Spring 3.1 M2: Testing with @Configuration Classes and Profiles

 ENGINEERING

https://gravatar.com/avatar/64310de9c07f30458f9baae33d91ccdc?s=20&d=mm [SAM BRANNEN](https://spring.io/team/sbrannen)

 JUNE 21, 2011

[9 COMMENTS](https://spring.io/blog/2011/06/21/spring-3-1-m2-testing-with-configuration-classes-and-profiles#disqus_thread)

As Jürgen Höller mentioned in his post announcing the [release of Spring 3.1 M2](https://bit.ly/jPg8az), the [Spring TestContext Framework](https://bit.ly/m3UHhv)(\*) has been overhauled to provide first-class testing support for @Configuration classes and environment profiles.

In this post I’ll first walk you through some examples that demonstrate these new testing features. I’ll then cover some of the new extension points in the TestContext framework that make these new features possible.

      Please note: this is a cross post from my company blog [www.swiftmind.com](https://bit.ly/ixacQb).

**Background**

In Spring 2.5 we introduced the *Spring TestContext Framework* which provides annotation-driven integration testing support that can be used with JUnit or TestNG. The examples in this blog will focus on JUnit-based tests, but all features used here apply to TestNG as well.

At its core, the TestContext framework allows you to annotate test classes with @ContextConfiguration to specify which configuration files to use to load the ApplicationContext for your test. By default the ApplicationContext is loaded using the GenericXmlContextLoader which loads a context from XML Spring configuration files. You can then access beans from the ApplicationContext by annotating fields in your test class with @Autowired, @Resource, or @Inject.

Spring 3.0 introduced support for Java-based configuration via @Configuration classes, but the TestContext framework did not supply an appropriate ContextLoader to support @Configuration classes in tests until now. Spring 3.1 M2 introduces a new AnnotationConfigContextLoader for this purpose, and the @ContextConfiguration annotation has been updated to support declaration of @Configuration classes via a new classesattribute.

Let’s take a look at some examples now.

**Integration Testing with XML-based Configuration**

The [Testing](https://bit.ly/m3UHhv) chapter of the Spring Reference Manual provides numerous examples of how to configure integration tests using XML configuration files, but we’ll include an example here as a quick introduction.

*If you’re already familiar with the Spring TestContext Framework, feel free to skip to the next section.*

<?xml version="1.0" encoding="UTF-8"?>

<beans ...>

<!-- this bean will be injected into the OrderServiceTest class -->

<bean id="orderService" class="com.example.OrderServiceImpl">

<!-- set properties, etc. -->

</bean>

<!-- other beans -->

</beans>

package com.example;

@RunWith(SpringJUnit4ClassRunner.class)

// ApplicationContext will be loaded from "classpath:/com/example/OrderServiceTest-context.xml"

@ContextConfiguration

public class OrderServiceTest {

@Autowired

private OrderService orderService;

@Test

public void testOrderService() {

// test the orderService

}

}

In the preceding example we configure JUnit to use the SpringJUnit4ClassRunner to run our tests. We do this using JUnit’s @RunWith annotation. We also annotate our test class with Spring’s @ContextConfiguration annotation without specifying any attributes. In this case the default GenericXmlContextLoader will be used, and following the principle of *convention over configuration* Spring will load our ApplicationContext from classpath:/com/example/OrderServiceTest-context.xml. Within the testOrderService()method we can directly test the OrderService that was injected into our test instance using @Autowired. Note that the orderService is defined as a bean in OrderServiceTest-context.xml.

**Integration Testing with @Configuration Classes**

Spring 3.1 M2’s support for integration testing with @Configuration classes is analogous to the XML-based example above. So let’s rework that example to use a @Configuration class and the new AnnotationConfigContextLoader.

package com.example;

@RunWith(SpringJUnit4ClassRunner.class)

// ApplicationContext will be loaded from the static inner ContextConfiguration class

@ContextConfiguration(loader=AnnotationConfigContextLoader.class)

public class OrderServiceTest {

@Configuration

static class ContextConfiguration {

// this bean will be injected into the OrderServiceTest class

@Bean

public OrderService orderService() {

OrderService orderService = new OrderServiceImpl();

// set properties, etc.

return orderService;

}

}

@Autowired

private OrderService orderService;

@Test

public void testOrderService() {

// test the orderService

}

}

There are a few notable differences between this example and the XML-based one:

1. There is no XML file.
2. The bean definitions have been converted from XML to Java using @Configuration and @Bean in the static inner ContextConfiguration class.
3. The AnnotationConfigContextLoader has been specified via the loader attribute of @ContextConfiguration.

Otherwise, the configuration and implementation of the test remain unchanged.

So, how does Spring know to use the static inner ContextConfiguration class to load the ApplicationContext? The answer is *convention over configuration*. By default, if no classes are explicitly declared, AnnotationConfigContextLoader will look for a static inner class of the test class named ContextConfiguration. Per the requirements of @Configuration classes, this static inner class must be non-final and non-private.

*Note: as of Spring 3.1 M2, the default configuration class must be named exactly ContextConfiguration. As of Spring 3.1 RC1, however, the naming restriction has been lifted. In other words, from RC1 forward, you can choose to name your default configuration class whatever you want, but the other requirements still apply.*

In the following example we’ll see how to declare explicit configuration classes.

package com.example;

@Configuration

public class OrderServiceConfig {

// this bean will be injected into the OrderServiceTest class

@Bean

public OrderService orderService() {

OrderService orderService = new OrderServiceImpl();

// set properties, etc.

return orderService;

}

}

package com.example;

@RunWith(SpringJUnit4ClassRunner.class)

// ApplicationContext will be loaded from the OrderServiceConfig class

@ContextConfiguration(classes=OrderServiceConfig.class, loader=AnnotationConfigContextLoader.class)

public class OrderServiceTest {

@Autowired

private OrderService orderService;

@Test

public void testOrderService() {

// test the orderService

}

}

We have now extracted the static inner ContextConfiguration class into a top-level class named OrderServiceConfig. To instruct the AnnotationConfigContextLoader to use this configuration class instead of relying on the default, we simply declare OrderServiceConfig.class via the new classes attribute of @ContextConfiguration. As with @ContextConfiguration’s locations attribute for resource locations, we can declare multiple configuration classes by supplying a Class[] array to the classes attribute — for example: @ContextConfiguration(classes={Config1.class, Config2.class}, … ).

This ends the coverage of integration testing with @Configuration classes. Now let’s take a look at Spring’s testing support for environment profiles.

**Integration Testing with Environment Profiles**

As Chris Beams discussed in his release announcement for [Spring 3.1 M1](https://bit.ly/g8Eiv6) and his follow-up blog [Introducing @Profile](https://bit.ly/mbOdHa), Spring 3.1 introduces first-class support in the framework for the notion of environments and profiles (a.k.a., *bean definition profiles*). As of Spring 3.1 M2, integration tests can also be configured to activate particular bean definition profiles for various testing scenarios. This is achieved by annotating a test class with the new @ActiveProfiles annotation and supplying a list of profiles that should be activated when loading the ApplicationContext for the test.

*Note: @ActiveProfiles may be used with any implementation of the new SmartContextLoaderSPI (see later discussion), but @ActiveProfiles is****not****supported with implementations of the simpler ContextLoader SPI.*

Let’s take a look at some examples with XML configuration and @Configuration classes.

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

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

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

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

xsi:schemaLocation="...">

<bean id="transferService" class="com.bank.service.internal.DefaultTransferService">

<constructor-arg ref="accountRepository"/>

<constructor-arg ref="feePolicy"/>

</bean>

<bean id="accountRepository" class="com.bank.repository.internal.JdbcAccountRepository">

<constructor-arg ref="dataSource"/>

</bean>

<bean id="feePolicy" class="com.bank.service.internal.ZeroFeePolicy"/>

<beans profile="dev">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

</beans>

package com.bank.service;

@RunWith(SpringJUnit4ClassRunner.class)

// ApplicationContext will be loaded from "classpath:/app-config.xml"

@ContextConfiguration("/app-config.xml")

@ActiveProfiles("dev")

public class TransferServiceTest {

@Autowired

private TransferService transferService;

@Test

public void testTransferService() {

// test the transferService

}

}

When TransferServiceTest is run, its ApplicationContext will be loaded from the app-config.xml configuration file in the root of the classpath. If you inspect app-config.xmlyou’ll notice that the accountRepository bean has a dependency on a dataSource bean; however, dataSource is not defined as a top-level bean. Instead, dataSource is defined twice: once in the *production* profile and once in the *dev* profile.

By annotating TransferServiceTest with @ActiveProfiles(“dev”) we instruct the Spring TestContext Framework to load the ApplicationContext with the active profiles set to {“dev”}. As a result, an embedded database will be created, and the accountRepository bean will be wired with a reference to the development DataSource. And that’s likely what we want in an integration test!

The following code listings demonstrate how to implement the same configuration and integration test but using @Configuration classes instead of XML.

@Configuration

@Profile("dev")

public class StandaloneDataConfig {

@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.addScript("classpath:com/bank/config/sql/test-data.sql")

.build();

}

}

@Configuration

@Profile("production")

public class JndiDataConfig {

@Bean

public DataSource dataSource() throws Exception {

Context ctx = new InitialContext();

return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

}

@Configuration

public class TransferServiceConfig {

@Autowired DataSource dataSource;

@Bean

public TransferService transferService() {

return new DefaultTransferService(accountRepository(), feePolicy());

}

@Bean

public AccountRepository accountRepository() {

return new JdbcAccountRepository(dataSource);

}

@Bean

public FeePolicy feePolicy() {

return new ZeroFeePolicy();

}

}

package com.bank.service;

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(loader=AnnotationConfigContextLoader.class,

classes={TransferServiceConfig.class, StandaloneDataConfig.class, JndiDataConfig.class})

@ActiveProfiles("dev")

public class TransferServiceTest {

@Autowired

private TransferService transferService;

@Test

public void testTransferService() {

// test the transferService

}

}

In this variation, we have split the XML configuration into three independent @Configurationclasses:

* TransferServiceConfig: acquires a dataSource via dependency injection using @Autowired
* StandaloneDataConfig: defines a dataSource for an embedded database suitable for developer tests
* JndiDataConfig: defines a dataSource that is retrieved from JNDI in a production environment

As with the XML-based configuration example, we still annotate TransferServiceTest with @ActiveProfiles(“dev”), but this time we specify the AnnotationConfigContextLoader and all three configuration classes via the @ContextConfiguration annotation. The body of the test class itself remains completely unchanged.

For details on how to simplify the above @Configuration classes consult the [Spring 3.1 M1: Introducing @Profile](https://bit.ly/mbOdHa) blog post.

**ApplicationContext Caching**

Since Spring 2.5 the *Spring TestContext Framework* has [cached ApplicationContexts for integration tests](https://bit.ly/jzG3Ie) based on a key that was generated from all merged context resource locations for a given test. Since the ContextLoader SPI only supported locations, this key generation algorithm was sufficient for uniquely identifying the configuration used to load an ApplicationContext. With the added support for configuration classes and profiles, however, the old algorithm is no longer adequate.

As a result, the context cache key generation algorithm has been updated in Spring 3.1 M2 to include the all of the following:

* locations *(from @ContextConfiguration)*
* classes *(from @ContextConfiguration)*
* contextLoader *(from @ContextConfiguration)*
* activeProfiles *(from @ActiveProfiles)*

What this means for you as a developer is that you can implement a base test class that declares a certain set of resource locations or configuration classes. Then, if you want to run tests against that base configuration but with different active profiles, you can extend that base test class and annotate each concrete subclass with @ActiveProfiles, supplying a different set of profiles to activate per subclass. Each of these subclasses would therefore define a unique set of configuration attributes that would result in different ApplicationContexts being loaded and cached.

**SmartContextLoader Supersedes ContextLoader SPI**

As hinted at earlier in this post, Spring 3.1 M2 introduces a new SmartContextLoader SPI that supersedes the existing ContextLoader SPI. If you plan to develop or already have developed your own custom ContextLoader, you will likely want to take a closer look at the new [SmartContextLoader](https://bit.ly/m4Yg73) interface. In contrast to the old ContextLoader interface, a SmartContextLoader can process both resource locations and configuration classes. Furthermore, a SmartContextLoader can set active bean definition profiles in the context that it loads.

ContextLoader will continue to be supported, and any existing implementations of that SPI should continue to work *as is*; however, if you want to support configuration classes or environment profiles in your custom loader, you will need to implement SmartContextLoader.

**DelegatingSmartContextLoader**

If you have been paying close attention to the examples presented thus far, you may have noticed that we always had to explicitly declare AnnotationConfigContextLoader.class for @ContextConfiguration’s loader attribute when using configuration classes. But when we specified XML configuration files (or relied on convention over configuration), the GenericXmlContextLoader was used by default.

*Wouldn’t it be nice if Spring could just notice whether we are using configuration classes or XML resource locations and then automatically pick the right ContextLoader to load our application context?*

Yeah, we think so, too! ;)

So for Spring 3.1 RC1 we plan to introduce a DelegatingSmartContextLoader that will delegate to a list of candidate SmartContextLoaders (i.e., GenericXmlContextLoader and AnnotationConfigContextLoader) to determine which context loader is appropriate for a given test class’s configuration. The winning candidate will then be used to actually load the context.

Once this work is complete, DelegatingSmartContextLoader will replace GenericXmlContextLoader as the default loader. Feel free to follow the progress of this development in JIRA: [SPR-8387](https://bit.ly/l5eKWS).

**Summary**

Spring 3.1 provides first-class testing support for @Configuration classes and environment profiles, and we encourage you to try out these features as soon as you can. M2 is the last milestone in the 3.1 release train. So if you find any bugs or have any suggestions for improvements, now is the time to [take action](https://jira.springsource.org/browse/SPR)!

<https://spring.io/blog/2011/06/21/spring-3-1-m2-testing-with-configuration-classes-and-profiles>