Testing

Version 5.1.0.RC3

Table of Contents

The adoption of the test-driven-development (TDD) approach to software development is certainly advocated by the Spring team, and so coverage of Spring's support for integration testing is covered (alongside best practices for unit testing). The Spring team has found that the correct use of IoC certainly does make both unit and integration testing easier (in that the presence of setter methods and appropriate constructors on classes makes them easier to wire together in a test without having to set up service locator registries and suchlike)… the chapter dedicated solely to testing will hopefully convince you of this as well.

Chapter 1. Introduction to Spring Testing

Testing is an integral part of enterprise software development. This chapter focuses on the value-add of the IoC principle to unit testing and on the benefits of the Spring Framework's support for integration testing. *(A thorough treatment of testing in the enterprise is beyond the scope of this reference manual.)*

Chapter 2. Unit Testing

Dependency Injection should make your code less dependent on the container than it would be with traditional Java EE development. The POJOs that make up your application should be testable in JUnit or TestNG tests, with objects simply instantiated using the `new` operator, *without Spring or any other container*. You can use mock objects (in conjunction with other valuable testing techniques) to test your code in isolation. If you follow the architecture recommendations for Spring, the resulting clean layering and componentization of your codebase will facilitate easier unit testing. For example, you can test service layer objects by stubbing or mocking DAO or Repository interfaces, without needing to access persistent data while running unit tests.

True unit tests typically run extremely quickly, as there is no runtime infrastructure to set up. Emphasizing true unit tests as part of your development methodology will boost your productivity. You may not need this section of the testing chapter to help you write effective unit tests for your IoC-based applications. For certain unit testing scenarios, however, the Spring Framework provides the following mock objects and testing support classes.

2.1. Mock Objects

2.1.1. Environment

The `org.springframework.mock.env` package contains mock implementations of the `Environment` and `PropertySource` abstractions (see Bean definition profiles and PropertySource abstraction). `MockEnvironment` and `MockPropertySource` are useful for developing *out-of-container* tests for code that depends on environment-specific properties.

2.1.2. JNDI

The `org.springframework.mock.jndi` package contains an implementation of the JNDI SPI, which you can use to set up a simple JNDI environment for test suites or stand-alone applications. If, for example, JDBC `DataSource`s get bound to the same JNDI names in test code as within a Java EE container, you can reuse both application code and configuration in testing scenarios without modification.

2.1.3. Servlet API

The `org.springframework.mock.web` package contains a comprehensive set of Servlet API mock objects that are useful for testing web contexts, controllers, and filters. These mock objects are targeted at usage with Spring's Web MVC framework and are generally more convenient to use than dynamic mock objects such as EasyMock or alternative Servlet API mock objects such as MockObjects.

Since Spring Framework 5.0, the mock objects in `org.springframework.mock.web` are based on the Servlet 4.0 API.

The Spring MVC Test framework builds on the mock Servlet API objects to provide an integration testing framework for Spring MVC. See *Spring MVC Test*.

2.1.4. Spring Web Reactive

The package `org.springframework.mock.http.server.reactive` contains mock implementations of `ServerHttpRequest` and `ServerHttpResponse` for use in WebFlux applications. The package `org.springframework.mock.web.server` contains a mock `ServerWebExchange` that depends on those mock request and response objects.

Both `MockServerHttpRequest` and `MockServerHttpResponse` extend from the same abstract base classes as server-specific implementations do and share behavior with them. For example a mock request is immutable once created but you can use the `mutate()` method from `ServerHttpRequest` to create a modified instance.

In order for the mock response to properly implement the write contract and return a write completion handle (i.e. `Mono<Void>`), by default it uses a `Flux` with `cache().then()`, which buffers the data and makes it available for assertions in tests. Applications can set a custom write function for example to test an infinite stream.

The WebTestClient builds on the mock request and response to provide support for testing WebFlux applications without an HTTP server. The client can also be used for end-to-end tests with a running server.

2.2. Unit Testing support Classes

2.2.1. General testing utilities

The `org.springframework.test.util` package contains several general purpose utilities for use in unit and integration testing.

`ReflectionTestUtils` is a collection of reflection-based utility methods. Developers use these methods in testing scenarios where they need to change the value of a constant, set a non-`public` field, invoke a non-`public` setter method, or invoke a non-`public` *configuration* or *lifecycle* callback method when testing application code involving use cases such as the following.

- ORM frameworks such as JPA and Hibernate that condone `private` or `protected` field access as opposed to `public` setter methods for properties in a domain entity.
- Spring's support for annotations such as `@Autowired`, `@Inject`, and `@Resource`, which provides dependency injection for `private` or `protected` fields, setter methods, and configuration methods.
- Use of annotations such as `@PostConstruct` and `@PreDestroy` for lifecycle callback methods.

`AopTestUtils` is a collection of AOP-related utility methods. These methods can be used to obtain a reference to the underlying target object hidden behind one or more Spring proxies. For example, if you have configured a bean as a dynamic mock using a library like EasyMock or Mockito and the mock is wrapped in a Spring proxy, you may need direct access to the underlying mock in order to configure expectations on it and perform verifications. For Spring's core AOP utilities, see `AopUtils` and `AopProxyUtils`.

2.2.2. Spring MVC

The `org.springframework.test.web` package contains `ModelAndViewAssert`, which you can use in combination with JUnit, TestNG, or any other testing framework for unit tests dealing with Spring MVC `ModelAndView` objects.

Unit testing Spring MVC Controllers

To unit test your Spring MVC `Controller`s as POJOs, use `ModelAndViewAssert` combined with `MockHttpServletRequest`, `MockHttpSession`, and so on from Spring's Servlet API mocks. For thorough integration testing of your Spring MVC and REST `Controller`s in conjunction with your `WebApplicationContext` configuration for Spring MVC, use the *Spring MVC Test Framework* instead.

Chapter 3. Integration Testing

3.1. Overview

It is important to be able to perform some integration testing without requiring deployment to your application server or connecting to other enterprise infrastructure. This will enable you to test things such as:

- The correct wiring of your Spring IoC container contexts.
- Data access using JDBC or an ORM tool. This would include such things as the correctness of SQL statements, Hibernate queries, JPA entity mappings, etc.

The Spring Framework provides first-class support for integration testing in the `spring-test` module. The name of the actual JAR file might include the release version and might also be in the long `org.springframework.test` form, depending on where you get it from (see the section on Dependency Management for an explanation). This library includes the `org.springframework.test` package, which contains valuable classes for integration testing with a Spring container. This testing does not rely on an application server or other deployment environment. Such tests are slower to run than unit tests but much faster than the equivalent Selenium tests or remote tests that rely on deployment to an application server.

In Spring 2.5 and later, unit and integration testing support is provided in the form of the annotation-driven Spring TestContext Framework. The TestContext framework is agnostic of the actual testing framework in use, thus allowing instrumentation of tests in various environments including JUnit, TestNG, and so on.

3.2. Goals of Integration Testing

Spring's integration testing support has the following primary goals:

- To manage Spring IoC container caching between test execution.
- To provide Dependency Injection of test fixture instances.
- To provide transaction management appropriate to integration testing.
- To supply Spring-specific base classes that assist developers in writing integration tests.

The next few sections describe each goal and provide links to implementation and configuration details.

3.2.1. Context management and caching

The Spring TestContext Framework provides consistent loading of Spring `ApplicationContext`s and `WebApplicationContext`s as well as caching of those contexts. Support for the caching of loaded contexts is important, because startup time can become an issue — not because of the overhead of Spring itself, but because the objects instantiated by the Spring container take time to instantiate. For example, a project with 50 to 100 Hibernate mapping files might take 10 to 20 seconds to load the mapping files, and incurring that cost before running every test in every test fixture leads to

slower overall test runs that reduce developer productivity.

Test classes typically declare either an array of *resource locations* for XML or Groovy configuration metadata — often in the classpath — or an array of *annotated classes* that is used to configure the application. These locations or classes are the same as or similar to those specified in `web.xml` or other configuration files for production deployments.

By default, once loaded, the configured `ApplicationContext` is reused for each test. Thus the setup cost is incurred only once per test suite, and subsequent test execution is much faster. In this context, the term *test suite* means all tests run in the same JVM — for example, all tests run from an Ant, Maven, or Gradle build for a given project or module. In the unlikely case that a test corrupts the application context and requires reloading — for example, by modifying a bean definition or the state of an application object — the TestContext framework can be configured to reload the configuration and rebuild the application context before executing the next test.

See Context management and Context caching with the TestContext framework.

3.2.2. Dependency Injection of test fixtures

When the TestContext framework loads your application context, it can optionally configure instances of your test classes via Dependency Injection. This provides a convenient mechanism for setting up test fixtures using preconfigured beans from your application context. A strong benefit here is that you can reuse application contexts across various testing scenarios (e.g., for configuring Spring-managed object graphs, transactional proxies, `DataSource`s, etc.), thus avoiding the need to duplicate complex test fixture setup for individual test cases.

As an example, consider the scenario where we have a class, `HibernateTitleRepository`, that implements data access logic for a `Title` domain entity. We want to write integration tests that test the following areas:

- The Spring configuration: basically, is everything related to the configuration of the `HibernateTitleRepository` bean correct and present?
- The Hibernate mapping file configuration: is everything mapped correctly, and are the correct lazy-loading settings in place?
- The logic of the `HibernateTitleRepository`: does the configured instance of this class perform as anticipated?

See dependency injection of test fixtures with the TestContext framework.

3.2.3. Transaction management

One common issue in tests that access a real database is their effect on the state of the persistence store. Even when you're using a development database, changes to the state may affect future tests. Also, many operations — such as inserting or modifying persistent data — cannot be performed (or verified) outside a transaction.

The TestContext framework addresses this issue. By default, the framework will create and roll back a transaction for each test. You simply write code that can assume the existence of a transaction. If you call transactionally proxied objects in your tests, they will behave correctly,

according to their configured transactional semantics. In addition, if a test method deletes the contents of selected tables while running within the transaction managed for the test, the transaction will roll back by default, and the database will return to its state prior to execution of the test. Transactional support is provided to a test via a `PlatformTransactionManager` bean defined in the test's application context.

If you want a transaction to commit — unusual, but occasionally useful when you want a particular test to populate or modify the database — the TestContext framework can be instructed to cause the transaction to commit instead of roll back via the `@Commit` annotation.

See transaction management with the TestContext framework.

3.2.4. Support classes for integration testing

The Spring TestContext Framework provides several `abstract` support classes that simplify the writing of integration tests. These base test classes provide well-defined hooks into the testing framework as well as convenient instance variables and methods, which enable you to access:

- The `ApplicationContext`, for performing explicit bean lookups or testing the state of the context as a whole.
- A `JdbcTemplate`, for executing SQL statements to query the database. Such queries can be used to confirm database state both *prior to* and *after* execution of database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid false positives.

In addition, you may want to create your own custom, application-wide superclass with instance variables and methods specific to your project.

See support classes for the TestContext framework.

3.3. JDBC Testing Support

The `org.springframework.test.jdbc` package contains `JdbcTestUtils`, which is a collection of JDBC related utility functions intended to simplify standard database testing scenarios. Specifically, `JdbcTestUtils` provides the following static utility methods.

- `countRowsInTable(..)`: counts the number of rows in the given table
- `countRowsInTableWhere(..)`: counts the number of rows in the given table, using the provided `WHERE` clause
- `deleteFromTables(..)`: deletes all rows from the specified tables
- `deleteFromTableWhere(..)`: deletes rows from the given table, using the provided `WHERE` clause
- `dropTables(..)`: drops the specified tables

Note that `AbstractTransactionalJUnit4SpringContextTests` *and* `AbstractTransactionalTestNGSpringContextTests` *provide convenience methods which delegate to the aforementioned methods in* `JdbcTestUtils`.

The `spring-jdbc` module provides support for configuring and launching an embedded database which can be used in integration tests that interact with a database. For details, see Embedded database support and Testing data access logic with an embedded database.

3.4. Annotations

3.4.1. Spring Testing Annotations

The Spring Framework provides the following set of *Spring-specific* annotations that you can use in your unit and integration tests in conjunction with the TestContext framework. Refer to the corresponding javadocs for further information, including default attribute values, attribute aliases, and so on.

@BootstrapWith

`@BootstrapWith` is a class-level annotation that is used to configure how the *Spring TestContext Framework* is bootstrapped. Specifically, `@BootstrapWith` is used to specify a custom `TestContextBootstrapper`. Consult the Bootstrapping the TestContext framework section for further details.

@ContextConfiguration

`@ContextConfiguration` defines class-level metadata that is used to determine how to load and configure an `ApplicationContext` for integration tests. Specifically, `@ContextConfiguration` declares the application context resource `locations` or the annotated `classes` that will be used to load the context.

Resource locations are typically XML configuration files or Groovy scripts located in the classpath; whereas, annotated classes are typically `@Configuration` classes. However, resource locations can also refer to files and scripts in the file system, and annotated classes can be component classes, etc.

```
<strong>@ContextConfiguration</strong>("/test-config.xml")
public class XmlApplicationContextTests {
    // class body...
}
```

```
<strong>@ContextConfiguration</strong>(<strong>classes</strong> = TestConfig.class)
public class ConfigClassApplicationContextTests {
    // class body...
}
```

As an alternative or in addition to declaring resource locations or annotated classes, `@ContextConfiguration` may be used to declare `ApplicationContextInitializer` classes.

```
<strong>@ContextConfiguration</strong>(<strong>initializers</strong> =
CustomContextIntializer.class)
public class ContextInitializerTests {
    // class body...
}
```

`@ContextConfiguration` may optionally be used to declare the `ContextLoader` strategy as well. Note, however, that you typically do not need to explicitly configure the loader since the default loader supports either resource `locations` or annotated `classes` as well as `initializers`.

```
<strong>@ContextConfiguration</strong>(<strong>locations</strong> = "/test-
context.xml", <strong>loader</strong> = CustomContextLoader.class)
public class CustomLoaderXmlApplicationContextTests {
    // class body...
}
```

`@ContextConfiguration` provides support for *inheriting* resource locations or configuration classes as well as context initializers declared by superclasses by default.

See Context management and the `@ContextConfiguration` javadocs for further details.

@WebAppConfiguration

`@WebAppConfiguration` is a class-level annotation that is used to declare that the `ApplicationContext` loaded for an integration test should be a `WebApplicationContext`. The mere presence of `@WebAppConfiguration` on a test class ensures that a `WebApplicationContext` will be loaded for the test, using the default value of `"file:src/main/webapp"` for the path to the root of the web application (i.e., the *resource base path*). The resource base path is used behind the scenes to create a `MockServletContext` which serves as the `ServletContext` for the test's `WebApplicationContext`.

```
@ContextConfiguration
<strong>@WebAppConfiguration</strong>
public class WebAppTests {
    // class body...
}
```

To override the default, specify a different base resource path via the *implicit* `value` attribute. Both `classpath:` and `file:` resource prefixes are supported. If no resource prefix is supplied the path is assumed to be a file system resource.

```
@ContextConfiguration
<strong>@WebAppConfiguration("classpath:test-web-resources")</strong>
public class WebAppTests {
    // class body...
}
```

Note that `@WebAppConfiguration` must be used in conjunction with `@ContextConfiguration`, either within a single test class or within a test class hierarchy. See the `@WebAppConfiguration` javadocs for further details.

@ContextHierarchy

`@ContextHierarchy` is a class-level annotation that is used to define a hierarchy of `ApplicationContext`s for integration tests. `@ContextHierarchy` should be declared with a list of one or more `@ContextConfiguration` instances, each of which defines a level in the context hierarchy. The following examples demonstrate the use of `@ContextHierarchy` within a single test class; however, `@ContextHierarchy` can also be used within a test class hierarchy.

```
@ContextHierarchy({
    @ContextConfiguration("/parent-config.xml"),
    @ContextConfiguration("/child-config.xml")
})
public class ContextHierarchyTests {
    // class body...
}
```

```
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = AppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
public class WebIntegrationTests {
    // class body...
}
```

If you need to merge or override the configuration for a given level of the context hierarchy within a test class hierarchy, you must explicitly name that level by supplying the same value to the `name` attribute in `@ContextConfiguration` at each corresponding level in the class hierarchy. See Context hierarchies and the `@ContextHierarchy` javadocs for further examples.

@ActiveProfiles

`@ActiveProfiles` is a class-level annotation that is used to declare which *bean definition profiles* should be active when loading an `ApplicationContext` for an integration test.

```
@ContextConfiguration
<strong>@ActiveProfiles</strong>("dev")
public class DeveloperTests {
    // class body...
}
```

```
@ContextConfiguration
<strong>@ActiveProfiles</strong>({"dev", "integration"})
public class DeveloperIntegrationTests {
    // class body...
}
```

`@ActiveProfiles` provides support for *inheriting* active bean definition profiles declared by superclasses by default. It is also possible to resolve active bean definition profiles programmatically by implementing a custom `ActiveProfilesResolver` and registering it via the `resolver` attribute of `@ActiveProfiles`.

See Context configuration with environment profiles and the `@ActiveProfiles` javadocs for examples and further details.

@TestPropertySource

`@TestPropertySource` is a class-level annotation that is used to configure the locations of properties files and inlined properties to be added to the set of `PropertySources` in the `Environment` for an `ApplicationContext` loaded for an integration test.

Test property sources have higher precedence than those loaded from the operating system's environment or Java system properties as well as property sources added by the application declaratively via `@PropertySource` or programmatically. Thus, test property sources can be used to selectively override properties defined in system and application property sources. Furthermore, inlined properties have higher precedence than properties loaded from resource locations.

The following example demonstrates how to declare a properties file from the classpath.

```
@ContextConfiguration
<strong>@TestPropertySource</strong>("/test.properties")
public class MyIntegrationTests {
    // class body...
}
```

The following example demonstrates how to declare *inlined* properties.

```
@ContextConfiguration
<strong>@TestPropertySource</strong>(properties = { "timezone = GMT", "port: 4242" })
public class MyIntegrationTests {
    // class body...
}
```

@DirtiesContext

`@DirtiesContext` indicates that the underlying Spring `ApplicationContext` has been *dirtied* during the execution of a test (i.e., modified or corrupted in some manner — for example, by changing the state of a singleton bean) and should be closed. When an application context is marked *dirty*, it is removed from the testing framework's cache and closed. As a consequence, the underlying Spring container will be rebuilt for any subsequent test that requires a context with the same configuration metadata.

`@DirtiesContext` can be used as both a class-level and method-level annotation within the same class or class hierarchy. In such scenarios, the `ApplicationContext` is marked as *dirty* before or after any such annotated method as well as before or after the current test class, depending on the configured `methodMode` and `classMode`.

The following examples explain when the context would be dirtied for various configuration scenarios:

- Before the current test class, when declared on a class with class mode set to `BEFORE_CLASS`.

  ```
  <strong>@DirtiesContext(classMode = BEFORE_CLASS)</strong>
  public class FreshContextTests {
      // some tests that require a new Spring container
  }
  ```

- After the current test class, when declared on a class with class mode set to `AFTER_CLASS` (i.e., the default class mode).

  ```
  <strong>@DirtiesContext</strong>
  public class ContextDirtyingTests {
      // some tests that result in the Spring container being dirtied
  }
  ```

- Before each test method in the current test class, when declared on a class with class mode set to `BEFORE_EACH_TEST_METHOD.`

  ```
  <strong>@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD)</strong>
  public class FreshContextTests {
      // some tests that require a new Spring container
  }
  ```

- After each test method in the current test class, when declared on a class with class mode set to `AFTER_EACH_TEST_METHOD`.

```
<strong>@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD)</strong>
public class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

- Before the current test, when declared on a method with the method mode set to `BEFORE_METHOD`.

```
<strong>@DirtiesContext(methodMode = BEFORE_METHOD)</strong>
@Test
public void testProcessWhichRequiresFreshAppCtx() {
    // some logic that requires a new Spring container
}
```

- After the current test, when declared on a method with the method mode set to `AFTER_METHOD` (i.e., the default method mode).

```
<strong>@DirtiesContext</strong>
@Test
public void testProcessWhichDirtiesAppCtx() {
    // some logic that results in the Spring container being dirtied
}
```

If `@DirtiesContext` is used in a test whose context is configured as part of a context hierarchy via `@ContextHierarchy`, the `hierarchyMode` flag can be used to control how the context cache is cleared. By default an *exhaustive* algorithm will be used that clears the context cache including not only the current level but also all other context hierarchies that share an ancestor context common to the current test; all `ApplicationContext`s that reside in a sub-hierarchy of the common ancestor context will be removed from the context cache and closed. If the *exhaustive* algorithm is overkill for a particular use case, the simpler *current level* algorithm can be specified instead, as seen below.

```
@ContextHierarchy({
    @ContextConfiguration("/parent-config.xml"),
    @ContextConfiguration("/child-config.xml")
})
public class BaseTests {
    // class body...
}

public class ExtendedTests extends BaseTests {

    @Test
    @DirtiesContext(<strong>hierarchyMode = CURRENT_LEVEL</strong>)
    public void test() {
        // some logic that results in the child context being dirtied
    }
}
```

For further details regarding the `EXHAUSTIVE` and `CURRENT_LEVEL` algorithms see the `DirtiesContext.HierarchyMode` javadocs.

@TestExecutionListeners

`@TestExecutionListeners` defines class-level metadata for configuring the `TestExecutionListener` implementations that should be registered with the `TestContextManager`. Typically, `@TestExecutionListeners` is used in conjunction with `@ContextConfiguration`.

```
@ContextConfiguration
<strong>@TestExecutionListeners</strong>({CustomTestExecutionListener.class,
AnotherTestExecutionListener.class})
public class CustomTestExecutionListenerTests {
    // class body...
}
```

`@TestExecutionListeners` supports *inherited* listeners by default. See the javadocs for an example and further details.

@Commit

`@Commit` indicates that the transaction for a transactional test method should be *committed* after the test method has completed. `@Commit` can be used as a direct replacement for `@Rollback(false)` in order to more explicitly convey the intent of the code. Analogous to `@Rollback`, `@Commit` may also be declared as a class-level or method-level annotation.

```
<strong>@Commit</strong>
@Test
public void testProcessWithoutRollback() {
    // ...
}
```

@Rollback

`@Rollback` indicates whether the transaction for a transactional test method should be *rolled back* after the test method has completed. If `true`, the transaction is rolled back; otherwise, the transaction is committed (see also `@Commit`). Rollback semantics for integration tests in the Spring TestContext Framework default to `true` even if `@Rollback` is not explicitly declared.

When declared as a class-level annotation, `@Rollback` defines the default rollback semantics for all test methods within the test class hierarchy. When declared as a method-level annotation, `@Rollback` defines rollback semantics for the specific test method, potentially overriding class-level `@Rollback` or `@Commit` semantics.

```
<strong>@Rollback</strong>(false)
@Test
public void testProcessWithoutRollback() {
    // ...
}
```

@BeforeTransaction

`@BeforeTransaction` indicates that the annotated `void` method should be executed *before* a transaction is started for test methods configured to run within a transaction via Spring's `@Transactional` annotation. As of Spring Framework 4.3, `@BeforeTransaction` methods are not required to be `public` and may be declared on Java 8 based interface default methods.

```
<strong>@BeforeTransaction</strong>
void beforeTransaction() {
    // logic to be executed before a transaction is started
}
```

@AfterTransaction

`@AfterTransaction` indicates that the annotated `void` method should be executed *after* a transaction is ended for test methods configured to run within a transaction via Spring's `@Transactional` annotation. As of Spring Framework 4.3, `@AfterTransaction` methods are not required to be `public` and may be declared on Java 8 based interface default methods.

```
<strong>@AfterTransaction</strong>
void afterTransaction() {
    // logic to be executed after a transaction has ended
}
```

@Sql

`@Sql` is used to annotate a test class or test method to configure SQL scripts to be executed against a given database during integration tests.

```
@Test
<strong>@Sql</strong>({"/test-schema.sql", "/test-user-data.sql"})
public void userTest {
    // execute code that relies on the test schema and test data
}
```

See Executing SQL scripts declaratively with @Sql for further details.

@SqlConfig

`@SqlConfig` defines metadata that is used to determine how to parse and execute SQL scripts configured via the `@Sql` annotation.

```
@Test
@Sql(
    scripts = "/test-user-data.sql",
    config = <strong>@SqlConfig</strong>(commentPrefix = "`", separator = "@@")
)
public void userTest {
    // execute code that relies on the test data
}
```

@SqlGroup

`@SqlGroup` is a container annotation that aggregates several `@Sql` annotations. `@SqlGroup` can be used natively, declaring several nested `@Sql` annotations, or it can be used in conjunction with Java 8's support for repeatable annotations, where `@Sql` can simply be declared several times on the same class or method, implicitly generating this container annotation.

```
@Test
<strong>@SqlGroup</strong>({
    @Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`")),
    @Sql("/test-user-data.sql")
)}
public void userTest {
    // execute code that uses the test schema and test data
}
```

3.4.2. Standard Annotation Support

The following annotations are supported with standard semantics for all configurations of the Spring TestContext Framework. Note that these annotations are not specific to tests and can be used anywhere in the Spring Framework.

- `@Autowired`
- `@Qualifier`
- `@Resource` (javax.annotation) *if JSR-250 is present*
- `@ManagedBean` (javax.annotation) *if JSR-250 is present*
- `@Inject` (javax.inject) *if JSR-330 is present*
- `@Named` (javax.inject) *if JSR-330 is present*
- `@PersistenceContext` (javax.persistence) *if JPA is present*
- `@PersistenceUnit` (javax.persistence) *if JPA is present*
- `@Required`
- `@Transactional`

> *JSR-250 Lifecycle Annotations*
>
> In the Spring TestContext Framework `@PostConstruct` and `@PreDestroy` may be used with standard semantics on any application components configured in the `ApplicationContext`; however, these lifecycle annotations have limited usage within an actual test class.
>
> If a method within a test class is annotated with `@PostConstruct`, that method will be executed before any *before* methods of the underlying test framework (e.g., methods annotated with JUnit Jupiter's `@BeforeEach`), and that will apply for every test method in the test class. On the other hand, if a method within a test class is annotated with `@PreDestroy`, that method will *never* be executed. Within a test class it is therefore recommended to use test lifecycle callbacks from the underlying test framework instead of `@PostConstruct` and `@PreDestroy`.

3.4.3. Spring JUnit 4 Testing Annotations

The following annotations are *only* supported when used in conjunction with the SpringRunner, Spring's JUnit 4 rules, or Spring's JUnit 4 support classes.

@IfProfileValue

`@IfProfileValue` indicates that the annotated test is enabled for a specific testing environment. If the configured `ProfileValueSource` returns a matching `value` for the provided `name`, the test is enabled. Otherwise, the test will be disabled and effectively *ignored*.

`@IfProfileValue` can be applied at the class level, the method level, or both. Class-level usage of `@IfProfileValue` takes precedence over method-level usage for any methods within that class or its subclasses. Specifically, a test is enabled if it is enabled both at the class level *and* at the method level; the absence of `@IfProfileValue` means the test is implicitly enabled. This is analogous to the semantics of JUnit 4's `@Ignore` annotation, except that the presence of `@Ignore` always disables a test.

```
<strong>@IfProfileValue</strong>(<strong>name</strong>="java.vendor", <strong>value<
/strong>="Oracle Corporation")
@Test
public void testProcessWhichRunsOnlyOnOracleJvm() {
    // some logic that should run only on Java VMs from Oracle Corporation
}
```

Alternatively, you can configure `@IfProfileValue` with a list of `values` (with *OR* semantics) to achieve TestNG-like support for *test groups* in a JUnit 4 environment. Consider the following example:

```
<strong>@IfProfileValue</strong>(<strong>name</strong>="test-groups", <strong>values<
/strong>={"unit-tests", "integration-tests"})
@Test
public void testProcessWhichRunsForUnitOrIntegrationTestGroups() {
    // some logic that should run only for unit and integration test groups
}
```

@ProfileValueSourceConfiguration

`@ProfileValueSourceConfiguration` is a class-level annotation that specifies what type of `ProfileValueSource` to use when retrieving *profile values* configured through the `@IfProfileValue` annotation. If `@ProfileValueSourceConfiguration` is not declared for a test, `SystemProfileValueSource` is used by default.

```
<strong>@ProfileValueSourceConfiguration</strong>(CustomProfileValueSource.class)
public class CustomProfileValueSourceTests {
    // class body...
}
```

@Timed

`@Timed` indicates that the annotated test method must finish execution in a specified time period (in milliseconds). If the text execution time exceeds the specified time period, the test fails.

The time period includes execution of the test method itself, any repetitions of the test (see `@Repeat`),

as well as any *set up* or *tear down* of the test fixture.

```
<strong>@Timed</strong>(millis=1000)
public void testProcessWithOneSecondTimeout() {
    // some logic that should not take longer than 1 second to execute
}
```

Spring's `@Timed` annotation has different semantics than JUnit 4's `@Test(timeout=…)` support. Specifically, due to the manner in which JUnit 4 handles test execution timeouts (that is, by executing the test method in a separate `Thread`), `@Test(timeout=…)` preemptively fails the test if the test takes too long. Spring's `@Timed`, on the other hand, does not preemptively fail the test but rather waits for the test to complete before failing.

@Repeat

`@Repeat` indicates that the annotated test method must be executed repeatedly. The number of times that the test method is to be executed is specified in the annotation.

The scope of execution to be repeated includes execution of the test method itself as well as any *set up* or *tear down* of the test fixture.

```
<strong>@Repeat</strong>(10)
@Test
public void testProcessRepeatedly() {
    // ...
}
```

3.4.4. Spring JUnit Jupiter Testing Annotations

The following annotations are *only* supported when used in conjunction with the `SpringExtension` and JUnit Jupiter (i.e., the programming model in JUnit 5).

@SpringJUnitConfig

`@SpringJUnitConfig` is a *composed annotation* that combines `@ExtendWith(SpringExtension.class)` from JUnit Jupiter with `@ContextConfiguration` from the Spring TestContext Framework. It can be used at the class level as a drop-in replacement for `@ContextConfiguration`. With regard to configuration options, the only difference between `@ContextConfiguration` and `@SpringJUnitConfig` is that annotated classes may be declared via the `value` attribute in `@SpringJUnitConfig`.

```
<strong>@SpringJUnitConfig</strong>(TestConfig.class)
class ConfigurationClassJUnitJupiterSpringTests {
    // class body...
}
```

```
<strong>@SpringJUnitConfig</strong>(<strong>locations</strong> = "/test-config.xml")
class XmlJUnitJupiterSpringTests {
    // class body...
}
```

See Context management as well as the javadocs for `@SpringJUnitConfig` and `@ContextConfiguration` for further details.

@SpringJUnitWebConfig

`@SpringJUnitWebConfig` is a *composed annotation* that combines `@ExtendWith(SpringExtension.class)` from JUnit Jupiter with `@ContextConfiguration` and `@WebAppConfiguration` from the Spring TestContext Framework. It can be used at the class level as a drop-in replacement for `@ContextConfiguration` and `@WebAppConfiguration`. With regard to configuration options, the only difference between `@ContextConfiguration` and `@SpringJUnitWebConfig` is that annotated classes may be declared via the `value` attribute in `@SpringJUnitWebConfig`. In addition, the `value` attribute from `@WebAppConfiguration` can only be overridden via the `resourcePath` attribute in `@SpringJUnitWebConfig`.

```
<strong>@SpringJUnitWebConfig</strong>(TestConfig.class)
class ConfigurationClassJUnitJupiterSpringWebTests {
    // class body...
}
```

```
<strong>@SpringJUnitWebConfig</strong>(<strong>locations</strong> = "/test-config.xml
")
class XmlJUnitJupiterSpringWebTests {
    // class body...
}
```

See Context management as well as the javadocs for `@SpringJUnitWebConfig`, `@ContextConfiguration`, and `@WebAppConfiguration` for further details.

@EnabledIf

`@EnabledIf` is used to signal that the annotated JUnit Jupiter test class or test method is *enabled* and should be executed if the supplied `expression` evaluates to `true`. Specifically, if the expression evaluates to `Boolean.TRUE` or a `String` equal to `"true"` (ignoring case), the test will be *enabled*. When applied at the class level, all test methods within that class are automatically enabled by default as well.

Expressions can be any of the following.

- Spring Expression Language (SpEL) expression – for example:
  - `@EnabledIf("#{systemProperties['os.name'].toLowerCase().contains('mac')}")`
- Placeholder for a property available in the Spring `Environment` – for example:

    - `@EnabledIf("${smoke.tests.enabled}")`
- Text literal – for example:
    - `@EnabledIf("true")`

Note, however, that a text literal which is *not* the result of dynamic resolution of a property placeholder is of zero practical value since `@EnabledIf("false")` is equivalent to `@Disabled` and `@EnabledIf("true")` is logically meaningless.

`@EnabledIf` may be used as a meta-annotation to create custom composed annotations. For example, a custom `@EnabledOnMac` annotation can be created as follows.

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@EnabledIf(
    expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
    reason = "Enabled on Mac OS"
)
public @interface EnabledOnMac {}
```

@DisabledIf

`@DisabledIf` is used to signal that the annotated JUnit Jupiter test class or test method is *disabled* and should not be executed if the supplied `expression` evaluates to `true`. Specifically, if the expression evaluates to `Boolean.TRUE` or a `String` equal to `"true"` (ignoring case), the test will be *disabled*. When applied at the class level, all test methods within that class are automatically disabled as well.

Expressions can be any of the following.

- Spring Expression Language (SpEL) expression – for example:
    - `@DisabledIf("#{systemProperties['os.name'].toLowerCase().contains('mac')}")`
- Placeholder for a property available in the Spring `Environment` – for example:
    - `@DisabledIf("${smoke.tests.disabled}")`
- Text literal – for example:
    - `@DisabledIf("true")`

Note, however, that a text literal which is *not* the result of dynamic resolution of a property placeholder is of zero practical value since `@DisabledIf("true")` is equivalent to `@Disabled` and `@DisabledIf("false")` is logically meaningless.

`@DisabledIf` may be used as a meta-annotation to create custom composed annotations. For example, a custom `@DisabledOnMac` annotation can be created as follows.

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@DisabledIf(
    expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
    reason = "Disabled on Mac OS"
)
public @interface DisabledOnMac {}
```

3.4.5. Meta-Annotation Support for Testing

It is possible to use most test-related annotations as meta-annotations in order to create custom *composed annotations* and reduce configuration duplication across a test suite.

Each of the following may be used as meta-annotations in conjunction with the TestContext framework.

- `@BootstrapWith`
- `@ContextConfiguration`
- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DirtiesContext`
- `@WebAppConfiguration`
- `@TestExecutionListeners`
- `@Transactional`
- `@BeforeTransaction`
- `@AfterTransaction`
- `@Commit`
- `@Rollback`
- `@Sql`
- `@SqlConfig`
- `@SqlGroup`
- `@Repeat` *(only supported on JUnit 4)*
- `@Timed` *(only supported on JUnit 4)*
- `@IfProfileValue` *(only supported on JUnit 4)*
- `@ProfileValueSourceConfiguration` *(only supported on JUnit 4)*
- `@SpringJUnitConfig` *(only supported on JUnit Jupiter)*
- `@SpringJUnitWebConfig` *(only supported on JUnit Jupiter)*
- `@EnabledIf` *(only supported on JUnit Jupiter)*
- `@DisabledIf` *(only supported on JUnit Jupiter)*

For example, if we discover that we are repeating the following configuration across our *JUnit 4* based test suite…

```
@RunWith(SpringRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public class UserRepositoryTests { }
```

We can reduce the above duplication by introducing a custom *composed annotation* that centralizes the common test configuration for Spring like this:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTestConfig { }
```

Then we can use our custom `@TransactionalDevTestConfig` annotation to simplify the configuration of individual JUnit 4 based test classes as follows:

```
@RunWith(SpringRunner.class)
@TransactionalDevTestConfig
public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)
@TransactionalDevTestConfig
public class UserRepositoryTests { }
```

If we are writing tests using JUnit Jupiter, we can reduce code duplication even further since annotations in JUnit 5 can also be used as meta-annotations. For example, if we discover that we are repeating the following configuration across our JUnit Jupiter based test suite…

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
class OrderRepositoryTests { }

@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
class UserRepositoryTests { }
```

We can reduce the above duplication by introducing a custom *composed annotation* that centralizes the common test configuration for Spring and JUnit Jupiter like this:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTestConfig { }
```

Then we can use our custom `@TransactionalDevTestConfig` annotation to simplify the configuration of individual JUnit Jupiter based test classes as follows:

```
@TransactionalDevTestConfig
class OrderRepositoryTests { }

@TransactionalDevTestConfig
class UserRepositoryTests { }
```

Since JUnit Jupiter supports the use of `@Test`, `@RepeatedTest`, `ParameterizedTest`, etc. as meta-annotations, it is also possible to create custom composed annotations at the test method level. For example, if we wish to create a *composed annotation* that combines the `@Test` and `@Tag` annotations from JUnit Jupiter with the `@Transactional` annotation from Spring, we could create an `@TransactionalIntegrationTest` annotation as follows.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Transactional
@Tag("integration-test") // org.junit.jupiter.api.Tag
@Test // org.junit.jupiter.api.Test
public @interface TransactionalIntegrationTest { }
```

Then we can use our custom `@TransactionalIntegrationTest` annotation to simplify the

configuration of individual JUnit Jupiter based test methods as follows:

```
@TransactionalIntegrationTest
void saveOrder() { }

@TransactionalIntegrationTest
void deleteOrder() { }
```

For further details, consult the Spring Annotation Programming Model.

3.5. Spring TestContext Framework

The *Spring TestContext Framework* (located in the `org.springframework.test.context` package) provides generic, annotation-driven unit and integration testing support that is agnostic of the testing framework in use. The TestContext framework also places a great deal of importance on *convention over configuration* with reasonable defaults that can be overridden through annotation-based configuration.

In addition to generic testing infrastructure, the TestContext framework provides explicit support for JUnit 4, JUnit Jupiter (a.k.a., JUnit 5), and TestNG. For JUnit 4 and TestNG, Spring provides `abstract` support classes. Furthermore, Spring provides a custom JUnit `Runner` and custom JUnit `Rules` for *JUnit 4* as well as a custom `Extension` for *JUnit Jupiter* that allow one to write so-called *POJO test classes*. POJO test classes are not required to extend a particular class hierarchy such as the `abstract` support classes.

The following section provides an overview of the internals of the TestContext framework. If you are only interested in *using* the framework and not necessarily interested in *extending* it with your own custom listeners or custom loaders, feel free to go directly to the configuration (context management, dependency injection, transaction management), support classes, and annotation support sections.

3.5.1. Key abstractions

The core of the framework consists of the `TestContextManager` class and the `TestContext`, `TestExecutionListener`, and `SmartContextLoader` interfaces. A `TestContextManager` is created per test class (e.g., for the execution of all test methods within a single test class in JUnit Jupiter). The `TestContextManager` in turn manages a `TestContext` that holds the context of the current test. The `TestContextManager` also updates the state of the `TestContext` as the test progresses and delegates to `TestExecutionListener` implementations, which instrument the actual test execution by providing dependency injection, managing transactions, and so on. A `SmartContextLoader` is responsible for loading an `ApplicationContext` for a given test class. Consult the javadocs and the Spring test suite for further information and examples of various implementations.

TestContext

`TestContext` encapsulates the context in which a test is executed, agnostic of the actual testing framework in use, and provides context management and caching support for the test instance for which it is responsible. The `TestContext` also delegates to a `SmartContextLoader` to load an

`ApplicationContext` if requested.

TestContextManager

`TestContextManager` is the main entry point into the *Spring TestContext Framework* and is responsible for managing a single `TestContext` and signaling events to each registered `TestExecutionListener` at well-defined test execution points:

- prior to any *before class* or *before all* methods of a particular testing framework
- test instance post-processing
- prior to any *before* or *before each* methods of a particular testing framework
- immediately before execution of the test method but after test setup
- immediately after execution of the test method but before test tear down
- after any *after* or *after each* methods of a particular testing framework
- after any *after class* or *after all* methods of a particular testing framework

TestExecutionListener

`TestExecutionListener` defines the API for reacting to test execution events published by the `TestContextManager` with which the listener is registered. See TestExecutionListener configuration.

Context Loaders

`ContextLoader` is a strategy interface that was introduced in Spring 2.5 for loading an `ApplicationContext` for an integration test managed by the Spring TestContext Framework. Implement `SmartContextLoader` instead of this interface in order to provide support for annotated classes, active bean definition profiles, test property sources, context hierarchies, and `WebApplicationContext` support.

`SmartContextLoader` is an extension of the `ContextLoader` interface introduced in Spring 3.1. The `SmartContextLoader` SPI supersedes the `ContextLoader` SPI that was introduced in Spring 2.5. Specifically, a `SmartContextLoader` can choose to process resource `locations`, annotated `classes`, or context `initializers`. Furthermore, a `SmartContextLoader` can set active bean definition profiles and test property sources in the context that it loads.

Spring provides the following implementations:

- `DelegatingSmartContextLoader`: one of two default loaders which delegates internally to an `AnnotationConfigContextLoader`, a `GenericXmlContextLoader`, or a `GenericGroovyXmlContextLoader` depending either on the configuration declared for the test class or on the presence of default locations or default configuration classes. Groovy support is only enabled if Groovy is on the classpath.
- `WebDelegatingSmartContextLoader`: one of two default loaders which delegates internally to an `AnnotationConfigWebContextLoader`, a `GenericXmlWebContextLoader`, or a `GenericGroovyXmlWebContextLoader` depending either on the configuration declared for the test class or on the presence of default locations or default configuration classes. A web `ContextLoader` will only be used if `@WebAppConfiguration` is present on the test class. Groovy

support is only enabled if Groovy is on the classpath.

- `AnnotationConfigContextLoader`: loads a standard `ApplicationContext` from *annotated classes*.
- `AnnotationConfigWebContextLoader`: loads a `WebApplicationContext` from *annotated classes*.
- `GenericGroovyXmlContextLoader`: loads a standard `ApplicationContext` from *resource locations* that are either Groovy scripts or XML configuration files.
- `GenericGroovyXmlWebContextLoader`: loads a `WebApplicationContext` from *resource locations* that are either Groovy scripts or XML configuration files.
- `GenericXmlContextLoader`: loads a standard `ApplicationContext` from XML *resource locations*.
- `GenericXmlWebContextLoader`: loads a `WebApplicationContext` from XML *resource locations*.
- `GenericPropertiesContextLoader`: loads a standard `ApplicationContext` from Java Properties files.

3.5.2. Bootstrapping the TestContext framework

The default configuration for the internals of the Spring TestContext Framework is sufficient for all common use cases. However, there are times when a development team or third party framework would like to change the default `ContextLoader`, implement a custom `TestContext` or `ContextCache`, augment the default sets of `ContextCustomizerFactory` and `TestExecutionListener` implementations, etc. For such low level control over how the TestContext framework operates, Spring provides a bootstrapping strategy.

`TestContextBootstrapper` defines the SPI for *bootstrapping* the TestContext framework. A `TestContextBootstrapper` is used by the `TestContextManager` to load the `TestExecutionListener` implementations for the current test and to build the `TestContext` that it manages. A custom bootstrapping strategy can be configured for a test class (or test class hierarchy) via `@BootstrapWith`, either directly or as a meta-annotation. If a bootstrapper is not explicitly configured via `@BootstrapWith`, either the `DefaultTestContextBootstrapper` or the `WebTestContextBootstrapper` will be used, depending on the presence of `@WebAppConfiguration`.

Since the `TestContextBootstrapper` SPI is likely to change in the future in order to accommodate new requirements, implementers are strongly encouraged not to implement this interface directly but rather to extend `AbstractTestContextBootstrapper` or one of its concrete subclasses instead.

3.5.3. TestExecutionListener configuration

Spring provides the following `TestExecutionListener` implementations that are registered by default, exactly in this order.

- `ServletTestExecutionListener`: configures Servlet API mocks for a `WebApplicationContext`
- `DirtiesContextBeforeModesTestExecutionListener`: handles the `@DirtiesContext` annotation for *before* modes
- `DependencyInjectionTestExecutionListener`: provides dependency injection for the test instance
- `DirtiesContextTestExecutionListener`: handles the `@DirtiesContext` annotation for *after* modes
- `TransactionalTestExecutionListener`: provides transactional test execution with default rollback semantics

- `SqlScriptsTestExecutionListener`: executes SQL scripts configured via the `@Sql` annotation

Registering custom TestExecutionListeners

Custom `TestExecutionListener`s can be registered for a test class and its subclasses via the `@TestExecutionListeners` annotation. See annotation support and the javadocs for `@TestExecutionListeners` for details and examples.

Automatic discovery of default TestExecutionListeners

Registering custom `TestExecutionListener`s via `@TestExecutionListeners` is suitable for custom listeners that are used in limited testing scenarios; however, it can become cumbersome if a custom listener needs to be used across a test suite. Since Spring Framework 4.1, this issue is addressed via support for automatic discovery of *default* `TestExecutionListener` implementations via the `SpringFactoriesLoader` mechanism.

Specifically, the `spring-test` module declares all core default `TestExecutionListener`s under the `org.springframework.test.context.TestExecutionListener` key in its `META-INF/spring.factories` properties file. Third-party frameworks and developers can contribute their own `TestExecutionListener`s to the list of default listeners in the same manner via their own `META-INF/spring.factories` properties file.

Ordering TestExecutionListeners

When the TestContext framework discovers default `TestExecutionListener`s via the aforementioned `SpringFactoriesLoader` mechanism, the instantiated listeners are sorted using Spring's `AnnotationAwareOrderComparator` which honors Spring's `Ordered` interface and `@Order` annotation for ordering. `AbstractTestExecutionListener` and all default `TestExecutionListener`s provided by Spring implement `Ordered` with appropriate values. Third-party frameworks and developers should therefore make sure that their *default* `TestExecutionListener`s are registered in the proper order by implementing `Ordered` or declaring `@Order`. Consult the javadocs for the `getOrder()` methods of the core default `TestExecutionListener`s for details on what values are assigned to each core listener.

Merging TestExecutionListeners

If a custom `TestExecutionListener` is registered via `@TestExecutionListeners`, the *default* listeners will not be registered. In most common testing scenarios, this effectively forces the developer to manually declare all default listeners in addition to any custom listeners. The following listing demonstrates this style of configuration.

```
@ContextConfiguration
@TestExecutionListeners({
    MyCustomTestExecutionListener.class,
    ServletTestExecutionListener.class,
    DirtiesContextBeforeModesTestExecutionListener.class,
    DependencyInjectionTestExecutionListener.class,
    DirtiesContextTestExecutionListener.class,
    TransactionalTestExecutionListener.class,
    SqlScriptsTestExecutionListener.class
})
public class MyTest {
    // class body...
}
```

The challenge with this approach is that it requires that the developer know exactly which listeners are registered by default. Moreover, the set of default listeners can change from release to release — for example, `SqlScriptsTestExecutionListener` was introduced in Spring Framework 4.1, and `DirtiesContextBeforeModesTestExecutionListener` was introduced in Spring Framework 4.2. Furthermore, third-party frameworks like Spring Security register their own default `TestExecutionListener`s via the aforementioned automatic discovery mechanism.

To avoid having to be aware of and re-declare **all** *default* listeners, the `mergeMode` attribute of `@TestExecutionListeners` can be set to `MergeMode.MERGE_WITH_DEFAULTS`. `MERGE_WITH_DEFAULTS` indicates that locally declared listeners should be merged with the default listeners. The merging algorithm ensures that duplicates are removed from the list and that the resulting set of merged listeners is sorted according to the semantics of `AnnotationAwareOrderComparator` as described in Ordering TestExecutionListeners. If a listener implements `Ordered` or is annotated with `@Order` it can influence the position in which it is merged with the defaults; otherwise, locally declared listeners will simply be appended to the list of default listeners when merged.

For example, if the `MyCustomTestExecutionListener` class in the previous example configures its `order` value (for example, `500`) to be less than the order of the `ServletTestExecutionListener` (which happens to be `1000`), the `MyCustomTestExecutionListener` can then be automatically merged with the list of defaults *in front of* the `ServletTestExecutionListener`, and the previous example could be replaced with the following.

```
@ContextConfiguration
@TestExecutionListeners(
    listeners = MyCustomTestExecutionListener.class,
    mergeMode = MERGE_WITH_DEFAULTS
)
public class MyTest {
    // class body...
}
```

3.5.4. Context management

Each `TestContext` provides context management and caching support for the test instance it is responsible for. Test instances do not automatically receive access to the configured `ApplicationContext`. However, if a test class implements the `ApplicationContextAware` interface, a reference to the `ApplicationContext` is supplied to the test instance. Note that `AbstractJUnit4SpringContextTests` and `AbstractTestNGSpringContextTests` implement `ApplicationContextAware` and therefore provide access to the `ApplicationContext` automatically.

@Autowired ApplicationContext

As an alternative to implementing the `ApplicationContextAware` interface, you can inject the application context for your test class through the `@Autowired` annotation on either a field or setter method. For example:

```
@RunWith(SpringRunner.class)
@ContextConfiguration
public class MyTest {

    <strong>@Autowired</strong>
    private ApplicationContext applicationContext;

    // class body...
}
```

Similarly, if your test is configured to load a `WebApplicationContext`, you can inject the web application context into your test as follows:

```
@RunWith(SpringRunner.class)
<strong>@WebAppConfiguration</strong>
@ContextConfiguration
public class MyWebAppTest {
    <strong>@Autowired</strong>
    private WebApplicationContext wac;

    // class body...
}
```

Dependency injection via `@Autowired` is provided by the `DependencyInjectionTestExecutionListener` which is configured by default (see Dependency injection of test fixtures).

Test classes that use the TestContext framework do not need to extend any particular class or implement a specific interface to configure their application context. Instead, configuration is achieved simply by declaring the `@ContextConfiguration` annotation at the class level. If your test class does not explicitly declare application context resource `locations` or annotated `classes`, the configured `ContextLoader` determines how to load a context from a default location or default configuration classes. In addition to context resource `locations` and annotated `classes`, an

application context can also be configured via application context `initializers`.

The following sections explain how to configure an `ApplicationContext` via XML configuration files, Groovy scripts, annotated classes (typically `@Configuration` classes), or context initializers using Spring's `@ContextConfiguration` annotation. Alternatively, you can implement and configure your own custom `SmartContextLoader` for advanced use cases.

Context configuration with XML resources

To load an `ApplicationContext` for your tests using XML configuration files, annotate your test class with `@ContextConfiguration` and configure the `locations` attribute with an array that contains the resource locations of XML configuration metadata. A plain or relative path — for example `"context.xml"` — will be treated as a classpath resource that is relative to the package in which the test class is defined. A path starting with a slash is treated as an absolute classpath location, for example `"/org/example/config.xml"`. A path which represents a resource URL (i.e., a path prefixed with `classpath:`, `file:`, `http:`, etc.) will be used *as is*.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from "/app-config.xml" and
// "/test-config.xml" in the root of the classpath
<strong>@ContextConfiguration(locations={"/app-config.xml", "/test-config.xml"})<
/strong>
public class MyTest {
    // class body...
}
```

`@ContextConfiguration` supports an alias for the `locations` attribute through the standard Java `value` attribute. Thus, if you do not need to declare additional attributes in `@ContextConfiguration`, you can omit the declaration of the `locations` attribute name and declare the resource locations by using the shorthand format demonstrated in the following example.

```
@RunWith(SpringRunner.class)
<strong>@ContextConfiguration({"/app-config.xml", "/test-config.xml"})</strong>
public class MyTest {
    // class body...
}
```

If you omit both the `locations` and `value` attributes from the `@ContextConfiguration` annotation, the TestContext framework will attempt to detect a default XML resource location. Specifically, `GenericXmlContextLoader` and `GenericXmlWebContextLoader` detect a default location based on the name of the test class. If your class is named `com.example.MyTest`, `GenericXmlContextLoader` loads your application context from `"classpath:com/example/MyTest-context.xml"`.

```
package com.example;

@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTest-context.xml"
<strong>@ContextConfiguration</strong>
public class MyTest {
    // class body...
}
```

Context configuration with Groovy scripts

To load an `ApplicationContext` for your tests using Groovy scripts that utilize the Groovy Bean Definition DSL, annotate your test class with `@ContextConfiguration` and configure the `locations` or `value` attribute with an array that contains the resource locations of Groovy scripts. Resource lookup semantics for Groovy scripts are the same as those described for XML configuration files.

Enabling Groovy script support

Support for using Groovy scripts to load an `ApplicationContext` in the Spring TestContext Framework is enabled automatically if Groovy is on the classpath.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from "/AppConfig.groovy" and
// "/TestConfig.groovy" in the root of the classpath
<strong>@ContextConfiguration({"/AppConfig.groovy", "/TestConfig.Groovy"})</strong>
public class MyTest {
    // class body...
}
```

If you omit both the `locations` and `value` attributes from the `@ContextConfiguration` annotation, the TestContext framework will attempt to detect a default Groovy script. Specifically, `GenericGroovyXmlContextLoader` and `GenericGroovyXmlWebContextLoader` detect a default location based on the name of the test class. If your class is named `com.example.MyTest`, the Groovy context loader will load your application context from `"classpath:com/example/MyTestContext.groovy"`.

```
package com.example;

@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTestContext.groovy"
<strong>@ContextConfiguration</strong>
public class MyTest {
    // class body...
}
```

Declaring XML config and Groovy scripts simultaneously

Both XML configuration files and Groovy scripts can be declared simultaneously via the `locations` or `value` attribute of `@ContextConfiguration`. If the path to a configured resource location ends with `.xml` it will be loaded using an `XmlBeanDefinitionReader`; otherwise it will be loaded using a `GroovyBeanDefinitionReader`.

The following listing demonstrates how to combine both in an integration test.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from
// "/app-config.xml" and "/TestConfig.groovy"
@ContextConfiguration({ "/app-config.xml", "/TestConfig.groovy" })
public class MyTest {
    // class body...
}
```

Context configuration with annotated classes

To load an `ApplicationContext` for your tests using *annotated classes* (see Java-based container configuration), annotate your test class with `@ContextConfiguration` and configure the `classes` attribute with an array that contains references to annotated classes.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from AppConfig and TestConfig
<strong>@ContextConfiguration(classes = {AppConfig.class, TestConfig.class})</strong>
public class MyTest {
    // class body...
}
```

Annotated Classes

The term *annotated class* can refer to any of the following.

- A class annotated with `@Configuration`
- A component (i.e., a class annotated with `@Component`, `@Service`, `@Repository`, etc.)
- A JSR-330 compliant class that is annotated with `javax.inject` annotations
- Any other class that contains `@Bean`-methods

Consult the javadocs of `@Configuration` and `@Bean` for further information regarding the configuration and semantics of *annotated classes*, paying special attention to the discussion of `*@Bean* Lite Mode`.

If you omit the `classes` attribute from the `@ContextConfiguration` annotation, the TestContext framework will attempt to detect the presence of default configuration classes. Specifically, `AnnotationConfigContextLoader` and `AnnotationConfigWebContextLoader` will detect all `static` nested

classes of the test class that meet the requirements for configuration class implementations as specified in the `@Configuration` javadocs. In the following example, the `OrderServiceTest` class declares a `static` nested configuration class named `Config` that will be automatically used to load the `ApplicationContext` for the test class. Note that the name of the configuration class is arbitrary. In addition, a test class can contain more than one `static` nested configuration class if desired.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from the
// static nested Config class
<strong>@ContextConfiguration</strong>
public class OrderServiceTest {

    @Configuration
    static class Config {

        // 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
    }

}
```

Mixing XML, Groovy scripts, and annotated classes

It may sometimes be desirable to mix XML configuration files, Groovy scripts, and annotated classes (i.e., typically `@Configuration` classes) to configure an `ApplicationContext` for your tests. For example, if you use XML configuration in production, you may decide that you want to use `@Configuration` classes to configure specific Spring-managed components for your tests, or vice versa.

Furthermore, some third-party frameworks (like Spring Boot) provide first-class support for loading an `ApplicationContext` from different types of resources simultaneously (e.g., XML configuration files, Groovy scripts, and `@Configuration` classes). The Spring Framework historically has not supported this for standard deployments. Consequently, most of the `SmartContextLoader` implementations that the Spring Framework delivers in the `spring-test` module support only one resource type per test context; however, this does not mean that you cannot use both. One exception to the general rule is that the `GenericGroovyXmlContextLoader` and

`GenericGroovyXmlWebContextLoader` support both XML configuration files and Groovy scripts simultaneously. Furthermore, third-party frameworks may choose to support the declaration of both `locations` and `classes` via `@ContextConfiguration`, and with the standard testing support in the TestContext framework, you have the following options.

If you want to use resource locations (e.g., XML or Groovy) *and* `@Configuration` classes to configure your tests, you will have to pick one as the *entry point*, and that one will have to include or import the other. For example, in XML or Groovy scripts you can include `@Configuration` classes via component scanning or define them as normal Spring beans; whereas, in a `@Configuration` class you can use `@ImportResource` to import XML configuration files or Groovy scripts. Note that this behavior is semantically equivalent to how you configure your application in production: in production configuration you will define either a set of XML or Groovy resource locations or a set of `@Configuration` classes that your production `ApplicationContext` will be loaded from, but you still have the freedom to include or import the other type of configuration.

Context configuration with context initializers

To configure an `ApplicationContext` for your tests using context initializers, annotate your test class with `@ContextConfiguration` and configure the `initializers` attribute with an array that contains references to classes that implement `ApplicationContextInitializer`. The declared context initializers will then be used to initialize the `ConfigurableApplicationContext` that is loaded for your tests. Note that the concrete `ConfigurableApplicationContext` type supported by each declared initializer must be compatible with the type of `ApplicationContext` created by the `SmartContextLoader` in use (i.e., typically a `GenericApplicationContext`). Furthermore, the order in which the initializers are invoked depends on whether they implement Spring's `Ordered` interface or are annotated with Spring's `@Order` annotation or the standard `@Priority` annotation.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from TestConfig
// and initialized by TestAppCtxInitializer
<strong>@ContextConfiguration(
    classes = TestConfig.class,
    initializers = TestAppCtxInitializer.class)</strong>
public class MyTest {
    // class body...
}
```

It is also possible to omit the declaration of XML configuration files, Groovy scripts, or annotated classes in `@ContextConfiguration` entirely and instead declare only `ApplicationContextInitializer` classes which are then responsible for registering beans in the context — for example, by programmatically loading bean definitions from XML files or configuration classes.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be initialized by EntireAppInitializer
// which presumably registers beans in the context
<strong>@ContextConfiguration(initializers = EntireAppInitializer.class)</strong>
public class MyTest {
    // class body...
}
```

Context configuration inheritance

`@ContextConfiguration` supports boolean `inheritLocations` and `inheritInitializers` attributes that denote whether resource locations or annotated classes and context initializers declared by superclasses should be *inherited*. The default value for both flags is `true`. This means that a test class inherits the resource locations or annotated classes as well as the context initializers declared by any superclasses. Specifically, the resource locations or annotated classes for a test class are appended to the list of resource locations or annotated classes declared by superclasses. Similarly, the initializers for a given test class will be added to the set of initializers defined by test superclasses. Thus, subclasses have the option of *extending* the resource locations, annotated classes, or context initializers.

If the `inheritLocations` or `inheritInitializers` attribute in `@ContextConfiguration` is set to `false`, the resource locations or annotated classes and the context initializers, respectively, for the test class *shadow* and effectively replace the configuration defined by superclasses.

In the following example that uses XML resource locations, the `ApplicationContext` for `ExtendedTest` will be loaded from *"base-config.xml" and "extended-config.xml"*, in that order. Beans defined in *"extended-config.xml"* may therefore *override* (i.e., replace) those defined in *"base-config.xml"*.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from "/base-config.xml"
// in the root of the classpath
<strong>@ContextConfiguration("/base-config.xml")</strong>
public class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from "/base-config.xml" and
// "/extended-config.xml" in the root of the classpath
<strong>@ContextConfiguration("/extended-config.xml")</strong>
public class ExtendedTest extends BaseTest {
    // class body...
}
```

Similarly, in the following example that uses annotated classes, the `ApplicationContext` for `ExtendedTest` will be loaded from the `BaseConfig` *and* `ExtendedConfig` classes, in that order. Beans defined in `ExtendedConfig` may therefore override (i.e., replace) those defined in `BaseConfig`.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from BaseConfig
<strong>@ContextConfiguration(classes = BaseConfig.class)</strong>
public class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from BaseConfig and ExtendedConfig
<strong>@ContextConfiguration(classes = ExtendedConfig.class)</strong>
public class ExtendedTest extends BaseTest {
    // class body...
}
```

In the following example that uses context initializers, the `ApplicationContext` for `ExtendedTest` will be initialized using `BaseInitializer` *and* `ExtendedInitializer`. Note, however, that the order in which the initializers are invoked depends on whether they implement Spring's `Ordered` interface or are annotated with Spring's `@Order` annotation or the standard `@Priority` annotation.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be initialized by BaseInitializer
<strong>@ContextConfiguration(initializers = BaseInitializer.class)</strong>
public class BaseTest {
    // class body...
}

// ApplicationContext will be initialized by BaseInitializer
// and ExtendedInitializer
<strong>@ContextConfiguration(initializers = ExtendedInitializer.class)</strong>
public class ExtendedTest extends BaseTest {
    // class body...
}
```

Context configuration with environment profiles

Spring 3.1 introduced first-class support in the framework for the notion of environments and profiles (a.k.a., *bean definition profiles*), and integration tests can be configured to activate particular bean definition profiles for various testing scenarios. This is achieved by annotating a test class with the `@ActiveProfiles` annotation and supplying a list of profiles that should be activated when loading the `ApplicationContext` for the test.

`@ActiveProfiles` may be used with any implementation of the new `SmartContextLoader` SPI, but `@ActiveProfiles` is not supported with implementations of the older `ContextLoader` SPI.

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

```
<!-- app-config.xml -->
<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 profile="default">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script
                location="classpath:com/bank/config/sql/schema.sql"/>
        </jdbc:embedded-database>
    </beans>

</beans>
```

```
package com.bank.service;

@RunWith(SpringRunner.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.xml` you'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 three times: in the *production* profile, the *dev* profile, and the *default* 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 populated with test data, 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.

It is sometimes useful to assign beans to a `default` profile. Beans within the default profile are only included when no other profile is specifically activated. This can be used to define *fallback* beans to be used in the application's default state. For example, you may explicitly provide a data source for `dev` and `production` profiles, but define an in-memory data source as a default when neither of these is active.

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(destroyMethod="")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

```
@Configuration
@Profile("default")
public class DefaultDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .build();
    }
}
```

```
@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(SpringRunner.class)
@ContextConfiguration(classes = {
        TransferServiceConfig.class,
        StandaloneDataConfig.class,
        JndiDataConfig.class,
        DefaultDataConfig.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 four independent `@Configuration` classes:

- `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
- `DefaultDataConfig`: defines a `dataSource` for a default embedded database in case no profile is active

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

It is often the case that a single set of profiles is used across multiple test classes within a given project. Thus, to avoid duplicate declarations of the `@ActiveProfiles` annotation it is possible to declare `@ActiveProfiles` once on a base class, and subclasses will automatically inherit the `@ActiveProfiles` configuration from the base class. In the following example, the declaration of `@ActiveProfiles` (as well as other annotations) has been moved to an abstract superclass, `AbstractIntegrationTest`.

```
package com.bank.service;

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {
        TransferServiceConfig.class,
        StandaloneDataConfig.class,
        JndiDataConfig.class,
        DefaultDataConfig.class})
@ActiveProfiles("dev")
public abstract class AbstractIntegrationTest {
}
```

```
package com.bank.service;

// "dev" profile inherited from superclass
public class TransferServiceTest extends AbstractIntegrationTest {

    @Autowired
    private TransferService transferService;

    @Test
    public void testTransferService() {
        // test the transferService
    }
}
```

`@ActiveProfiles` also supports an `inheritProfiles` attribute that can be used to disable the inheritance of active profiles.

```
package com.bank.service;

// "dev" profile overridden with "production"
@ActiveProfiles(profiles = "production", inheritProfiles = false)
public class ProductionTransferServiceTest extends AbstractIntegrationTest {
    // test body
}
```

Furthermore, it is sometimes necessary to resolve active profiles for tests *programmatically* instead of declaratively — for example, based on:

- the current operating system
- whether tests are being executed on a continuous integration build server
- the presence of certain environment variables
- the presence of custom class-level annotations
- etc.

To resolve active bean definition profiles programmatically, simply implement a custom `ActiveProfilesResolver` and register it via the `resolver` attribute of `@ActiveProfiles`. The following example demonstrates how to implement and register a custom `OperatingSystemActiveProfilesResolver`. For further information, refer to the corresponding javadocs.

```
package com.bank.service;

// "dev" profile overridden programmatically via a custom resolver
@ActiveProfiles(
    resolver = OperatingSystemActiveProfilesResolver.class,
    inheritProfiles = false)
public class TransferServiceTest extends AbstractIntegrationTest {
    // test body
}
```

```
package com.bank.service.test;

public class OperatingSystemActiveProfilesResolver implements ActiveProfilesResolver {

    @Override
    String[] resolve(Class<?> testClass) {
        String profile = ...;
        // determine the value of profile based on the operating system
        return new String[] {profile};
    }
}
```

Context configuration with test property sources

Spring 3.1 introduced first-class support in the framework for the notion of an environment with a hierarchy of *property sources*, and since Spring 4.1 integration tests can be configured with test-specific property sources. In contrast to the `@PropertySource` annotation used on `@Configuration` classes, the `@TestPropertySource` annotation can be declared on a test class to declare resource locations for test properties files or *inlined* properties. These test property sources will be added to the set of `PropertySources` in the `Environment` for the `ApplicationContext` loaded for the annotated integration test.

`@TestPropertySource` may be used with any implementation of the `SmartContextLoader` SPI, but `@TestPropertySource` is not supported with implementations of the older `ContextLoader` SPI.

Implementations of `SmartContextLoader` gain access to merged test property source values via the `getPropertySourceLocations()` and `getPropertySourceProperties()` methods in `MergedContextConfiguration`.

Declaring test property sources

Test properties files can be configured via the `locations` or `value` attribute of `@TestPropertySource` as shown in the following example.

Both traditional and XML-based properties file formats are supported — for example, `"classpath:/com/example/test.properties"` or `"file:///path/to/file.xml"`.

Each path will be interpreted as a Spring `Resource`. A plain path — for example, `"test.properties"` — will be treated as a classpath resource that is *relative* to the package in which the test class is defined. A path starting with a slash will be treated as an *absolute* classpath resource, for example: `"/org/example/test.xml"`. A path which references a URL (e.g., a path prefixed with `classpath:`, `file:`, `http:`, etc.) will be loaded using the specified resource protocol. Resource location wildcards (e.g. `*/.properties`) are not permitted: each location must evaluate to exactly one `.properties` or `.xml` resource.

```
@ContextConfiguration
@TestPropertySource("/test.properties")
public class MyIntegrationTests {
    // class body...
}
```

Inlined properties in the form of key-value pairs can be configured via the `properties` attribute of `@TestPropertySource` as shown in the following example. All key-value pairs will be added to the enclosing `Environment` as a single test `PropertySource` with the highest precedence.

The supported syntax for key-value pairs is the same as the syntax defined for entries in a Java properties file:

- `"key=value"`
- `"key:value"`

- `"key value"`

```
@ContextConfiguration
@TestPropertySource(properties = {"timezone = GMT", "port: 4242"})
public class MyIntegrationTests {
    // class body...
}
```

Default properties file detection

If `@TestPropertySource` is declared as an empty annotation (i.e., without explicit values for the `locations` or `properties` attributes), an attempt will be made to detect a *default* properties file relative to the class that declared the annotation. For example, if the annotated test class is `com.example.MyTest`, the corresponding default properties file is `"classpath:com/example/MyTest.properties"`. If the default cannot be detected, an `IllegalStateException` will be thrown.

Precedence

Test property sources have higher precedence than those loaded from the operating system's environment or Java system properties as well as property sources added by the application declaratively via `@PropertySource` or programmatically. Thus, test property sources can be used to selectively override properties defined in system and application property sources. Furthermore, inlined properties have higher precedence than properties loaded from resource locations.

In the following example, the `timezone` and `port` properties as well as any properties defined in `"/test.properties"` will override any properties of the same name that are defined in system and application property sources. Furthermore, if the `"/test.properties"` file defines entries for the `timezone` and `port` properties those will be overridden by the *inlined* properties declared via the `properties` attribute.

```
@ContextConfiguration
@TestPropertySource(
    locations = "/test.properties",
    properties = {"timezone = GMT", "port: 4242"}
)
public class MyIntegrationTests {
    // class body...
}
```

Inheriting and overriding test property sources

`@TestPropertySource` supports boolean `inheritLocations` and `inheritProperties` attributes that denote whether resource locations for properties files and inlined properties declared by superclasses should be *inherited*. The default value for both flags is `true`. This means that a test class inherits the locations and inlined properties declared by any superclasses. Specifically, the locations and inlined properties for a test class are appended to the locations and inlined properties declared by superclasses. Thus, subclasses have the option of *extending* the locations and inlined properties.

Note that properties that appear later will *shadow* (i.e., override) properties of the same name that appear earlier. In addition, the aforementioned precedence rules apply for inherited test property sources as well.

If the `inheritLocations` or `inheritProperties` attribute in `@TestPropertySource` is set to `false`, the locations or inlined properties, respectively, for the test class *shadow* and effectively replace the configuration defined by superclasses.

In the following example, the `ApplicationContext` for `BaseTest` will be loaded using only the `"base.properties"` file as a test property source. In contrast, the `ApplicationContext` for `ExtendedTest` will be loaded using the `"base.properties"` **and** `"extended.properties"` files as test property source locations.

```
@TestPropertySource("base.properties")
@ContextConfiguration
public class BaseTest {
    // ...
}

@TestPropertySource("extended.properties")
@ContextConfiguration
public class ExtendedTest extends BaseTest {
    // ...
}
```

In the following example, the `ApplicationContext` for `BaseTest` will be loaded using only the *inlined* `key1` property. In contrast, the `ApplicationContext` for `ExtendedTest` will be loaded using the *inlined* `key1` and `key2` properties.

```
@TestPropertySource(properties = "key1 = value1")
@ContextConfiguration
public class BaseTest {
    // ...
}

@TestPropertySource(properties = "key2 = value2")
@ContextConfiguration
public class ExtendedTest extends BaseTest {
    // ...
}
```

Loading a WebApplicationContext

Spring 3.2 introduced support for loading a `WebApplicationContext` in integration tests. To instruct the TestContext framework to load a `WebApplicationContext` instead of a standard `ApplicationContext`, simply annotate the respective test class with `@WebAppConfiguration`.

The presence of `@WebAppConfiguration` on your test class instructs the TestContext framework (TCF)

that a `WebApplicationContext` (WAC) should be loaded for your integration tests. In the background the TCF makes sure that a `MockServletContext` is created and supplied to your test's WAC. By default the base resource path for your `MockServletContext` will be set to *"src/main/webapp"*. This is interpreted as a path relative to the root of your JVM (i.e., normally the path to your project). If you're familiar with the directory structure of a web application in a Maven project, you'll know that *"src/main/webapp"* is the default location for the root of your WAR. If you need to override this default, simply provide an alternate path to the `@WebAppConfiguration` annotation (e.g., `@WebAppConfiguration("src/test/webapp")`). If you wish to reference a base resource path from the classpath instead of the file system, just use Spring's *classpath:* prefix.

Please note that Spring's testing support for `WebApplicationContexts` is on par with its support for standard `ApplicationContexts`. When testing with a `WebApplicationContext` you are free to declare XML configuration files, Groovy scripts, or `@Configuration` classes via `@ContextConfiguration`. You are of course also free to use any other test annotations such as `@ActiveProfiles`, `@TestExecutionListeners`, `@Sql`, `@Rollback`, etc.

The following examples demonstrate some of the various configuration options for loading a `WebApplicationContext`.

Conventions

```
@RunWith(SpringRunner.class)

// defaults to "file:src/main/webapp"
@WebAppConfiguration

// detects "WacTests-context.xml" in same package
// or static nested @Configuration class
@ContextConfiguration

public class WacTests {
    //...
}
```

The above example demonstrates the TestContext framework's support for *convention over configuration*. If you annotate a test class with `@WebAppConfiguration` without specifying a resource base path, the resource path will effectively default to *"file:src/main/webapp"*. Similarly, if you declare `@ContextConfiguration` without specifying resource `locations`, annotated `classes`, or context `initializers`, Spring will attempt to detect the presence of your configuration using conventions (i.e., *"WacTests-context.xml"* in the same package as the `WacTests` class or static nested `@Configuration` classes).

Default resource semantics

```
@RunWith(SpringRunner.class)

// file system resource
@WebAppConfiguration("webapp")

// classpath resource
@ContextConfiguration("/spring/test-servlet-config.xml")

public class WacTests {
    //...
}
```

This example demonstrates how to explicitly declare a resource base path with `@WebAppConfiguration` and an XML resource location with `@ContextConfiguration`. The important thing to note here is the different semantics for paths with these two annotations. By default, `@WebAppConfiguration` resource paths are file system based; whereas, `@ContextConfiguration` resource locations are classpath based.

Explicit resource semantics

```
@RunWith(SpringRunner.class)

// classpath resource
@WebAppConfiguration("classpath:test-web-resources")

// file system resource
@ContextConfiguration("file:src/main/webapp/WEB-INF/servlet-config.xml")

public class WacTests {
    //...
}
```

In this third example, we see that we can override the default resource semantics for both annotations by specifying a Spring resource prefix. Contrast the comments in this example with the previous example.

Working with Web Mocks

To provide comprehensive web testing support, Spring 3.2 introduced a `ServletTestExecutionListener` that is enabled by default. When testing against a `WebApplicationContext` this TestExecutionListener sets up default thread-local state via Spring Web's `RequestContextHolder` before each test method and creates a `MockHttpServletRequest`, `MockHttpServletResponse`, and `ServletWebRequest` based on the base resource path configured via `@WebAppConfiguration`. `ServletTestExecutionListener` also ensures that the `MockHttpServletResponse` and `ServletWebRequest` can be injected into the test instance, and once the test is complete it cleans up thread-local state.

Once you have a `WebApplicationContext` loaded for your test you might find that you need to interact

with the web mocks — for example, to set up your test fixture or to perform assertions after invoking your web component. The following example demonstrates which mocks can be autowired into your test instance. Note that the `WebApplicationContext` and `MockServletContext` are both cached across the test suite; whereas, the other mocks are managed per test method by the `ServletTestExecutionListener`.

Injecting mocks

```
@WebAppConfiguration
@ContextConfiguration
public class WacTests {

    @Autowired
    WebApplicationContext wac; // cached

    @Autowired
    MockServletContext servletContext; // cached

    @Autowired
    MockHttpSession session;

    @Autowired
    MockHttpServletRequest request;

    @Autowired
    MockHttpServletResponse response;

    @Autowired
    ServletWebRequest webRequest;

    //...
}
```

Context caching

Once the TestContext framework loads an `ApplicationContext` (or `WebApplicationContext`) for a test, that context will be cached and reused for *all* subsequent tests that declare the same unique context configuration within the same test suite. To understand how caching works, it is important to understand what is meant by *unique* and *test suite*.

An `ApplicationContext` can be *uniquely* identified by the combination of configuration parameters that is used to load it. Consequently, the unique combination of configuration parameters is used to generate a *key* under which the context is cached. The TestContext framework uses the following configuration parameters to build the context cache key:

- `locations` *(from @ContextConfiguration)*
- `classes` *(from @ContextConfiguration)*
- `contextInitializerClasses` *(from @ContextConfiguration)*
- `contextCustomizers` *(from ContextCustomizerFactory)*

- `contextLoader` *(from @ContextConfiguration)*
- `parent` *(from @ContextHierarchy)*
- `activeProfiles` *(from @ActiveProfiles)*
- `propertySourceLocations` *(from @TestPropertySource)*
- `propertySourceProperties` *(from @TestPropertySource)*
- `resourceBasePath` *(from @WebAppConfiguration)*

For example, if `TestClassA` specifies `{"app-config.xml", "test-config.xml"}` for the `locations` (or `value`) attribute of `@ContextConfiguration`, the TestContext framework will load the corresponding `ApplicationContext` and store it in a `static` context cache under a key that is based solely on those locations. So if `TestClassB` also defines `{"app-config.xml", "test-config.xml"}` for its locations (either explicitly or implicitly through inheritance) but does not define `@WebAppConfiguration`, a different `ContextLoader`, different active profiles, different context initializers, different test property sources, or a different parent context, then the same `ApplicationContext` will be shared by both test classes. This means that the setup cost for loading an application context is incurred only once (per test suite), and subsequent test execution is much faster.

Test suites and forked processes

The Spring TestContext framework stores application contexts in a *static* cache. This means that the context is literally stored in a `static` variable. In other words, if tests execute in separate processes the static cache will be cleared between each test execution, and this will effectively disable the caching mechanism.

To benefit from the caching mechanism, all tests must run within the same process or test suite. This can be achieved by executing all tests as a group within an IDE. Similarly, when executing tests with a build framework such as Ant, Maven, or Gradle it is important to make sure that the build framework does not *fork* between tests. For example, if the forkMode for the Maven Surefire plug-in is set to `always` or `pertest`, the TestContext framework will not be able to cache application contexts between test classes and the build process will run significantly slower as a result.

Since Spring Framework 4.3, the size of the context cache is bounded with a default maximum size of 32. Whenever the maximum size is reached, a *least recently used* (LRU) eviction policy is used to evict and close stale contexts. The maximum size can be configured from the command line or a build script by setting a JVM system property named `spring.test.context.cache.maxSize`. As an alternative, the same property can be set programmatically via the `SpringProperties` API.

Since having a large number of application contexts loaded within a given test suite can cause the suite to take an unnecessarily long time to execute, it is often beneficial to know exactly how many contexts have been loaded and cached. To view the statistics for the underlying context cache, simply set the log level for the `org.springframework.test.context.cache` logging category to `DEBUG`.

In the unlikely case that a test corrupts the application context and requires reloading — for example, by modifying a bean definition or the state of an application object — you can annotate your test class or test method with `@DirtiesContext` (see the discussion of `@DirtiesContext` in Spring

Testing Annotations). This instructs Spring to remove the context from the cache and rebuild the application context before executing the next test. Note that support for the `@DirtiesContext` annotation is provided by the `DirtiesContextBeforeModesTestExecutionListener` and the `DirtiesContextTestExecutionListener` which are enabled by default.

Context hierarchies

When writing integration tests that rely on a loaded Spring `ApplicationContext`, it is often sufficient to test against a single context; however, there are times when it is beneficial or even necessary to test against a hierarchy of `ApplicationContext`s. For example, if you are developing a Spring MVC web application you will typically have a root `WebApplicationContext` loaded via Spring's `ContextLoaderListener` and a child `WebApplicationContext` loaded via Spring's `DispatcherServlet`. This results in a parent-child context hierarchy where shared components and infrastructure configuration are declared in the root context and consumed in the child context by web-specific components. Another use case can be found in Spring Batch applications where you often have a parent context that provides configuration for shared batch infrastructure and a child context for the configuration of a specific batch job.

Since Spring Framework 3.2.2, it is possible to write integration tests that use context hierarchies by declaring context configuration via the `@ContextHierarchy` annotation, either on an individual test class or within a test class hierarchy. If a context hierarchy is declared on multiple classes within a test class hierarchy it is also possible to merge or override the context configuration for a specific, named level in the context hierarchy. When merging configuration for a given level in the hierarchy the configuration resource type (i.e., XML configuration files or annotated classes) must be consistent; otherwise, it is perfectly acceptable to have different levels in a context hierarchy configured using different resource types.

The following JUnit 4 based examples demonstrate common configuration scenarios for integration tests that require the use of context hierarchies.

Single test class with context hierarchy

`ControllerIntegrationTests` represents a typical integration testing scenario for a Spring MVC web application by declaring a context hierarchy consisting of two levels, one for the *root* WebApplicationContext (loaded using the `TestAppConfig` `@Configuration` class) and one for the *dispatcher servlet* `WebApplicationContext` (loaded using the `WebConfig` `@Configuration` class). The `WebApplicationContext` that is *autowired* into the test instance is the one for the child context (i.e., the lowest context in the hierarchy).

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = TestAppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
public class ControllerIntegrationTests {

    @Autowired
    private WebApplicationContext wac;

    // ...
}
```

Class hierarchy with implicit parent context

The following test classes define a context hierarchy within a test class hierarchy. `AbstractWebTests` declares the configuration for a root `WebApplicationContext` in a Spring-powered web application. Note, however, that `AbstractWebTests` does not declare `@ContextHierarchy`; consequently, subclasses of `AbstractWebTests` can optionally participate in a context hierarchy or simply follow the standard semantics for `@ContextConfiguration`. `SoapWebServiceTests` and `RestWebServiceTests` both extend `AbstractWebTests` and define a context hierarchy via `@ContextHierarchy`. The result is that three application contexts will be loaded (one for each declaration of `@ContextConfiguration`), and the application context loaded based on the configuration in `AbstractWebTests` will be set as the parent context for each of the contexts loaded for the concrete subclasses.

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration("file:src/main/webapp/WEB-INF/applicationContext.xml")
public abstract class AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/soap-ws-config.xml")
public class SoapWebServiceTests extends AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/rest-ws-config.xml")
public class RestWebServiceTests extends AbstractWebTests {}
```

Class hierarchy with merged context hierarchy configuration

The following classes demonstrate the use of *named* hierarchy levels in order to *merge* the configuration for specific levels in a context hierarchy. `BaseTests` defines two levels in the hierarchy, `parent` and `child`. `ExtendedTests` extends `BaseTests` and instructs the Spring TestContext Framework to merge the context configuration for the `child` hierarchy level, simply by ensuring that the names declared via the `name` attribute in `@ContextConfiguration` are both `"child"`. The result is that three application contexts will be loaded: one for `"/app-config.xml"`, one for `"/user-config.xml"`, and one for `{"/user-config.xml", "/order-config.xml"}`. As with the previous example, the application context loaded from `"/app-config.xml"` will be set as the parent context for the contexts loaded from `"/user-config.xml"` and `{"/user-config.xml", "/order-config.xml"}`.

```
@RunWith(SpringRunner.class)
@ContextHierarchy({
    @ContextConfiguration(name = "parent", locations = "/app-config.xml"),
    @ContextConfiguration(name = "child", locations = "/user-config.xml")
})
public class BaseTests {}

@ContextHierarchy(
    @ContextConfiguration(name = "child", locations = "/order-config.xml")
)
public class ExtendedTests extends BaseTests {}
```

Class hierarchy with overridden context hierarchy configuration

In contrast to the previous example, this example demonstrates how to *override* the configuration for a given named level in a context hierarchy by setting the `inheritLocations` flag in `@ContextConfiguration` to `false`. Consequently, the application context for `ExtendedTests` will be loaded only from `"/test-user-config.xml"` and will have its parent set to the context loaded from `"/app-config.xml"`.

```
@RunWith(SpringRunner.class)
@ContextHierarchy({
    @ContextConfiguration(name = "parent", locations = "/app-config.xml"),
    @ContextConfiguration(name = "child", locations = "/user-config.xml")
})
public class BaseTests {}

@ContextHierarchy(
    @ContextConfiguration(
        name = "child",
        locations = "/test-user-config.xml",
        inheritLocations = false
))
public class ExtendedTests extends BaseTests {}
```

Dirtying a context within a context hierarchy

If `@DirtiesContext` is used in a test whose context is configured as part of a context hierarchy, the `hierarchyMode` flag can be used to control how the context cache is cleared. For further details consult the discussion of `@DirtiesContext` in Spring Testing Annotations and the `@DirtiesContext` javadocs.

3.5.5. Dependency injection of test fixtures

When you use the `DependencyInjectionTestExecutionListener` — which is configured by default — the dependencies of your test instances are *injected* from beans in the application context that you configured with `@ContextConfiguration`. You may use setter injection, field injection, or both, depending on which annotations you choose and whether you place them on setter methods

or fields. For consistency with the annotation support introduced in Spring 2.5 and 3.0, you can use Spring's `@Autowired` annotation or the `@Inject` annotation from JSR 330.

The TestContext framework does not instrument the manner in which a test instance is instantiated. Thus the use of `@Autowired` or `@Inject` for constructors has no effect for test classes.

Because `@Autowired` is used to perform *autowiring by type*, if you have multiple bean definitions of the same type, you cannot rely on this approach for those particular beans. In that case, you can use `@Autowired` in conjunction with `@Qualifier`. As of Spring 3.0 you may also choose to use `@Inject` in conjunction with `@Named`. Alternatively, if your test class has access to its `ApplicationContext`, you can perform an explicit lookup by using (for example) a call to `applicationContext.getBean("titleRepository")`.

If you do not want dependency injection applied to your test instances, simply do not annotate fields or setter methods with `@Autowired` or `@Inject`. Alternatively, you can disable dependency injection altogether by explicitly configuring your class with `@TestExecutionListeners` and omitting `DependencyInjectionTestExecutionListener.class` from the list of listeners.

Consider the scenario of testing a `HibernateTitleRepository` class, as outlined in the Goals section. The next two code listings demonstrate the use of `@Autowired` on fields and setter methods. The application context configuration is presented after all sample code listings.

The dependency injection behavior in the following code listings is not specific to JUnit 4. The same DI techniques can be used in conjunction with any testing framework.

The following examples make calls to static assertion methods such as `assertNotNull()` but without prepending the call with `Assert`. In such cases, assume that the method was properly imported through an `import static` declaration that is not shown in the example.

The first code listing shows a JUnit 4 based implementation of the test class that uses `@Autowired` for field injection.

```
@RunWith(SpringRunner.class)
// specifies the Spring configuration to load for this test fixture
<strong>@ContextConfiguration("repository-config.xml")</strong>
public class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    <strong>@Autowired</strong>
    private HibernateTitleRepository titleRepository;

    @Test
    public void findById() {
        Title title = titleRepository.findById(new Long(10));
        assertNotNull(title);
    }
}
```

Alternatively, you can configure the class to use `@Autowired` for setter injection as seen below.

```
@RunWith(SpringRunner.class)
// specifies the Spring configuration to load for this test fixture
<strong>@ContextConfiguration("repository-config.xml")</strong>
public class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    private HibernateTitleRepository titleRepository;

    <strong>@Autowired</strong>
    public void setTitleRepository(HibernateTitleRepository titleRepository) {
        this.titleRepository = titleRepository;
    }

    @Test
    public void findById() {
        Title title = titleRepository.findById(new Long(10));
        assertNotNull(title);
    }
}
```

The preceding code listings use the same XML context file referenced by the `@ContextConfiguration` annotation (that is, `repository-config.xml`), which looks like this:

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

    <!-- this bean will be injected into the HibernateTitleRepositoryTests class -->
    <bean id="<strong>titleRepository</strong>" class="
<strong>com.foo.repository.hibernate.HibernateTitleRepository</strong>">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <bean id="sessionFactory" class=
"org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <!-- configuration elided for brevity -->
    </bean>

</beans>
```

If you are extending from a Spring-provided test base class that happens to use `@Autowired` on one of its setter methods, you might have multiple beans of the affected type defined in your application context: for example, multiple `DataSource` beans. In such a case, you can override the setter method and use the `@Qualifier` annotation to indicate a specific target bean as follows, but make sure to delegate to the overridden method in the superclass as well.

```
// ...

    @Autowired
    @Override
    public void setDataSource(<strong>@Qualifier("myDataSource")<
/strong> DataSource dataSource) {
        <strong>super</strong>.setDataSource(dataSource);
    }

// ...
```

The specified qualifier value indicates the specific `DataSource` bean to inject, narrowing the set of type matches to a specific bean. Its value is matched against `<qualifier>` declarations within the corresponding `<bean>` definitions. The bean name is used as a fallback qualifier value, so you may effectively also point to a specific bean by name there (as shown above, assuming that "myDataSource" is the bean id).

3.5.6. Testing request and session scoped beans

Request and session scoped beans have been supported by Spring since the early years, and since

Spring 3.2 it's a breeze to test your request-scoped and session-scoped beans by following these steps.

- Ensure that a `WebApplicationContext` is loaded for your test by annotating your test class with `@WebAppConfiguration`.
- Inject the mock request or session into your test instance and prepare your test fixture as appropriate.
- Invoke your web component that you retrieved from the configured `WebApplicationContext` (i.e., via dependency injection).
- Perform assertions against the mocks.

The following code snippet displays the XML configuration for a login use case. Note that the `userService` bean has a dependency on a request-scoped `loginAction` bean. Also, the `LoginAction` is instantiated using SpEL expressions that retrieve the username and password from the current HTTP request. In our test, we will want to configure these request parameters via the mock managed by the TestContext framework.

Request-scoped bean configuration

```
<beans>

    <bean id="userService" class="com.example.SimpleUserService"
            c:loginAction-ref="loginAction"/>

    <bean id="loginAction" class="com.example.LoginAction"
            c:username="#{request.getParameter('user')}"
            c:password="#{request.getParameter('pswd')}"
            scope="request">
        <aop:scoped-proxy/>
    </bean>

</beans>
```

In `RequestScopedBeanTests` we inject both the `UserService` (i.e., the subject under test) and the `MockHttpServletRequest` into our test instance. Within our `requestScope()` test method we set up our test fixture by setting request parameters in the provided `MockHttpServletRequest`. When the `loginUser()` method is invoked on our `userService` we are assured that the user service has access to the request-scoped `loginAction` for the current `MockHttpServletRequest` (i.e., the one we just set parameters in). We can then perform assertions against the results based on the known inputs for the username and password.

Request-scoped bean test

```
@RunWith(SpringRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class RequestScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpServletRequest request;

    @Test
    public void requestScope() {
        request.setParameter("user", "enigma");
        request.setParameter("pswd", "$pr!ng");

        LoginResults results = userService.loginUser();
        // assert results
    }
}
```

The following code snippet is similar to the one we saw above for a request-scoped bean; however, this time the `userService` bean has a dependency on a session-scoped `userPreferences` bean. Note that the `UserPreferences` bean is instantiated using a SpEL expression that retrieves the *theme* from the current HTTP session. In our test, we will need to configure a theme in the mock session managed by the TestContext framework.

Session-scoped bean configuration

```
<beans>

    <bean id="userService" class="com.example.SimpleUserService"
            c:userPreferences-ref="userPreferences" />

    <bean id="userPreferences" class="com.example.UserPreferences"
            c:theme="#{session.getAttribute('theme')}"
            scope="session">
        <aop:scoped-proxy/>
    </bean>

</beans>
```

In `SessionScopedBeanTests` we inject the `UserService` and the `MockHttpSession` into our test instance. Within our `sessionScope()` test method we set up our test fixture by setting the expected "theme" attribute in the provided `MockHttpSession`. When the `processUserPreferences()` method is invoked on our `userService` we are assured that the user service has access to the session-scoped `userPreferences` for the current `MockHttpSession`, and we can perform assertions against the results based on the configured theme.

Session-scoped bean test

```
@RunWith(SpringRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class SessionScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpSession session;

    @Test
    public void sessionScope() throws Exception {
        session.setAttribute("theme", "blue");

        Results results = userService.processUserPreferences();
        // assert results
    }
}
```

3.5.7. Transaction management

In the TestContext framework, transactions are managed by the `TransactionalTestExecutionListener` which is configured by default, even if you do not explicitly declare `@TestExecutionListeners` on your test class. To enable support for transactions, however, you must configure a `PlatformTransactionManager` bean in the `ApplicationContext` that is loaded via `@ContextConfiguration` semantics (further details are provided below). In addition, you must declare Spring's `@Transactional` annotation either at the class or method level for your tests.

Test-managed transactions

Test-managed transactions are transactions that are managed *declaratively* via the `TransactionalTestExecutionListener` or *programmatically* via `TestTransaction` (see below). Such transactions should not be confused with *Spring-managed transactions* (i.e., those managed directly by Spring within the `ApplicationContext` loaded for tests) or *application-managed transactions* (i.e., those managed programmatically within application code that is invoked via tests). Spring-managed and application-managed transactions will typically participate in test-managed transactions; however, caution should be taken if Spring-managed or application-managed transactions are configured with any *propagation* type other than `REQUIRED` or `SUPPORTS` (see the discussion on transaction propagation for details).

Enabling and disabling transactions

Annotating a test method with `@Transactional` causes the test to be run within a transaction that will, by default, be automatically rolled back after completion of the test. If a test class is annotated with `@Transactional`, each test method within that class hierarchy will be run within a transaction. Test methods that are not annotated with `@Transactional` (at the class or method level) will not be run within a transaction. Furthermore, tests that are annotated with `@Transactional` but have the `propagation` type set to `NOT_SUPPORTED` will not be run within a transaction.

Note that `AbstractTransactionalJUnit4SpringContextTests` *and* `AbstractTransactionalTestNGSpringContextTests` *are preconfigured for transactional support at the class level.*

The following example demonstrates a common scenario for writing an integration test for a Hibernate-based `UserRepository`. As explained in Transaction rollback and commit behavior, there is no need to clean up the database after the `createUser()` method is executed since any changes made to the database will be automatically rolled back by the `TransactionalTestExecutionListener`. See PetClinic Example for an additional example.

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = TestConfig.class)
@Transactional
public class HibernateUserRepositoryTests {

    @Autowired
    HibernateUserRepository repository;

    @Autowired
    SessionFactory sessionFactory;

    JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Test
    public void createUser() {
        // track initial state in test database:
        final int count = countRowsInTable("user");

        User user = new User(...);
        repository.save(user);

        // Manual flush is required to avoid false positive in test
        sessionFactory.getCurrentSession().flush();
        assertNumUsers(count + 1);
    }

    protected int countRowsInTable(String tableName) {
        return JdbcTestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    protected void assertNumUsers(int expected) {
        assertEquals("Number of rows in the [user] table.", expected,
countRowsInTable("user"));
    }
}
```

Transaction rollback and commit behavior

By default, test transactions will be automatically rolled back after completion of the test; however, transactional commit and rollback behavior can be configured declaratively via the `@Commit` and `@Rollback` annotations. See the corresponding entries in the annotation support section for further details.

Programmatic transaction management

Since Spring Framework 4.1, it is possible to interact with test-managed transactions *programmatically* via the static methods in `TestTransaction`. For example, `TestTransaction` may be used within *test* methods, *before* methods, and *after* methods to start or end the current test-managed transaction or to configure the current test-managed transaction for rollback or commit. Support for `TestTransaction` is automatically available whenever the `TransactionalTestExecutionListener` is enabled.

The following example demonstrates some of the features of `TestTransaction`. Consult the javadocs for `TestTransaction` for further details.

```
@ContextConfiguration(classes = TestConfig.class)
public class ProgrammaticTransactionManagementTests extends
        AbstractTransactionalJUnit4SpringContextTests {

    @Test
    public void transactionalTest() {
        // assert initial state in test database:
        assertNumUsers(2);

        deleteFromTables("user");

        // changes to the database will be committed!
        TestTransaction.flagForCommit();
        TestTransaction.end();
        assertFalse(TestTransaction.isActive());
        assertNumUsers(0);

        TestTransaction.start();
        // perform other actions against the database that will
        // be automatically rolled back after the test completes...
    }

    protected void assertNumUsers(int expected) {
        assertEquals("Number of rows in the [user] table.", expected,
countRowsInTable("user"));
    }
}
```

Executing code outside of a transaction

Occasionally you need to execute certain code before or after a transactional test method but outside the transactional context — for example, to verify the initial database state prior to execution of your test or to verify expected transactional commit behavior after test execution (if the test was configured to commit the transaction). `TransactionalTestExecutionListener` supports the `@BeforeTransaction` and `@AfterTransaction` annotations exactly for such scenarios. Simply annotate any `void` method in a test class or any `void` default method in a test interface with one of these annotations, and the `TransactionalTestExecutionListener` ensures that your *before transaction*

method or *after transaction method* is executed at the appropriate time.

Any *before methods* (such as methods annotated with JUnit Jupiter's `@BeforeEach`) and any *after methods* (such as methods annotated with JUnit Jupiter's `@AfterEach`) are executed *within* a transaction. In addition, methods annotated with `@BeforeTransaction` or `@AfterTransaction` are naturally not executed for test methods that are not configured to run within a transaction.

Configuring a transaction manager

`TransactionalTestExecutionListener` expects a `PlatformTransactionManager` bean to be defined in the Spring `ApplicationContext` for the test. In case there are multiple instances of `PlatformTransactionManager` within the test's `ApplicationContext`, a *qualifier* may be declared via `@Transactional("myTxMgr")` or `@Transactional(transactionManager = "myTxMgr")`, or `TransactionManagementConfigurer` can be implemented by an `@Configuration` class. Consult the javadocs for `TestContextTransactionUtils.retrieveTransactionManager()` for details on the algorithm used to look up a transaction manager in the test's `ApplicationContext`.

Demonstration of all transaction-related annotations

The following JUnit 4 based example displays a *fictitious* integration testing scenario highlighting all transaction-related annotations. The example is **not** intended to demonstrate best practices but rather to demonstrate how these annotations can be used. Consult the annotation support section for further information and configuration examples. Transaction management for `@Sql` contains an additional example using `@Sql` for declarative SQL script execution with default transaction rollback semantics.

```
@RunWith(SpringRunner.class)
@ContextConfiguration
<strong>@Transactional(transactionManager = "txMgr")</strong>
<strong>@Commit</strong>
public class FictitiousTransactionalTest {

    <strong>@BeforeTransaction</strong>
    void verifyInitialDatabaseState() {
        // logic to verify the initial state before a transaction is started
    }

    @Before
    public void setUpTestDataWithinTransaction() {
        // set up test data within the transaction
    }

    @Test
    // overrides the class-level @Commit setting
    <strong>@Rollback</strong>
    public void modifyDatabaseWithinTransaction() {
        // logic which uses the test data and modifies database state
    }

    @After
    public void tearDownWithinTransaction() {
        // execute "tear down" logic within the transaction
    }

    <strong>@AfterTransaction</strong>
    void verifyFinalDatabaseState() {
        // logic to verify the final state after transaction has rolled back
    }

}
```

Avoid false positives when testing ORM code

When you test application code that manipulates the state of a Hibernate session or JPA persistence context, make sure to *flush* the underlying unit of work within test methods that execute that code. Failing to flush the underlying unit of work can produce *false positives*: your test may pass, but the same code throws an exception in a live, production environment. In the following Hibernate-based example test case, one method demonstrates a false positive, and the other method correctly exposes the results of flushing the session. Note that this applies to any ORM frameworks that maintain an in-memory *unit of work*.

```
// ...

@Autowired
SessionFactory sessionFactory;

@Transactional
@Test // no expected exception!
public void falsePositive() {
    updateEntityInHibernateSession();
    // False positive: an exception will be thrown once the Hibernate
    // Session is finally flushed (i.e., in production code)
}

@Transactional
@Test(expected = ...)
public void updateWithSessionFlush() {
    updateEntityInHibernateSession();
    // Manual flush is required to avoid false positive in test
    sessionFactory.getCurrentSession().flush();
}

// ...
```

Or for JPA:

```
// ...

@PersistenceContext
EntityManager entityManager;

@Transactional
@Test // no expected exception!
public void falsePositive() {
    updateEntityInJpaPersistenceContext();
    // False positive: an exception will be thrown once the JPA
    // EntityManager is finally flushed (i.e., in production code)
}

@Transactional
@Test(expected = ...)
public void updateWithEntityManagerFlush() {
    updateEntityInJpaPersistenceContext();
    // Manual flush is required to avoid false positive in test
    entityManager.flush();
}

// ...
```

3.5.8. Executing SQL scripts

When writing integration tests against a relational database, it is often beneficial to execute SQL scripts to modify the database schema or insert test data into tables. The `spring-jdbc` module provides support for *initializing* an embedded or existing database by executing SQL scripts when the Spring `ApplicationContext` is loaded. See Embedded database support and Testing data access logic with an embedded database for details.

Although it is very useful to initialize a database for testing *once* when the `ApplicationContext` is loaded, sometimes it is essential to be able to modify the database *during* integration tests. The following sections explain how to execute SQL scripts programmatically and declaratively during integration tests.

Executing SQL scripts programmatically

Spring provides the following options for executing SQL scripts programmatically within integration test methods.

- `org.springframework.jdbc.datasource.init.ScriptUtils`
- `org.springframework.jdbc.datasource.init.ResourceDatabasePopulator`
- `org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests`
- `org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests`

`ScriptUtils` provides a collection of static utility methods for working with SQL scripts and is mainly intended for internal use within the framework. However, if you require full control over how SQL scripts are parsed and executed, `ScriptUtils` may suit your needs better than some of the other alternatives described below. Consult the javadocs for individual methods in `ScriptUtils` for further details.

`ResourceDatabasePopulator` provides a simple object-based API for programmatically populating, initializing, or cleaning up a database using SQL scripts defined in external resources. `ResourceDatabasePopulator` provides options for configuring the character encoding, statement separator, comment delimiters, and error handling flags used when parsing and executing the scripts, and each of the configuration options has a reasonable default value. Consult the javadocs for details on default values. To execute the scripts configured in a `ResourceDatabasePopulator`, you can invoke either the `populate(Connection)` method to execute the populator against a `java.sql.Connection` or the `execute(DataSource)` method to execute the populator against a `javax.sql.DataSource`. The following example specifies SQL scripts for a test schema and test data, sets the statement separator to `"@@"`, and then executes the scripts against a `DataSource`.

```
@Test
public void databaseTest {
    ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
    populator.addScripts(
            new ClassPathResource("test-schema.sql"),
            new ClassPathResource("test-data.sql"));
    populator.setSeparator("@@");
    populator.execute(this.dataSource);
    // execute code that uses the test schema and data
}
```

Note that `ResourceDatabasePopulator` internally delegates to `ScriptUtils` for parsing and executing SQL scripts. Similarly, the `executeSqlScript(..)` methods in `AbstractTransactionalJUnit4SpringContextTests` and `AbstractTransactionalTestNGSpringContextTests` internally use a `ResourceDatabasePopulator` for executing SQL scripts. Consult the javadocs for the various `executeSqlScript(..)` methods for further details.

Executing SQL scripts declaratively with @Sql

In addition to the aforementioned mechanisms for executing SQL scripts *programmatically*, SQL scripts can also be configured *declaratively* in the Spring TestContext Framework. Specifically, the `@Sql` annotation can be declared on a test class or test method to configure the resource paths to SQL scripts that should be executed against a given database either before or after an integration test method. Note that method-level declarations override class-level declarations and that support for `@Sql` is provided by the `SqlScriptsTestExecutionListener` which is enabled by default.

Path resource semantics

Each path will be interpreted as a Spring `Resource`. A plain path — for example, `"schema.sql"` — will be treated as a classpath resource that is *relative* to the package in which the test class is defined. A path starting with a slash will be treated as an *absolute* classpath resource, for example: `"/org/example/schema.sql"`. A path which references a URL (e.g., a path prefixed with `classpath:`, `file:`, `http:`, etc.) will be loaded using the specified resource protocol.

The following example demonstrates how to use `@Sql` at the class level and at the method level within a JUnit Jupiter based integration test class.

```
@SpringJUnitConfig
@Sql("/test-schema.sql")
class DatabaseTests {

    @Test
    void emptySchemaTest {
        // execute code that uses the test schema without any test data
    }

    @Test
    @Sql({"/test-schema.sql", "/test-user-data.sql"})
    void userTest {
        // execute code that uses the test schema and test data
    }
}
```

Default script detection

If no SQL scripts are specified, an attempt will be made to detect a `default` script depending on where `@Sql` is declared. If a default cannot be detected, an `IllegalStateException` will be thrown.

- *class-level declaration*: if the annotated test class is `com.example.MyTest`, the corresponding default script is `"classpath:com/example/MyTest.sql"`.
- *method-level declaration*: if the annotated test method is named `testMethod()` and is defined in the class `com.example.MyTest`, the corresponding default script is `"classpath:com/example/MyTest.testMethod.sql"`.

Declaring multiple `@Sql` sets

If multiple sets of SQL scripts need to be configured for a given test class or test method but with different syntax configuration, different error handling rules, or different execution phases per set, it is possible to declare multiple instances of `@Sql`. With Java 8, `@Sql` can be used as a *repeatable* annotation. Otherwise, the `@SqlGroup` annotation can be used as an explicit container for declaring multiple instances of `@Sql`.

The following example demonstrates the use of `@Sql` as a repeatable annotation using Java 8. In this scenario the `test-schema.sql` script uses a different syntax for single-line comments.

```
@Test
@Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`"))
@Sql("/test-user-data.sql")
public void userTest {
    // execute code that uses the test schema and test data
}
```

The following example is identical to the above except that the `@Sql` declarations are grouped together within `@SqlGroup` for compatibility with Java 6 and Java 7.

```
@Test
@SqlGroup({
    @Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`")),
    @Sql("/test-user-data.sql")
)}
public void userTest {
    // execute code that uses the test schema and test data
}
```

Script execution phases

By default, SQL scripts will be executed *before* the corresponding test method. However, if a particular set of scripts needs to be executed *after* the test method — for example, to clean up database state — the `executionPhase` attribute in `@Sql` can be used as seen in the following example. Note that `ISOLATED` and `AFTER_TEST_METHOD` are statically imported from `Sql.TransactionMode` and `Sql.ExecutionPhase` respectively.

```
@Test
@Sql(
    scripts = "create-test-data.sql",
    config = @SqlConfig(transactionMode = ISOLATED)
)
@Sql(
    scripts = "delete-test-data.sql",
    config = @SqlConfig(transactionMode = ISOLATED),
    executionPhase = AFTER_TEST_METHOD
)
public void userTest {
    // execute code that needs the test data to be committed
    // to the database outside of the test's transaction
}
```

Script configuration with `@SqlConfig`

Configuration for script parsing and error handling can be configured via the `@SqlConfig` annotation. When declared as a class-level annotation on an integration test class, `@SqlConfig` serves as *global* configuration for all SQL scripts within the test class hierarchy. When declared directly via the `config` attribute of the `@Sql` annotation, `@SqlConfig` serves as *local* configuration for the SQL scripts declared within the enclosing `@Sql` annotation. Every attribute in `@SqlConfig` has an implicit default value which is documented in the javadocs of the corresponding attribute. Due to the rules defined for annotation attributes in the Java Language Specification, it is unfortunately not possible to assign a value of `null` to an annotation attribute. Thus, in order to support overrides of inherited global configuration, `@SqlConfig` attributes have an explicit default value of either `""` for Strings or `DEFAULT` for Enums. This approach allows local declarations of `@SqlConfig` to selectively override individual attributes from global declarations of `@SqlConfig` by providing a value other than `""` or `DEFAULT`. Global `@SqlConfig` attributes are inherited whenever local `@SqlConfig` attributes do not supply an explicit value other than `""` or `DEFAULT`. Explicit *local* configuration therefore overrides

global configuration.

The configuration options provided by `@Sql` and `@SqlConfig` are equivalent to those supported by `ScriptUtils` and `ResourceDatabasePopulator` but are a superset of those provided by the `<jdbc:initialize-database/>` XML namespace element. Consult the javadocs of individual attributes in `@Sql` and `@SqlConfig` for details.

Transaction management for `@Sql`

By default, the `SqlScriptsTestExecutionListener` will infer the desired transaction semantics for scripts configured via `@Sql`. Specifically, SQL scripts will be executed without a transaction, within an existing Spring-managed transaction — for example, a transaction managed by the `TransactionalTestExecutionListener` for a test annotated with `@Transactional` — or within an isolated transaction, depending on the configured value of the `transactionMode` attribute in `@SqlConfig` and the presence of a `PlatformTransactionManager` in the test's `ApplicationContext`. As a bare minimum however, a `javax.sql.DataSource` must be present in the test's `ApplicationContext`.

If the algorithms used by `SqlScriptsTestExecutionListener` to detect a `DataSource` and `PlatformTransactionManager` and infer the transaction semantics do not suit your needs, you may specify explicit names via the `dataSource` and `transactionManager` attributes of `@SqlConfig`. Furthermore, the transaction propagation behavior can be controlled via the `transactionMode` attribute of `@SqlConfig` — for example, if scripts should be executed in an isolated transaction. Although a thorough discussion of all supported options for transaction management with `@Sql` is beyond the scope of this reference manual, the javadocs for `@SqlConfig` and `SqlScriptsTestExecutionListener` provide detailed information, and the following example demonstrates a typical testing scenario using JUnit Jupiter and transactional tests with `@Sql`. Note that there is no need to clean up the database after the `usersTest()` method is executed since any changes made to the database (either within the test method or within the `/test-data.sql` script) will be automatically rolled back by the `TransactionalTestExecutionListener` (see transaction management for details).

```
@SpringJUnitConfig(TestDatabaseConfig.class)
@Transactional
class TransactionalSqlScriptsTests {

    final JdbcTemplate jdbcTemplate;

    @Autowired
    TransactionalSqlScriptsTests(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Test
    @Sql("/test-data.sql")
    void usersTest() {
        // verify state in test database:
        assertNumUsers(2);
        // execute code that uses the test data...
    }

    int countRowsInTable(String tableName) {
        return JdbcTestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    void assertNumUsers(int expected) {
        assertEquals(expected, countRowsInTable("user"),
            "Number of rows in the [user] table.");
    }
}
```

3.5.9. Parallel test execution

Spring Framework 5.0 introduces basic support for executing tests in parallel within a single JVM when using the *Spring TestContext Framework*. In general this means that most test classes or test methods can be executed in parallel without any changes to test code or configuration.

For details on how to set up parallel test execution, consult the documentation for your testing framework, build tool, or IDE.

Keep in mind that the introduction of concurrency into your test suite can result in unexpected side effects, strange runtime behavior, and tests that only fail intermittently or seemingly randomly. The Spring Team therefore provides the following general guidelines for when *not* to execute tests in parallel.

Do not execute tests in parallel if:

- Tests make use of Spring's `@DirtiesContext` support.
- Tests make use of JUnit 4's `@FixMethodOrder` support or any testing framework feature that is designed to ensure that test methods execute in a particular order. Note, however, that this does

not apply if entire test classes are executed in parallel.

- Tests change the state of shared services or systems such as a database, message broker, filesystem, etc. This applies to both in-memory and external systems.

If parallel test execution fails with an exception stating that the `ApplicationContext` for the current test is no longer active, this typically means that the `ApplicationContext` was removed from the `ContextCache` in a different thread.

This may be due to the use of `@DirtiesContext` or due to automatic eviction from the `ContextCache`. If `@DirtiesContext` is the culprit, you will either need to find a way to avoid using `@DirtiesContext` or exclude such tests from parallel execution. If the maximum size of the `ContextCache` has been exceeded, you can increase the maximum size of the cache. See the discussion on context caching for details.

Parallel test execution in the Spring TestContext Framework is only possible if the underlying `TestContext` implementation provides a *copy constructor* as explained in the javadocs for `TestContext`. The `DefaultTestContext` used in Spring provides such a constructor; however, if you use a third-party library that provides a custom `TestContext` implementation, you will need to verify if it is suitable for parallel test execution.

3.5.10. TestContext Framework support classes

Spring JUnit 4 Runner

The *Spring TestContext Framework* offers full integration with JUnit 4 through a custom runner (supported on JUnit 4.12 or higher). By annotating test classes with `@RunWith(SpringJUnit4ClassRunner.class)` or the shorter `@RunWith(SpringRunner.class)` variant, developers can implement standard JUnit 4 based unit and integration tests and simultaneously reap the benefits of the TestContext framework such as support for loading application contexts, dependency injection of test instances, transactional test method execution, and so on. If you would like to use the Spring TestContext Framework with an alternative runner such as JUnit 4's `Parameterized` or third-party runners such as the `MockitoJUnitRunner`, you may optionally use Spring's support for JUnit rules instead.

The following code listing displays the minimal requirements for configuring a test class to run with the custom Spring `Runner`. `@TestExecutionListeners` is configured with an empty list in order to disable the default listeners, which otherwise would require an `ApplicationContext` to be configured through `@ContextConfiguration`.

```
@RunWith(SpringRunner.class)
@TestExecutionListeners({})
public class SimpleTest {

    @Test
    public void testMethod() {
        // execute test logic...
    }
}
```

Spring JUnit 4 Rules

The `org.springframework.test.context.junit4.rules` package provides the following JUnit 4 rules (supported on JUnit 4.12 or higher).

- `SpringClassRule`
- `SpringMethodRule`

`SpringClassRule` is a JUnit `TestRule` that supports *class-level* features of the *Spring TestContext Framework*; whereas, `SpringMethodRule` is a JUnit `MethodRule` that supports instance-level and method-level features of the *Spring TestContext Framework*.

In contrast to the `SpringRunner`, Spring's rule-based JUnit support has the advantage that it is independent of any `org.junit.runner.Runner` implementation and can therefore be combined with existing alternative runners like JUnit 4's `Parameterized` or third-party runners such as the `MockitoJUnitRunner`.

In order to support the full functionality of the TestContext framework, a `SpringClassRule` must be combined with a `SpringMethodRule`. The following example demonstrates the proper way to declare these rules in an integration test.

```
// Optionally specify a non-Spring Runner via @RunWith(...)
@ContextConfiguration
public class IntegrationTest {

    @ClassRule
    public static final SpringClassRule springClassRule = new SpringClassRule();

    @Rule
    public final SpringMethodRule springMethodRule = new SpringMethodRule();

    @Test
    public void testMethod() {
        // execute test logic...
    }
}
```

JUnit 4 support classes

The `org.springframework.test.context.junit4` package provides the following support classes for JUnit 4 based test cases (supported on JUnit 4.12 or higher).

- `AbstractJUnit4SpringContextTests`
- `AbstractTransactionalJUnit4SpringContextTests`

`AbstractJUnit4SpringContextTests` is an abstract base test class that integrates the *Spring TestContext Framework* with explicit `ApplicationContext` testing support in a JUnit 4 environment. When you extend `AbstractJUnit4SpringContextTests`, you can access a `protected applicationContext` instance variable that can be used to perform explicit bean lookups or to test the state of the context as a whole.

`AbstractTransactionalJUnit4SpringContextTests` is an abstract *transactional* extension of `AbstractJUnit4SpringContextTests` that adds some convenience functionality for JDBC access. This class expects a `javax.sql.DataSource` bean and a `PlatformTransactionManager` bean to be defined in the `ApplicationContext`. When you extend `AbstractTransactionalJUnit4SpringContextTests` you can access a `protected jdbcTemplate` instance variable that can be used to execute SQL statements to query the database. Such queries can be used to confirm database state both *prior to* and *after* execution of database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid false positives. As mentioned in JDBC Testing Support, `AbstractTransactionalJUnit4SpringContextTests` also provides convenience methods which delegate to methods in `JdbcTestUtils` using the aforementioned `jdbcTemplate`. Furthermore, `AbstractTransactionalJUnit4SpringContextTests` provides an `executeSqlScript(..)` method for executing SQL scripts against the configured `DataSource`.

These classes are a convenience for extension. If you do not want your test classes to be tied to a Spring-specific class hierarchy, you can configure your own custom test classes by using `@RunWith(SpringRunner.class)` or Spring's JUnit rules.

SpringExtension for JUnit Jupiter

The *Spring TestContext Framework* offers full integration with the *JUnit Jupiter* testing framework introduced in JUnit 5. By annotating test classes with `@ExtendWith(SpringExtension.class)`, developers can implement standard JUnit Jupiter based unit and integration tests and simultaneously reap the benefits of the TestContext framework such as support for loading application contexts, dependency injection of test instances, transactional test method execution, and so on.

Furthermore, thanks to the rich extension API in JUnit Jupiter, Spring is able to provide the following features above and beyond the feature set that Spring supports for JUnit 4 and TestNG.

- Dependency injection for test constructors, test methods, and test lifecycle callback methods
  - See Dependency Injection with the SpringExtension for further details.
- Powerful support for *conditional test execution* based on SpEL expressions, environment variables, system properties, etc.

    - See the documentation for `@EnabledIf` and `@DisabledIf` in Spring JUnit Jupiter Testing Annotations for further details and examples.
- Custom *composed annotations* that combine annotations from Spring **and** JUnit Jupiter.
    - See the `@TransactionalDevTestConfig` and `@TransactionalIntegrationTest` examples in Meta-Annotation Support for Testing for further details.

The following code listing demonstrates how to configure a test class to use the `SpringExtension` in conjunction with `@ContextConfiguration`.

```
// Instructs JUnit Jupiter to extend the test with Spring support.
@ExtendWith(SpringExtension.class)
// Instructs Spring to load an ApplicationContext from TestConfig.class
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // execute test logic...
    }
}
```

Since annotations in JUnit 5 can also be used as meta-annotations, Spring is able to provide `@SpringJUnitConfig` and `@SpringJUnitWebConfig` *composed annotations* to simplify the configuration of the test `ApplicationContext` and JUnit Jupiter.

For example, the following example uses `@SpringJUnitConfig` to reduce the amount of configuration used in the previous example.

```
// Instructs Spring to register the SpringExtension with JUnit
// Jupiter and load an ApplicationContext from TestConfig.class
@SpringJUnitConfig(TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // execute test logic...
    }
}
```

Similarly, the following example uses `@SpringJUnitWebConfig` to create a `WebApplicationContext` for use with JUnit Jupiter.

```
// Instructs Spring to register the SpringExtension with JUnit
// Jupiter and load a WebApplicationContext from TestWebConfig.class
@SpringJUnitWebConfig(TestWebConfig.class)
class SimpleWebTests {

    @Test
    void testMethod() {
        // execute test logic...
    }
}
```

See the documentation for `@SpringJUnitConfig` and `@SpringJUnitWebConfig` in Spring JUnit Jupiter Testing Annotations for further details.

Dependency Injection with the SpringExtension

The `SpringExtension` implements the `ParameterResolver` extension API from JUnit Jupiter which allows Spring to provide dependency injection for test constructors, test methods, and test lifecycle callback methods.

Specifically, the `SpringExtension` is able to inject dependencies from the test's `ApplicationContext` into test constructors and methods annotated with `@BeforeAll`, `@AfterAll`, `@BeforeEach`, `@AfterEach`, `@Test`, `@RepeatedTest`, `@ParameterizedTest`, etc.

Constructor Injection

If a parameter in a constructor for a JUnit Jupiter test class is of type `ApplicationContext` (or a sub-type thereof) or is annotated or meta-annotated with `@Autowired`, `@Qualifier`, or `@Value`, Spring will inject the value for that specific parameter with the corresponding bean from the test's `ApplicationContext`. A test constructor can also be directly annotated with `@Autowired` if all of the parameters should be supplied by Spring.

If the constructor for a test class is itself annotated with `@Autowired`, Spring will assume the responsibility for resolving **all** parameters in the constructor. Consequently, no other `ParameterResolver` registered with JUnit Jupiter will be able to resolve parameters for such a constructor.

In the following example, Spring will inject the `OrderService` bean from the `ApplicationContext` loaded from `TestConfig.class` into the `OrderServiceIntegrationTests` constructor. Note as well that this feature allows test dependencies to be `final` and therefore *immutable*.

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    private final OrderService orderService;

    @Autowired
    OrderServiceIntegrationTests(OrderService orderService) {
        this.orderService = orderService.
    }

    // tests that use the injected OrderService
}
```

Method Injection

If a parameter in a JUnit Jupiter test method or test lifecycle callback method is of type `ApplicationContext` (or a sub-type thereof) or is annotated or meta-annotated with `@Autowired`, `@Qualifier`, or `@Value`, Spring will inject the value for that specific parameter with the corresponding bean from the test's `ApplicationContext`.

In the following example, Spring will inject the `OrderService` from the `ApplicationContext` loaded from `TestConfig.class` into the `deleteOrder()` test method.

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    @Test
    void deleteOrder(@Autowired OrderService orderService) {
        // use orderService from the test's ApplicationContext
    }
}
```

Due to the robustness of the `ParameterResolver` support in JUnit Jupiter, it is also possible to have multiple dependencies injected into a single method not only from Spring but also from JUnit Jupiter itself or other third-party extensions.

The following example demonstrates how to have both Spring and JUnit Jupiter inject dependencies into the `placeOrderRepeatedly()` test method simultaneously. Note that the use of `@RepeatedTest` from JUnit Jupiter allows the test method to gain access to the `RepetitionInfo`.

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    @RepeatedTest(10)
    void placeOrderRepeatedly(RepetitionInfo repetitionInfo,
            @Autowired OrderService orderService) {

        // use orderService from the test's ApplicationContext
        // and repetitionInfo from JUnit Jupiter
    }
}
```

TestNG support classes

The `org.springframework.test.context.testng` package provides the following support classes for TestNG based test cases.

- `AbstractTestNGSpringContextTests`
- `AbstractTransactionalTestNGSpringContextTests`

`AbstractTestNGSpringContextTests` is an abstract base test class that integrates the *Spring TestContext Framework* with explicit `ApplicationContext` testing support in a TestNG environment. When you extend `AbstractTestNGSpringContextTests`, you can access a `protected applicationContext` instance variable that can be used to perform explicit bean lookups or to test the state of the context as a whole.

`AbstractTransactionalTestNGSpringContextTests` is an abstract *transactional* extension of `AbstractTestNGSpringContextTests` that adds some convenience functionality for JDBC access. This class expects a `javax.sql.DataSource` bean and a `PlatformTransactionManager` bean to be defined in the `ApplicationContext`. When you extend `AbstractTransactionalTestNGSpringContextTests` you can access a `protected jdbcTemplate` instance variable that can be used to execute SQL statements to query the database. Such queries can be used to confirm database state both *prior to* and *after* execution of database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid false positives. As mentioned in JDBC Testing Support, `AbstractTransactionalTestNGSpringContextTests` also provides convenience methods which delegate to methods in `JdbcTestUtils` using the aforementioned `jdbcTemplate`. Furthermore, `AbstractTransactionalTestNGSpringContextTests` provides an `executeSqlScript(..)` method for executing SQL scripts against the configured `DataSource`.

These classes are a convenience for extension. If you do not want your test classes to be tied to a Spring-specific class hierarchy, you can configure your own custom test classes by using `@ContextConfiguration`, `@TestExecutionListeners`, and so on, and by manually instrumenting your test class with a `TestContextManager`. See the source code of `AbstractTestNGSpringContextTests` for an example of how to instrument your test class.

3.6. Spring MVC Test Framework

The *Spring MVC Test framework* provides first class support for testing Spring MVC code using a fluent API that can be used with JUnit, TestNG, or any other testing framework. It's built on the Servlet API mock objects from the `spring-test` module and hence does *not* use a running Servlet container. It uses the `DispatcherServlet` to provide full Spring MVC runtime behavior and provides support for loading actual Spring configuration with the *TestContext framework* in addition to a standalone mode in which controllers may be instantiated manually and tested one at a time.

Spring MVC Test also provides client-side support for testing code that uses the `RestTemplate`. Client-side tests mock the server responses and also do *not* use a running server.

Spring Boot provides an option to write full, end-to-end integration tests that include a running server. If this is your goal please have a look at the Spring Boot reference page. For more information on the differences between out-of-container and end-to-end integration tests, see Differences between Out-of-Container and End-to-End Integration Tests.

3.6.1. Server-Side Tests

It's easy to write a plain unit test for a Spring MVC controller using JUnit or TestNG: simply instantiate the controller, inject it with mocked or stubbed dependencies, and call its methods passing `MockHttpServletRequest`, `MockHttpServletResponse`, etc., as necessary. However, when writing such a unit test, much remains untested: for example, request mappings, data binding, type conversion, validation, and much more. Furthermore, other controller methods such as `@InitBinder`, `@ModelAttribute`, and `@ExceptionHandler` may also be invoked as part of the request processing lifecycle.

The goal of *Spring MVC Test* is to provide an effective way for testing controllers by performing requests and generating responses through the actual `DispatcherServlet`.

Spring MVC Test builds on the familiar "mock" implementations of the Servlet API available in the `spring-test` module. This allows performing requests and generating responses without the need for running in a Servlet container. For the most part everything should work as it does at runtime with a few notable exceptions as explained in Differences between Out-of-Container and End-to-End Integration Tests. Here is a JUnit Jupiter based example of using Spring MVC Test:

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringJUnitWebConfig(locations = "test-servlet-context.xml")
class ExampleTests {

    private MockMvc mockMvc;

    @BeforeEach
    void setup(WebApplicationContext wac) {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

    @Test
    void getAccount() throws Exception {
        this.mockMvc.perform(get("/accounts/1")
                .accept(MediaType.parseMediaType("application/json;charset=UTF-8")))
            .andExpect(status().isOk())
            .andExpect(content().contentType("application/json"))
            .andExpect(jsonPath("$.name").value("Lee"));
    }
}
```

The above test relies on the `WebApplicationContext` support of the *TestContext framework* for loading Spring configuration from an XML configuration file located in the same package as the test class, but Java-based and Groovy-based configuration are also supported. See these sample tests.

The `MockMvc` instance is used to perform a `GET` request to `"/accounts/1"` and verify that the resulting response has status 200, the content type is `"application/json"`, and the response body has a JSON property called "name" with the value "Lee". The `jsonPath` syntax is supported through the Jayway JsonPath project. There are lots of other options for verifying the result of the performed request that will be discussed below.

Static Imports

The fluent API in the example above requires a few static imports such as `MockMvcRequestBuilders.*`, `MockMvcResultMatchers.*`, and `MockMvcBuilders.*`. An easy way to find these classes is to search for types matching *"MockMvc*"*. If using Eclipse, be sure to add them as "favorite static members" in the Eclipse preferences under *Java → Editor → Content Assist → Favorites*. That will allow use of content assist after typing the first character of the static method name. Other IDEs (e.g. IntelliJ) may not require any additional configuration. Just check the support for code completion on static members.

Setup Choices

There are two main options for creating an instance of `MockMvc`. The first is to load Spring MVC configuration through the *TestContext framework*, which loads the Spring configuration and injects a `WebApplicationContext` into the test to use to build a `MockMvc` instance:

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration("my-servlet-context.xml")
public class MyWebTests {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    }

    // ...

}
```

The second is to simply create a controller instance manually without loading Spring configuration. Instead basic default configuration, roughly comparable to that of the MVC JavaConfig or the MVC namespace, is automatically created and can be customized to a degree:

```
public class MyWebTests {

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.standaloneSetup(new AccountController()).build
();
    }

    // ...

}
```

Which setup option should you use?

The *"webAppContextSetup"* loads your actual Spring MVC configuration resulting in a more complete integration test. Since the *TestContext framework* caches the loaded Spring configuration, it helps keep tests running fast, even as you introduce more tests in your test suite. Furthermore, you can inject mock services into controllers through Spring configuration in order to remain focused on testing the web layer. Here is an example of declaring a mock service with Mockito:

```
<bean id="accountService" class="org.mockito.Mockito" factory-method="mock">
    <constructor-arg value="org.example.AccountService"/>
</bean>
```

You can then inject the mock service into the test in order set up and verify expectations:

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration("test-servlet-context.xml")
public class AccountTests {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Autowired
    private AccountService accountService;

    // ...

}
```

The *"standaloneSetup"* on the other hand is a little closer to a unit test. It tests one controller at a time: the controller can be injected with mock dependencies manually, and it doesn't involve loading Spring configuration. Such tests are more focused on style and make it easier to see which controller is being tested, whether any specific Spring MVC configuration is required to work, and so on. The "standaloneSetup" is also a very convenient way to write ad-hoc tests to verify specific behavior or to debug an issue.

Just like with any "integration vs. unit testing" debate, there is no right or wrong answer. However, using the "standaloneSetup" does imply the need for additional "webAppContextSetup" tests in order to verify your Spring MVC configuration. Alternatively, you may choose to write all tests with "webAppContextSetup" in order to always test against your actual Spring MVC configuration.

Setup Features

No matter which MockMvc builder you use all `MockMvcBuilder` implementations provide some common and very useful features. For example you can declare an `Accept` header for all requests and expect a status of 200 as well as a `Content-Type` header in all responses as follows:

```
// static import of MockMvcBuilders.standaloneSetup

MockMvc mockMvc = standaloneSetup(new MusicController())
        .defaultRequest(get("/").accept(MediaType.APPLICATION_JSON))
        .alwaysExpect(status().isOk())
        .alwaysExpect(content().contentType("application/json;charset=UTF-8"))
        .build();
```

In addition 3rd party frameworks (and applications) may pre-package setup instructions like the ones through a `MockMvcConfigurer`. The Spring Framework has one such built-in implementation that helps to save and re-use the HTTP session across requests. It can be used as follows:

```
// static import of SharedHttpSessionConfigurer.sharedHttpSession

MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new TestController())
        .apply(sharedHttpSession())
        .build();

// Use mockMvc to perform requests...
```

See `ConfigurableMockMvcBuilder` for a list of all MockMvc builder features or use the IDE to explore the available options.

Performing Requests

It's easy to perform requests using any HTTP method:

```
mockMvc.perform(post("/hotels/{id}", 42).accept(MediaType.APPLICATION_JSON));
```

You can also perform file upload requests that internally use `MockMultipartHttpServletRequest` so that there is no actual parsing of a multipart request but rather you have to set it up:

```
mockMvc.perform(multipart("/doc").file("a1", "ABC".getBytes("UTF-8")));
```

You can specify query parameters in URI template style:

```
mockMvc.perform(get("/hotels?foo={foo}", "bar"));
```

Or you can add Servlet request parameters representing either query of form parameters:

```
mockMvc.perform(get("/hotels").param("foo", "bar"));
```

If application code relies on Servlet request parameters and doesn't check the query string

explicitly (as is most often the case) then it doesn't matter which option you use. Keep in mind however that query params provided with the URI template will be decoded while request parameters provided through the `param(…)` method are expected to already be decoded.

In most cases it's preferable to leave out the context path and the Servlet path from the request URI. If you must test with the full request URI, be sure to set the `contextPath` and `servletPath` accordingly so that request mappings will work:

```
mockMvc.perform(get("/app/main/hotels/{id}").contextPath("/app").servletPath("/main"))
```

Looking at the above example, it would be cumbersome to set the contextPath and servletPath with every performed request. Instead you can set up default request properties:

```
public class MyWebTests {

    private MockMvc mockMvc;

    @Before
    public void setup() {
        mockMvc = standaloneSetup(new AccountController())
            .defaultRequest(get("/")
            .contextPath("/app").servletPath("/main")
            .accept(MediaType.APPLICATION_JSON)).build();
    }
```

The above properties will affect every request performed through the `MockMvc` instance. If the same property is also specified on a given request, it overrides the default value. That is why the HTTP method and URI in the default request don't matter since they must be specified on every request.

Defining Expectations

Expectations can be defined by appending one or more `.andExpect(..)` calls after performing a request:

```
mockMvc.perform(get("/accounts/1")).andExpect(status().isOk());
```

`MockMvcResultMatchers.*` provides a number of expectations, some of which are further nested with more detailed expectations.

Expectations fall in two general categories. The first category of assertions verifies properties of the response: for example, the response status, headers, and content. These are the most important results to assert.

The second category of assertions goes beyond the response. These assertions allow one to inspect Spring MVC specific aspects such as which controller method processed the request, whether an exception was raised and handled, what the content of the model is, what view was selected, what flash attributes were added, and so on. They also allow one to inspect Servlet specific aspects such

as request and session attributes.

The following test asserts that binding or validation failed:

```
mockMvc.perform(post("/persons"))
    .andExpect(status().isOk())
    .andExpect(model().attributeHasErrors("person"));
```

Many times when writing tests, it's useful to *dump* the results of the performed request. This can be done as follows, where `print()` is a static import from `MockMvcResultHandlers`:

```
mockMvc.perform(post("/persons"))
    .andDo(print())
    .andExpect(status().isOk())
    .andExpect(model().attributeHasErrors("person"));
```

As long as request processing does not cause an unhandled exception, the `print()` method will print all the available result data to `System.out`. Spring Framework 4.2 introduced a `log()` method and two additional variants of the `print()` method, one that accepts an `OutputStream` and one that accepts a `Writer`. For example, invoking `print(System.err)` will print the result data to `System.err`; while invoking `print(myWriter)` will print the result data to a custom writer. If you would like to have the result data *logged* instead of printed, simply invoke the `log()` method which will log the result data as a single `DEBUG` message under the `org.springframework.test.web.servlet.result` logging category.

In some cases, you may want to get direct access to the result and verify something that cannot be verified otherwise. This can be achieved by appending `.andReturn()` after all other expectations:

```
MvcResult mvcResult = mockMvc.perform(post("/persons")).andExpect(status().isOk())
.andReturn();
// ...
```

If all tests repeat the same expectations you can set up common expectations once when building the `MockMvc` instance:

```
standaloneSetup(new SimpleController())
    .alwaysExpect(status().isOk())
    .alwaysExpect(content().contentType("application/json;charset=UTF-8"))
    .build()
```

Note that common expectations are *always* applied and cannot be overridden without creating a separate `MockMvc` instance.

When JSON response content contains hypermedia links created with Spring HATEOAS, the resulting links can be verified using JsonPath expressions:

```
mockMvc.perform(get("/people").accept(MediaType.APPLICATION_JSON))
    .andExpect(jsonPath("$.links[?(@.rel == 'self')].href").value(
"http://localhost:8080/people"));
```

When XML response content contains hypermedia links created with Spring HATEOAS, the resulting links can be verified using XPath expressions:

```
Map<String, String> ns = Collections.singletonMap("ns", "http://www.w3.org/2005/Atom"
);
mockMvc.perform(get("/handle").accept(MediaType.APPLICATION_XML))
    .andExpect(xpath("/person/ns:link[@rel='self']/@href", ns).string(
"http://localhost:8080/people"));
```

Filter Registrations

When setting up a `MockMvc` instance, you can register one or more Servlet `Filter` instances:

```
mockMvc = standaloneSetup(new PersonController()).addFilters(new
CharacterEncodingFilter()).build();
```

Registered filters will be invoked through via the `MockFilterChain` from `spring-test`, and the last filter will delegate to the `DispatcherServlet`.

Differences between Out-of-Container and End-to-End Integration Tests

As mentioned earlier *Spring MVC Test* is built on the Servlet API mock objects from the `spring-test` module and does not use a running Servlet container. Therefore there are some important differences compared to full end-to-end integration tests with an actual client and server running.

The easiest way to think about this is starting with a blank `MockHttpServletRequest`. Whatever you add to it is what the request will be. Things that may catch you by surprise are that there is no context path by default, no `jsessionid` cookie, no forwarding, error, or async dispatches, and therefore no actual JSP rendering. Instead, "forwarded" and "redirected" URLs are saved in the `MockHttpServletResponse` and can be asserted with expectations.

This means if you are using JSPs you can verify the JSP page to which the request was forwarded, but there won't be any HTML rendered. In other words, the JSP will not be *invoked*. Note however that all other rendering technologies which don't rely on forwarding such as Thymeleaf and Freemarker will render HTML to the response body as expected. The same is true for rendering JSON, XML, and other formats via `@ResponseBody` methods.

Alternatively you may consider the full end-to-end integration testing support from Spring Boot via `@WebIntegrationTest`. See the Spring Boot reference.

There are pros and cons for each approach. The options provided in *Spring MVC Test* are different stops on the scale from classic unit testing to full integration testing. To be certain, none of the options in Spring MVC Test fall under the category of classic unit testing, but they *are* a little closer

to it. For example, you can isolate the web layer by injecting mocked services into controllers, in which case you're testing the web layer only through the `DispatcherServlet` but with actual Spring configuration, just like you might test the data access layer in isolation from the layers above. Or you can use the standalone setup focusing on one controller at a time and manually providing the configuration required to make it work.

Another important distinction when using *Spring MVC Test* is that conceptually such tests are on the *inside* of the server-side so you can check what handler was used, if an exception was handled with a HandlerExceptionResolver, what the content of the model is, what binding errors there were, etc. That means it's easier to write expectations since the server is not a black box as it is when testing it through an actual HTTP client. This is generally an advantage of classic unit testing, that it's easier to write, reason about, and debug but does not replace the need for full integration tests. At the same time it's important not to lose sight of the fact that the response is the most important thing to check. In short, there is room here for multiple styles and strategies of testing even within the same project.

Further Server-Side Test Examples

The framework's own tests include many sample tests intended to demonstrate how to use Spring MVC Test. Browse these examples for further ideas. Also the spring-mvc-showcase has full test coverage based on Spring MVC Test.

3.6.2. HtmlUnit Integration

Spring provides integration between MockMvc and HtmlUnit. This simplifies performing end-to-end testing when using HTML based views. This integration enables developers to:

- Easily test HTML pages using tools such as HtmlUnit, WebDriver, & Geb without the need to deploy to a Servlet container
- Test JavaScript within pages
- Optionally test using mock services to speed up testing
- Share logic between in-container end-to-end tests and out-of-container integration tests

`MockMvc` works with templating technologies that do not rely on a Servlet Container (e.g., Thymeleaf, FreeMarker, etc.), but it does not work with JSPs since they rely on the Servlet container.

Why HtmlUnit Integration?

The most obvious question that comes to mind is, "Why do I need this?". The answer is best found by exploring a very basic sample application. Assume you have a Spring MVC web application that supports CRUD operations on a `Message` object. The application also supports paging through all messages. How would you go about testing it?

With Spring MVC Test, we can easily test if we are able to create a `Message`.

```
MockHttpServletRequestBuilder createMessage = post("/messages/")
        .param("summary", "Spring Rocks")
        .param("text", "In case you didn't know, Spring Rocks!");

mockMvc.perform(createMessage)
        .andExpect(status().is3xxRedirection())
        .andExpect(redirectedUrl("/messages/123"));
```

What if we want to test our form view that allows us to create the message? For example, assume our form looks like the following snippet:

```
<form id="messageForm" action="/messages/" method="post">
    <div class="pull-right"><a href="/messages/">Messages</a></div>

    <label for="summary">Summary</label>
    <input type="text" class="required" id="summary" name="summary" value="" />

    <label for="text">Message</label>
    <textarea id="text" name="text"></textarea>

    <div class="form-actions">
        <input type="submit" value="Create" />
    </div>
</form>
```

How do we ensure that our form will produce the correct request to create a new message? A naive attempt would look like this:

```
mockMvc.perform(get("/messages/form"))
        .andExpect(xpath("//input[@name='summary']").exists())
        .andExpect(xpath("//textarea[@name='text']").exists());
```

This test has some obvious drawbacks. If we update our controller to use the parameter `message` instead of `text`, our form test would continue to pass even though the HTML form is out of synch with the controller. To resolve this we can combine our two tests.

```
String summaryParamName = "summary";
String textParamName = "text";
mockMvc.perform(get("/messages/form"))
        .andExpect(xpath("//input[@name='" + summaryParamName + "']").exists())
        .andExpect(xpath("//textarea[@name='" + textParamName + "']").exists());

MockHttpServletRequestBuilder createMessage = post("/messages/")
        .param(summaryParamName, "Spring Rocks")
        .param(textParamName, "In case you didn't know, Spring Rocks!");

mockMvc.perform(createMessage)
        .andExpect(status().is3xxRedirection())
        .andExpect(redirectedUrl("/messages/123"));
```

This would reduce the risk of our test incorrectly passing, but there are still some problems.

- What if we have multiple forms on our page? Admittedly we could update our xpath expressions, but they get more complicated the more factors we take into account (Are the fields the correct type? Are the fields enabled? etc.).
- Another issue is that we are doing double the work we would expect. We must first verify the view, and then we submit the view with the same parameters we just verified. Ideally this could be done all at once.
- Finally, there are some things that we still cannot account for. For example, what if the form has JavaScript validation that we wish to test as well?

The overall problem is that testing a web page does not involve a single interaction. Instead, it is a combination of how the user interacts with a web page and how that web page interacts with other resources. For example, the result of a form view is used as the input to a user for creating a message. In addition, our form view may potentially utilize additional resources which impact the behavior of the page, such as JavaScript validation.

Integration testing to the rescue?

To resolve the issues above we could perform end-to-end integration testing, but this has some obvious drawbacks. Consider testing the view that allows us to page through the messages. We might need the following tests.

- Does our page display a notification to the user indicating that no results are available when the messages are empty?
- Does our page properly display a single message?
- Does our page properly support paging?

To set up these tests, we would need to ensure our database contained the proper messages in it. This leads to a number of additional challenges.

- Ensuring the proper messages are in the database can be tedious; consider foreign key constraints.

- Testing can become slow since each test would need to ensure that the database is in the correct state.
- Since our database needs to be in a specific state, we cannot run tests in parallel.
- Performing assertions on things like auto-generated ids, timestamps, etc. can be difficult.

These challenges do not mean that we should abandon end-to-end integration testing altogether. Instead, we can reduce the number of end-to-end integration tests by refactoring our detailed tests to use mock services which will execute much faster, more reliably, and without side effects. We can then implement a small number of *true* end-to-end integration tests that validate simple workflows to ensure that everything works together properly.

Enter HtmlUnit Integration

So how can we achieve a balance between testing the interactions of our pages and still retain good performance within our test suite? The answer is: "By integrating MockMvc with HtmlUnit."

HtmlUnit Integration Options

There are a number of ways to integrate `MockMvc` with HtmlUnit.

- MockMvc and HtmlUnit: Use this option if you want to use the raw HtmlUnit libraries.
- MockMvc and WebDriver: Use this option to ease development and reuse code between integration and end-to-end testing.
- MockMvc and Geb: Use this option if you would like to use Groovy for testing, ease development, and reuse code between integration and end-to-end testing.

MockMvc and HtmlUnit

This section describes how to integrate `MockMvc` and HtmlUnit. Use this option if you want to use the raw HtmlUnit libraries.

MockMvc and HtmlUnit Setup

First, make sure that you have included a test dependency on `net.sourceforge.htmlunit:htmlunit`. In order to use HtmlUnit with Apache HttpComponents 4.5+, you will need to use HtmlUnit 2.18 or higher.

We can easily create an HtmlUnit `WebClient` that integrates with `MockMvc` using the `MockMvcWebClientBuilder` as follows.

```
@Autowired
WebApplicationContext context;

WebClient webClient;

@Before
public void setup() {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build();
}
```

This is a simple example of using `MockMvcWebClientBuilder`. For advanced usage see Advanced MockMvcWebClientBuilder

This will ensure that any URL referencing `localhost` as the server will be directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL will be requested using a network connection as normal. This allows us to easily test the use of CDNs.

MockMvc and HtmlUnit Usage

Now we can use HtmlUnit as we normally would, but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following.

```
HtmlPage createMsgFormPage = webClient.getPage("http://localhost/messages/form");
```

The default context path is "". Alternatively, we can specify the context path as illustrated in Advanced MockMvcWebClientBuilder.

Once we have a reference to the `HtmlPage`, we can then fill out the form and submit it to create a message.

```
HtmlForm form = createMsgFormPage.getHtmlElementById("messageForm");
HtmlTextInput summaryInput = createMsgFormPage.getHtmlElementById("summary");
summaryInput.setValueAttribute("Spring Rocks");
HtmlTextArea textInput = createMsgFormPage.getHtmlElementById("text");
textInput.setText("In case you didn't know, Spring Rocks!");
HtmlSubmitInput submit = form.getOneHtmlElementByAttribute("input", "type", "submit");
HtmlPage newMessagePage = submit.click();
```

Finally, we can verify that a new message was created successfully. The following assertions use the AssertJ library.

```
assertThat(newMessagePage.getUrl().toString()).endsWith("/messages/123");
String id = newMessagePage.getHtmlElementById("id").getTextContent();
assertThat(id).isEqualTo("123");
String summary = newMessagePage.getHtmlElementById("summary").getTextContent();
assertThat(summary).isEqualTo("Spring Rocks");
String text = newMessagePage.getHtmlElementById("text").getTextContent();
assertThat(text).isEqualTo("In case you didn't know, Spring Rocks!");
```

This improves on our MockMvc test in a number of ways. First we no longer have to explicitly verify our form and then create a request that looks like the form. Instead, we request the form, fill it out, and submit it, thereby significantly reducing the overhead.

Another important factor is that HtmlUnit uses the Mozilla Rhino engine to evaluate JavaScript. This means that we can test the behavior of JavaScript within our pages as well!

Refer to the HtmlUnit documentation for additional information about using HtmlUnit.

Advanced MockMvcWebClientBuilder

In the examples so far, we have used `MockMvcWebClientBuilder` in the simplest way possible, by building a `WebClient` based on the `WebApplicationContext` loaded for us by the Spring TestContext Framework. This approach is repeated here.

```
@Autowired
WebApplicationContext context;

WebClient webClient;

@Before
public void setup() {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build();
}
```

We can also specify additional configuration options.

```
WebClient webClient;

@Before
public void setup() {
    webClient = MockMvcWebClientBuilder
        // demonstrates applying a MockMvcConfigurer (Spring Security)
        .webAppContextSetup(context, springSecurity())
        // for illustration only - defaults to ""
        .contextPath("")
        // By default MockMvc is used for localhost only;
        // the following will use MockMvc for example.com and example.org as well
        .useMockMvcForHosts("example.com","example.org")
        .build();
}
```

As an alternative, we can perform the exact same setup by configuring the `MockMvc` instance separately and supplying it to the `MockMvcWebClientBuilder` as follows.

```
MockMvc mockMvc = MockMvcBuilders
        .webAppContextSetup(context)
        .apply(springSecurity())
        .build();

webClient = MockMvcWebClientBuilder
        .mockMvcSetup(mockMvc)
        // for illustration only - defaults to ""
        .contextPath("")
        // By default MockMvc is used for localhost only;
        // the following will use MockMvc for example.com and example.org as well
        .useMockMvcForHosts("example.com","example.org")
        .build();
```

This is more verbose, but by building the `WebClient` with a `MockMvc` instance we have the full power of `MockMvc` at our fingertips.

For additional information on creating a `MockMvc` instance refer to Setup Choices.

MockMvc and WebDriver

In the previous sections, we have seen how to use `MockMvc` in conjunction with the raw HtmlUnit APIs. In this section, we will leverage additional abstractions within the Selenium WebDriver to make things even easier.

Why WebDriver and MockMvc?

We can already use HtmlUnit and `MockMvc`, so why would we want to use `WebDriver`? The Selenium `WebDriver` provides a very elegant API that allows us to easily organize our code. To better understand, let's explore an example.

Despite being a part of Selenium, WebDriver does not require a Selenium Server to run your tests.

Suppose we need to ensure that a message is created properly. The tests involve finding the HTML form input elements, filling them out, and making various assertions.

This approach results in numerous, separate tests because we want to test error conditions as well. For example, we want to ensure that we get an error if we fill out only part of the form. If we fill out the entire form, the newly created message should be displayed afterwards.

If one of the fields were named "summary", then we might have something like the following repeated in multiple places within our tests.

```
HtmlTextInput summaryInput = currentPage.getHtmlElementById("summary");
summaryInput.setValueAttribute(summary);
```

So what happens if we change the `id` to "smmry"? Doing so would force us to update all of our tests to incorporate this change! Of course, this violates the *DRY Principle*; so we should ideally extract this code into its own method as follows.

```
public HtmlPage createMessage(HtmlPage currentPage, String summary, String text) {
    setSummary(currentPage, summary);
    // ...
}

public void setSummary(HtmlPage currentPage, String summary) {
    HtmlTextInput summaryInput = currentPage.getHtmlElementById("summary");
    summaryInput.setValueAttribute(summary);
}
```

This ensures that we do not have to update all of our tests if we change the UI.

We might even take this a step further and place this logic within an Object that represents the `HtmlPage` we are currently on.

```
public class CreateMessagePage {

    final HtmlPage currentPage;

    final HtmlTextInput summaryInput;

    final HtmlSubmitInput submit;

    public CreateMessagePage(HtmlPage currentPage) {
        this.currentPage = currentPage;
        this.summaryInput = currentPage.getHtmlElementById("summary");
        this.submit = currentPage.getHtmlElementById("submit");
    }

    public <T> T createMessage(String summary, String text) throws Exception {
        setSummary(summary);

        HtmlPage result = submit.click();
        boolean error = CreateMessagePage.at(result);

        return (T) (error ? new CreateMessagePage(result) : new ViewMessagePage(
result));
    }

    public void setSummary(String summary) throws Exception {
        summaryInput.setValueAttribute(summary);
    }

    public static boolean at(HtmlPage page) {
        return "Create Message".equals(page.getTitleText());
    }
}
```

Formerly, this pattern is known as the Page Object Pattern. While we can certainly do this with HtmlUnit, WebDriver provides some tools that we will explore in the following sections to make this pattern much easier to implement.

MockMvc and WebDriver Setup

To use Selenium WebDriver with the Spring MVC Test framework, make sure that your project includes a test dependency on `org.seleniumhq.selenium:selenium-htmlunit-driver`.

We can easily create a Selenium `WebDriver` that integrates with `MockMvc` using the `MockMvcHtmlUnitDriverBuilder` as follows.

```
    @Autowired
    WebApplicationContext context;

    WebDriver driver;

    @Before
    public void setup() {
        driver = MockMvcHtmlUnitDriverBuilder
                .webAppContextSetup(context)
                .build();
}
```

This is a simple example of using `MockMvcHtmlUnitDriverBuilder`. For more advanced usage, refer to Advanced MockMvcHtmlUnitDriverBuilder

This will ensure that any URL referencing `localhost` as the server will be directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL will be requested using a network connection as normal. This allows us to easily test the use of CDNs.

MockMvc and WebDriver Usage

Now we can use WebDriver as we normally would, but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following.

```
CreateMessagePage page = CreateMessagePage.to(driver);
```

We can then fill out the form and submit it to create a message.

```
ViewMessagePage viewMessagePage =
        page.createMessage(ViewMessagePage.class, expectedSummary, expectedText);
```

This improves on the design of our HtmlUnit test by leveraging the *Page Object Pattern*. As we mentioned in Why WebDriver and MockMvc?, we can use the Page Object Pattern with HtmlUnit, but it is much easier with WebDriver. Let's take a look at our new `CreateMessagePage` implementation.

```
public class CreateMessagePage
        extends AbstractPage { ①

    ②
    private WebElement summary;
    private WebElement text;

    ③
    @FindBy(css = "input[type=submit]")
    private WebElement submit;

    public CreateMessagePage(WebDriver driver) {
        super(driver);
    }

    public <T> T createMessage(Class<T> resultPage, String summary, String details) {
        this.summary.sendKeys(summary);
        this.text.sendKeys(details);
        this.submit.click();
        return PageFactory.initElements(driver, resultPage);
    }

    public static CreateMessagePage to(WebDriver driver) {
        driver.get("http://localhost:9990/mail/messages/form");
        return PageFactory.initElements(driver, CreateMessagePage.class);
    }
}
```

① The first thing you will notice is that `CreateMessagePage` extends the `AbstractPage`. We won't go over the details of `AbstractPage`, but in summary it contains common functionality for all of our pages. For example, if our application has a navigational bar, global error messages, etc., this logic can be placed in a shared location.

② The next thing you will notice is that we have a member variable for each of the parts of the HTML page that we are interested in. These are of type `WebElement`. `WebDriver`'s PageFactory allows us to remove a lot of code from the HtmlUnit version of `CreateMessagePage` by automatically resolving each `WebElement`. The PageFactory#initElements(WebDriver,Class<T>) method will automatically resolve each `WebElement` by using the field name and looking it up by the `id` or `name` of the element within the HTML page.

③ We can use the @FindBy annotation to override the default lookup behavior. Our example demonstrates how to use the `@FindBy` annotation to look up our submit button using a css selector, **input[type=submit]**.

Finally, we can verify that a new message was created successfully. The following assertions use the FEST assertion library.

```
assertThat(viewMessagePage.getMessage()).isEqualTo(expectedMessage);
assertThat(viewMessagePage.getSuccess()).isEqualTo("Successfully created a new
message");
```

We can see that our `ViewMessagePage` allows us to interact with our custom domain model. For example, it exposes a method that returns a `Message` object.

```
public Message getMessage() throws ParseException {
    Message message = new Message();
    message.setId(getId());
    message.setCreated(getCreated());
    message.setSummary(getSummary());
    message.setText(getText());
    return message;
}
```

We can then leverage the rich domain objects in our assertions.

Lastly, don't forget to *close* the `WebDriver` instance when the test is complete.

```
@After
public void destroy() {
    if (driver != null) {
        driver.close();
    }
}
```

For additional information on using WebDriver, refer to the Selenium WebDriver documentation.

Advanced MockMvcHtmlUnitDriverBuilder

In the examples so far, we have used `MockMvcHtmlUnitDriverBuilder` in the simplest way possible, by building a `WebDriver` based on the `WebApplicationContext` loaded for us by the Spring TestContext Framework. This approach is repeated here.

```
@Autowired
WebApplicationContext context;

WebDriver driver;

@Before
public void setup() {
    driver = MockMvcHtmlUnitDriverBuilder
            .webAppContextSetup(context)
            .build();
}
```

We can also specify additional configuration options.

```
    WebDriver driver;

    @Before
    public void setup() {
        driver = MockMvcHtmlUnitDriverBuilder
                // demonstrates applying a MockMvcConfigurer (Spring Security)
                .webAppContextSetup(context, springSecurity())
                // for illustration only - defaults to ""
                .contextPath("")
                // By default MockMvc is used for localhost only;
                // the following will use MockMvc for example.com and example.org as
well
                .useMockMvcForHosts("example.com","example.org")
                .build();
}
```

As an alternative, we can perform the exact same setup by configuring the `MockMvc` instance separately and supplying it to the `MockMvcHtmlUnitDriverBuilder` as follows.

```
MockMvc mockMvc = MockMvcBuilders
        .webAppContextSetup(context)
        .apply(springSecurity())
        .build();

driver = MockMvcHtmlUnitDriverBuilder
        .mockMvcSetup(mockMvc)
        // for illustration only - defaults to ""
        .contextPath("")
        // By default MockMvc is used for localhost only;
        // the following will use MockMvc for example.com and example.org as well
        .useMockMvcForHosts("example.com","example.org")
        .build();
```

This is more verbose, but by building the `WebDriver` with a `MockMvc` instance we have the full power of `MockMvc` at our fingertips.

For additional information on creating a `MockMvc` instance refer to Setup Choices.

MockMvc and Geb

In the previous section, we saw how to use `MockMvc` with `WebDriver`. In this section, we will use Geb to make our tests even Groovy-er.

Why Geb and MockMvc?

Geb is backed by WebDriver, so it offers many of the same benefits that we get from WebDriver.

However, Geb makes things even easier by taking care of some of the boilerplate code for us.

MockMvc and Geb Setup

We can easily initialize a Geb `Browser` with a Selenium `WebDriver` that uses `MockMvc` as follows.

```
def setup() {
    browser.driver = MockMvcHtmlUnitDriverBuilder
        .webAppContextSetup(context)
        .build()
}
```

This is a simple example of using `MockMvcHtmlUnitDriverBuilder`. For more advanced usage, refer to Advanced MockMvcHtmlUnitDriverBuilder

This will ensure that any URL referencing `localhost` as the server will be directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL will be requested using a network connection as normal. This allows us to easily test the use of CDNs.

MockMvc and Geb Usage

Now we can use Geb as we normally would, but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following:

```
to CreateMessagePage
```

We can then fill out the form and submit it to create a message.

```
when:
form.summary = expectedSummary
form.text = expectedMessage
submit.click(ViewMessagePage)
```

Any unrecognized method calls or property accesses/references that are not found will be forwarded to the current page object. This removes a lot of the boilerplate code we needed when using WebDriver directly.

As with direct WebDriver usage, this improves on the design of our HtmlUnit test by leveraging the *Page Object Pattern*. As mentioned previously, we can use the Page Object Pattern with HtmlUnit and WebDriver, but it is even easier with Geb. Let's take a look at our new Groovy-based `CreateMessagePage` implementation.

```
class CreateMessagePage extends Page {
    static url = 'messages/form'
    static at = { assert title == 'Messages : Create'; true }
    static content =  {
        submit { $('input[type=submit]') }
        form { $('form') }
        errors(required:false) { $('label.error, .alert-error')?.text() }
    }
}
```

The first thing you will notice is that our `CreateMessagePage` extends `Page`. We won't go over the details of `Page`, but in summary it contains common functionality for all of our pages. The next thing you will notice is that we define a URL in which this page can be found. This allows us to navigate to the page as follows.

```
to CreateMessagePage
```

We also have an `at` closure that determines if we are at the specified page. It should return `true` if we are on the correct page. This is why we can assert that we are on the correct page as follows.

```
then:
at CreateMessagePage
errors.contains('This field is required.')
```

We use an assertion in the closure, so that we can determine where things went wrong if we were at the wrong page.

Next we create a `content` closure that specifies all the areas of interest within the page. We can use a jQuery-ish Navigator API to select the content we are interested in.

Finally, we can verify that a new message was created successfully.

```
then:
at ViewMessagePage
success == 'Successfully created a new message'
id
date
summary == expectedSummary
message == expectedMessage
```

For further details on how to get the most out of Geb, consult The Book of Geb user's manual.

3.6.3. Client-Side REST Tests

Client-side tests can be used to test code that internally uses the `RestTemplate`. The idea is to declare

expected requests and to provide "stub" responses so that you can focus on testing the code in isolation, i.e. without running a server. Here is an example:

```
RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer = MockRestServiceServer.bindTo(restTemplate).build();
mockServer.expect(requestTo("/greeting")).andRespond(withSuccess());

// Test code that uses the above RestTemplate ...

mockServer.verify();
```

In the above example, `MockRestServiceServer`, the central class for client-side REST tests, configures the `RestTemplate` with a custom `ClientHttpRequestFactory` that asserts actual requests against expectations and returns "stub" responses. In this case we expect a request to "/greeting" and want to return a 200 response with "text/plain" content. We could define as additional expected requests and stub responses as needed. When expected requests and stub responses are defined, the `RestTemplate` can be used in client-side code as usual. At the end of testing `mockServer.verify()` can be used to verify that all expectations have been satisfied.

By default requests are expected in the order in which expectations were declared. You can set the `ignoreExpectOrder` option when building the server in which case all expectations are checked (in order) to find a match for a given request. That means requests are allowed to come in any order. Here is an example:

```
server = MockRestServiceServer.bindTo(restTemplate).ignoreExpectOrder(true).build();
```

Even with unordered requests by default each request is allowed to execute once only. The `expect` method provides an overloaded variant that accepts an `ExpectedCount` argument that specifies a count range, e.g. `once`, `manyTimes`, `max`, `min`, `between`, and so on. Here is an example:

```
RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer = MockRestServiceServer.bindTo(restTemplate).build();
mockServer.expect(times(2), requestTo("/foo")).andRespond(withSuccess());
mockServer.expect(times(3), requestTo("/bar")).andRespond(withSuccess());

// ...

mockServer.verify();
```

Note that when `ignoreExpectOrder` is not set (the default), and therefore requests are expected in order of declaration, then that order only applies to the first of any expected request. For example if "/foo" is expected 2 times followed by "/bar" 3 times, then there should be a request to "/foo" before there is a request to "/bar" but aside from that subsequent "/foo" and "/bar" requests can come at any time.

As an alternative to all of the above the client-side test support also provides a `ClientHttpRequestFactory` implementation that can be configured into a `RestTemplate` to bind it to a `MockMvc` instance. That allows processing requests using actual server-side logic but without running a server. Here is an example:

```
MockMvc mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
this.restTemplate = new RestTemplate(new MockMvcClientHttpRequestFactory(mockMvc));

// Test code that uses the above RestTemplate ...
```

Static Imports

Just like with server-side tests, the fluent API for client-side tests requires a few static imports. Those are easy to find by searching *"MockRest*"*. Eclipse users should add `"MockRestRequestMatchers.*"` and `"MockRestResponseCreators.*"` as "favorite static members" in the Eclipse preferences under *Java → Editor → Content Assist → Favorites*. That allows using content assist after typing the first character of the static method name. Other IDEs (e.g. IntelliJ) may not require any additional configuration. Just check the support for code completion on static members.

Further Examples of Client-side REST Tests

Spring MVC Test's own tests include example tests of client-side REST tests.

3.7. WebTestClient

`WebTestClient` is a thin shell around WebClient, using it to perform requests and exposing a dedicated, fluent API for verifying responses. `WebTestClient` bind to a WebFlux application using a mock request and response, or it can test any web server over an HTTP connection.

Kotlin users, please see this section related to use of the `WebTestClient`.

3.7.1. Setup

To create a `WebTestClient` you must choose one of several server setup options. Effectively you're either configuring the WebFlux application to bind to, or using a URL to connect to a running server.

Bind to controller

Use this server setup to test one `@Controller` at a time:

```
    client = WebTestClient.bindToController(new TestController()).build();
```

The above loads the WebFlux Java config and registers the given controller. The resulting WebFlux application will be tested without an HTTP server using mock request and response objects. There

are more methods on the builder to customize the default WebFlux Java config.

Bind to RouterFunction

Use this option to set up a server from a RouterFunction:

```
RouterFunction<?> route = ...
client = WebTestClient.bindToRouterFunction(route).build();
```

Internally the provided configuration is passed to `RouterFunctions.toWebHandler`. The resulting WebFlux application will be tested without an HTTP server using mock request and response objects.

Bind to ApplicationContext

Use this option to setup a server from the Spring configuration of your application, or some subset of it:

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = WebConfig.class) ①
public class MyTests {

    @Autowired
    private ApplicationContext context; ②

    private WebTestClient client;

    @Before
    public void setUp() {
        client = WebTestClient.bindToApplicationContext(context).build(); ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

Internally the provided configuration is passed to `WebHttpHandlerBuilder` to set up the request processing chain, see WebHandler API for more details. The resulting WebFlux application will be tested without an HTTP server using mock request and response objects.

Bind to server

This server setup option allows you to connect to a running server:

```
client = WebTestClient.bindToServer().baseUrl("http://localhost:8080").build();
```

Client builder

In addition to the server setup options above, you can also configure client options including base URL, default headers, client filters, and others. These options are readily available following `bindToServer`. For all others, you need to use `configureClient()` to transition from server to client configuration as shown below:

```
client = WebTestClient.bindToController(new TestController())
        .configureClient()
        .baseUrl("/test")
        .build();
```

3.7.2. Writing tests

`WebTestClient` is a thin shell around WebClient. It provides an identical API up to the point of performing a request via `exchange()`. What follows after `exchange()` is a chained API workflow to verify responses.

Typically you start by asserting the response status and headers:

```
client.get().uri("/persons/1")
        .accept(MediaType.APPLICATION_JSON_UTF8)
        .exchange()
        .expectStatus().isOk()
        .expectHeader().contentType(MediaType.APPLICATION_JSON_UTF8)
        // ...
```

Then you specify how to decode and consume the response body:

- `expectBody(Class<T>)` — decode to single object.
- `expectBodyList(Class<T>)` — decode and collect objects to `List<T>`.
- `expectBody()` — decode to `byte[]` for JSON content or empty body.

Then you can use built-in assertions for the body. Here is one example:

```
client.get().uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectBodyList(Person.class).hasSize(3).contains(person);
```

You can go beyond the built-in assertions and create your own:

```
client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody(Person.class)
        .consumeWith(result -> {
            // custom assertions (e.g. AssertJ)...
        });
```

You can also exit the workflow and get a result:

```
EntityExchangeResult<Person> result = client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody(Person.class)
        .returnResult();
```

When you need to decode to a target type with generics, look for the overloaded methods that accept ParameterizedTypeReference instead of `Class<T>`.

No content

If the response has no content, or you don't care if it does, use `Void.class` which ensures that resources are released:

```
client.get().uri("/persons/123")
        .exchange()
        .expectStatus().isNotFound()
        .expectBody(Void.class);
```

Or if you want to assert there is no response content, use this:

```
client.post().uri("/persons")
        .body(personMono, Person.class)
        .exchange()
        .expectStatus().isCreated()
        .expectBody().isEmpty();
```

JSON content

When you use `expectBody()` the response is consumed as a `byte[]`. This is useful for raw content assertions. For example you can use JSONAssert to verify JSON content:

```
client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody()
        .json("{\"name\":\"Jane\"}")
```

You can also use JSONPath expressions:

```
client.get().uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectBody()
        .jsonPath("$[0].name").isEqualTo("Jane")
        .jsonPath("$[1].name").isEqualTo("Jason");
```

Streaming responses

To test infinite streams (e.g. `"text/event-stream"`, `"application/stream+json"`), you'll need to exit the chained API, via `returnResult`, immediately after response status and header assertions, as shown below:

```
FluxExchangeResult<MyEvent> result = client.get().uri("/events")
        .accept(TEXT_EVENT_STREAM)
        .exchange()
        .expectStatus().isOk()
        .returnResult(MyEvent.class);
```

Now you can consume the `Flux<T>`, assert decoded objects as they come, and then cancel at some point when test objects are met. We recommend using the `StepVerifier` from the `reactor-test` module to do that, for example:

```
Flux<Event> eventFux = result.getResponseBody();

StepVerifier.create(eventFlux)
        .expectNext(person)
        .expectNextCount(4)
        .consumeNextWith(p -> ...)
        .thenCancel()
        .verify();
```

Request body

When it comes to building requests, the `WebTestClient` offers an identical API as the `WebClient` and the implementation is mostly a simple pass-through. Please refer to the WebClient documentation for examples on how to prepare a request with a body including submitting form data, multipart

requests, and more.

3.8. PetClinic Example

The PetClinic application, available on GitHub, illustrates several features of the *Spring TestContext Framework* in a JUnit 4 environment. Most test functionality is included in the `AbstractClinicTests`, for which a partial listing is shown below:

```
import static org.junit.Assert.assertEquals;
// import ...

<strong>@ContextConfiguration</strong>
public abstract class AbstractClinicTests <strong>extends
AbstractTransactionalJUnit4SpringContextTests</strong> {

    <strong>@Autowired</strong>
    protected Clinic clinic;

    @Test
    public void getVets() {
        Collection<Vet> vets = this.clinic.getVets();
        assertEquals("JDBC query must show the same number of vets",
            <strong>super.countRowsInTable("VETS")</strong>, vets.size());
        Vet v1 = EntityUtils.getById(vets, Vet.class, 2);
        assertEquals("Leary", v1.getLastName());
        assertEquals(1, v1.getNrOfSpecialties());
        assertEquals("radiology", (v1.getSpecialties().get(0)).getName());
        // ...
    }

    // ...
}
```

Notes:

- This test case extends the `AbstractTransactionalJUnit4SpringContextTests` class, from which it inherits configuration for Dependency Injection (through the `DependencyInjectionTestExecutionListener`) and transactional behavior (through the `TransactionalTestExecutionListener`).
- The `clinic` instance variable — the application object being tested — is set by Dependency Injection through `@Autowired` semantics.
- The `getVets()` method illustrates how you can use the inherited `countRowsInTable()` method to easily verify the number of rows in a given table, thus verifying correct behavior of the application code being tested. This allows for stronger tests and lessens dependency on the exact test data. For example, you can add additional rows in the database without breaking tests.
- Like many integration tests that use a database, most of the tests in `AbstractClinicTests` depend

on a minimum amount of data already in the database before the test cases run. Alternatively, you might choose to populate the database within the test fixture set up of your test cases — again, within the same transaction as the tests.

The PetClinic application supports three data access technologies: JDBC, Hibernate, and JPA. By declaring `@ContextConfiguration` without any specific resource locations, the `AbstractClinicTests` class will have its application context loaded from the default location, `AbstractClinicTests-context.xml`, which declares a common `DataSource`. Subclasses specify additional context locations that must declare a `PlatformTransactionManager` and a concrete implementation of `Clinic`.

For example, the Hibernate implementation of the PetClinic tests contains the following implementation. For this example, `HibernateClinicTests` does not contain a single line of code: we only need to declare `@ContextConfiguration`, and the tests are inherited from `AbstractClinicTests`. Because `@ContextConfiguration` is declared without any specific resource locations, the *Spring TestContext Framework* loads an application context from all the beans defined in `AbstractClinicTests-context.xml` (i.e., the inherited locations) and `HibernateClinicTests-context.xml`, with `HibernateClinicTests-context.xml` possibly overriding beans defined in `AbstractClinicTests-context.xml`.

```
<strong>@ContextConfiguration</strong>
public class HibernateClinicTests extends AbstractClinicTests { }
```

In a large-scale application, the Spring configuration is often split across multiple files. Consequently, configuration locations are typically specified in a common base class for all application-specific integration tests. Such a base class may also add useful instance variables — populated by Dependency Injection, naturally — such as a `SessionFactory` in the case of an application using Hibernate.

As far as possible, you should have exactly the same Spring configuration files in your integration tests as in the deployed environment. One likely point of difference concerns database connection pooling and transaction infrastructure. If you are deploying to a full-blown application server, you will probably use its connection pool (available through JNDI) and JTA implementation. Thus in production you will use a `JndiObjectFactoryBean` or `<jee:jndi-lookup>` for the `DataSource` and `JtaTransactionManager`. JNDI and JTA will not be available in out-of-container integration tests, so you should use a combination like the Commons DBCP `BasicDataSource` and `DataSourceTransactionManager` or `HibernateTransactionManager` for them. You can factor out this variant behavior into a single XML file, having the choice between application server and a 'local' configuration separated from all other configuration, which will not vary between the test and production environments. In addition, it is advisable to use properties files for connection settings. See the PetClinic application for an example.

Chapter 4. Further Resources

Consult the following resources for more information about testing:

- JUnit: "*A programmer-oriented testing framework for Java*". Used by the Spring Framework in its test suite.
- TestNG: A testing framework inspired by JUnit with added support for annotations, test groups, data-driven testing, distributed testing, etc.
- AssertJ: "*Fluent assertions for Java*" including support for Java 8 lambdas, streams, etc.
- Mock Objects: Article in Wikipedia.
- MockObjects.com: Web site dedicated to mock objects, a technique for improving the design of code within test-driven development.
- Mockito: Java mock library based on the test spy pattern.
- EasyMock: Java library "*that provides Mock Objects for interfaces (and objects through the class extension) by generating them on the fly using Java's proxy mechanism.*" Used by the Spring Framework in its test suite.
- JMock: Library that supports test-driven development of Java code with mock objects.
- DbUnit: JUnit extension (also usable with Ant and Maven) targeted for database-driven projects that, among other things, puts your database into a known state between test runs.
- The Grinder: Java load testing framework.