

Reference Documentation

**Jan Machacek
Anirvan Chakraborty
Mark Harrison**

Reference Documentation

by Jan Machacek, Anirvan Chakraborty, and Mark Harrison

0.4

Copyright © 2011-2012 Jan Machacek, Anirvan Chakraborty, Mark Harrison

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

I. Overview of Specs2 Spring	1
1. Introducing Specs2 Spring	2
What is Specs2 Spring	2
Spring testing approaches	3
Spring Integration Testing	4
Getting started	5
II. Using Specs2 Spring	7
2. Using Specs2 Spring Specification	8
Using Specs2 Spring	8
Understanding the Specs2 setup annotations	8
Understanding the Specs2 behavioural annotations	18
3. Testing large systems using Specs2 Spring	19
What are large systems?	19
Testing large systems	20
Multiple environment entries	22
4. Test data setup Specs2 Spring	23
Inserting test data	23
HibernateDataAccess	25
III. Technical background	27
5. Improving Specs2 Spring	28
Building Specs2 Spring from sources	28
Contributing to Specs2 Spring	29

Part I. Overview of Specs2 Spring

Chapter 1. Introducing Specs2 Spring

What is Specs2 Spring

This chapter covers the motivation, implementation and typical usage of the Spring Specification. The Spring Specification is an extension of the Specs2 [<http://implicit.ly/specs2-10>] framework.

The Specs2 Spring includes code that will help you set up the context for the entire integration test--and by context, we mean the appropriate entries in the JNDI environment as well as the beans under test, autowired in to the instance of the test under execution. Finally, Specs2 Spring can be easily configured to run every example in its own transaction that rolls back automatically when the example completes.

The extension is meant to help you write tests in Scala to test your Spring code (whether implemented in Java or Scala). You will be able to take advantage of all the features of the Specs2 framework and apply them to the Spring test code.

Why bother, you ask? Because Specs2 and Scala allow you to be much more expressive in your tests. Consequently, your tests can focus on the essence of what is being tested, reducing the noise that the traditional Java code requires. A motivational example shows how to prepare test data, insert them to the RDBMS and then verify that some service method works as expected.

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.orm.hibernate3.HibernateTemplate
import org.specs2.spring.{BeanTables, HibernateDataAccess, Specification}

@IntegrationTest
class SomeComponentSpec extends Specification
  with HibernateDataAccess with BeanTables {
  @Autowired var someComponent: SomeComponent = _

  /**
   * Shows the usage of BeanTables and HibernateDataAccess to
   * set up and insert test objects using the convenient tabular notation.
   */
  "getByUsername finds existing Rider" in {
    "age" | "username" | "name" | "teamName" |
      32 ! "janm"      ! "Jan"  ! "Wheelers" |
      30 ! "anic"      ! "Ani"  ! "Team GB"  |> insert[Rider]

    this.someComponent.getByUsername("janm").getName must_== ("Jan")
  }
}
```

There are several things at play here: the custom `IntegrationTest` annotation defines the environment for the Spring components under test, the `Autowired` annotation on the `someComponent` variable tells Specs2 Spring to inject the constructed bean. Finally, the example sets up some test data using `BeanTables`, bulk-inserts them using the Hibernate ORM (method `insert[T]: (T => Result)` in `HibernateDataAccess`). Once the set-up work is done, we proceed to verify the correct behaviour of the `getByUsername` method in `SomeComponent`. Notably, the `"getByUsername finds`

existing `Rider` example runs in its own transaction. When the example completes (whether successfully or unsuccessfully), the transaction is rolled back!

The equivalent Java code would be *much, much* longer: you would be able to use the `spring-test` artifact, giving you the ability to inject dependencies into your test and run the test methods in their own transactions. Unfortunately, you would still be left to your own devices to set up the JNDI environment (which is non-trivial and *very* repetitive work). Moreover, the body of the test method would contain much more noise, distracting you from the test code.

Specs2 Spring contains support to set-up:

- Spring `ApplicationContext` from the specified list of configuration files,
- Multiple XA `DataSources` to RDBMS as `javax.sql.DataSource`,
- Single XA transaction support as `javax.transaction.UserTransaction`,
- Multiple JMS queue and topic support `javax.jms.Queue`, `javax.jms.Topic`, `javax.jms.ConnectionFactory`,
- Multiple Javamail sessions as `javax.mail.Session`,
- Multiple `WorkManagers`--both the `commonj.work.WorkManager` and the `javax.spi.resource.work.WorkManager`,
- Multiple arbitrary beans, as long as the types include accessible nullary constructor,
- Ability to register a class that can inject arbitrary entries into the JNDI environment.

In addition to the set-up phase, Specs2 Spring extension enhances the behaviour of the examples by isolating them (when required) in their own transactions.

But there's much more to Specs2 Spring: on top of the set-up code, Specs2 Spring comes with convenience traits that simplify setting up test data, manipulating data in ORM tools and relational databases.

Spring testing approaches

In an integration test, you exercise code in multiple components working together. These components sometimes need *external* components, such as `javax.sql.DataSource`, `javax.mail.Session`, `javax.jms.Queue`, `javax.jms.Topic`, `javax.transaction.TransactionManager` and many other components of the Java EE world.

To keep the source code of our application as maintainable as possible and to remove as much complexity from the build process as possible, we should try to keep the source code that is running on the developers' machines *exactly* the same as the source code that is deployed on the pre-production servers as well as the production servers.

The development machine may have its "own" RDBMS, specific configuration of the JMS infrastructure; the pre-production machine will have its specific configuration, and same will apply to the live machine. Moreover, some of the details of the configuration should be inaccessible to us. (For example, the connection details to the live RDBMS should be kept secret, not left to the developers to maintain.)

Java offers a good way to achieve this: register the appropriate resources in JNDI. Taking a *huge* simplification, you may think of JNDI as `Map<String, Object>` (`Map[String, AnyRef]` in

Scala speak). The application consumes objects from JNDI and uses them as the variable resources we explored in the previous few paragraphs.

So, in addition to the test that makes up the test, we will need to include code that sets up JNDI for the test. The trouble is that such code would be repetitive if it were to be used more than once and we would need to ensure that it runs before any other code. Consider the following code snippet as an example.

```
class SomeTest extends Specification {

  "specification" in {
    // set-up JNDI environment
    // code that depends (consumes) the entries from JNDI
    success
  }

  "another specification" in {
    // set-up JNDI environment
    // code that depends (consumes) the entries from JNDI
    success
  }
}
```

Spring Integration Testing

The situation with JNDI is similar to the operations we need to set up the `ApplicationContext` that contains the beans under test. Once we have the `ApplicationContext`, we can inject the beans from that context into the test we are running. Expanding the code from the previous example, we can write:

```
class SomeTest extends Specification {
  // set-up JNDI environment

  "specification" in {
    // set-up JNDI environment
    // the Spring application that consumes the JNDI entries
    val someService = new ClassPathXmlApplicationContext("...").
      getBean(classOf[SomeService])
    success
  }

  "another specification" in {
    // set-up JNDI environment
    // the Spring application that consumes the JNDI entries
    val someService = new ClassPathXmlApplicationContext("...").
      getBean(classOf[SomeService])
    success
  }
}
```

Additionally, we would like to examine whether the specifications hold (in the examples) in a way that isolates each example; when the example completes its work, we would like to roll back any changes

we may have made to the system. In other words, it should be possible to run each example in its own transaction and that transaction should be automatically rolled back when the example finishes. We could write all this code by hand, but that would make the bodies of the examples far too noisy, it would distract us from the code that makes up the examples!

The code in `SomeTest` now sets up everything we will need in the specifications. Specs2 Spring sets out to eliminate the duplication in setting up the `ApplicationContext` and the JNDI environment; and, if requested, executes each example in its own transaction and rolls back that transaction automatically when the example finishes.

Getting started

You can delegate all this work to the code that makes up Specs2 Spring extension. Let's start with an example and then explore the details.

```
import org.springframework.beans.factory.annotation.Autowired
import org.specs2.spring.Specification
import org.springframework.test.context.ContextConfiguration
import org.specs2.spring.annotation.DataSource
import org.hsqldb.jdbc.JDBCDriver

@DataSource(name = "java:comp/env/jdbc/test",
  driverClass = classOf[JDBCDriver], url = "jdbc:hsqldb:mem:test")
@ContextConfiguration(Array("classpath*/META-INF/spring/module-context.xml"))
class SomeTest extends Specification {
  @Autowired
  var someService: SomeService = _

  "specification" in {
    // call methods of someService that will hopefully return success!
    success
  }
}
```

The code in `SomeTest` includes *variable* of type `SomeService` that carries the `Autowired` annotation. This indicates to the Specs2 Spring extension that this field should be set to be the bean of type `SomeService` looked up from the `ApplicationContext` constructed by loading the configuration in `META-INF/spring/module-context.xml` files on the entire classpath. The code in one of the `META-INF/spring/module-context.xml` includes JNDI lookup of the `javax.sql.DataSource` under name `java:comp/env/jdbc/test`.

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

```
    http://www.springframework.org/schema/jee/spring-jee.xsd">

    <context:component-scan base-package="org.specs2.springexample"/>
    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/test"
        expected-type="javax.sql.DataSource"/>

</beans>
```

Notice the `jee:jndi-lookup` element, which queries the JNDI environment for the `java:comp/env/jdbc/test`, whose expected type is `javax.sql.DataSource`.

To repeat the introductory paragraph, the Specs2 Spring includes code that will help you set up the context for the entire integration test--and by context, we mean the appropriate entries in the JNDI environment as well as the beans under test, autowired in to the instance of the test under execution. Finally, Specs2 Spring can be easily configured to run every example in its own transaction that rolls back automatically when the example completes.

Part II. Using Specs2 Spring

Chapter 2. Using Specs2 Spring Specification

Using Specs2 Spring

To configure the Spring `ApplicationContext`, you need to use the `ContextConfiguration` annotation on a subclass of the `org.specs2.spring.Specification`. The code in the `org.specs2.spring.Specification` will build the `ApplicationContext` by instantiating the `ClassPathXmlApplicationContext` using the configuration files specified in the values property of the `ContextConfiguration`.

Once the `org.specs2.spring.Specification` is done constructing the `ApplicationContext`, it will autowire the variables of the test using the beans in the context. (Obviously, you have to use the `Autowired` annotation on the variables that you want to have autowired.) Once the Spring `ApplicationContext` is ready and the variables autowired, you can use them in the bodies of the examples.

In addition to setting up properly configured Spring `ApplicationContext`, Specs2 Spring sets up the JNDI environment for the test. This allows you to have the same Spring context configuration files, regardless of whether you are running tests, trying out the application on your own machine, deploying the application to pre-production servers or even running your application live.

In this section, you will find out how to configure your Spring `ApplicationContext` and how to configure the JNDI environment for your tests. Finally, we will discuss using running the examples in your specifications in their own transactions and controlling the behaviour of those transactions

Understanding the Specs2 setup annotations

The annotations on a Specs2 Spring test control & define how the Spring `ApplicationContext` is constructed, define the elements of the JNDI environment required for the test. (See the section called “Understanding the Specs2 behavioural annotations” [18]). The set-up annotations are:

Table 2.1. Specs2 Spring annotations

Annotation	Description
<code>ContextConfiguration</code>	Drives the way in which the Specs2 Spring extension will create the <code>ClassPathXmlApplicationContext</code>
<code>DataSource</code>	Adds the <code>javax.sql.DataSource</code> to the environment
<code>TransactionManager</code>	Adds the <code>javax.transaction.UserTransaction</code> to the environment
<code>Jms</code>	Adds the <code>javax.jms.ConnectionFactory</code> to the environment
<code>JmsTopic</code>	Adds the <code>javax.jms.Topic</code> to the environment
<code>JmsQueue</code>	Adds the <code>javax.jms.Queue</code> to the environment
<code>MailSession</code>	Adds the <code>javax.mail.Session</code> to the environment
<code>WorkManager</code>	Adds the <code>commonj.work.WorkManager</code> or <code>javax.spi.resource.work.WorkManager</code> to the environment

Annotation	Description
Transactional	Indicates that every example should run in its own transaction, the <code>TransactionConfiguration</code> annotation refines the transactional behaviour further
TransactionConfiguration	Tunes the transactional semantics that is applied to the transactional examples

But there is one more annotation, `Jndi`, which aggregates all other annotations; moreover, it lets you specify *multiple* `DataSources`, `Queues`, `Topics`, (Suppose you needed two `DataSources`, you must use the `Jndi` annotation, you cannot annotate your specification with two `DataSource` annotations.) Its usage is straight-forward:

```
@Jndi(
  dataSources = {
    @DataSource(name = "java:comp/env/jdbc/test",
      driverClass = JDBCDriver.class, url = "jdbc:hsqldb:mem:test"),
    @DataSource(name = "java:comp/env/jdbc/external",
      driverClass = JDBCDriver.class, url = "jdbc:hsqldb:mem:external")
  },
  mailSessions = @MailSession(name = "java:comp/env/mail/foo"),
  transactionManager =
    @TransactionManager(name = "java:comp/TransactionManager"),
  jms = @Jms(
    connectionFactoryName = "java:comp/env/jms/connectionFactory",
    queues = {
      @JmsQueue(name = "java:comp/env/jms/requests"),
      @JmsQueue(name = "java:comp/env/jms/responses")
    },
    topics = {
      @JmsTopic(name = "java:comp/env/jms/cacheFlush"),
      @JmsTopic(name = "java:comp/env/jms/ruleUpdate")
    }
  ),
  workManagers = @WorkManager(name = "java:comp/env/work/WorkManager",
    kind = WorkManager.Kind.CommonJ)
)
@Transactional
@TransactionConfiguration(defaultRollback = true)
@ContextConfiguration("classpath*/META-INF/spring/module-context.xml")
public class SomeTest... {
}
```

In Java, we can use (for example) the `DataSource` annotation as the element of the `dataSources` property of the `Jndi` annotation. Unfortunately, we have used the annotation on a Java class. You cannot use annotations as elements of the properties of an annotation in Scala. If you want to use the `Jndi` annotation in your Scala code, you have to create an annotation and *annotate the custom annotation with the `Jndi` annotation*. Typically, you will also include the `ContextConfiguration`, `Transactional` and `TransactionDefinition` on the custom annotation and use the custom annotation throughout your Specs2 code.

```
@Jndi(
  dataSources = {
```

```

    @DataSource(name = "java:comp/env/jdbc/test",
        driverClass = JDBCDriver.class, url = "jdbc:hsqldb:mem:test"),
    @DataSource(name = "java:comp/env/jdbc/external",
        driverClass = JDBCDriver.class, url = "jdbc:hsqldb:mem:external")
    },
    mailSessions = @MailSession(name = "java:comp/env/mail/foo"),
    transactionManager =
        @TransactionManager(name = "java:comp/TransactionManager"),
    jms = @Jms(
        connectionFactoryName = "java:comp/env/jms/connectionFactory",
        queues = {
            @JmsQueue(name = "java:comp/env/jms/requests"),
            @JmsQueue(name = "java:comp/env/jms/responses")},
        topics = {
            @JmsTopic(name = "java:comp/env/jms/cacheFlush"),
            @JmsTopic(name = "java:comp/env/jms/ruleUpdate")}
    ),
    workManagers = @WorkManager(name = "java:comp/env/work/WorkManager",
        kind = WorkManager.Kind.CommonJ)
    )
@Transactional
@TransactionalConfiguration(defaultRollback = true)
@ContextConfiguration("classpath*/META-INF/spring/module-context.xml")
@Retention(RetentionPolicy.RUNTIME)
public @interface IntegrationTest {
}

```

You now have the `IntegrationTest` annotation (in Java), which you can use on your Scala specifications:

```

import org.specs2.spring.Specification
...

@IntegrationTest
class SomeComponentSpec extends Specification {
    ...
}

```

The ContextConfiguration annotation

The `ContextConfiguration` annotation is the core of the Specs2 Spring extension. This annotation configures how Specs2 Spring is going to construct the Spring `ApplicationContext`. Version 1.0 of Specs2 Scala examines only the value property; it ignores the `locations`, `inheritLocations` and `loader` properties that you may be used to in your Spring testing using the `spring-test` artifact.

The `String[]` value property defines the array of locations the context files for the `ClassPathXmlApplicationContext` to use. For example, writing `@ContextConfiguration(Array("classpath*/a.xml", "classpath*/b.xml"))` will instruct the Specs2 Spring extension to create the Spring `ApplicationContext` by (in effect) calling `new ClassPathXmlApplicationContext(new String[]{"classpath*/a.xml", "classpath*/b.xml"})`.

Once the `ApplicationContext` is constructed, the Specs2 Scala extension will inject the configured dependencies into the specification. Just like ordinary Java Spring code, you need to instruct the Specs2 Spring to perform the DI--in other words, use the `Autowired` annotation on fields (vars in Scala) or on the setters. *For completeness, you may not use constructor injection..* A complete example of Specs2 Spring specification class that uses the `ContextConfiguration` annotation could be:

```
import org.specs2.spring.Specification
...

@ContextConfiguration(
  Array("classpath*/META-INF/spring/module-context.xml"))
class SomeComponentSpec extends Specification {
  @Autowired var someComponent: SomeComponent = _
  @Autowired var hibernateTemplate: HibernateTemplate = _

  "specification" in {
    ...
    success
  }
}
```

Notice in particular the `import org.specs2.spring.Specification`, application of the `@ContextConfiguration(Array("classpath*/META-INF/spring/module-context.xml"))` and the two variables marked with the `Autowired` annotation. When you run the test, by the time the code of your examples (the `"specification" in { ... }`) runs, the variables will have been injected.

The DataSource annotation

While the `ContextConfiguration` annotation drives the setup of the Spring `ApplicationContext`, the JNDI annotations inject the appropriate values to the environment for the test. The first JNDI annotation is `DataSource`.

The `DataSource` annotations adds the `javax.sql.DataSource` object to JNDI; the properties of the annotations specify what JDBC driver to use, set the RDBMS connection parameters and finally define the name for the `javax.sql.DataSource` in the environment. If you include the `TransactionManager`, the `DataSource` will be an XA (two-phase commit) one; otherwise, it will be the ordinary one.

The following code snippet shows typical configuration of the `DataSource` annotation:

```
@DataSource(
  name = "java:comp/env/jdbc/test",
  driverClass = JDBCDriver.class,
  url = "jdbc:hsqldb:mem:test",
  username = "sa",
  password = "")
```

The code listing shows all properties of the `DataSource` annotation.

Table 2.2. Properties of the DataSource annotation

Property	Description
name	The name in JNDI environment that the created <code>javax.sql.DataSource</code> should be registered under. This name must match the <code>jndi-name</code> of the <code>jee:jndi-lookup</code> element of the Spring configuration file.
driverClass	The class of the JDBC driver (must be subtype of <code>java.sql.Driver</code>)
url	The JDBC connection URL; this string will vary according to the used JDBC driver.
username	The username to be used to create the connection to the RDBMS.
password	The password to be used to create the connection to the RDBMS.

The complete example that ties the `DataSource` annotation and the Spring configuration file is simply:

```
import org.specs2.spring.Specification
import org.hsqldb.jdbc.JDBCdriver
...

@DataSource(
  name = "java:comp/env/jdbc/test",
  driverClass = classOf[JDBCdriver],
  url = "jdbc:hsqldb:mem:test",
  username = "sa",
  password = "")
@ContextConfiguration(
  Array("classpath*/META-INF/spring/module-context.xml"))
class SomeSpecification extends Specification {
  ...
}
```

The annotation on the specification class instructs the Specs2 Spring to add the `javax.sql.DataSource` (that connects to HSQL DB using the `org.hsqldb.jdbc.JDBCdriver`, URL `jdbc:hsqldb:mem:test` and credentials `sa/`) *before* constructing the Spring `ApplicationContext` using the configuration files located at `/META-INF/spring/module-context.xml`. To consume the registered JNDI entry in one of the context files, simply use the `jee:jndi-lookup` element.

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

```

<context:component-scan base-package="org.specs2.springexample"/>
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/test"
  expected-type="javax.sql.DataSource"/>

</beans>

```

In the body of the specification, you can now request autowiring of the `dataSource` bean (in addition to any beans that may be discovered by the `context:component-scan`).

The TransactionManager annotation

It is typical to use some kind of transactional semantics in Spring applications. Spring Framework abstracts out the details of how it is going to achieve this transactional behaviour by defining the `PlatformTransactionManager` interface and providing several implementations of the interface. There are, to name a few, `DataSourceTransactionManager`, `JpaTransactionManager`, `HibernateTransactionManager` and many other *local* transaction managers.

Imagine now that you would like the transaction to span across multiple transactional resources, for example two RDBMSs, JMS queues and topics. In that scenario, these local transaction managers will not work: being *local*, they can only manage the transactional semantics on a single resource. The only transaction manager that can apply transactional semantics to *multiple* resources is the `JtaTransactionManager`. As the name suggests, the `JtaTransactionManager` is the implementation of the `PlatformTransactionManager` that uses the Java transaction API-compliant manager. The manager is the implementation of the `javax.transaction.TransactionManager` or `javax.transaction.UserTransaction`. The Specs2 Spring includes support for the global transaction manager; if you use the `TransactionManager` annotation, Specs2 Spring will also create all `DataSources` that are aware of the global transaction manager.

The `TransactionManager` annotation includes only one property, `name`.

Table 2.3. Properties of the TransactionManager annotation

Property	Description
<code>name</code>	The name in JNDI environment that the created <code>javax.transaction.UserTransaction</code> should be registered under. Typically, the value of this property is <code>java:comp/TransactionManager</code>

Spring's `<tx:jta-transaction-manager />` automatically scans the JNDI environment for the JTA elements under `java:comp/TransactionManager` name. This makes typical usage of this annotation very simple:

```

import org.specs2.spring.Specification
import org.hsqldb.jdbc.JDBCdriver
...

@DataSource(
  name = "java:comp/env/jdbc/test",
  driverClass = classOf[JDBCdriver],
  url = "jdbc:hsqldb:mem:test",
  username = "sa",
  password = "")
@TransactionManager(name = "java:comp/TransactionManager")

```

```
@ContextConfiguration(
    Array("classpath*/META-INF/spring/module-context.xml"))
class SomeSpecification extends Specification {
    ...
}
```

The annotation on the specification class instructs the Specs2 Spring to add the `javax.transaction.UserTransaction` to the environment. To consume the registered JNDI entry in one of the context files, simply use the `tx:jta-transaction-manager` element.

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

    <context:component-scan base-package="org.specs2.springexample"/>
    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/test"
        expected-type="javax.sql.DataSource"/>
    <tx:jta-transaction-manager />
    <tx:annotation-driven />

</beans>
```

In the body of the specification, you now have the `transactionManager` bean that has been obtained by looking up the `java:comp/TransactionManager` name. Typically, you will also need to include the `tx:annotation-driven` element to weave in the transactional advices, allowing you to use the `Transactional` annotation on types and methods. (And, ideally, you will use load- or compile-time weaving; tuning the `tx:annotation-driven` element by specifying the `mode` attribute: `<tx:annotation-driven mode="aspectj"/>`.)

JMS annotations

The JMS infrastructure combines multiple queues and topics under a connection factory. The `javax.jms.ConnectionFactory` is the main "access point" to the JMS infrastructure. Spring applications make use of the connection factory to register code to be called whenever a message appears on a queue; the code of JMS-based Spring applications typically also needs to have access to the queues and topics.

The `javax.jms.ConnectionFactory`, `javax.jms.Queue` and `javax.jms.Topic` objects are usually looked up from JNDI.

The basic setup of the JMS infrastructure therefore requires that you use the `Jms` annotation, specifying the name for the connection factory and either a `JmsTopic` and/or `JmsQueue` that define the JMS topic and queue, respectively. All three annotations only require you to set the the name in the environment (and which must match the `jndi-name` attribute of the `jee:jndi-lookup` element).

The following code snippet shows typical configuration of the JMS annotations:

```
@Jms(connectionFactoryName = "java:comp/env/jms/connectionFactory")
@JmsQueue(name = "java:comp/env/jms/requests")
@JmsTopic(name = "java:comp/env/jms/cacheFlush")
class SomeSpecification extends Specification {
    ...
}
```

Exploring the properties of each annotation further, we have:

Table 2.4. Properties of the JMS annotation

Property	Description
connectionFactoryName	The name in JNDI environment that the created <code>javax.jms.ConnectionFactory</code> should be registered under.

Table 2.5. Properties of the JMS queue

Property	Description
name	The name in JNDI environment that the created <code>javax.jms.Queue</code> should be registered under.

Table 2.6. Properties of the JmsTopic annotation

Property	Description
name	The name in JNDI environment that the created <code>javax.jms.Topic</code> should be registered under.

Expanding the previous examples with the JMS annotations, we now have:

```
import org.specs2.spring.Specification
import org.sqljdbc.jdbc.JDBCdriver
...

@DataSource(
  name = "java:comp/env/jdbc/test",
  driverClass = classOf[JDBCdriver],
  url = "jdbc:sqljdbc:mem:test",
  username = "sa",
  password = "")
@Jms(connectionFactoryName = "java:comp/env/jms/connectionFactory")
@JmsQueue(name = "java:comp/env/jms/requests")
@JmsTopic(name = "java:comp/env/jms/cacheFlush")
@ContextConfiguration(
  Array("classpath*/META-INF/spring/module-context.xml"))
class SomeSpecification extends Specification {
    ...
}
```

In addition to the RDBMS and the global transaction manager, we define a JMS infrastructure and one queue and one topic. We consume the newly defined elements in the Spring configuration file using the `jee:jndi-lookup` element.

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

  <context:component-scan base-package="org.specs2.springexample"/>
  <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/test"
    expected-type="javax.sql.DataSource"/>
  <tx:jta-transaction-manager />
  <tx:annotation-driven />
  <jee:jndi-lookup id="connectionFactory"
    jndi-name="java:comp/env/jms/connectionFactory"
    expected-type="javax.jms.ConnectionFactory"/>
  <jee:jndi-lookup id="requests"
    jndi-name="java:comp/env/jms/requests"
    expected-type="javax.jms.Queue"/>
  <jee:jndi-lookup id="cacheFlush"
    jndi-name="java:comp/env/jms/cacheFlush"
    expected-type="javax.jms.Topic"/>

</beans>
```

Note that in the sample code, the global transaction manager (registered using the `TransactionManager` annotation) can manage transactions over all transactional resources; in this case, the RDBMS *and* the JMS queue or topic.

Most applications require more than one queue or topic; some applications even require more than one connection factory. In that situation, you must use the *annotated annotation* approach we already demonstrated (see the section called “Understanding the Specs2 setup annotation{9} or ***advanced-xref***).

The MailSession annotation

If you are sending e-mails from your application, you will need the `javax.mail.Session`, which allows you to construct `javax.mail.Message` (including the complex multipart messages, deal with attachments and many other e-mail tasks). Just like all other JNDI annotations, you need to specify the name for the entry in JNDI and configure the remaining properties of the Javamail session.

The following code snippet shows typical configuration of the `MailSession` annotation:

```
@MailSession(
```

```
name = "java:comp/env/mail/foo",
properties = Array("mail.smtp.host=localhost", "mail.mime.charset=UTF-8"))
```

Exploring the properties of the annotation further, we have:

Table 2.7. Properties of the MailSession annotation

Property	Description
name	The name in JNDI environment that the created <code>javax.mail.Session</code> should be registered under.
properties	The array of Strings that represent the Javamail properties, in form of <i>property-key=property-value</i> . For example, if you wish to specify the <code>mail.smtp.host</code> to <code>localhost</code> and <code>mail.mime.charset</code> to <code>UTF-8</code> , you write <code>properties = Array("mail.smtp.host=localhost", "mail.mime.charset=UTF-8")</code>

To consume the `javax.mail.Session` in your Spring application, use the familiar `jee:jndi-lookup` element and make sure that the value of the `jndi-name` attribute matches the value of the name property of the `MailSession` annotation.

The WorkManager annotation

Finally, some applications use one of the two common `WorkManager` implementations to submit work to [a pool of] threads. Typically, the application server maintains and monitors the `WorkManager` instance. The application servers allow the administrators to configure the threads that should become parts of the `WorkManager` instances. Finally, the application servers usually then report on the load of the `WorkManager`. All that is left to us is to consume the `WorkManager` and use it in our application. *Having the application server maintain the `WorkManager` instances is far better approach than creating your own thread pools within your application.*

To configure the `WorkManager`, all you need to specify is the JNDI name and set the remaining properties:

```
@WorkManager(name = "java:comp/env/work/WorkManager",
    kind = WorkManager.Kind.CommonJ,
    maximumThreads = 5,
    minimumThreads = 3)
```

Exploring the properties of the annotation further, we have:

Table 2.8. Properties of the WorkManager annotation

Property	Description
name	The name in JNDI environment that the created <code>javax.mail.Session</code> should be registered under.
kind	Defines the kind of the created <code>WorkManager</code> instance. If the value is <code>WorkManager.Kind.CommonJ</code> , then the instance will

Property	Description
	implement the the <code>commonj.work.WorkManager</code> . If the value is <code>WorkManager.Kind.Javax</code> , then the instance will implement the <code>javax.spi.resource.work.WorkManager</code> .
<code>maximumThreads</code>	The maximum number of threads the created work manager should allow to be created. Must be greater than 1 and greater than <code>minimumThreads</code> .
<code>minimumThreads</code>	The minimum number of threads the created work manager should create. Must be greater than 0 and smaller than <code>minimumThreads</code> .

To consume the work manager in your Spring application, use the familiar `jee:jndi-lookup` element and make sure that the value of the `jndi-name` attribute matches the value of the `name` property of the `WorkManager` annotation.

Understanding the Specs2 behavioural annotations

The annotations on a Specs2 Spring behavioural annotations control the runtime of the examples. The behaviour defines the transactional semantics that will be applied to the examples. The transactional example behavioural support needs to have access to the `PlatformTransactionManager` bean. This `PlatformTransactionManager` can be any of its implementations, even the local one, if it is appropriate for your code. Typically, though, you will use the `TransactionManager` annotation together with the `tx:jta-transaction-manager` element, which exposes the transaction manager that the Specs2 Spring will use to apply the transactional behaviour to the examples. (See the section called “Understanding the Specs2 setup annotations” [8]).

Table 2.9. Specs2 Spring behavioural annotations

Annotation	Description
<code>Transactional</code>	Indicates that every example should run in its own transaction, the <code>TransactionConfiguration</code> annotation refines the transactional behaviour further
<code>TransactionConfiguration</code>	Tunes the transactional semantics that is applied to the transactional examples

When you annotate your test class with the `Transactional` annotation, every example will run in its own transaction and the transaction will be rolled back automatically when the example finishes. You can tune whether the transaction rolls back or commits by using the `defaultRollback` property of the `TransactionConfiguration` annotation. If `defaultRollback` is `false`, the transaction will *commit* when the example finishes; if the value is `true` (the default), the transaction will roll back.

Chapter 3. Testing large systems using Specs2 Spring

What are large systems?

Before we explore the intricacies of the code involved in testing large systems, we need to explore what we mean by *large system*. In what we call large systems, there are many components that need to be tested and those components depend on many other Java EE components. Moreover, the structure of the code should be as *elegant* and *flexible* as possible, without imposing technical restrictions.

Let's take a quick detour to explore the structure of large Spring applications. Enterprise [Spring] applications are usually made up of modules; in the smallest applications, these modules represent the tiers. The application is then simple packaging of these modules. Let's start with a small application that consists of four such modules. We have the domain, repository, services and the webapp.

The most efficient configuration structure is to create *self-configuring modules*. This means that every module that contains Spring functionality should carry just enough configuration to configure itself. Consequently, if we want to add functionality to the application, all that it take is to add the module to the application; no other portion of our application needs to change. Consider structure of application called `org.specs2.app`:

```
org.specs2.app.domain
src
main
  java
  org.specs2.app.domain
  scala
  org.specs2.app.domain
test
  java
  org.specs2.app.domain
  scala
  org.specs2.app.domain
org.specs2.app.repository
src
main
  resources
  META-INF
  spring
  module-context.xml
org.specs2.app.services
src
main
  resources
  META-INF
  spring
  module-context.xml
  scala
  org.specs2.app.services
  SomeService.scala
```

```
test
  scala
  org.specs2.app.services
    SomeServiceSpecification.scala
org.specs2.app.webapp
src
main
test
webapp
```

This rather long listing of directories shows that every module contains the `/META-INF/spring/module-context.xml` file. This file contains the Spring elements that are defined in the module.

Consider this example: in the `repository`, the `module-context.xml` defines, for example, the `HibernateTemplate` bean. The `services` *depends on* the `repository` module, therefore all beans defined in the `repository` module are "visible" in the `module-context.xml` file in the `services` module. The `webapp` then assembles two Spring application contexts: one for the user interface portion of the application and another application context that contains the headless beans.

Notice in particular the naming convention; particularly the `module-context.xml` files. If all other modules in your application follow the *self-configured* route, then adding new functionality to the application is as easy as adding a dependency.

Testing large systems

To test these large systems, we need to set up the environment for the test. The configuration that brings together the individual configuration files from the different modules is easy. If you had followed our naming advice, all you have to do is to annotate your tests with `@ContextConfiguration(Array("classpath*/META-INF/spring/module-context.xml"))`. This will load the configuration from *all module-context.xml-named files in all modules on the classpath!*

In addition to setting up the Spring `ApplicationContext`, we need to set up other Java EE components that the beans depend on. Such components include, for example, a JNDI-bound `javax.sql.DataSource`, `javax.mail.Session`, `javax.transaction.UserTransaction`, and many others. Furthermore, we need to set up the transactional behaviour of the tests itself.

The annotations you need to define all these components are the common Specs2 Spring annotations (viz the section called "Understanding the Specs2 setup annotations" [8]). Repeating the Specs2 annotations on every test results in clear code duplication. To reduce the code duplication, you should create custom Java annotation and annotate that annotation with the required Specs2 annotations; and use your custom annotation on your tests. Instead of having multiple tests that duplicate the same annotations:

```
import org.specs2.spring.Specification
...

@DataSource(name = "java:comp/env/jdbc/test",
  driverClass = classOf[JDBCDriver], url = "jdbc:hsqldb:mem:test")
@Transactional
@ContextConfiguration(Array("classpath*/META-INF/spring/module-context.xml"))
```

```

class SomeServiceTest extends Specification {
    @Autowired
    var someService: SomeService = _

    "specification" in {
        // call methods of someService that will hopefully return success!
        success
    }
}

import org.specs2.spring.Specification
...

@DataSource(name = "java:comp/env/jdbc/test",
    driverClass = classOf[JDBCdriver], url = "jdbc:hsqldb:mem:test")
@Transactional
@ContextConfiguration(Array("classpath*/META-INF/spring/module-context.xml"))
class AnotherServiceTest extends Specification {
    @Autowired
    var anotherService: AnotherService = _

    "specification" in {
        // call methods of anotherService that will hopefully return success!
        success
    }
}

```

We will create a custom annotation, and annotate it with the Specs2 annotations and use it on our tests:

```

@DataSource(name = "java:comp/env/jdbc/test",
    driverClass = classOf[JDBCdriver], url = "jdbc:hsqldb:mem:test")
@Transactional
@ContextConfiguration(Array("classpath*/META-INF/spring/module-context.xml"))
@Retention(RetentionPolicy.RUNTIME)
public @interface IntegrationTest {
}

import org.specs2.spring.Specification
...

@IntegrationTest
class SomeServiceTest extends Specification {
    @Autowired
    var someService: SomeService = _

    "specification" in {
        // call methods of someService that will hopefully return success!
        success
    }
}

import org.specs2.spring.Specification

```

```
...  
  
@IntegrationTest  
class AnotherServiceTest extends Specification {  
    @Autowired  
    var anotherService: AnotherService = _  
  
    "specification" in {  
        // call methods of anotherService that will hopefully return success!  
        success  
    }  
}
```

Multiple environment entries

In the previous example, we have used a single `javax.sql.DataSource`, `javax.mail.Session` (and many others). However, some applications require *many* of those Java EE components. Unfortunately, it is not possible to duplicate annotations on a class. We can't, for example, use two `DataSource` annotations.

Instead, we must use the `Jndi` annotation and specify the `DataSources` in its `dataSource` property. *Even more unfortunately, we cannot use the `Jndi` annotation in our Scala code, we must use the custom Java annotation route.*

The JNDI annotation defines the following properties, which let you define multiple Java EE components of the required type.

Chapter 4. Test data setup Specs2 Spring

Inserting test data

For a lot of the tests that you will be writing, you will need insert data to support your tests. You have several options; you can either:

- Use pre-populated test database,
- Write scripts that will insert the required test data,
- Use your application's *repository* code to insert the test data.

Now, each approach has its advantages and drawbacks. Using pre-populated database means that each developer needs to maintain his or her own copy of the database. In case of writing scripts, we are faced with scripts that are disconnected from the rest of our application's code. Finally, using the persistence tier in our application in the tests is cumbersome.

You can use the `HibernateDataAccess`, `HibernateTemplateDataAccess`, (and the future `JpaDataAccess`, `SqlDataAccess`, `DocumentDataAccess` or `GraphDataAccess`) traits together with the `BeanTables` trait to create the test data in a convenient tabular manner that is half-way between writing scripts that are completely disconnected and creating the objects manually and using your repository objects to persist them.

In other words, Specs2 Spring gives us half-way house that combines the scripts and using the repository tier to insert the objects. You can combine the script-like approach with manually creating and inserting the records. Specs2 Spring comes with `BeanTables`, a trait that allows you to construct lists of objects with their properties written in a convenient tabular form.

```
"username" | "firstName" |
"janm"     !! "Jan"      |
"marco"    !! "Marc"    |
```

Assuming there is a persistent class `User`, defined as:

```
@Entity
case class User() {
  @Id
  @GeneratedValue
  @BeanProperty
  var id: Long = _
  @Version
  @BeanProperty
  var version: Int = _
  @BeanProperty
  var username: String = _
  @BeanProperty
```

```

var firstName: String = _
@BeanProperty
var lastName: String = _

}

```

We would now expect that this table somehow creates two `User` instances whose `username` and `firstName` properties are set to the values in the rows of the table. We now need to be able to obtain the created objects or--as we'll see later--insert them into some underlying persistence mechanism. The "table" object that `BeanTables` creates contains functions `|>` and `|<`. Their names hint at the direction in which the objects will flow.

Let's start with obtaining the objects into a `List`. The `|>` function sends the rows of the constructed objects towards the left and top; that is, towards the variable definition.

```

class SpecificationSpec extends Specification with BeanTables {

  "user objects using BeanTables:" in {
    val inferredUsers: List[User] =
      "username" | "firstName" |
      "janm"     !! "Jan"       |
      "marco"    !! "Marc"      |<

    val typedUsers =
      "username" | "firstName" |
      "janm"     !! "Jan"       |
      "marco"    !! "Marc"      |<classOf[User]

    // assert something about typedUsers and inferredUsers
  }
}

```

The two variables, `inferredUsers` and `typedUsers` show the application of the `|>` function. In the first case, the function's type is inferred from the type of the `inferredUsers` variable. In the second case, we supply the type of the elements in the list (and we don't need to explicitly specify the type of the `List`). Once you have the instances from the table, you can then use them as any other instance in your code.

In a lot of cases, though, you will want to insert the data using some underlying persistence mechanism. To do that, you need to use the `|>` functions of the table. The `|>` function expects a function that takes some type `B` and returns `Result`. `Specs2 Spring` includes traits with functions that return just the function that `|>` expects. For example, you can use the `insert` function from the `HibernateDataAccess` trait.

```

class SpecificationSpec extends Specification
  with BeanTables with HibernateDataAccess {

  "user objects using BeanTables:" in {
    val sessionFactory = make-hibernate-SessionFactory()

    "username" | "firstName" |

```

```

    "janm"    !! "Jan"    |
    "marco"   !! "Marc"   |> insert[User](sessionFactory)

    // assert something about the users in the DB
  }
}

```

We will explore the details of the traits with functions that can be used in the `|>` function in the table.

HibernateDataAccess

The `HibernateDataAccess` trait includes functions `insert` and `deleteAll`; functions that can be used with the `BeanTable`'s `|>` functions.

We have the `insert[T](implicit sessionFactory: SessionFactory): (T => Result)` and `insert[T, R](f: T => R)(implicit sessionFactory: SessionFactory): (T => Result)`. The first `insert` returns a function that takes an instance `T` and performs the `Hibernate saveOrUpdate` operation. The second `insert` function takes a function that is applied to every instance `T` before it is inserted. Both functions accept the implicit parameter of type `SessionFactory`.

`insert[T](implicit sessionFactory: SessionFactory): (T => Result)`

The first `insert` requires only the `SessionFactory` that will be used to perform the `saveOrUpdate` operation. In a typical Spring Framework application, the `SessionFactory` is usually a Spring bean. Therefore, it can be injected into the specification.

```

@ContextConfiguration(Array("classpath*/META-INF/spring/module-context.xml"))
class SpecificationSpec extends Specification
  with BeanTables with HibernateDataAccess {

  @Autowired implicit var sessionFactory: SessionFactory = _
  @Autowired var springComponent: SpringComponent = _

  "springComponent must:" in {
    "username" | "firstName" |
    "janm"     !! "Jan"      |
    "marco"    !! "Marc"     |> insert[User]

    springComponent.findAll[User].size() must_== (2)
  }
}

```

`insert[T, R](f: T => R)(implicit sessionFactory: SessionFactory): (T => Result)`

The usage of the second `insert` function requires a function that will operate on every instance of type `T` from the table (with its properties set) and the `SessionFactory`. Again, in a typical Spring

Framework application, the `SessionFactory` is usually a Spring bean. Therefore, it can be injected into the specification.

```
@ContextConfiguration(Array("classpath*/META-INF/spring/module-context.xml"))
class SpecificationSpec extends Specification
  with BeanTables with HibernateDataAccess {

  @Autowired implicit var sessionFactory: SessionFactory = _
  @Autowired var springComponent: SpringComponent = _

  "springComponent must:" in {
    "username" | "firstName" |
    "janm"     !! "Jan"       |
    "marco"    !! "Marc"      |> insert[User]

    springComponent.findAll[User].size() must_== (2)
  }
}
```

Part III. Technical background

Chapter 5. Improving Specs2 Spring

Building Specs2 Spring from sources

If you are interested in peeking under the hood of Specs2 Spring; if you want to try to implement a new feature, you'll need to build Specs2 Spring from the sources; you might also want to set up your IDE to make it easier to write the Scala and Java code.

To build Specs2 Spring from sources, you'll need:

- GPG to and keyset to sign the JARs. Download GPG from <http://www.gnupg.org/>,
- Paul Phillips's SBT Extras at <https://github.com/paulp/sbt-extras>,
- Clones of <https://github.com/janm399/sbt-docbook-plugin> and <https://github.com/janm399/specs2-spring>

Starting with the most user-friendly approach, install & configure GPG keychain. The details will be different, depending on your platform, but on UNIX systems, you should be able to run:

```
$ gpg --list-secret-keys
/Users/janmachacek/.gnupg/secring.gpg
-----
sec   2048R/90A468A9 2012-01-30 [expires: 2016-01-30]
uid                               Jan Machacek <jan.machacek@gmail.com>
ssb   2048R/A9ED23D0 2012-01-30
```

If you cannot see any keys, you will need to generate a keypair by running `gpg --gen-key`. Onwards! Once you download the SBT Extras shell script, put it somewhere you remember and add it to your `PATH`. The common location for Specs2 Spring team at Cake Solutions is in `/usr/share/scala/sbt`. We also modify the `PATH` environment variable in `/etc/profile` so we can simply run `sbt`.

```
export PATH=$PATH:/usr/share/scala/sbt
```

Next, clone the two repositories to some directory, say `~/Sandbox`. Then you need to publish both projects to your local Ivy repository (this is where they are going to be picked up from later on).

```
~/Sandbox$ git clone https://github.com/janm399/specs2-spring
~/Sandbox$ git clone https://github.com/janm399/sbt-docbook-plugin

~/Sandbox$ cd sbt-docbook-plugin
~/Sandbox/sbt-docbook-plugin$ sbt publish-local

~/Sandbox/sbt-docbook-plugin$ cd ../specs2-spring
~/Sandbox/specs2-spring$ sbt publish-local
```

And you're ready to go. The `"de.undercouch" % "sbt-docbook-plugin" % "0.2-SNAPSHOT"` and `"org.specs2" % "spring" % "0.4"` are now available in your local Ivy repository; and you can use them in your Maven or SBT projects.

Contributing to Specs2 Spring

If you are interested in helping out, the first thing to do is to take a look at the currently open issues at <https://github.com/janm399/specs2-spring/issues> and decide which one you'd like to tackle. To start making changes to the code, you'll need to fork the repository from `janm399/specs2-spring` to your own Github account. The clone in your account will give you read-write access. Clone your repository from Github to your machine and hack away! All Specs2 Spring authors will be delighted to help you, please contact us at janm@cakesolutions.net [<mailto:janm@cakesolutions.net>], anirvanc@cakesolutions.net [<mailto:anirvanc@cakesolutions.net>] or markh@cakesolutions.net [<mailto:markh@cakesolutions.net>]. Follow us on twitter, too--we are [@honzam399](https://twitter.com/honzam399) [<https://twitter.com/honzam399>], [@anirvan_c](https://twitter.com/anirvan_c) [https://twitter.com/anirvan_c] and [@markglh](https://twitter.com/markglh) [<https://twitter.com/markglh>].

When your work is done, send a pull request. We will review your code, sort out any last niggles and then merge the code into our Specs2 Spring repository. As you gain experience, we will be delighted to add you as the contributor to the main `janm399/specs2-spring` repository so you don't have to worry about the merge dances.