tld` tag library.

8.2. The argument tag

Argument tag based on the JSTL fmt:param tag. The purpose is to support arguments inside the message and theme tags.

Table 1. Attributes

Attribute	Required?	Runtime Expression?	Description
value	false	true	The value of the argument.

8.3. The bind tag

Provides BindStatus object for the given bind path. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table 2. Attributes

Attribute	Required?	Runtime Expression?	Description
htmlEscape	false	true	Set HTML escaping for this tag, as boolean value. Overrides the default HTML escaping setting for the current page.
ignoreNestedPath	false	true	Set whether to ignore a nested path, if any. Default is to not ignore.
path	true	true	The path to the bean or bean property to bind status information for. For instance account.name, company.address.zipCode or just employee. The status object will exported to the page scope, specifically for this bean or bean property

8.4. The escapeBody tag

Escapes its enclosed body content, applying HTML escaping and/or JavaScript escaping. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table 3. Attributes

Attribute	Required?	Runtime Expression?	Description
htmlEscape	false	true	Set HTML escaping for this tag, as boolean value. Overrides the default HTML escaping setting for the current page.
javaScriptEscape	false	true	Set JavaScript escaping for this tag, as boolean value. Default is false.

8.5. The eval tag

Evaluates a Spring expression (SpEL) and either prints the result or assigns it to a variable.

Table 4. Attributes

Attribute	Required?	Runtime Expression?	Description
expression	true	true	The expression to evaluate.
htmlEscape	false	true	Set HTML escaping for this tag, as a boolean value. Overrides the default HTML escaping setting for the current page.
javaScriptEscape	false	true	Set JavaScript escaping for this tag, as a boolean value. Default is false.

Attribute	Required?	Runtime Expression?	Description
scope	false	true	The scope for the var. 'application', 'session', 'request' and 'page' scopes are supported. Defaults to page scope. This attribute has no effect unless the var attribute is also defined.
var	false	true	The name of the variable to export the evaluation result to. If not specified the evaluation result is converted to a String and written as output.

8.6. The hasBindErrors tag

Provides Errors instance in case of bind errors. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table 5. Attributes

Attribute	Required?	Runtime Expression?	Description
htmlEscape	false	true	Set HTML escaping for this tag, as boolean value. Overrides the default HTML escaping setting for the current page.
name	true	true	The name of the bean in the request, that needs to be inspected for errors. If errors are available for this bean, they will be bound under the 'errors' key.

8.7. The htmlEscape tag

Sets default HTML escape value for the current page. Overrides a "defaultHtmlEscape" context-param in web.xml, if any.

Table 6. Attributes

Attribute	Required?	Runtime Expression?	Description
defaultHtmlEscape	true	true	Set the default value for HTML escaping, to be put into the current PageContext.

8.8. The message tag

Retrieves the message with the given code, or text if code isn't resolvable. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table 7. Attributes

Attribute	Required?	Runtime Expression?	Description
arguments	false	true	Set optional message arguments for this tag, as a (comma-)delimited String (each String argument can contain JSP EL), an Object array (used as argument array), or a single Object (used as single argument).
argumentSeparator	false	true	The separator character to be used for splitting the arguments string value; defaults to a 'comma' (',').
code	false	true	The code (key) to use when looking up the message. If code is not provided, the text attribute will be used.
htmlEscape	false	true	Set HTML escaping for this tag, as boolean value. Overrides the default HTML escaping setting for the current page.
javaScriptEscape	false	true	Set JavaScript escaping for this tag, as boolean value. Default is false.
message	false	true	A MessageSourceResolvable argument (direct or through JSP EL). Fits nicely when used in conjunction with Spring's own validation error classes which all implement the MessageSourceResolvable interface. For example, this allows you to iterate over all of the errors in a form, passing each error (using a runtime expression) as the value of this 'message' attribute, thus effecting the easy display of such error messages.
scope	false	true	The scope to use when exporting the result to a variable. This attribute is only used when var is also set. Possible values are page, request, session and application.
text	false	true	Default text to output when a message for the given code could not be found. If both text and code are not set, the tag will output null.
var	false	true	The string to use when binding the result to the page, request, session or application scope. If not specified, the result gets outputted to the writer (i.e. typically directly to the JSP).

8.9. The nestedPath tag

Sets a nested path to be used by the bind tag's path.

Table 8. Attributes

Attribute	Required?	Runtime Expression?	Description
path	true	true	Set the path that this tag should apply. E.g. 'customer' to allow bind paths like 'address.street' rather than 'customer.address.street'.

8.10. The param tag

Parameter tag based on the JSTL c:param tag. The sole purpose is to support params inside the url tag.

Table 9. Attributes

Attribute	Required?	Runtime Expression?	Description
name	true	true	The name of the parameter.
value	false	true	The value of the parameter.

8.11. The theme tag

Retrieves the theme message with the given code, or text if code isn't resolvable. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table 10. Attributes

Attribute	Required?	Runtime Expression?	Description
arguments	false	true	Set optional message arguments for this tag, as a (comma-)delimited String (each String argument can contain JSP EL), an Object array (used as argument array), or a single Object (used as single argument).
argumentSeparator	false	true	The separator character to be used for splitting the arguments string value; defaults to a 'comma' (',').
code	false	true	The code (key) to use when looking up the message. If code is not provided, the text attribute will be used.
htmlEscape	false	true	Set HTML escaping for this tag, as boolean value. Overrides the default HTML escaping setting for the current page.
javaScriptEscape	false	true	Set JavaScript escaping for this tag, as boolean value. Default is false.
message	false	true	A MessageSourceResolvable argument (direct or through JSP EL).

Attribute	Required?	Runtime Expression?	Description
scope	false	true	The scope to use when exporting the result to a variable. This attribute is only used when var is also set. Possible values are page, request, session and application.
text	false	true	Default text to output when a message for the given code could not be found. If both text and code are not set, the tag will output null.
var	false	true	The string to use when binding the result to the page, request, session or application scope. If not specified, the result gets outputted to the writer (i.e. typically directly to the JSP).

8.12. The transform tag

Provides transformation of variables to Strings, using an appropriate custom PropertyEditor from BindTag (can only be used inside BindTag). The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a 'defaultHtmlEscape' context-param in web.xml).

Table 11. Attributes

Attribute	Required?	Runtime Expression?	Description
htmlEscape	false	true	Set HTML escaping for this tag, as boolean value. Overrides the default HTML escaping setting for the current page.
scope	false	true	The scope to use when exported the result to a variable. This attribute is only used when var is also set. Possible values are page, request, session and application.
value	true	true	The value to transform. This is the actual object you want to have transformed (for instance a Date). Using the PropertyEditor that is currently in use by the 'spring:bind' tag.
var	false	true	The string to use when binding the result to the page, request, session or application scope. If not specified, the result gets outputted to the writer (i.e. typically directly to the JSP).

8.13. The url tag

Creates URLs with support for URI template variables, HTML/XML escaping, and Javascript escaping. Modeled after the JSTL c:url tag with backwards compatibility in mind.

Table 12. Attributes

Attribute	Required?	Runtime Expression?	Description
value	true	true	The URL to build. This value can include template {placeholders} that are replaced with the URL encoded value of the named parameter. Parameters must be defined using the param tag inside the body of this tag.
context	false	true	Specifies a remote application context path. The default is the current application context path.
var	false	true	The name of the variable to export the URL value to. If not specified the URL is written as output.
scope	false	true	The scope for the var. 'application', 'session', 'request' and 'page' scopes are supported. Defaults to page scope. This attribute has no effect unless the var attribute is also defined.
htmlEscape	false	true	Set HTML escaping for this tag, as a boolean value. Overrides the default HTML escaping setting for the current page.
javaScriptEscape	false	true	Set JavaScript escaping for this tag, as a boolean value. Default is false.

Chapter 9. spring-form JSP Tag Library

9.1. Introduction

One of the view technologies you can use with the Spring Framework is Java Server Pages (JSPs). To help you implement views using Java Server Pages the Spring Framework provides you with some tags for evaluating errors, setting themes and outputting internationalized messages.

Please note that the various tags generated by this form tag library are compliant with the XHTML-1.0-Strict specification and attendant DTD.

This appendix describes the `spring-form.tld` tag library.

9.2. The button tag

Renders a form field label in an HTML 'button' tag.

Table 13. Attributes

Attribute	Required?	Runtime Expression?	Description
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
id	false	true	HTML Standard Attribute
name	false	true	The name attribute for the HTML button tag

Attribute	Required?	Runtime Expression?	Description
value	false	true	The name attribute for the HTML button tag

9.3. The checkbox tag

Renders an HTML 'input' tag with type 'checkbox'.

Table 14. Attributes

Attribute	Required?	Runtime Expression?	Description
accesskey	false	true	HTML Standard Attribute
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
dir	false	true	HTML Standard Attribute
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
label	false	true	Value to be displayed as part of the tag
lang	false	true	HTML Standard Attribute
onblur	false	true	HTML Event Attribute
onchange	false	true	HTML Event Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onfocus	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
path	true	true	Path to property for data binding
tabindex	false	true	HTML Standard Attribute

Attribute	Required?	Runtime Expression?	Description
title	false	true	HTML Standard Attribute
value	false	true	HTML Optional Attribute

9.4. The checkboxes tag

Renders multiple HTML 'input' tags with type 'checkbox'.

Table 15. Attributes

Attribute	Required?	Runtime Expression?	Description
accesskey	false	true	HTML Standard Attribute
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
delimiter	false	true	Delimiter to use between each 'input' tag with type 'checkbox'. There is no delimiter by default.
dir	false	true	HTML Standard Attribute
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
element	false	true	Specifies the HTML element that is used to enclose each 'input' tag with type 'checkbox'. Defaults to 'span'.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
itemLabel	false	true	Value to be displayed as part of the 'input' tags with type 'checkbox'
items	true	true	The Collection, Map or array of objects used to generate the 'input' tags with type 'checkbox'
itemValue	false	true	Name of the property mapped to 'value' attribute of the 'input' tags with type 'checkbox'
lang	false	true	HTML Standard Attribute
onblur	false	true	HTML Event Attribute
onchange	false	true	HTML Event Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onfocus	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute

Attribute	Required?	Runtime Expression?	Description
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
path	true	true	Path to property for data binding
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute

9.5. The errors tag

Renders field errors in an HTML 'span' tag.

Table 16. Attributes

Attribute	Required?	Runtime Expression?	Description
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
delimiter	false	true	Delimiter for displaying multiple error messages. Defaults to the br tag.
dir	false	true	HTML Standard Attribute
element	false	true	Specifies the HTML element that is used to render the enclosing errors.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
lang	false	true	HTML Standard Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute

Attribute	Required?	Runtime Expression?	Description
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
path	false	true	Path to errors object for data binding
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute

9.6. The form tag

Renders an HTML 'form' tag and exposes a binding path to inner tags for binding.

Table 17. Attributes

Attribute	Required?	Runtime Expression?	Description
acceptCharset	false	true	Specifies the list of character encodings for input data that is accepted by the server processing this form. The value is a space- and/or comma-delimited list of charset values. The client must interpret this list as an exclusive-or list, i.e., the server is able to accept any single character encoding per entity received.
action	false	true	HTML Required Attribute
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
dir	false	true	HTML Standard Attribute
enctype	false	true	HTML Optional Attribute
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
lang	false	true	HTML Standard Attribute
method	false	true	HTML Optional Attribute
methodParam	false	true	The parameter name used for HTTP methods other then GET and POST. Default is '_method'.
modelAttribute	false	true	Name of the model attribute under which the form object is exposed. Defaults to 'command'.
name	false	true	HTML Standard Attribute - added for backwards compatibility cases
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute

Attribute	Required?	Runtime Expression?	Description
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
onreset	false	true	HTML Event Attribute
onsubmit	false	true	HTML Event Attribute
servletRelativeAction	false	true	Action reference to be appended to the current servlet path
target	false	true	HTML Optional Attribute
title	false	true	HTML Standard Attribute

9.7. The hidden tag

Renders an HTML 'input' tag with type 'hidden' using the bound value.

Table 18. Attributes

Attribute	Required?	Runtime Expression?	Description
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
path	true	true	Path to property for data binding

9.8. The input tag

Renders an HTML 'input' tag with type 'text' using the bound value.

Table 19. Attributes

Attribute	Required?	Runtime Expression?	Description
accesskey	false	true	HTML Standard Attribute
alt	false	true	HTML Optional Attribute
autocomplete	false	true	Common Optional Attribute
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute

Attribute	Required?	Runtime Expression?	Description
dir	false	true	HTML Standard Attribute
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
lang	false	true	HTML Standard Attribute
maxlength	false	true	HTML Optional Attribute
onblur	false	true	HTML Event Attribute
onchange	false	true	HTML Event Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onfocus	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
onselect	false	true	HTML Event Attribute
path	true	true	Path to property for data binding
readonly	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will make the HTML element readonly.
size	false	true	HTML Optional Attribute
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute

9.9. The label tag

Renders a form field label in an HTML 'label' tag.

Table 20. Attributes

Attribute	Required?	Runtime Expression?	Description
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute.
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used only when errors are present.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
dir	false	true	HTML Standard Attribute
for	false	true	HTML Standard Attribute
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
lang	false	true	HTML Standard Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
path	true	true	Path to errors object for data binding
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute

9.10. The option tag

Renders a single HTML 'option'. Sets 'selected' as appropriate based on bound value.

Table 21. Attributes

Attribute	Required?	Runtime Expression?	Description
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
dir	false	true	HTML Standard Attribute

Attribute	Required?	Runtime Expression?	Description
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
label	false	true	HTML Optional Attribute
lang	false	true	HTML Standard Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute
value	true	true	HTML Optional Attribute

9.11. The options tag

Renders a list of HTML 'option' tags. Sets 'selected' as appropriate based on bound value.

Table 22. Attributes

Attribute	Required?	Runtime Expression?	Description
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
dir	false	true	HTML Standard Attribute
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.

Attribute	Required?	Runtime Expression?	Description
id	false	true	HTML Standard Attribute
itemLabel	false	true	Name of the property mapped to the inner text of the 'option' tag
items	true	true	The Collection, Map or array of objects used to generate the inner 'option' tags
itemValue	false	true	Name of the property mapped to 'value' attribute of the 'option' tag
lang	false	true	HTML Standard Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute

9.12. The password tag

Renders an HTML 'input' tag with type 'password' using the bound value.

Table 23. Attributes

Attribute	Required?	Runtime Expression?	Description
accesskey	false	true	HTML Standard Attribute
alt	false	true	HTML Optional Attribute
autocomplete	false	true	Common Optional Attribute
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
dir	false	true	HTML Standard Attribute

Attribute	Required?	Runtime Expression?	Description
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
lang	false	true	HTML Standard Attribute
maxlength	false	true	HTML Optional Attribute
onblur	false	true	HTML Event Attribute
onchange	false	true	HTML Event Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onfocus	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
onselect	false	true	HTML Event Attribute
path	true	true	Path to property for data binding
readonly	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will make the HTML element readonly.
showPassword	false	true	Is the password value to be shown? Defaults to false.
size	false	true	HTML Optional Attribute
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute

9.13. The radiobutton tag

Renders an HTML 'input' tag with type 'radio'.

Table 24. Attributes

Attribute	Required?	Runtime Expression?	Description
accesskey	false	true	HTML Standard Attribute
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
dir	false	true	HTML Standard Attribute
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
label	false	true	Value to be displayed as part of the tag
lang	false	true	HTML Standard Attribute
onblur	false	true	HTML Event Attribute
onchange	false	true	HTML Event Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onfocus	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
path	true	true	Path to property for data binding
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute
value	false	true	HTML Optional Attribute

9.14. The radiobuttons tag

Renders multiple HTML 'input' tags with type 'radio'.

Table 25. Attributes

Attribute	Required?	Runtime Expression?	Description
accesskey	false	true	HTML Standard Attribute
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
delimiter	false	true	Delimiter to use between each 'input' tag with type 'radio'. There is no delimiter by default.
dir	false	true	HTML Standard Attribute
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
element	false	true	Specifies the HTML element that is used to enclose each 'input' tag with type 'radio'. Defaults to 'span'.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
itemLabel	false	true	Value to be displayed as part of the 'input' tags with type 'radio'
items	true	true	The Collection, Map or array of objects used to generate the 'input' tags with type 'radio'
itemValue	false	true	Name of the property mapped to 'value' attribute of the 'input' tags with type 'radio'
lang	false	true	HTML Standard Attribute
onblur	false	true	HTML Event Attribute
onchange	false	true	HTML Event Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onfocus	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
path	true	true	Path to property for data binding

Attribute	Required?	Runtime Expression?	Description
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute

9.15. The select tag

Renders an HTML 'select' element. Supports databinding to the selected option.

Table 26. Attributes

Attribute	Required?	Runtime Expression?	Description
accesskey	false	true	HTML Standard Attribute
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
dir	false	true	HTML Standard Attribute
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
itemLabel	false	true	Name of the property mapped to the inner text of the 'option' tag
items	false	true	The Collection, Map or array of objects used to generate the inner 'option' tags
itemValue	false	true	Name of the property mapped to 'value' attribute of the 'option' tag
lang	false	true	HTML Standard Attribute
multiple	false	true	HTML Optional Attribute
onblur	false	true	HTML Event Attribute
onchange	false	true	HTML Event Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onfocus	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute

Attribute	Required?	Runtime Expression?	Description
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
path	true	true	Path to property for data binding
size	false	true	HTML Optional Attribute
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute

9.16. The textarea tag

Renders an HTML 'textarea'.

Table 27. Attributes

Attribute	Required?	Runtime Expression?	Description
accesskey	false	true	HTML Standard Attribute
cols	false	true	HTML Required Attribute
cssClass	false	true	Equivalent to "class" - HTML Optional Attribute
cssErrorClass	false	true	Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors.
cssStyle	false	true	Equivalent to "style" - HTML Optional Attribute
dir	false	true	HTML Standard Attribute
disabled	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will disable the HTML element.
htmlEscape	false	true	Enable/disable HTML escaping of rendered values.
id	false	true	HTML Standard Attribute
lang	false	true	HTML Standard Attribute
onblur	false	true	HTML Event Attribute
onchange	false	true	HTML Event Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onfocus	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute

Attribute	Required?	Runtime Expression?	Description
onmousedown	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
onselect	false	true	HTML Event Attribute
path	true	true	Path to property for data binding
readonly	false	true	HTML Optional Attribute. Setting the value of this attribute to 'true' will make the HTML element readonly.
rows	false	true	HTML Required Attribute
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute