SPRING

Ejemplo1:

public class FileReader {

private StringBuilder builder = null;

private Scanner scanner = null;

public FileReader(String fileName) {

scanner = new Scanner(new File(fileName));

builder = new StringBuilder();

}

public String read() {

while (scanner.hasNext()) {

builder.append(scanner.next());

}

return builder.toString();

}

}

Ejemplo2:

public interface Reader {

String read();

}

private class XXXReader implements Reader {

public String read() {

//impl goes here

}

}

public class DataReaderClient {

private Reader reader = null;

public DataReaderClient(Reader reader) {

this.reader = reader;

}

private String fetchData() {

return reader.read();

}

public static void main(String[] args) {

}

}

The Spring container is simply a holder of the bean instances that were created from the XML file.

public class DataReaderClient

{

private ApplicationContext ctx = null;

private Reader reader = null;

public DataReaderClient() {

ctx = new ClasspathXmlApplicationContext(“reader-beans.xml”);

}

public String getData() {

reader = (Reader) ctx.getBean(“fileReader”);

reader.fetchData();

}

public static void main(String[] args) {

DataReaderClient client = new DataReaderClient();

System.out.println(“Data:”+client.getData());

}

}

<bean name="fileReader" class="com.oreilly.justspring.ch1.FileReader"

<constructor-arg value="src/main/resources/myfile.txt"/>

</bean>

The ctx = new ClasspathXmlApplicationContext(“reader-beans.xml”) statement creates

this container of beans defined in the *reader-beans.xml*. In our XML file, we have

defined a single bean: FileReader. The bean was given an unique name: fileReader.

Once the container is created, the client will have to use an API provided by the Context in order to access all the beans that were defined in the XML file.

En Spring todas las dependencias y asociaciones son creadas por el contenedor de IoC (Inversión de Control) y estas son inyectadas dentro del programa principal por medio de propiedades.

ApplicationContext y ClasspathXmlApplicationContext son puntos claves para entender Spring.

The instantiation of the ApplicationContext creates the container that consists of the objects defined in that XML file.

En un programa normal el método main crea todas las dependencias, conjunto de propiedades e instancia las instancias de la aplicación en progreso whereas (mientras) en Spring esta responsabilidad es tomada por el Spring Container.

Ciclo de vida

• The framework factory loads the bean definitions and creates the bean.

• The bean is then populated with the properties as declared in the bean definitions.

If the property is a reference to another bean, that other bean will be created and

populated (poblado), and the reference is injected prior to injecting it into this bean.

• If your bean implements any of Spring’s interfaces, such as **BeanNameAware** or **BeanFactoryAware**, appropriate methods will be called.

• The framework also invokes any **BeanPostProcessor’s** associated with your bean for pre-initialzation.

• The init-method, if specified, is invoked on the bean.

• The post-initialization will be performed if specified on the bean.

Modos Hooks (gancho)

Cuando el bean es creado por la inversión de control, se le puede pedir a Spring que invoque un método especifico en la inicialización del bean. Este método provee una oportunidad a el bean para realizar algo de inicialización, como son crear estructuras de datos, thread pool, etc.

**<bean name="fileReader" class="com.oreilly.justspring.ch2.FileReader"**

**init-method="init">**

**destroy-method**

Similar a la inicialización, el framework también puede invocar un método destroy para limpiar antes de destruir el bean.

**<bean name="fileReader" class="com.oreilly.justspring.ch2.FileReader"**

**destroy-method="cleanUp">**

**Bean Post Processors**

Spring provee un par de interfaces con un solo método de implementación para la inicialización y el procesamiento interno (housekeeping).

**-InitializingBean** public void afterPropertiesSet() { //inicilización }

**-DisposableBean** public void destroy() {}

La desventaja de utilizar estas interfaces, es que estamos cerrándonos a utilizar Spring, lo cual no es conveniente; es mejor utilizar los métodos init y destroy.

**Bean Scopes (**singleton and prototype**)**

**Singleton:** Cuando se necesita una y una única instancia de un bean es necesario poner está propiedad a true, tal es el caso de un service o un factory. Esto es por contexto o contenedor y el de java es por proceso o por clase cargada.

**<bean name="fileReader" class="com.ipn.FileReader" singleton="true">**

El valor por default es singleton

**prototype:** Se crea una instancia nueva cada vez que la llamada es realizada para recuperar el bean.

**<bean name="trade" ref="com.ipn.Trade" singleton="false"**

**init-method="initId"/>**

**<bean id="ticket"**

**class="com.springinaction.springidol.Ticket" scope="prototype" />**

Se puede realizar **Wiring** properties con el name space **p**, lo cual no es muy alentador. Se debe de agregar al esquema

xmlns:p="<http://www.springframework.org/schema/p>"

<bean id="kenny" class="com.springinaction.springidol.Instrumentalist"

p:song = "Jingle Bells"

p:instrument-ref = "saxophone" />

![A description...](data:None;base64,)

**Injecting Java Collections**

**propiedades**

![A description...](data:None;base64,)

![A description...](data:None;base64,)

**Listas**

![A description...](data:None;base64,)

![A description...](data:None;base64,)

![A description...](data:None;base64,)

**private Set sourceProps = null;**

![A description...](data:None;base64,)

**private Map sourceProps = null;**

![A description...](data:None;base64,)

Injection Inner Beans.

![A description...](data:None;base64,)

Wiring nothing (null)

**<property name="someNonNullProperty"><null/></property>**

A Spring container is the backbone of the framework.

**Containers**

Basicamente es un pool de beans creados en memoria por el framework cuando la aplicación inicia.

Durante este proceso los bean son instanciados en forma de Inversión de control, todas las relaciones y dependencias son creadas. **Todas las relaciones son satisfactorias y las dependencias son injectadas.** Todos los nombre de los beans están disponibles para ser consultados utilizando la API. Los beans son leídos en forma lazily dentro del contenedor, Lo que significa es que a menos que es solicitados por nosotros o por otro bean (parte de la dependdencia) la factory no instanciara este bean. Este mismo concepto utiliza el contenedor de Spring excepto para los Singleton

**Spring Container**

In a Spring-based application, your application objects will live within the Spring container. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from cradle to grave (cuna hasta la tumba)

(or new to finalize(), as the case may be).

it’s important to get to know the container where your objects will be hanging out. Understanding the container helps you grasp (agarrar) how your objects will be managed.

The container is at the core of the Spring Framework. Spring’s container uses

dependency injection (DI) to manage the components that make up an application.

This includes creating associations between collaborating components. As such, these

objects are cleaner and easier to understand, support reuse, and are easy to unit test.

Cae dentro de dos categorias Bean Factories y Application Context.

Los nombres son una especie de nombres inapropiados, ya que no dan ninguna pista sobre si son contenedores y lo que hacen.

Bean Factory Es un contenedor soportado básicamente por inyección de dependencias. ApplicationContext es una extensión del BeanFactory.

![A description...](data:None;base64,)

**BeanFactory Container**

Es el más simple de todos los contenedores que Spring provee. Lo importante de está factory es crear e instanciar beans con todas sus dependientes configuraciones. Cuando se consulta un bean, este es obtenido y todas sus asociaciones y relaciones son ya han sido creadas. Pueden invocar al metodo init y destroy.

Una de las implementaciones más comunes es XmlBeanFactory,

BeanFactory factory = XmlBeanFactory(new FileInputStream("trade-beans.xml"));

TradeService service = (TradeService) factory.getBean("tradeService");

Es más utilizado en pequeñas aplicaciones como mobiles.

**ApplicationContext Container**

Es una extensión del BeanFactory, sus implementaciones más comunes son:

* **FileSystemXmlApplicationContext**: This container loads the definitions of the beans found in the XML file located at a specified file system location. You should provide the full path of the file to the constructor.
* **ClassPathXmlApplicationContext**: In this container, the XML config file is read from the classpath by the container. The only difference between the FileSystemXmlApplicationContext and this one is that this context does not require the full path of the file.
* **WebXmlApplicationContext**: This container loads the XML file with definitions of all beans from within a web application

**Using Static Methods**

Es adecuada si las clases que tienen factories static para crear objetos de instancia.

![A description...](data:None;base64,)

**<bean name="tradeService" class="com.oreilly.justspring.ch3.TradeService"**

**factory-method="getTradeService"/>**

![A description...](data:None;base64,)

**Using Factory Methods**

Si se quiere crear instancias sin utilizar métodos estáticos

**<bean name="empCreator" factor-bean="employeeCreator" factory- method="createEmployee" />**

**<bean name="execCreator" factor-bean="employeeCreator" factory- method="createExecutive" />**

**<bean name="employeeCreator" class="com.oreilly.EmployeeCreator" />**

**Bean Post Processors**

Es una interface que es utilizada para customizar el ciclo de vida del bean.

Tiene dos métodos **postProcessBeforeInitialization** es invocado justo antes de invocar al init-method ó afterPropertiesSet. p**ostProcessAfterInitialization** es llamado justo antes de que la inicialización del bean este completa.

Le tenemos que decir al Framework que estamos utilizando un post processor Si utilizamos el BeanFactory tenemos que invocar ***addBeanPostProcessor.*** Si el contenedor es ApplicationContext definiendolo en el archivo es suficiente. El contenedor automáticamente busca en la clase si es un post processor.

**<bean name="beanPostProcessor" class="com.oreilly..TradePostProcessor"/>**

**Event Handling**

Sometimes, you may want to react to an event that happened in the container so you

can do some custom processing. If you wish to let other beans know that you have finished processing a huge file, for example. Spring provides a way to listen to the events that may allow you to react on the context. And yes, you can use Spring’s framework to publish your custom events, too.

**Listening to Context Events**

In order to listen to the context events, our bean should implement the Application

Listener interface. This interface has just one method: onApplicationEvent(Applica

tionEvent event).

**public class TradeContextEventListener implements ApplicationListener {**

**public onApplicationEvent(ApplicationEvent event) {}**

**<bean id="tradeCtxListener" class="com.ipn.TradeContextEventListener" />**

Spring context publishes the following types of events:

* ContextStartedEvent: This event is published when the ApplicationContext is started. The beans receive a start signal once the ApplicationContext is started. The activities such as polling to database or observing a file system can be started once you receive this type of event.
* ContextStoppedEvent: This is the opposite of the start event. This event is published when the ApplicationContext is stopped. Your bean receives a stop signal from the framework so you can do housekeeping if you wish.
* ContextRefreshedEvent: A refresh event is emitted when the context is either re-

freshed or initialized.

* ContextClosedEvent: This event occurs when the ApplicationContext is closed
* RequestHandledEvent: This is a web-specific event informing the receivers that a web request has been received.

![A description...](data:None;base64,)

**Auto Wiring**

Spring has an advanced concept of autowiring the relationships and dependencies. This means that you don’t have to explicitly mention the properties and their values. but setting autowire property to a value allows the framework to wire them with appropriate properties.

**Autowiring byName**

When autowiring byName is enabled, the framework tries to inject the dependencies by

matching the names of the beans. You have to set the value of autowire to byName when defining the bean in the config. The container looks at the properties of the respective bean on which autowiring byName is set. It then tries to match with the beans defined by the same name in the config file. If matches are found, it will inject **those** beans straight away; otherwise, it will throw exceptions.

![A description...](data:None;base64,)

<bean name="tradeReceiver" class="com.ipn.TradeReceiver"

**autowire="byName"/>**

<bean name="tradePersistor" class="com.ipn.TradePersistor" />

<bean name="tradeTransformer" class="com.ipn.TradeTransformer"/>

We did not declare any properties such as tradePesistor or tradeTransformer for tradeReceiver! How does this work then? Well, the attribute **autowire="byName"** does the magic for you behind the scenes. In our TradeReceiver, we

defined two properties named tradePersistor and tradeTransformer.

**Autowiring byType**

Similar to byName, we need to set the autowire property to byType in order to enable this type of autowiring. In this case, instead of looking for a bean with the same names, the container searches for the same types. Taking the same example of TradeReceiver, setting the autowire="byType" tells the container that it should look for a bean of type

TradePersistor and another one with a type of TradeTransformer. If the container finds the appropriate types, it will inject them into the bean. However, if it finds more than

one bean with the same type defined in the config, a fatal exception is thrown.

But there’s a limitation to autowiring by type. What happens if Spring finds more

than one bean whose type is assignable to the autowired property? In such a case,

Spring isn’t going to guess which bean to autowire and will instead throw an excep-

tion. Consequently, you’re allowed to have only one bean configured that matches the

autowired property. Spring offers two options: you can either identify a primary candidate for autowiring or you can eliminate beans from autowiring candidacy.

To identify a primary autowire candidate, you’ll work with the <bean> element’s

primary attribute.

**Autowiring by Constructor**

You may have already gotten the gist (escencia) of using autowire by constructor. If a bean has a constructor that takes an argument of another bean type, the container looks for that reference and injects it. For example, we define a TradePersistor class with a single constructor that takes a datasource object,

![A description...](data:None;base64,)

If we enable autowiring by constructor, the container looks for an object of type Data

Source and injects into the TradePersistor bean.

**<bean name="tradePersistor" class="com.ipn.TradePersitor"**

**autowire="constructor"/>**

**Mixing Autowiring with Explicit Wiring**

You can get the best of both worlds using auto and explicit wiring. Any ambiguities

encountered while autowiring can be dealt with using explicit wiring. For example, the

TradePersitor can be injected explicitly while the TradeTransformer can be wired au-

tomatically using byName variation.

<bean name="tradeReceiver" class="com.ipn.TradeReceiver" **autowire="byName"**>

<property name="tradePersistor" ref="tradePersistor"/>

</bean>

<bean name="tradePersistor" class="com.ipn.TradePersistor"/>

<bean name="tradeTransformer" class="com.ipn.TradeTransformer"/>

**Autodetect**

Attempts to apply constructor autowiring first. If that fails, byType will be tried.

If you want to autowire your beans, but you can’t decide which type of autowiring to use, have no fear. You can set the autowire attribute to autodetect to let Spring make

the decision for you. For example:

**<bean id="duke" class="com.springinaction.springidol.PoeticJuggler"**

**autowire="autodetect" />**

When a bean has been configured to autowire by autodetect, Spring will attempt to

autowire by constructor first. If a suitable constructor-to-bean match can’t be found,

then Spring will attempt to autowire by type.

**Default Autowiring**

If you find yourself putting the same autowire attribute on every bean in your applica-tion context (or even most of them), you can simplify things by asking Spring to apply

the same autowiring style to all beans that it creates.

**<beans xmlns=".... “ default-autowire="byType"> //default none solo en su archivo**

**Default Autowiring**

Since Spring 2.5, one of the most interesting ways of wiring beans in Spring has been to use annotations to automatically wire bean properties.

Annotation wiring isn’t turned on in the Spring container by default.

<context:annotation-config />

JMS Messaging Models

**Point-to-Point Messaging (**Queue**)**

In point-to-point mode, a message is delivered to a single consumer via a destination.

A publisher publishes a message onto a destination, while a consumer consumes the

message off that destination. When a message is published in P2P mode to a queue,

**one and only one consumer can receive the message**. Even if there are hundreds of consumers connected to that queue, still only one consumer will get that message delivered. Of course, you never know who’s the lucky winner though!

The destination point-to-point model is called Queue.

**Pub/Sub Messaging (**Topic**)**

On the other hand, if a message is published to a destination, **there could be several**

**subscribers each receiving a copy of the message**. The publisher obviously publishes the message once. The subscribers interested will listen to the same destination to consume that message. As mentioned earlier, the JMS Provider will make sure each of the subscribers receives a copy of the message.

The destination in a Pub/Sub model is called a Topic.

**Mother of All: the JmsTemplate class**

You can use the JmsTemplate class for both sending and receiving the messages. The

template class hides all the plumbing needed for connecting to a provider and publishes or receives messages. It has to be wired with a few properties, of which the ConnectionFactory is a must. There are other properties that are required for more features (such as defaultDestination and receiveTimeout parameters). Basically, it uses callbacks such as MessageCreator for the message creation, SessionCallback for associating with a Session, and ProducerCallback for creating a message producer.

![A description...](data:None;base64,)

En el libro Just Spring viene un poco más de este tema.

**Spring JDBC**

The core of the JDBC package revolves (gira) around one class: JdbcTemplate. This class plays the key role in accessing data from your components.

**JdbcTemplate**

The basic and most useful class from the framework is the JdbcTemplate. There are two variants of JdbcTemplate the

**SimpleJdbcTemplate** and **NamedParameterJdbcTemplate.**

The JdbcTemplate class provides the common aspects of database operations, such as inserting and updating data using prepared statements, querying tables using standard SQL queries, invoking stored procedures, etc. It can also iterate over the ResultSet data. The connection management is hidden from the user, and so is the resource pooling and exception management. Regarding (respecto a) the exceptions, one does not have to clutter (desorden) the code with try-catch blocks because the database

specific exceptions are wrapped by Spring’s Runtime Exceptions.

Following the template design pattern, the JdbTemplate provides some callback inter-

faces for you to implement. In these callbacks, you create the necessary business logic.

For example, PreparedStatementCallback is used for creating PreparedStatements, while RowCallbackHandler is where you extract the ResultSet into your domain objects. The CallableStatementCallback is used when executing a stored procedure.

**Configuring JdbcTemplate**

![A description...](data:None;base64,)

Once you have the datasource created, your next job is to create the JdbcTemplate. You

have primarily two options: First, you can **instantiate a JdbcTemplate** in your code base (in your DAO), injecting the DataSource into it. Alternatively, you can **define the**

**JdbcTemplate in the XML file,** wiring the datasource to it. You then inject the JdbcTemplate reference into your DAO class. The JdbcTemplate is a threadsafe object, so you can inject it into any number of DAOs.

<bean id="movieDao" class="com.ipn.MovieDAO" destroy-method="close">

<property name="jdbcTemplate" ref="jdbcTemplate"/>

</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

<property name="dataSource" ref="movieDataSource"/>

</bean>

<bean id="movieDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

...

</bean>

![A description...](data:None;base64,)

![A description...](data:None;base64,)

**What’s new in Spring 2.5**

In November 2007, the Spring team released version 2.5 of the Spring Framework.

* Annotation-driven dependency injection through the @Autowired annotation

and fine-grained auto-wiring control with @Qualifier.

* Support for JSR-250 annotations, including @Resource for dependency injec-

tion of a named resource, as well as @PostConstruct and @PreDestroy for life-

cycle methods.

* Auto-detection of Spring components that are annotated with @Component (or

one of several stereotype annotations).

* An all-new annotation-driven Spring MVC programming model that greatly sim-

plifies Spring web development.

* A new integration test framework that’s based on JUnit 4 and annotations.
* Full Java 6 and Java EE 5 support, including JDBC 4.0, JTA 1.1, JavaMail 1.4, and

JAX-WS 2.0.

* A new bean-name pointcut expression for weaving aspects into Spring beans by

their name.

* Built-in support for AspectJ load-time weaving.
* New XML configuration namespaces, including the context namespace for configuring application context details and a jms namespace for configuring

message-driven beans.

* Support for named parameters in SqlJdbcTemplate.

**What’s new in Spring 3.0**

* AnnFull-scale REST support in Spring MVC, including Spring MVC controllers that

respond to REST-style URLs with XML, JSON, RSS, or any other appropriate

response.

* A new expression language that brings Spring dependency injection to a new

level by enabling injection of values from a variety of sources, including other

beans and system properties.

* New annotations for Spring MVC, including @CookieValue and @RequestHeader, to pull values from cookies and request headers, respectively.
* A new XML namespace for easing configuration of Spring MVC.
* Support for declarative validation with JSR-303 (Bean Validation) annotations.
* Support for the new JSR-330 dependency injection specification.
* Annotation-oriented declaration of asynchronous and scheduled methods.
* A new annotation-based configuration model that allows for nearly XML-free

Spring configuration.

* The Object-to-XML (OXM) mapping functionality from the Spring Web Services project has been moved into the core Spring Framework.

Just as important as what’s new in Spring 3.0 is what’s not in Spring 3.0. Specifically,

starting with Spring 3.0, Java 5 is now required, as Java 1.4 has reached end-of-life and

will no longer be supported in Spring

**Using @Autowired**

When Spring sees that you’ve annotated setInstrument() with @Autowired it’ll try to perform byType autowiring on the method.

@Autowired

public void setInstrument(Instrument instrument) {

this.instrument = instrument;

}

@Autowired

public void heresYourInstrument(Instrument instrument) {

this.instrument = instrument;

}

@Autowired

public Instrumentalist(Instrument instrument) {

this.instrument = instrument;

}

**Optional Autowiring**

If no bean can be wired into the @Autowired-annotated property or argument, then autowiring fails (with a nasty NoSuchBeanDefinitionException).

But it’s also possible that the property being wired is truly optional and a null

value is acceptable.

*@Autowired(required=false)*

private Instrument instrument;

When used with constructors, only one constructor can be annotated with @Autowired and required set to true. All other @Autowired-annotated constructors must have required set to false.

**Qualifying ambiguous dependencies**

Maybe it’s an abundance of (or at least two) beans. For example, suppose you have two beans that implement Instrument. In that event, there’s no way for @Autowired to choose which one you really want. So, instead of guessing, a NoSuchBeanDefinitionException will be thrown and wiring will fail.

@Autowired

@Qualifier("guitar") //dentro de esta anotación implicitamente realiza el **by-name**

private Instrument instrument;

In addition to narrowing (limitar) by a bean’s ID, it’s also possible to narrow by a qualifier that’s applied to a bean itself. For example, suppose that the guitar bean were

declared in XML as follows:

<bean class="com.springinaction.springidol.Guitar">

<qualifier value="stringed" />

</bean>

You could have also annotated the Guitar class itself with the @Qualifier annotation:

@Qualifier("stringed")

public class Guitar implements Instrument {

...

}

**Creating Custom Qualifier**

To create a custom qualifier annotation, all you need to do is to define an annotation that’s itself annotated with @Qualifier. For example, let’s create our own @StringedInstrument annotation to serve as a qualifier. The following listing shows the custom qualifier annotation.

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

import org.springframework.beans.factory.annotation.Qualifier; //**SR-330 has its own @Qualifier annotation in the javax.inject**

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface StringedInstrument {

}

With the @StringedInstrument annotation defined, you can now use it instead of

@Qualifier to annotate Guitar

@Qualifier

public @interface StringedInstrument {

........

}

Then, you can qualify the @Autowired instrument property with @StringedInstrument:

@Autowired

@StringedInstrument

private Instrument instrument;

When Spring tries to autowire the instrument property, it’ll narrow the selection of all

Instrument beans down to just those that are annotated with @StringedInstrument.

As long as only one bean is annotated with @StringedInstrument, it’ll be wired into

the instrument property.

*If there’s more than one @StringedInstrument-annotated bean*, then you’ll need

to provide further qualification to narrow it down. For example, suppose that in addi-

tion to the Guitar bean, you also have a HammeredDulcimer bean which is also anno-

tated with @StringedInstrument. One key difference between a guitar and a

hammered dulcimer is that guitars are strummed whereas hammered dulcimers are

hit with small wooden sticks (called hammers).

So, to qualify the Guitar class further, you could define another qualifier annota-

tion called @Strummed:

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Strummed {

}

Now you can annotate the instrument property with @Strummed to narrow the selec-

tion down to strummed string instruments:

@Autowired

@StringedInstrument

@Strummed

private Instrument instrument;

**Applying standards-based autowiring with @Inject**

In an effort to unify the programming model among the various dependency injection frameworks, the Java Community Process recently published the Dependency Injection for Java specification. Known in the Java Community Process as JSR-330 or more commonly as at inject, this specification brings a common dependency injection model to Java. As of Spring 3, Spring supports the at inject model.

The centerpiece of JSR-330 is the @Inject annotation. This annotation is an almost

complete drop-in replacement for Spring’s @Autowired annotation. So, instead of

using the Spring-specific @Autowired annotation, you might choose to use @Inject on

the instrument property.

**@Inject**

**private Instrument instrument;**

A diferencia de @Autowired, @Inject no tiene un atributo obligatorio. Por lo tanto, Se espera que las dependencias vengan llenas, en su defecto, lanza una excepción.

QUALIFYING @INJECTED PROPERTIES

As you’ve seen, @Inject and @Autowired have a lot in common. And like @Autowired,

the @Inject annotation is prone (propenso) to ambiguous bean definitions. @Inject’s answer to the @Qualifier annotation is the @Named annotation.

The @Named annotation works much like Spring’s @Qualifier, as you can see here:

@Inject

@Named("guitar")

private Instrument instrument;

The key difference between Spring’s @Qualifier and JSR-330’s @Named is one of

semantics. Whereas @Qualifier helps narrow the selection of matching beans (using

the bean’s ID by default), @Named specifically identifies a selected bean by its ID.

|  |  |  |
| --- | --- | --- |
| @Resource | javax.annotation | Java |
| @Inject | javax.inject | Java |
| @Qualifier | javax.inject | Java |
| @Autowired | org.springframework.bean.factory | Spring |

U**sing expressions with annotation injection**

@Value("Eruption")

private String song;

@Value("#{systemProperties.myFavoriteSong}")

private String song;

**Automatically discovering beans**

When you added **<context:annotation-config>** to your Spring configuration, you told Spring that you wanted it to honor a certain set of annotations in the beans that

you declared and to use those beans to guide bean wiring.

Even though **<context:annotation-config>** can go a long way toward eliminating most uses of <property>

and <constructor-arg> elements from your Spring configuration, you still must

explicitly declare beans using <bean>.

But Spring has another trick up its sleeve. The **<context:component-scan>** ele-

ment does everything that <context:annotation-config> does, plus it configures

Spring to automatically discover beans and declare them for you. What this means is

that most (or all) of the beans in your Spring application can be declared and wired

without using <bean>.

To configure Spring for autodiscovery, use <context:component-scan> instead of

<context:annotation-config>:

![A description...](data:None;base64,)

The <context:component-scan> element works by scanning a package and all of its

subpackages, looking for classes that could be automatically registered as beans in the Spring container.

**Annotating beans for autodiscovery**

By default, <context:component-scan> looks for classes that are annotated with one

of a handful of special stereotype annotations:

* @Component—A general-purpose stereotype annotation indicating that the class is a Spring component
* @Controller—Indicates that the class defines a Spring MVC controller
* @Repository—Indicates that the class defines a data repository
* @Service—Indicates that the class defines a service
* Any custom annotation that is itself annotated with @Component

**Using Spring’s Java-based configuration**

You have the option of configuring a Spring application with almost no XML, using

pure Java.

<context:component-scan base-package="com.springinaction.springidol" />

<context:component-scan> automatically registers beans that are annotated with certain stereotype annotations. But it also automatically loads in Java-based configuration classes that are annotated with **@Configuration**.

In this case, the base-package attribute tells Spring to look in com.springinaction.springidol to find classes that are annotated with @Configuration.

package com.springinaction.springidol;

import org.springframework.context.annotation.Configuration;

@Configuration

public class SpringIdolConfig {

// Bean declaration methods go here

}

The @Configuration annotation serves as a clue to Spring that this class will contain

one or more Spring bean declarations. Those bean declarations are just methods that

are annotated with @Bean.

**Aspect-oriented Spring (Monitorear)**

In software, several activities are common to most applications. Logging, security,

and transaction management are important, but should they be activities that your

application objects are actively participating in? Or would it be better for your applica-

tion objects to focus on the business domain problems they’re designed for and leave

certain aspects to be handled by someone else?

In software development, functions that span (lapso) multiple points of an application are called cross-cutting concerns (preocupaciones transversales). Typically, these cross-cutting concerns are conceptually separate from (but often embedded directly within) the application’s business logic.

Separating these cross-cutting concerns from the business logic is where aspect-

oriented programming (AOP) goes to work.

**AOP helps you decouple cross-cutting concerns from the objects that they affect. Aspects help to modularize cross-cutting concerns.**

Figure 4.1 represents a typical application that’s broken down into modules. Each mod-

ule’s main concern is to provide services for its particular domain. But each module also requires similar ancillary (auxiliar) functionalities, such as security and transaction management.

![A description...](data:None;base64,)

With AOP, you still define the common functionality in one place, but you can declaratively define how and where this functionality is applied without having to modify the class to which you’re applying the new feature. Cross-cutting concerns can now be modularized into special classes called aspects. This has two benefits. First, the logic for each concern is now in one place, as opposed to being scattered all over the code base. Second, our service modules are now cleaner since they only contain code for their primary concern (or core functionality) and secondary concerns have been moved to aspects.

***ADVICE***

Aspects have a purpose—a job they’re meant to do (Que están destinados a hacer). In AOP terms, the job of an aspect is called advice.

![A description...](data:None;base64,)

Advice defines both the **what** and the **when** of an aspect.

In addition to describing the job that an aspect will perform, advice addresses the question of when to perform the job. Should it be applied before a method is invoked? After the method is invoked? Both before and after method invocation? Or should it only be applied if a method throws an exception?

*Spring aspects can work with five kinds of advice:*

* Before—The advice functionality takes place before the advised method is

invoked.

* After—The advice functionality takes place after the advised method completes,

regardless of the outcome.

* After-returning—The advice functionality takes place after the advised method

successfully completes.

* After-throwing—The advice functionality takes place after the advised method

throws an exception.

* Around—The advice wraps the advised method, providing some functionality

before and after the advised method is invoked.

***JOIN POINTS***

A join point is a point in the execution of the application where an aspect can be plugged in (conectado). This point could be a method being called, an exception being thrown, or even a field being modified.

These are the points where your aspect’s code can be inserted into the normal flow of

your application to add new behavior.

***POINTCUTS***

Pointcuts help narrow down (acortar) the join points advised by an aspect.

If advice defines the what and when of aspects, then **pointcuts define the where**. A

pointcut definition matches one or more join points at which advice should be woven (ligado).

Often you specify these pointcuts using explicit class and method names or through

regular expressions that define matching class and method name patterns. Some AOP

frameworks allow you to create dynamic pointcuts that determine whether to apply

advice based on runtime decisions, such as the value of method parameters.

***ASPECTS***

An aspect is the merger of advice and pointcuts. Taken together, advice and pointcuts define everything there is to know about an aspect—**what it does and where and when it does it.**

***INSTRODUCTIONS***

An introduction allows you to add new methods or attributes to existing classes

***WEAVING***

Weaving is the process of applying aspects to a target object (objeto de destino) to create a new proxied object (objeto proxy). The aspects are woven into the target object at the specified join points.

The weaving can take place at several points in the target object’s lifetime:

* Compile time—Aspects are woven in when the target class is compiled. This

requires a special compiler. AspectJ’s weaving compiler weaves aspects this way.

* Classload time—Aspects are woven in when the target class is loaded into the

JVM. This requires a special ClassLoader that enhances that target class’s byte-

code before the class is introduced into the application. AspectJ 5’s load-time

weaving (LTW) support weaves aspects in this way.

* Runtime—Aspects are woven in sometime during the execution of the applica-

tion. Typically, an AOP container will dynamically generate a proxy object that

will delegate to the target object while weaving in the aspects. This is how

Spring AOP aspects are woven.

**Spring’s AOP support**

Not all AOP frameworks are created equal.

Spring’s support for AOP comes in four flavors:

* Classic Spring proxy-based AOP
* @AspectJ annotation-driven aspects
* Pure-POJO aspects
* Injected AspectJ aspects (available in all versions of Spring)

SPRING ADVICE IS WRITTEN IN JAVA

SPRING ADVISES OBJECTS AT RUNTIME

SPRING ONLY SUPPORTS METHOD JOIN POINTS

Selecting join points with pointcuts

AspectJ, I strongly recommend Ramnivas Laddad’s AspectJ in Action, Second Edition.

![A description...](data:None;base64,)

The execution designator is the only one that actually performs matches. This means that execution is the primary designator you’ll use in every pointcut definition you write. You’ll use the other designators to constrain the pointcut’s reach.

**Writing pointcuts**

For example, the pointcut expression shown in figure 4.4 can be used to apply advice

whenever an Instrument’s play() method is executed:

![A description...](data:None;base64,)

We used the execution() designator to select the Instrument’s play() method.

The method specification starts with an asterisk, which indicates that we don’t care what type the method returns.

Then we specify the fully qualified class name and the name of the method we want to select. For the method’s parameter list, we use the doubledot (..), indicating that the pointcut should select any play() method, no matter what the argument list is.

Now let’s suppose that we want to confine the reach of that pointcut to only the

com.springinaction.springidol package. In that case, we could limit the match by

tacking on a within() designator, as shown in figure 4.5.

Note that we used the && operator to combine the execution() and within() designators in an “and” relationship (where both designators must match for the pointcut to match). Similarly, we could’ve used the || operator to indicate an “or” relationship. And the ! operator can be used to negate the effect of a designator.

![A description...](data:None;base64,)

Since ampersands have special meaning in XML, you’re free to use and in place of &&

when specifying pointcuts in a Spring XML-based configuration. Likewise, or and not

can be used in place of || and ! (respectively).

**Using Spring’s bean() designator**

In addition to the designators listed in table 4.1, Spring 2.5 introduced a new bean()

designator that lets you identify beans by their ID within a pointcut expression. bean()

takes a bean ID or name as an argument and limits the pointcut’s effect to that specific

bean. For example, consider the following pointcut:

*execution(\* com.springinaction.springidol.Instrument.play())*

*and bean(eddie)*

Here we’re saying that we want to apply aspect advice to the execution of an

Instruments play() method, but limited to the bean whose ID is eddie.

Narrowing a pointcut to a specific bean may be valuable in some cases, but we can

also use negation to apply an aspect to all beans that don’t have a specific ID:

execution(\* com.springinaction.springidol.Instrument.play())

and !bean(eddie)

In this case, the aspect’s advice will be woven into all beans whose ID isn’t eddie.

**Declaring aspects in XML**

Un proxy es un programa o dispositivo que realiza una tarea acceso a Internet en lugar de otro ordenador. Un proxy es un punto intermedio entre un ordenador conectado a Internet y el servidor que está accediendo. Cuando navegamos a través de un proxy, nosotros en realidad no estamos accediendo directamente al servidor, sino que realizamos una solicitud sobre el proxy y es éste quien se conecta con el servidor que queremos acceder y nos devuelve el resultado de la solicitud.

Cuando nos conectamos con un proxy, el servidor al que accedemos en realidad recibe la solicitud del proxy, en vez de recibirla directamente desde nuestro ordenador. Puede haber sistemas proxy que interceptan diversos servicios de Internet.

Lo más habitual es el proxy web, que sirve para interceptar las conexiones con la web y puede ser útil para incrementar la seguridad, rapidez de navegación o anonimato.

![A description...](data:None;base64,)

![A description...](data:None;base64,)

**<bean id="audience" class="com.springinaction.springidol.Audience" />**

**package com.springinaction.springidol;**

**public interface Performer {**

**void perform() throws PerformanceException;**

**}**

![A description...](data:None;base64,)

The first thing to notice about the Spring AOP configuration elements is that most of

them must be used within the context of the <aop:config> element.

There are a few exceptions to this rule, but when it comes to declaring beans as aspects you’ll always start with the <aop:config> element.

Within <aop:config> you may declare one or more advisors, aspects, or pointcuts.

In listing 4.2, we declared a single aspect using the <aop:aspect> element.

The **ref attribute** references the POJO bean that will be used to supply the functionality of the aspect—in this case, audience. The bean that’s referenced by the ref attribute will supply the methods called by any advice in the aspect.

The aspect has four different bits of advice. The two <aop:before> elements

define method before advice that will call the takeSeats() and turnOffCellPhones()

methods (declared by the method attribute) of the Audience bean before any methods

matching the pointcut are executed.

The <aop:after-returning> element defines an after-returning advice to call the applaud() method after the pointcut. Meanwhile, the <aop:after-throwing> element defines an after-throwing advice to call the demandRefund() method if any exceptions are thrown. Figure 4.6 shows how the advice logic is woven into the business logic.

**In all advice elements, the pointcut attribute defines the pointcut where the**

**advice will be applied.** The value given to the pointcut attribute is a pointcut defined

in AspectJ’s pointcut expression syntax.

You’ll notice that the value of the pointcut attribute is the same for all of the

advice elements. That’s because all of the advice is being applied to the same pointcut.

![A description...](data:None;base64,)

This presents a DRY (don’t repeat yourself) principle violation. If you decide later to change the pointcut, you must change it in four different places.

To avoid duplication of the pointcut definition, you may choose to define a named pointcut using the <aop:pointcut> element. The following XML shows how the <aop:pointcut> element is used within the <aop:aspect> element to define a named pointcut that can be used by all of the advice elements.

![A description...](data:None;base64,)

Now the pointcut is defined in a single location and is referenced across multiple advice elements. The <aop:pointcut> element defines the pointcut to have an id of performance. Meanwhile, all of the advice elements have been changed to reference the named pointcut with the pointcut-ref attribute.

As used in listing 4.3, the <aop:pointcut> element defines a pointcut that can be referenced by all advices within the same <aop:aspect> element. But you can also define pointcuts that can be used across multiple aspects by placing the <aop:point-cut> elements within the scope of the <aop:config> element.

**Declaring around advice**

Around advice has an advantage over before and after advice in this regard. With

around advice, you can accomplish the same thing as you can with distinct before and

after advice, but do it in a single method. Since the entire set of advice takes place in a

single method, there’s no need to retain state in a member variable.

The first thing you’ll notice about this new advice method is that it’s given a![A description...](data:None;base64,)

**ProceedingJoinPoint as a parameter. This object is necessary, as it’s how we’ll be able to invoke the advised method from within our advice.** The advice method will do everything it needs to do and, when it’s ready to pass control to the advised method, it’ll call ProceedingJoinPoint’s proceed() method.

Note that it’s crucial that you remember to include a call to the proceed() method. If you don’t, then your advice will effectively block access to the advised method. Maybe that’s what you want, but chances are good that you do want the advised method to be executed at some point.

What’s also interesting is that just as you can omit a call to the proceed() method to block access to the advised method, you can also invoke it multiple times from within the advice. One reason for doing this may be to implement retry logic to perform repeated attempts on the advised method should it fail.

In the case of the audience aspect, the watchPerformance() method contains all of the functionality of the previous four advice methods, but all of it’s contained in this single method, and this method is responsible for its own exception handling.

You’ll also note that just before the join point’s proceed() method is called, the current time is recorded in a local variable. Just after the method returns, the elapsed time is reported.

Declaring around advice isn’t dramatically different from declaring other types of

advice. All you need to do is use the <aop:around> element.

![A description...](data:None;base64,)

**Passing parameters to advice**

![A description...](data:None;base64,)

**Annotating aspects**

A key feature introduced in AspectJ 5 is the ability to use annotations to create aspects.

Prior (anterior) to AspectJ 5, writing AspectJ aspects involved learning a Java language extension.

But AspectJ’s annotation-oriented model makes it simple to turn any class into an

aspect by sprinkling a few annotations around. This new feature is commonly referred

to as @AspectJ.

![A description...](data:None;base64,)

![A description...](data:None;base64,)

The new Audience class is now annotated with @Aspect. This annotation indicates that

Audience isn’t just any POJO but is an aspect.

The @Pointcut annotation is used to define a reusable pointcut within an @AspectJ aspect. The value given to the @Pointcut annotation is an AspectJ pointcut

expression—here indicating that the pointcut should match the perform() method

of a Performer. The name of the pointcut is derived from the name of the method to

which the annotation is applied. Therefore, the name of this pointcut is performance().

The actual body of the performance() method is irrelevant and in fact should be empty.

The method itself is just a marker, giving the @Pointcut annotation something to attach itself to.

<bean id="audience" class="com.springinaction.springidol.Audience" />

There’s one last thing to do to make Spring apply Audience as an aspect. You must declare an autoproxy bean in the Spring context that knows how to turn @AspectJannotated beans into proxy advice.

For that purpose, Spring comes with an autoproxy creator class called AnnotationAwareAspectJAutoProxyCreator.

**<aop:aspectj-autoproxy />**

<aop:aspectj-autoproxy/> will create an AnnotationAwareAspectJAutoProxy Creator in the Spring context and will automatically proxy beans whose methods match the pointcuts defined with @Pointcut annotations in @Aspect-annotated beans.

To use the <aop:aspectj-autoproxy> configuration element, you’ll need to remember to include the aop namespace in your Spring configuration file:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

<http://www.springframework.org/schema/aop/spring-aop-3.0.xsd>">

<aop:aspect> has one distinct advantage over @AspectJ in that you don’t need the

source code of the class that’s to provide the aspect’s functionality. With @AspectJ, you

must annotate the class and methods, which requires having the source code. But <aop:aspect> can reference any bean.

**Annotating around advice**

![A description...](data:None;base64,)

**Hitting the data base**

DataAccessException is unchecked exception Not catch blocks, you don’t have to catch any of the data access exceptions thrown from Spring (although you’re perfectly welcome to if you’d like).

**Templating data access**

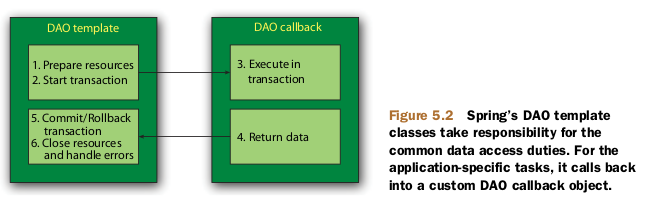
A template method defines the skeleton of a process. In our example, the process is moving luggage from departure city to arrival city. The process itself is fixed; it never changes.

Because each passenger’s luggage check-in is different, the implementation of this part of the process is determined by the passenger. In software terms, a template method delegates the implementation-specific portions of the process to an interface. Different implementations of this interface define specific implementations of this portion of the process.

Spring separates the fixed (fija) and variable parts of the data access process into two distinct classes: templates and callbacks. Templates manage the fixed part of the process, whereas your custom data access code is handled in the callbacks. Figure 5.2 shows the responsibilities of both of these classes.

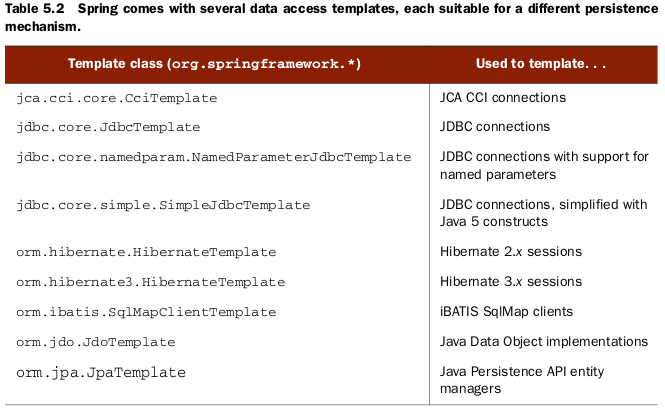
As you can see in figure 5.2, Spring’s template classes handle the fixed parts of data access controlling transactions, managing resources, and handling exceptions.

Meanwhile, the specifics of data access as they pertain to your application creating statements, binding parameters, and marshaling result sets—are handled in the call back implementation.

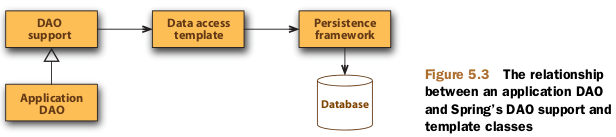


Spring comes with several templates to choose from, depending on your persis-

tence platform choice. If you’re using straight JDBC, then you’ll want to use JdbcTemplate. But if you favor one of the object relational mapping frameworks, then perhaps HibernateTemplate or JpaTemplate is more suitable. Table 5.2 lists all of Spring’s data access templates and their purposes.



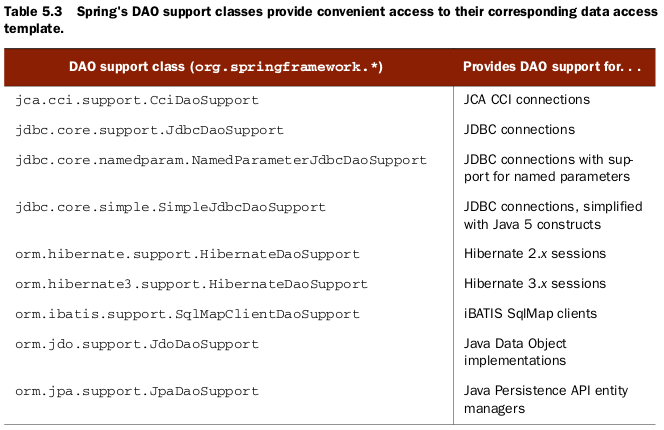
**Using DAO support classes**



DAO support classes provide convenient access to the template class that they support.

When writing your application DAO implementation, you can subclass a DAO support class and call a template retrieval method to have direct access to the underlying data access template. For example, if your application DAO subclasses JdbcDaoSupport, then you only need to call getJdbcTemplate() to get a JdbcTemplate to work with.

Plus, if you need access to the underlying persistence platform, each of the DAO support classes provides access to whatever class it uses to communicate with the data base. For instance, the JdbcDaoSupport class contains a getConnection() method for dealing directly with the JDBC connection.



**Configuring a data source**

Regardless of which form of Spring DAO support you use, you’ll likely need to configure a reference to a data source. Spring offers several options for configuring data source beans in your Spring application, including

* Data sources that are defined by a JDBC driver
* Data sources that are looked up by JNDI
* Data sources that pool connections

**Using JNDI data sources**

The benefit of configuring data sources in this way is that they can be managed completely external to the application, allowing the application to ask for a data source when it’s ready to access the database.

Moreover, data sources managed in an application server are often pooled for greater performance and can be hot-swapped by system administrators.

<jee:jndi-lookup id="dataSource"

jndi-name="/jdbc/SpitterDS"

resource-ref="true" />

The jndi-name attribute is used to specify the name of the resource in JNDI. If only the jndi-name property is set, then the data source will be looked up using the name given as is. But if the application is running within a Java application server, then you’ll want to set the resource-ref property to true so that the value given in jndi-name will be prepended with java:comp/env/.

**Using a pooled data source**

If you're unable to retrieve a data source from JNDI, the next best thing is to configure a pooled data source directly in Spring. Although Spring doesn’t provide a pooled data source, there’s a suitable one available in the Jakarta Commons Database Connection Pooling (DBCP) project (<http://jakarta.apache.org/commons/dbcp>).

DBCP includes several data sources that provide pooling, but the BasicDataSource is one that’s often used because it’s simple to configure in Spring and because it resembles Spring’s own DriverManagerDataSource.

For the Spitter application, we’ll configure a BasicDataSource bean as follows:

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="org.hsqldb.jdbcDriver" />

<property name="url" value="jdbc:hsqldb:hsql://localhost/spitter/spitter" />

<property name="username" value="sa" />

<property name="password" value="" />

<property name="initialSize" value="5" />

<property name="maxActive" value="10" />

</bean>

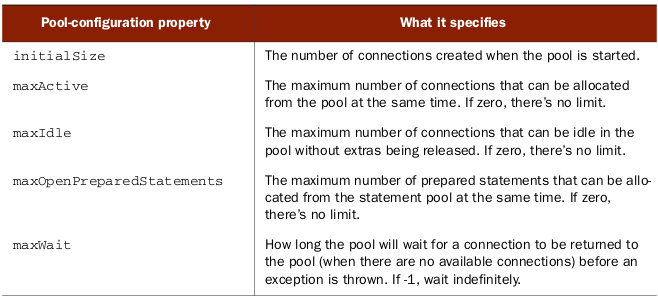
The first four properties are elemental to configuring a BasicDataSource. The

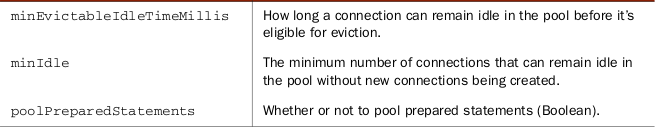
driverClassName property specifies the fully qualified name of the JDBC driver class.

Here we’ve configured it with the JDBC driver for the Hypersonic database. The url property is where we set the complete JDBC URL for the database. Finally, the username and password properties are used to authenticate when we’re connecting to the database.

Those four basic properties define connection information for BasicDataSource. In addition, several properties can be used to configure the data source

pool itself. Table 5.4 lists a few of the most useful pool-configuration properties of BasicDataSource.





**JDBC driver-based data source**

The simplest data source you can configure in Spring is one that’s defined through a JDBC driver. Spring offers two such data source classes to choose from (both in the org.springframework.jdbc.datasource package):

* DriverManagerDataSource —Returns a new connection every time that a connection is requested. Unlike DBCP’s BasicDataSource, the connections provided by DriverManagerDataSource aren’t pooled.
* SingleConnectionDataSource—Returns the same connection every time a

connection is requested. Although SingleConnectionDataSource isn’t exactly a pooled data source, you can think of it as a data source with a pool of exactly one connection.

Configuring either of these data sources is similar to how we configured DBCP’s

BasicDataSource:

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="org.hsqldb.jdbcDriver" />

<property name="url"

value="jdbc:hsqldb:hsql://localhost/spitter/spitter" />

<property name="username" value="sa" />

<property name="password" value="" />

</bean>

The only difference is that since neither DriverManagerDataSource nor SingleConnectionDataSource provides a connection pool, there are no pool configuration properties to set.

Although SingleConnectionDataSource and DriverManagerDataSource are great

for small applications and running in development, you should seriously consider the implications of using either in a production application. Because SingleConnectionDataSource has one and only one database connection to work with, it doesn’t work well in a multithreaded application. At the same time, even though DriverManagerDataSource is capable of supporting multiple threads, it incurs a performance cost for creating a new connection each time a connection is requested. Because of these limitations, I strongly recommend using pooled data sources.

Now that we’ve established a connection to the database through a data source, we’re ready to actually access the database. As I’ve already mentioned, Spring affords us several options for working with databases, including JDBC, Hibernate, and the Java Persistence API (JPA).

**Using JDBC with Spring**

JDBC lets you work with data at a much lower level than the persistence frameworks, allowing you to access and manipulate individual columns in a

database. This fine-grained approach to data access comes in handy in applications, such as reporting applications, where it doesn’t make sense to organize the data into objects, just to then unwind it back into raw data.

But all is not sunny in the world of JDBC. With its power, flexibility, and other niceties also come some not-so-niceties.

Though JDBC gives you an API that works closely with your database, you’re responsible for handling everything related to accessing the database. This includes managing database resources and handling exceptions.

**Working with JDBC templates**

Spring’s JDBC framework will clean up your JDBC code by shouldering the burden of resource management and exception handling. This leaves you free to write only the code necessary to move data to and from the database.

***JdbcTemplate***—The most basic of Spring’s JDBC templates, this class provides

simple access to a database through JDBC and simple indexed-parameter queries.

***NamedParameterJdbcTemplate***—This JDBC template class enables you to perform queries where values are bound to named parameters in SQL, rather than indexed parameters.

***SimpleJdbcTemplate***—This version of the JDBC template takes advantage of

Java 5 features such as autoboxing, generics, and variable parameter lists to simplify how a JDBC template is used.

ACCESSING DATA USING SIMPLEJDBCTEMPLATE

All that a SimpleJdbcTemplate needs to do its work is a DataSource. This makes it easy enough to configure a SimpleJdbcTemplate bean in Spring with the following XML:

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">

<constructor-arg ref="dataSource" />

</bean>

The actual DataSource being referred to by the dataSource property can be any implementation of javax.sql.DataSource.

Now we can wire the jdbcTemplate bean into our DAO and use it to access the database.

<bean id="spitterDao" class="com.ipn.SimpleJdbcTemplateSpitterDao">

<property name="jdbcTemplate" ref="jdbcTemplate" />

</bean>

USING SPRING’S DAO SUPPORT CLASSES FOR JDBC

For each of our application’s JDBC-backed DAO classes, we’ll need to be sure to add a

SimpleJdbcTemplate property and setter method. And we’ll need to be sure to wire

the SimpleJdbcTemplate bean into the SimpleJdbcTemplate property of each DAO.

That’s not a big deal if the application only has one DAO, but if you have multiple

DAOs, that’s a lot of repeated code.

One solution would be for you to create a common parent class for all your DAO

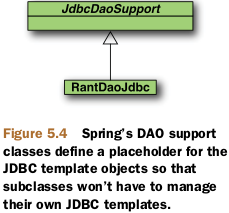
objects where the SimpleJdbcTemplate property resides. Then all of your DAO classes

would extend that class and use the parent class’s SimpleJdbcTemplate for its data

access.

The idea of creating a base DAO class that holds the JDBC template is such a good idea that Spring comes with just such a base class out of the box.

Actually, it comes with three such classes—JdbcDaoSupport, SimpleJdbcDaoSupport, and NamedParameterJdbcDaoSupport.



<bean id="spitterDao" class="com.ipn.JdbcSpitterDao">

<property name="dataSource" ref="dataSource" />

</bean>

When JdbcSpitterDao has its dataSource property configured, it’ll internally create a

**SimpleJdbcTemplate** instance for you. **This eliminates the need to explicitly declare a SimpleJdbcTemplate bean in Spring.**

JDBC is the most basic way to access data in a relational database. Spring’s JDBC

templates save you the hassle of dealing with the boilerplate code that handles con-

nection resources and exception handling, leaving you to focus on the actual work of

querying and updating data.

**HIBERNATE object-relational mapping (ORM)**

* Lazy loading—As our object graphs become more complex, we sometimes don’t want to fetch entire relationships immediately. To use a typical example, suppose we’re selecting a collection of PurchaseOrder objects, and each of these objects contains a collection of LineItem objects. If we’re only interested in PurchaseOrder attributes, it makes no sense to grab the LineItem data. This could be expensive. Lazy loading allows us to grab data only as it’s needed.
* Eager fetching—This is the opposite of lazy loading. Eager fetching allows you to grab an entire object graph in one query. In the cases where we know that we need a PurchaseOrder object and its associated LineItems, eager fetching lets us get this from the database in one operation, saving us from costly round trips.
* Cascading—Sometimes changes to a database table should result in changes to other tables as well. Going back to our purchase order example, when an Order object is deleted, we also want to delete the associated LineItems from the database.

Spring provides support for several persistence frameworks, including Hibernate, iBATIS, Java Data Objects (JDO), and the Java Persistence API (JPA).

As with Spring’s JDBC support, Spring's support for ORM frameworks provides integration points to the frameworks as well as some additional services:

* Integrated support for Spring declarative transactions
* Transparent exception handling
* Thread-safe, lightweight template classes
* DAO support classes
* Resource management

Historically, HibernateTemplate was the way to work with Hibernate in a Spring application. Like its JDBC counterpart, HibernateTemplate took care of the intricacies of working with Hibernate by catching Hibernate-specific exceptions and rethrowing them as one of Spring’s unchecked data access exceptions.

One of the responsibilities of HibernateTemplate is to manage Hibernate

Sessions. This involves opening and closing sessions as well as ensuring one session per transaction. Without HibernateTemplate, you’d have no choice but to clutter your DAOs with boilerplate session management code.

The downside of HibernateTemplate is that it’s somewhat intrusive. When we use Spring’s HibernateTemplate in a DAO (whether directly or through HibernateDaoSupport), the DAO class is coupled to the Spring API. Although this may not be of much concern to some developers, others may find Spring’s intrusion into their DAO code undesirable.

Even though HibernateTemplate is still around, it’s no longer considered the best way of working with Hibernate. Contextual sessions, introduced in Hibernate 3, are a way in which Hibernate itself manages one Session per transaction. There’s no need for HibernateTemplate to ensure this behavior. This keeps your DAO classes free of Spring-specific code.

**Declaring a Hibernate session factory**

Natively, the main interface for working with Hibernate is org.hibernate.Session.

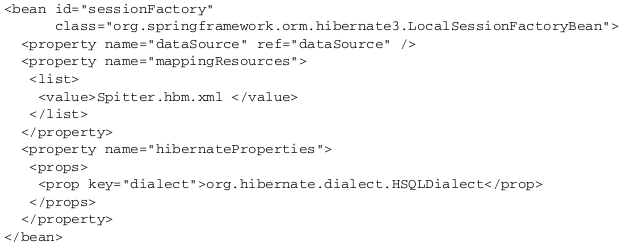
The Session interface provides basic data access functionality such as the ability to save, update, delete, and load objects from the database. Through the Hibernate Session, an application’s DAO will perform all of its persistence needs.

In Spring, the way to get a Hibernate SessionFactory is through one of Spring’s

Hibernate session factory beans. These session factory beans are implementations of Spring’s FactoryBean interface that produce a Hibernate SessionFactory when wired into any property of type SessionFactory. This makes it possible to configure your Hibernate session factory alongside the other beans in your application’s Spring context.

When it comes to configuring a Hibernate session factory bean, you have a choice to make. The decision hinges on whether you want to configure your persistent domain objects using Hibernate’s XML mapping files or with annotations.

If you choose to define your object-to-database mapping in XML, you’ll need to configure LocalSessionFactoryBean in Spring :



LocalSessionFactoryBean is configured here with three properties. The dataSource property is wired with a reference to a DataSource bean. The mappingResources property lists one or more Hibernate mapping files that define the persistence strategy for the application. Finally, hibernateProperties is where we configure the minutia of how Hibernate should operate. In this case, we’re saying that Hibernate will be working with a Hypersonic database and should use the HSQLDialect to construct SQL accordingly.

If annotation-oriented persistence is more your style, then you’ll need to use

AnnotationSessionFactoryBean instead of LocalSessionFactoryBean:

As with LocalSessionFactoryBean, the dataSource and hibernateProperties properties tell where to find a database connection and what kind of database we’ll be dealing with.

But instead of listing Hibernate mapping files, we can use the packagesToScan

property to tell Spring to scan one or more packages looking for domain classes that are annotated for persistence with Hibernate. This includes classes that are annotated with JPA’s @Entity or @MappedSuperclass and Hibernate’s own @Entity annotation.

**A list of one**

AnnotationSessionFactoryBean’s packagesToScan property takes an array of

Strings specifying the packages to look for persistent classes in. Normally, I might specify such a list as follows:

<property name="packagesToScan">

<list>

<value>com.habuma.spitter.domain</value>

</list>

</property>

But since I’m only asking it to scan a single package, I’m taking advantage of a

built-in property editor that automatically converts a single String value into a

String array.

If you’d prefer, you may also explicitly list out all of your application's persistent classes by specifying a list of fully qualified class names in the annotatedClasses property:

<property name="annotatedClasses">

<list>

<value>com.habuma.spitter.domain.Spitter</value>

<value>com.habuma.spitter.domain.Spittle</value>

</list>

</property>

The annotatedClasses property is fine for hand-picking a few domain classes. But packagesToScan is more appropriate if you have a lot of domain classes and don’t want to list them all or if you want the freedom to add or remove domain classes without revisiting the Spring configuration.

With a Hibernate session factory bean declared in the Spring application context, we’re ready to start creating our DAO classes.

**Spring and the Java Persistence API**

The Java Persistence API (JPA) emerged out of the rubble (escombros) of EJB 2’s entity beans as the next-generation Java persistence standard. JPA is a POJO-based persistence mechanism that draws ideas from both Hibernate and Java Data Objects (JDO), and mixes Java 5 annotations in for good measure.

With the Spring 2.0 release came the premiere of Spring integration with JPA.

**Configuring an entity manager factory**

In a nutshell (cascara de nuez), JPA-based applications use an implementation of EntityManagerFactory to get an instance of an EntityManager. The JPA specification defines two kinds of entity managers:

* Application-managed—Entity managers are created when an application directly requests one from an entity manager factory. With applicationmanaged entity managers, the application is responsible for opening or closing entity managers and involving the entity manager in transactions. This type of entity manager is most appropriate for use in standalone applications that don’t run within a Java EE container.
* Container-managed—Entity managers are created and managed by a Java EE container. The application doesn’t interact with the entity manager factory at all.

Instead, entity managers are obtained directly through injection or from JNDI. The container is responsible for configuring the entity manager factories. This type of entity manager is most appropriate for use by a Java EE container that wants to maintain some control over JPA configuration beyond what’s specified in persistence.xml.

Both kinds of entity manager implement the same EntityManager interface. The key difference isn’t in the EntityManager itself, but rather (más bien) in how the EntityManager is created and managed. **Application-managed** EntityManagers are created by an EntityManagerFactory obtained by calling the ***createEntityManagerFactory()*** method of the PersistenceProvider. Meanwhile, **container-managed** EntityManagerFactorys are obtained through PersistenceProvider’s ***createContainerEntityManagerFactory()*** method.

So what does this all mean for Spring developers wanting to use JPA? Not much. Regardless (independientemente) of which variety of EntityManagerFactory you want to use, Spring will take responsibility for managing EntityManagers for you.

If using an application-managed entity manager, Spring plays the role of an application and transparently deals with the EntityManager on your behalf (favor).

In the container-managed scenario, Spring plays the role of the container.

Each flavor of entity manager factory is produced by a corresponding Spring factory bean:

* LocalEntityManagerFactoryBean produces an application-managed

EntityManagerFactory.

* LocalContainerEntityManagerFactoryBean produces a container-managed

EntityManagerFactory.

It’s important to point out that the choice made between an application-managed EntityManagerFactory and a container-managed EntityManagerFactory is completely transparent to a Spring-based application.

Spring’s JpaTemplate hides the intricate details of dealing with either form of EntityManagerFactory, leaving your data access code to focus on its true purpose: data access.

The only real difference between application-managed and container-managed

entity manager factories, as far as Spring is concerned, is how each is configured within the Spring application context.

***application-managed LocalEntityManagerFactoryBean in Spring.***

Application-managed entity manager factories derive most of their configuration information from a configuration file called persistence.xml. This file must appear in the META-INF directory within the classpath.

The purpose of the persistence.xml file is to define one or more persistence units. A persistence unit is a grouping of one or more persistent classes that correspond to a single data source. In simple terms, persistence.xml enumerates one or more persistent classes along with any additional configuration such as data sources and XML based mapping files. Here’s a typical example of a persistence.xml file:

container-managed LocalContainerEntityManagerFactoryBean.