Testing

Version 6.0.0-M2

Table of Contents

This chapter covers Spring's support for integration testing and best practices for unit testing. The Spring team advocates test-driven development (TDD). The Spring team has found that the correct use of inversion of control (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 similar structures).

Chapter 1. Introduction to Spring Testing

Testing is an integral part of enterprise software development. This chapter focuses on the value added by 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 J2EE / Java EE development. The POJOs that make up your application should be testable in JUnit or TestNG tests, with objects instantiated by 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 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 can 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 mock objects and testing support classes, which are described in this chapter.

2.1. Mock Objects

Spring includes a number of packages dedicated to mocking:

- Environment
- JNDI
- Servlet API
- Spring Web Reactive

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 a partial 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` instances get bound to the same JNDI names in test code as they do in a Jakarta EE container, you can reuse both application code and configuration in testing scenarios without modification.

The mock JNDI support in the `org.springframework.mock.jndi` package is officially deprecated as of Spring Framework 5.2 in favor of complete solutions from third parties such as Simple-JNDI.

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 MockMvc.

2.1.4. Spring Web Reactive

The `org.springframework.mock.http.server.reactive` package contains mock implementations of `ServerHttpRequest` and `ServerHttpResponse` for use in WebFlux applications. The `org.springframework.mock.web.server` package 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 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 (that is, `Mono<Void>`), it by default 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

Spring includes a number of classes that can help with unit testing. They fall into two categories:

- General Testing Utilities
- Spring MVC Testing Utilities

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. You can use these methods in testing scenarios where you 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 for 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`), that provide 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. You can use these methods 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 by using a library such as EasyMock or Mockito, and the mock is wrapped in a Spring proxy, you may need direct access to the underlying mock to configure expectations on it and perform verifications. For Spring's core AOP utilities, see `AopUtils` and `AopProxyUtils`.

2.2.2. Spring MVC Testing Utilities

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 that deal with Spring MVC `ModelAndView` objects.

Unit testing Spring MVC Controllers

To unit test your Spring MVC `Controller` classes 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` classes in conjunction with your `WebApplicationContext` configuration for Spring MVC, use the Spring MVC Test Framework instead.

Chapter 3. Integration Testing

This section (most of the rest of this chapter) covers integration testing for Spring applications. It includes the following topics:

- Overview
- Goals of Integration Testing
- JDBC Testing Support
- Annotations
- Spring TestContext Framework
- MockMvc

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. Doing so lets you test things such as:

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

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.

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, which allows instrumentation of tests in various environments, including JUnit, TestNG, and others.

3.2. Goals of Integration Testing

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

- To manage Spring IoC container caching between tests.
- 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` instances and `WebApplicationContext` instances 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 component 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 by using Dependency Injection. This provides a convenient mechanism for setting up test fixtures by using preconfigured beans from your application context. A strong benefit here is that you can reuse application contexts across various testing scenarios (for example, for configuring Spring-managed object graphs, transactional proxies, `DataSource` instances, and others), thus avoiding the need to duplicate complex test fixture setup for individual test cases.

As an example, consider a 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 use 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 of a transaction.

The TestContext framework addresses this issue. By default, the framework creates and rolls back a transaction for each test. You can write code that can assume the existence of a transaction. If you call transactionally proxied objects in your tests, they 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 rolls back by default, and the database returns to its state prior to execution of the test. Transactional support is provided to a test by using 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), you can tell the TestContext framework to cause the transaction to commit instead of roll back by using 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 let you 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. You can use such queries to confirm database state both before 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 by using the provided `WHERE` clause.
- `deleteFromTables(..)`: Deletes all rows from the specified tables.
- `deleteFromTableWhere(..)`: Deletes rows from the given table by using the provided `WHERE` clause.
- `dropTables(..)`: Drops the specified tables.

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

The `spring-jdbc` module provides support for configuring and launching an embedded database, which you can use 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

This section covers annotations that you can use when you test Spring applications. It includes the following topics:

- Spring Testing Annotations
- Standard Annotation Support
- Spring JUnit 4 Testing Annotations
- Spring JUnit Jupiter Testing Annotations
- Meta-Annotation Support for Testing

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. See the corresponding javadoc for further information, including default attribute values, attribute aliases, and other details.

Spring's testing annotations include the following:

- `@BootstrapWith`
- `@ContextConfiguration`
- `@WebAppConfiguration`
- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DynamicPropertySource`
- `@DirtiesContext`

- `@TestExecutionListeners`
- `@RecordApplicationEvents`
- `@Commit`
- `@Rollback`
- `@BeforeTransaction`
- `@AfterTransaction`
- `@Sql`
- `@SqlConfig`
- `@SqlMergeMode`
- `@SqlGroup`

`@BootstrapWith`

`@BootstrapWith` is a class-level annotation that you can use to configure how the Spring TestContext Framework is bootstrapped. Specifically, you can use `@BootstrapWith` to specify a custom `TestContextBootstrapper`. See the section on bootstrapping the TestContext framework 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 component `classes` used to load the context.

Resource locations are typically XML configuration files or Groovy scripts located in the classpath, while component classes are typically `@Configuration` classes. However, resource locations can also refer to files and scripts in the file system, and component classes can be `@Component` classes, `@Service` classes, and so on. See Component Classes for further details.

The following example shows a `@ContextConfiguration` annotation that refers to an XML file:

Java

```
@ContextConfiguration("/test-config.xml") ①
class XmlApplicationContextTests {
    // class body...
}
```

① Referring to an XML file.

Kotlin

```
@ContextConfiguration("/test-config.xml") ①
class XmlApplicationContextTests {
    // class body...
}
```

① Referring to an XML file.

The following example shows a `@ContextConfiguration` annotation that refers to a class:

Java

```
@ContextConfiguration(classes = TestConfig.class) ①
class ConfigClassApplicationContextTests {
    // class body...
}
```

① Referring to a class.

Kotlin

```
@ContextConfiguration(classes = [TestConfig::class]) ①
class ConfigClassApplicationContextTests {
    // class body...
}
```

① Referring to a class.

As an alternative or in addition to declaring resource locations or component classes, you can use `@ContextConfiguration` to declare `ApplicationContextInitializer` classes. The following example shows such a case:

Java

```
@ContextConfiguration(initializers = CustomContextIntializer.class) ①
class ContextInitializerTests {
    // class body...
}
```

① Declaring an initializer class.

Kotlin

```
@ContextConfiguration(initializers = [CustomContextIntializer::class]) ①
class ContextInitializerTests {
    // class body...
}
```

① Declaring an initializer class.

You can optionally use `@ContextConfiguration` 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 `initializers` and either resource `locations` or component `classes`.

The following example uses both a location and a loader:

Java

```
@ContextConfiguration(locations = "/test-context.xml", loader =
CustomContextLoader.class) ①
class CustomLoaderXmlApplicationContextTests {
    // class body...
}
```

① Configuring both a location and a custom loader.

Kotlin

```
@ContextConfiguration("/test-context.xml", loader = CustomContextLoader::class) ①
class CustomLoaderXmlApplicationContextTests {
    // class body...
}
```

① Configuring both a location and a custom loader.

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

See Context Management, `@Nested` test class configuration, and the `@ContextConfiguration` javadocs for further details.

`@WebAppConfiguration`

`@WebAppConfiguration` is a class-level annotation that you can use 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` is loaded for the test, using the default value of `"file:src/main/webapp"` for the path to the root of the web application (that is, 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`.

The following example shows how to use the `@WebAppConfiguration` annotation:

Java

```
@ContextConfiguration
@WebAppConfiguration ①
class WebAppTests {
    // class body...
}
```

Kotlin

```
@ContextConfiguration
@WebAppConfiguration ①
class WebAppTests {
    // class body...
}
```

① The `@WebAppConfiguration` annotation.

To override the default, you can specify a different base resource path by using 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. The following example shows how to specify a classpath resource:

Java

```
@ContextConfiguration
@WebAppConfiguration("classpath:test-web-resources") ①
class WebAppTests {
    // class body...
}
```

① Specifying a classpath resource.

Kotlin

```
@ContextConfiguration
@WebAppConfiguration("classpath:test-web-resources") ①
class WebAppTests {
    // class body...
}
```

① Specifying a classpath resource.

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` javadoc for further details.

`@ContextHierarchy`

`@ContextHierarchy` is a class-level annotation that is used to define a hierarchy of `ApplicationContext` instances 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 (`@ContextHierarchy` can also be used within a test class hierarchy):

Java

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

Kotlin

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

Java

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

Kotlin

```
@WebAppConfiguration
@ContextHierarchy(
        ContextConfiguration(classes = [AppConfig::class]),
        ContextConfiguration(classes = [WebConfig::class]))
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` javadoc 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.

The following example indicates that the `dev` profile should be active:

Java

```
@ContextConfiguration
@ActiveProfiles("dev") ①
class DeveloperTests {
    // class body...
}
```

① Indicate that the `dev` profile should be active.

Kotlin

```
@ContextConfiguration
@ActiveProfiles("dev") ①
class DeveloperTests {
    // class body...
}
```

① Indicate that the `dev` profile should be active.

The following example indicates that both the `dev` and the `integration` profiles should be active:

Java

```
@ContextConfiguration
@ActiveProfiles({"dev", "integration"}) ①
class DeveloperIntegrationTests {
    // class body...
}
```

① Indicate that the `dev` and `integration` profiles should be active.

Kotlin

```
@ContextConfiguration
@ActiveProfiles(["dev", "integration"]) ①
class DeveloperIntegrationTests {
    // class body...
}
```

① Indicate that the `dev` and `integration` profiles should be active.

`@ActiveProfiles` provides support for inheriting active bean definition profiles declared by superclasses and enclosing classes by default. You can also resolve active bean definition profiles programmatically by implementing a custom `ActiveProfilesResolver` and registering it by using the `resolver` attribute of `@ActiveProfiles`.

See Context Configuration with Environment Profiles, `@Nested` test class configuration, and the `@ActiveProfiles` javadoc for examples and further details.

`@TestPropertySource`

`@TestPropertySource` is a class-level annotation that you can use 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.

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

Java

```
@ContextConfiguration
@TestPropertySource("/test.properties") ①
class MyIntegrationTests {
    // class body...
}
```

① Get properties from `test.properties` in the root of the classpath.

Kotlin

```
@ContextConfiguration
@TestPropertySource("/test.properties") ①
class MyIntegrationTests {
    // class body...
}
```

① Get properties from `test.properties` in the root of the classpath.

The following example demonstrates how to declare inlined properties:

Java

```
@ContextConfiguration
@TestPropertySource(properties = { "timezone = GMT", "port: 4242" }) ①
class MyIntegrationTests {
    // class body...
}
```

① Declare `timezone` and `port` properties.

Kotlin

```
@ContextConfiguration
@TestPropertySource(properties = ["timezone = GMT", "port: 4242"]) ①
class MyIntegrationTests {
    // class body...
}
```

① Declare `timezone` and `port` properties.

See Context Configuration with Test Property Sources for examples and further details.

`@DynamicPropertySource`

`@DynamicPropertySource` is a method-level annotation that you can use to register *dynamic* properties to be added to the set of `PropertySources` in the `Environment` for an `ApplicationContext` loaded for an integration test. Dynamic properties are useful when you do not know the value of the properties upfront – for example, if the properties are managed by an external resource such as for a container managed by the Testcontainers project.

The following example demonstrates how to register a dynamic property:

Java

```
@ContextConfiguration
class MyIntegrationTests {

    static MyExternalServer server = // ...

    @DynamicPropertySource ①
    static void dynamicProperties(DynamicPropertyRegistry registry) { ②
        registry.add("server.port", server::getPort); ③
    }

    // tests ...
}
```

① Annotate a `static` method with `@DynamicPropertySource`.

② Accept a `DynamicPropertyRegistry` as an argument.

③ Register a dynamic `server.port` property to be retrieved lazily from the server.

Kotlin

```
@ContextConfiguration
class MyIntegrationTests {

    companion object {

        @JvmStatic
        val server: MyExternalServer = // ...

        @DynamicPropertySource ①
        @JvmStatic
        fun dynamicProperties(registry: DynamicPropertyRegistry) { ②
            registry.add("server.port", server::getPort) ③
        }
    }

    // tests ...
}
```

① Annotate a `static` method with `@DynamicPropertySource`.

② Accept a `DynamicPropertyRegistry` as an argument.

③ Register a dynamic `server.port` property to be retrieved lazily from the server.

See Context Configuration with Dynamic Property Sources for further details.

`@DirtiesContext`

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

You can use `@DirtiesContext` as both a class-level and a 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`.

Java

```
@DirtiesContext(classMode = BEFORE_CLASS) ①
class FreshContextTests {
    // some tests that require a new Spring container
}
```

① Dirty the context before the current test class.

Kotlin

```
@DirtiesContext(classMode = BEFORE_CLASS) ①
class FreshContextTests {
    // some tests that require a new Spring container
}
```

① Dirty the context before the current test class.

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

Java

```
@DirtiesContext ①
class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

① Dirty the context after the current test class.

Kotlin

```
@DirtiesContext ①
class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

① Dirty the context after the current test class.

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

Java

```
@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD) ①
class FreshContextTests {
    // some tests that require a new Spring container
}
```

① Dirty the context before each test method.

Kotlin

```
@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD) ①
class FreshContextTests {
    // some tests that require a new Spring container
}
```

① Dirty the context before each test method.

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

Java

```
@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD) ①
class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

① Dirty the context after each test method.

Kotlin

```
@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD) ①
class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

① Dirty the context after each test method.

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

Java

```
@DirtiesContext(methodMode = BEFORE_METHOD) ①
@Test
void testProcessWhichRequiresFreshAppCtx() {
    // some logic that requires a new Spring container
}
```

① Dirty the context before the current test method.

Kotlin

```
@DirtiesContext(methodMode = BEFORE_METHOD) ①
@Test
fun testProcessWhichRequiresFreshAppCtx() {
    // some logic that requires a new Spring container
}
```

① Dirty the context before the current test method.

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

  Java

  ```
  @DirtiesContext ①
  @Test
  void testProcessWhichDirtiesAppCtx() {
      // some logic that results in the Spring container being dirtied
  }
  ```

  ① Dirty the context after the current test method.

  Kotlin

  ```
  @DirtiesContext ①
  @Test
  fun testProcessWhichDirtiesAppCtx() {
      // some logic that results in the Spring container being dirtied
  }
  ```

  ① Dirty the context after the current test method.

If you use `@DirtiesContext` in a test whose context is configured as part of a context hierarchy with `@ContextHierarchy`, you can use the `hierarchyMode` flag to control how the context cache is cleared. By default, an exhaustive algorithm is used to clear 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` instances that reside in a sub-hierarchy of the common ancestor context are removed from the context cache and closed. If the exhaustive algorithm is overkill for a particular use case, you can specify the simpler current level algorithm, as the following example shows.

Java

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

class ExtendedTests extends BaseTests {

    @Test
    @DirtiesContext(hierarchyMode = CURRENT_LEVEL) ①
    void test() {
        // some logic that results in the child context being dirtied
    }
}
```

① Use the current-level algorithm.

Kotlin

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

class ExtendedTests : BaseTests() {

    @Test
    @DirtiesContext(hierarchyMode = CURRENT_LEVEL) ①
    fun test() {
        // some logic that results in the child context being dirtied
    }
}
```

① Use the current-level algorithm.

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

`@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`.

The following example shows how to register two `TestExecutionListener` implementations:

Java

```
@ContextConfiguration
@TestExecutionListeners({CustomTestExecutionListener.class,
AnotherTestExecutionListener.class}) ①
class CustomTestExecutionListenerTests {
    // class body...
}
```

① Register two `TestExecutionListener` implementations.

Kotlin

```
@ContextConfiguration
@TestExecutionListeners(CustomTestExecutionListener::class,
AnotherTestExecutionListener::class) ①
class CustomTestExecutionListenerTests {
    // class body...
}
```

① Register two `TestExecutionListener` implementations.

By default, `@TestExecutionListeners` provides support for inheriting listeners from superclasses or enclosing classes. See `@Nested` test class configuration and the `@TestExecutionListeners` javadoc for an example and further details.

`@RecordApplicationEvents`

`@RecordApplicationEvents` is a class-level annotation that is used to instruct the *Spring TestContext Framework* to record all application events that are published in the `ApplicationContext` during the execution of a single test.

The recorded events can be accessed via the `ApplicationEvents` API within tests.

See Application Events and the `@RecordApplicationEvents` javadoc 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. You can use `@Commit` as a direct replacement for `@Rollback(false)` to more explicitly convey the intent of the code. Analogous to `@Rollback`, `@Commit` can also be declared as a class-level or method-level annotation.

The following example shows how to use the `@Commit` annotation:

Java

```
@Commit ①
@Test
void testProcessWithoutRollback() {
    // ...
}
```

① Commit the result of the test to the database.

Kotlin

```
@Commit ①
@Test
fun testProcessWithoutRollback() {
    // ...
}
```

① Commit the result of the test to the database.

`@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 for integration tests in the Spring TestContext Framework defaults 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.

The following example causes a test method's result to not be rolled back (that is, the result is committed to the database):

Java

```
@Rollback(false) ①
@Test
void testProcessWithoutRollback() {
    // ...
}
```

① Do not roll back the result.

Kotlin

```
@Rollback(false) ①
@Test
fun testProcessWithoutRollback() {
    // ...
}
```

① Do not roll back the result.

`@BeforeTransaction`

`@BeforeTransaction` indicates that the annotated `void` method should be run before a transaction is started, for test methods that have been configured to run within a transaction by using Spring's `@Transactional` annotation. `@BeforeTransaction` methods are not required to be `public` and may be declared on Java 8-based interface default methods.

The following example shows how to use the `@BeforeTransaction` annotation:

Java

```
@BeforeTransaction ①
void beforeTransaction() {
    // logic to be run before a transaction is started
}
```

① Run this method before a transaction.

Kotlin

```
@BeforeTransaction ①
fun beforeTransaction() {
    // logic to be run before a transaction is started
}
```

① Run this method before a transaction.

`@AfterTransaction`

`@AfterTransaction` indicates that the annotated `void` method should be run after a transaction is ended, for test methods that have been configured to run within a transaction by using Spring's `@Transactional` annotation. `@AfterTransaction` methods are not required to be `public` and may be declared on Java 8-based interface default methods.

Java

```
@AfterTransaction ①
void afterTransaction() {
    // logic to be run after a transaction has ended
}
```

① Run this method after a transaction.

Kotlin

```
@AfterTransaction ①
fun afterTransaction() {
    // logic to be run after a transaction has ended
}
```

① Run this method after a transaction.

`@Sql`

`@Sql` is used to annotate a test class or test method to configure SQL scripts to be run against a given database during integration tests. The following example shows how to use it:

Java

```
@Test
@Sql({"/test-schema.sql", "/test-user-data.sql"}) ①
void userTest() {
    // run code that relies on the test schema and test data
}
```

① Run two scripts for this test.

Kotlin

```
@Test
@Sql("/test-schema.sql", "/test-user-data.sql") ①
fun userTest() {
    // run code that relies on the test schema and test data
}
```

① Run two scripts for this test.

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

`@SqlConfig`

`@SqlConfig` defines metadata that is used to determine how to parse and run SQL scripts configured with the `@Sql` annotation. The following example shows how to use it:

Java

```
@Test
@Sql(
    scripts = "/test-user-data.sql",
    config = @SqlConfig(commentPrefix = "`", separator = "@@") ①
)
void userTest() {
    // run code that relies on the test data
}
```

① Set the comment prefix and the separator in SQL scripts.

Kotlin

```
@Test
@Sql("/test-user-data.sql", config = SqlConfig(commentPrefix = "`", separator = "@@"))
①
fun userTest() {
    // run code that relies on the test data
}
```

① Set the comment prefix and the separator in SQL scripts.

`@SqlMergeMode`

`@SqlMergeMode` is used to annotate a test class or test method to configure whether method-level `@Sql` declarations are merged with class-level `@Sql` declarations. If `@SqlMergeMode` is not declared on a test class or test method, the `OVERRIDE` merge mode will be used by default. With the `OVERRIDE` mode, method-level `@Sql` declarations will effectively override class-level `@Sql` declarations.

Note that a method-level `@SqlMergeMode` declaration overrides a class-level declaration.

The following example shows how to use `@SqlMergeMode` at the class level.

Java

```
@SpringJUnitConfig(TestConfig.class)
@Sql("/test-schema.sql")
@SqlMergeMode(MERGE) ①
class UserTests {

    @Test
    @Sql("/user-test-data-001.sql")
    void standardUserProfile() {
        // run code that relies on test data set 001
    }
}
```

① Set the `@Sql` merge mode to `MERGE` for all test methods in the class.

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
@Sql("/test-schema.sql")
@SqlMergeMode(MERGE) ①
class UserTests {

    @Test
    @Sql("/user-test-data-001.sql")
    fun standardUserProfile() {
        // run code that relies on test data set 001
    }
}
```

① Set the `@Sql` merge mode to `MERGE` for all test methods in the class.

The following example shows how to use `@SqlMergeMode` at the method level.

Java

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

    @Test
    @Sql("/user-test-data-001.sql")
    @SqlMergeMode(MERGE) ①
    void standardUserProfile() {
        // run code that relies on test data set 001
    }
}
```

① Set the `@Sql` merge mode to `MERGE` for a specific test method.

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
@Sql("/test-schema.sql")
class UserTests {

    @Test
    @Sql("/user-test-data-001.sql")
    @SqlMergeMode(MERGE) ①
    fun standardUserProfile() {
        // run code that relies on test data set 001
    }
}
```

① Set the `@Sql` merge mode to `MERGE` for a specific test method.

`@SqlGroup`

`@SqlGroup` is a container annotation that aggregates several `@Sql` annotations. You can use `@SqlGroup` natively to declare several nested `@Sql` annotations, or you can use it in conjunction with Java 8's support for repeatable annotations, where `@Sql` can be declared several times on the same class or method, implicitly generating this container annotation. The following example shows how to declare an SQL group:

Java

```
@Test
@SqlGroup({ ①
    @Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`")),
    @Sql("/test-user-data.sql")
)}
void userTest() {
    // run code that uses the test schema and test data
}
```

① Declare a group of SQL scripts.

Kotlin

```
@Test
@SqlGroup( ①
    Sql("/test-schema.sql", config = SqlConfig(commentPrefix = "`")),
    Sql("/test-user-data.sql"))
fun userTest() {
    // run code that uses the test schema and test data
}
```

① Declare a group of SQL scripts.

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`
- `@Value`
- `@Resource` (jakarta.annotation) if JSR-250 is present
- `@ManagedBean` (jakarta.annotation) if JSR-250 is present
- `@Inject` (jakarta.inject) if JSR-330 is present
- `@Named` (jakarta.inject) if JSR-330 is present
- `@PersistenceContext` (jakarta.persistence) if JPA is present

- `@PersistenceUnit` (jakarta.persistence) if JPA is present
- `@Required`
- `@Transactional` (org.springframework.transaction.annotation) *with limited attribute support*

JSR-250 Lifecycle Annotations

In the Spring TestContext Framework, you can use `@PostConstruct` and `@PreDestroy` 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 runs before any before methods of the underlying test framework (for example, methods annotated with JUnit Jupiter's `@BeforeEach`), and that applies 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 never runs. Therefore, within a test class, we recommend that you 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 supported only when used in conjunction with the SpringRunner, Spring's JUnit 4 rules, or Spring's JUnit 4 support classes:

- `@IfProfileValue`
- `@ProfileValueSourceConfiguration`
- `@Timed`
- `@Repeat`

`@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 is disabled and, effectively, ignored.

You can apply `@IfProfileValue` 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.

The following example shows a test that has an `@IfProfileValue` annotation:

Java

```
@IfProfileValue(name="java.vendor", value="Oracle Corporation") ①
@Test
public void testProcessWhichRunsOnlyOnOracleJvm() {
    // some logic that should run only on Java VMs from Oracle Corporation
}
```

① Run this test only when the Java vendor is "Oracle Corporation".

Kotlin

```
@IfProfileValue(name="java.vendor", value="Oracle Corporation") ①
@Test
fun testProcessWhichRunsOnlyOnOracleJvm() {
    // some logic that should run only on Java VMs from Oracle Corporation
}
```

① Run this test only when the Java vendor is "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:

Java

```
@IfProfileValue(name="test-groups", values={"unit-tests", "integration-tests"}) ①
@Test
public void testProcessWhichRunsForUnitOrIntegrationTestGroups() {
    // some logic that should run only for unit and integration test groups
}
```

① Run this test for unit tests and integration tests.

Kotlin

```
@IfProfileValue(name="test-groups", values=["unit-tests", "integration-tests"]) ①
@Test
fun testProcessWhichRunsForUnitOrIntegrationTestGroups() {
    // some logic that should run only for unit and integration test groups
}
```

① Run this test for unit tests and integration tests.

`@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. The following example shows how to use `@ProfileValueSourceConfiguration`:

Java

```
@ProfileValueSourceConfiguration(CustomProfileValueSource.class) ①
public class CustomProfileValueSourceTests {
    // class body...
}
```

① Use a custom profile value source.

Kotlin

```
@ProfileValueSourceConfiguration(CustomProfileValueSource::class) ①
class CustomProfileValueSourceTests {
    // class body...
}
```

① Use a custom profile value source.

`@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 running the test method itself, any repetitions of the test (see `@Repeat`), as well as any setting up or tearing down of the test fixture. The following example shows how to use it:

Java

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

① Set the time period for the test to one second.

Kotlin

```
@Timed(millis = 1000) ①
fun testProcessWithOneSecondTimeout() {
    // some logic that should not take longer than 1 second to run
}
```

① Set the time period for the test to one second.

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 run repeatedly. The number of times that the test method is to be run is specified in the annotation.

The scope of execution to be repeated includes execution of the test method itself as well as any setting up or tearing down of the test fixture. When used with the `SpringMethodRule`, the scope additionally includes preparation of the test instance by `TestExecutionListener` implementations. The following example shows how to use the `@Repeat` annotation:

Java

```
@Repeat(10) ①
@Test
public void testProcessRepeatedly() {
    // ...
}
```

① Repeat this test ten times.

Kotlin

```
@Repeat(10) ①
@Test
fun testProcessRepeatedly() {
    // ...
}
```

① Repeat this test ten times.

3.4.4. Spring JUnit Jupiter Testing Annotations

The following annotations are supported when used in conjunction with the `SpringExtension` and JUnit Jupiter (that is, the programming model in JUnit 5):

- `@SpringJUnitConfig`
- `@SpringJUnitWebConfig`
- `@TestConstructor`
- `@NestedTestConfiguration`
- `@EnabledIf`
- `@DisabledIf`

`@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 component classes may be declared with the `value` attribute in `@SpringJUnitConfig`.

The following example shows how to use the `@SpringJUnitConfig` annotation to specify a configuration class:

Java

```
@SpringJUnitConfig(TestConfig.class) ①
class ConfigurationClassJUnitJupiterSpringTests {
    // class body...
}
```

① Specify the configuration class.

Kotlin

```
@SpringJUnitConfig(TestConfig::class) ①
class ConfigurationClassJUnitJupiterSpringTests {
    // class body...
}
```

① Specify the configuration class.

The following example shows how to use the `@SpringJUnitConfig` annotation to specify the location of a configuration file:

Java

```
@SpringJUnitConfig(locations = "/test-config.xml") ①
class XmlJUnitJupiterSpringTests {
    // class body...
}
```

① Specify the location of a configuration file.

Kotlin

```
@SpringJUnitConfig(locations = ["/test-config.xml"]) ①
class XmlJUnitJupiterSpringTests {
    // class body...
}
```

① Specify the location of a configuration file.

See Context Management as well as the javadoc 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. You can use it 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 you can declare component classes by using the `value` attribute in `@SpringJUnitWebConfig`. In addition, you can override the `value` attribute from `@WebAppConfiguration` only by using the `resourcePath` attribute in `@SpringJUnitWebConfig`.

The following example shows how to use the `@SpringJUnitWebConfig` annotation to specify a configuration class:

Java

```
@SpringJUnitWebConfig(TestConfig.class) ①
class ConfigurationClassJUnitJupiterSpringWebTests {
    // class body...
}
```

① Specify the configuration class.

Kotlin

```
@SpringJUnitWebConfig(TestConfig::class) ①
class ConfigurationClassJUnitJupiterSpringWebTests {
    // class body...
}
```

① Specify the configuration class.

The following example shows how to use the `@SpringJUnitWebConfig` annotation to specify the location of a configuration file:

Java

```
@SpringJUnitWebConfig(locations = "/test-config.xml") ①
class XmlJUnitJupiterSpringWebTests {
    // class body...
}
```

① Specify the location of a configuration file.

Kotlin

```
@SpringJUnitWebConfig(locations = ["/test-config.xml"]) ①
class XmlJUnitJupiterSpringWebTests {
    // class body...
}
```

① Specify the location of a configuration file.

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

`@TestConstructor`

`@TestConstructor` is a type-level annotation that is used to configure how the parameters of a test class constructor are autowired from components in the test's `ApplicationContext`.

If `@TestConstructor` is not present or meta-present on a test class, the default *test constructor autowire mode* will be used. See the tip below for details on how to change the default mode. Note, however, that a local declaration of `@Autowired` on a constructor takes precedence over both `@TestConstructor` and the default mode.

Changing the default test constructor autowire mode

The default *test constructor autowire mode* can be changed by setting the `spring.test.constructor.autowire.mode` JVM system property to `all`. Alternatively, the default mode may be set via the `SpringProperties` mechanism.

As of Spring Framework 5.3, the default mode may also be configured as a JUnit Platform configuration parameter.

If the `spring.test.constructor.autowire.mode` property is not set, test class constructors will not be automatically autowired.

As of Spring Framework 5.2, `@TestConstructor` is only supported in conjunction with the `SpringExtension` for use with JUnit Jupiter. Note that the `SpringExtension` is often automatically registered for you – for example, when using annotations such as `@SpringJUnitConfig` and `@SpringJUnitWebConfig` or various test-related annotations from Spring Boot Test.

`@NestedTestConfiguration`

`@NestedTestConfiguration` is a type-level annotation that is used to configure how Spring test configuration annotations are processed within enclosing class hierarchies for inner test classes.

If `@NestedTestConfiguration` is not present or meta-present on a test class, in its super type hierarchy, or in its enclosing class hierarchy, the default *enclosing configuration inheritance mode* will be used. See the tip below for details on how to change the default mode.

Changing the default enclosing configuration inheritance mode

The default *enclosing configuration inheritance mode* is `INHERIT`, but it can be changed by setting the `spring.test.enclosing.configuration` JVM system property to `OVERRIDE`. Alternatively, the default mode may be set via the `SpringProperties` mechanism.

The Spring TestContext Framework honors `@NestedTestConfiguration` semantics for the following annotations.

- `@BootstrapWith`
- `@ContextConfiguration`
- `@WebAppConfiguration`

- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DynamicPropertySource`
- `@DirtiesContext`
- `@TestExecutionListeners`
- `@RecordApplicationEvents`
- `@Transactional`
- `@Commit`
- `@Rollback`
- `@Sql`
- `@SqlConfig`
- `@SqlMergeMode`
- `@TestConstructor`

The use of `@NestedTestConfiguration` typically only makes sense in conjunction with `@Nested` test classes in JUnit Jupiter; however, there may be other testing frameworks with support for Spring and nested test classes that make use of this annotation.

See `@Nested` test class configuration for an example and further details.

`@EnabledIf`

`@EnabledIf` is used to signal that the annotated JUnit Jupiter test class or test method is enabled and should be run 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 is 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 that 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.

You can use `@EnabledIf` as a meta-annotation to create custom composed annotations. For example, you can create a custom `@EnabledOnMac` annotation as follows:

Java

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

Kotlin

```
@Target(AnnotationTarget.TYPE, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
@EnabledIf(
        expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
        reason = "Enabled on Mac OS"
)
annotation class EnabledOnMac {}
```

`@DisabledIf`

`@DisabledIf` is used to signal that the annotated JUnit Jupiter test class or test method is disabled and should not be run 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 is 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 that 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.

You can use `@DisabledIf` as a meta-annotation to create custom composed annotations. For example, you can create a custom `@DisabledOnMac` annotation as follows:

Java

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

Kotlin

```
@Target(AnnotationTarget.TYPE, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
@DisabledIf(
        expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
        reason = "Disabled on Mac OS"
)
annotation class DisabledOnMac {}
```

3.4.5. Meta-Annotation Support for Testing

You can use most test-related annotations as meta-annotations to create custom composed annotations and reduce configuration duplication across a test suite.

You can use each of the following as a meta-annotation in conjunction with the TestContext framework.

- `@BootstrapWith`
- `@ContextConfiguration`
- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DirtiesContext`
- `@WebAppConfiguration`
- `@TestExecutionListeners`
- `@Transactional`
- `@BeforeTransaction`
- `@AfterTransaction`
- `@Commit`
- `@Rollback`
- `@Sql`
- `@SqlConfig`

- `@SqlMergeMode`
- `@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)*
- `@TestConstructor` *(only supported on JUnit Jupiter)*
- `@NestedTestConfiguration` *(only supported on JUnit Jupiter)*
- `@EnabledIf` *(only supported on JUnit Jupiter)*
- `@DisabledIf` *(only supported on JUnit Jupiter)*

Consider the following example:

Java

```
@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 { }
```

Kotlin

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

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

If we discover that we are repeating the preceding configuration across our JUnit 4-based test suite, we can reduce the duplication by introducing a custom composed annotation that centralizes the

common test configuration for Spring, as follows:

Java

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

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@ContextConfiguration("/app-config.xml", "/test-data-access-config.xml")
@ActiveProfiles("dev")
@Transactional
annotation class TransactionalDevTestConfig { }
```

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

Java

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

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

Kotlin

```
@RunWith(SpringRunner::class)
@TransactionalDevTestConfig
class OrderRepositoryTests

@RunWith(SpringRunner::class)
@TransactionalDevTestConfig
class UserRepositoryTests
```

If we write tests that use JUnit Jupiter, we can reduce code duplication even further, since annotations in JUnit 5 can also be used as meta-annotations. Consider the following example:

Java

```
@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 { }
```

Kotlin

```
@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 { }
```

If we discover that we are repeating the preceding configuration across our JUnit Jupiter-based test suite, we can reduce the duplication by introducing a custom composed annotation that centralizes the common test configuration for Spring and JUnit Jupiter, as follows:

Java

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

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@ExtendWith(SpringExtension::class)
@ContextConfiguration("/app-config.xml", "/test-data-access-config.xml")
@ActiveProfiles("dev")
@Transactional
annotation class TransactionalDevTestConfig { }
```

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

Java

```
@TransactionalDevTestConfig
class OrderRepositoryTests { }

@TransactionalDevTestConfig
class UserRepositoryTests { }
```

Kotlin

```
@TransactionalDevTestConfig
class OrderRepositoryTests { }

@TransactionalDevTestConfig
class UserRepositoryTests { }
```

Since JUnit Jupiter supports the use of `@Test`, `@RepeatedTest`, `ParameterizedTest`, and others as meta-annotations, you can also 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:

Java

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

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@Transactional
@Tag("integration-test") // org.junit.jupiter.api.Tag
@Test // org.junit.jupiter.api.Test
annotation class TransactionalIntegrationTest { }
```

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

Java

```
@TransactionalIntegrationTest
void saveOrder() { }

@TransactionalIntegrationTest
void deleteOrder() { }
```

Kotlin

```
@TransactionalIntegrationTest
fun saveOrder() { }

@TransactionalIntegrationTest
fun deleteOrder() { }
```

For further details, see the Spring Annotation Programming Model wiki page.

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 you can override through annotation-based configuration.

In addition to generic testing infrastructure, the TestContext framework provides explicit support for JUnit 4, JUnit Jupiter (AKA 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 and a custom `Extension` for JUnit Jupiter that let you 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 interested only in using the framework and are not 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 for each test class (for example, 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. See the javadoc and the Spring test suite for further information and examples of various implementations.

`TestContext`

`TestContext` encapsulates the context in which a test is run (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 for loading an `ApplicationContext` for an integration test managed by the Spring TestContext Framework. You should implement `SmartContextLoader` instead of this interface to provide support for component classes, active bean definition profiles, test property sources, context hierarchies, and `WebApplicationContext` support.

`SmartContextLoader` is an extension of the `ContextLoader` interface that supersedes the original minimal `ContextLoader` SPI. Specifically, a `SmartContextLoader` can choose to process resource locations, component 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, it 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 enabled only if Groovy is on the classpath.
- `WebDelegatingSmartContextLoader`: One of two default loaders, it 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` is used only if `@WebAppConfiguration` is present on the test class. Groovy support is enabled only if Groovy is on the classpath.
- `AnnotationConfigContextLoader`: Loads a standard `ApplicationContext` from component classes.
- `AnnotationConfigWebContextLoader`: Loads a `WebApplicationContext` from component 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.

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, and so on. 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. You can configure a custom bootstrapping strategy for a test class (or test class hierarchy) by using `@BootstrapWith`, either directly or as a meta-annotation. If a bootstrapper is not explicitly configured by using `@BootstrapWith`, either the `DefaultTestContextBootstrapper` or the `WebTestContextBootstrapper` is used, depending on the presence of `@WebAppConfiguration`.

Since the `TestContextBootstrapper` SPI is likely to change in the future (to accommodate new requirements), we strongly encourage implementers 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 the following order:

- `ServletTestExecutionListener`: Configures Servlet API mocks for a `WebApplicationContext`.
- `DirtiesContextBeforeModesTestExecutionListener`: Handles the `@DirtiesContext` annotation for "before" modes.
- `ApplicationEventsTestExecutionListener`: Provides support for `ApplicationEvents`.
- `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`: Runs SQL scripts configured by using the `@Sql` annotation.
- `EventPublishingTestExecutionListener`: Publishes test execution events to the test's `ApplicationContext` (see Test Execution Events).

Registering `TestExecutionListener` Implementations

You can register `TestExecutionListener` implementations for a test class and its subclasses by using the `@TestExecutionListeners` annotation. See annotation support and the javadoc for `@TestExecutionListeners` for details and examples.

Automatic Discovery of Default `TestExecutionListener` Implementations

Registering `TestExecutionListener` implementations by using `@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 an entire test suite. This issue is addressed through support for automatic discovery of default `TestExecutionListener` implementations through the `SpringFactoriesLoader` mechanism.

Specifically, the `spring-test` module declares all core default `TestExecutionListener` implementations 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` implementations to the list of default listeners in the same manner through their own `META-INF/spring.factories` properties file.

Ordering `TestExecutionListener` Implementations

When the TestContext framework discovers default `TestExecutionListener` implementations through the aforementioned `SpringFactoriesLoader` mechanism, the instantiated listeners are sorted by using Spring's `AnnotationAwareOrderComparator`, which honors Spring's `Ordered` interface and `@Order` annotation for ordering. `AbstractTestExecutionListener` and all default `TestExecutionListener` implementations provided by Spring implement `Ordered` with appropriate values. Third-party frameworks and developers should therefore make sure that their default

`TestExecutionListener` implementations are registered in the proper order by implementing `Ordered` or declaring `@Order`. See the javadoc for the `getOrder()` methods of the core default `TestExecutionListener` implementations for details on what values are assigned to each core listener.

Merging `TestExecutionListener` Implementations

If a custom `TestExecutionListener` is registered via `@TestExecutionListeners`, the default listeners are not 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:

Java

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

Kotlin

```
@ContextConfiguration
@TestExecutionListeners(
    MyCustomTestExecutionListener::class,
    ServletTestExecutionListener::class,
    DirtiesContextBeforeModesTestExecutionListener::class,
    DependencyInjectionTestExecutionListener::class,
    DirtiesContextTestExecutionListener::class,
    TransactionalTestExecutionListener::class,
    SqlScriptsTestExecutionListener::class
)
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 Boot and Spring Security register their own

default `TestExecutionListener` implementations by using the aforementioned automatic discovery mechanism.

To avoid having to be aware of and re-declare all default listeners, you can set the `mergeMode` attribute of `@TestExecutionListeners` 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 `TestExecutionListener` Implementations. 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 are 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:

Java

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

Kotlin

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

3.5.4. Application Events

Since Spring Framework 5.3.3, the TestContext framework provides support for recording application events published in the `ApplicationContext` so that assertions can be performed against those events within tests. All events published during the execution of a single test are made available via the `ApplicationEvents` API which allows you to process the events as a `java.util.Stream`.

To use `ApplicationEvents` in your tests, do the following.

- Ensure that your test class is annotated or meta-annotated with `@RecordApplicationEvents`.
- Ensure that the `ApplicationEventsTestExecutionListener` is registered. Note, however, that `ApplicationEventsTestExecutionListener` is registered by default and only needs to be manually registered if you have custom configuration via `@TestExecutionListeners` that does not include the default listeners.
- Annotate a field of type `ApplicationEvents` with `@Autowired` and use that instance of `ApplicationEvents` in your test and lifecycle methods (such as `@BeforeEach` and `@AfterEach` methods in JUnit Jupiter).
    - When using the SpringExtension for JUnit Jupiter, you may declare a method parameter of type `ApplicationEvents` in a test or lifecycle method as an alternative to an `@Autowired` field in the test class.

The following test class uses the `SpringExtension` for JUnit Jupiter and AssertJ to assert the types of application events published while invoking a method in a Spring-managed component:

Java

```
@SpringJUnitConfig(/* ... */)
@RecordApplicationEvents ①
class OrderServiceTests {

    @Autowired
    OrderService orderService;

    @Autowired
    ApplicationEvents events; ②

    @Test
    void submitOrder() {
        // Invoke method in OrderService that publishes an event
        orderService.submitOrder(new Order(/* ... */));
        // Verify that an OrderSubmitted event was published
        long numEvents = events.stream(OrderSubmitted.class).count(); ③
        assertThat(numEvents).isEqualTo(1);
    }
}
```

① Annotate the test class with `@RecordApplicationEvents`.

② Inject the `ApplicationEvents` instance for the current test.

③ Use the `ApplicationEvents` API to count how many `OrderSubmitted` events were published.

Kotlin

```
@SpringJUnitConfig(/* ... */)
@RecordApplicationEvents ①
class OrderServiceTests {

    @Autowired
    lateinit var orderService: OrderService

    @Autowired
    lateinit var events: ApplicationEvents ②

    @Test
    fun submitOrder() {
        // Invoke method in OrderService that publishes an event
        orderService.submitOrder(Order(/* ... */))
        // Verify that an OrderSubmitted event was published
        val numEvents = events.stream(OrderSubmitted::class).count() ③
        assertThat(numEvents).isEqualTo(1)
    }
}
```

① Annotate the test class with `@RecordApplicationEvents`.

② Inject the `ApplicationEvents` instance for the current test.

③ Use the `ApplicationEvents` API to count how many `OrderSubmitted` events were published.

See the `ApplicationEvents` javadoc for further details regarding the `ApplicationEvents` API.

3.5.5. Test Execution Events

The `EventPublishingTestExecutionListener` introduced in Spring Framework 5.2 offers an alternative approach to implementing a custom `TestExecutionListener`. Components in the test's `ApplicationContext` can listen to the following events published by the `EventPublishingTestExecutionListener`, each of which corresponds to a method in the `TestExecutionListener` API.

- `BeforeTestClassEvent`
- `PrepareTestInstanceEvent`
- `BeforeTestMethodEvent`
- `BeforeTestExecutionEvent`
- `AfterTestExecutionEvent`
- `AfterTestMethodEvent`
- `AfterTestClassEvent`

These events are only published if the `ApplicationContext` has already been loaded.

These events may be consumed for various reasons, such as resetting mock beans or tracing test execution. One advantage of consuming test execution events rather than implementing a custom `TestExecutionListener` is that test execution events may be consumed by any Spring bean registered in the test `ApplicationContext`, and such beans may benefit directly from dependency injection and other features of the `ApplicationContext`. In contrast, a `TestExecutionListener` is not a bean in the `ApplicationContext`.

In order to listen to test execution events, a Spring bean may choose to implement the `org.springframework.context.ApplicationListener` interface. Alternatively, listener methods can be annotated with `@EventListener` and configured to listen to one of the particular event types listed above (see Annotation-based Event Listeners). Due to the popularity of this approach, Spring provides the following dedicated `@EventListener` annotations to simplify registration of test execution event listeners. These annotations reside in the `org.springframework.test.context.event.annotation` package.

- `@BeforeTestClass`
- `@PrepareTestInstance`
- `@BeforeTestMethod`
- `@BeforeTestExecution`
- `@AfterTestExecution`
- `@AfterTestMethod`
- `@AfterTestClass`

Exception Handling

By default, if a test execution event listener throws an exception while consuming an event, that exception will propagate to the underlying testing framework in use (such as JUnit or TestNG). For example, if the consumption of a `BeforeTestMethodEvent` results in an exception, the corresponding test method will fail as a result of the exception. In contrast, if an asynchronous test execution event listener throws an exception, the exception will not propagate to the underlying testing framework. For further details on asynchronous exception handling, consult the class-level javadoc for `@EventListener`.

Asynchronous Listeners

If you want a particular test execution event listener to process events asynchronously, you can use Spring's regular `@Async` support. For further details, consult the class-level javadoc for `@EventListener`.

3.5.6. Context Management

Each `TestContext` provides context management and caching support for the test instance for which it is responsible. 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, as the following example shows:

Java

```
@SpringJUnitConfig
class MyTest {

    @Autowired ①
    ApplicationContext applicationContext;

    // class body...
}
```

① Injecting the `ApplicationContext`.

Kotlin

```
@SpringJUnitConfig
class MyTest {

    @Autowired ①
    lateinit var applicationContext: ApplicationContext

    // class body...
}
```

① Injecting the `ApplicationContext`.

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

Java

```
@SpringJUnitWebConfig ①
class MyWebAppTest {

    @Autowired ②
    WebApplicationContext wac;

    // class body...
}
```

① Configuring the `WebApplicationContext`.

② Injecting the `WebApplicationContext`.

Kotlin

```
@SpringJUnitWebConfig ①
class MyWebAppTest {

    @Autowired ②
    lateinit var wac: WebApplicationContext
    // class body...
}
```

① Configuring the `WebApplicationContext`.

② Injecting the `WebApplicationContext`.

Dependency injection by using `@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 by declaring the `@ContextConfiguration` annotation at the class level. If your test class does not explicitly declare application context resource locations or component 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 component classes, an application context can also be configured through application context initializers.

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

- Context Configuration with XML resources
- Context Configuration with Groovy Scripts
- Context Configuration with Component Classes
- Mixing XML, Groovy Scripts, and Component Classes
- Context Configuration with Context Initializers
- Context Configuration Inheritance
- Context Configuration with Environment Profiles
- Context Configuration with Test Property Sources
- Context Configuration with Dynamic Property Sources
- Loading a `WebApplicationContext`
- Context Caching
- Context Hierarchies

Context Configuration with XML resources

To load an `ApplicationContext` for your tests by 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`) is 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 that represents a resource URL (i.e., a path prefixed with `classpath:`, `file:`, `http:`, etc.) is used *as is*.

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from "/app-config.xml" and
// "/test-config.xml" in the root of the classpath
@ContextConfiguration(locations={"/app-config.xml", "/test-config.xml"}) ①
class MyTest {
    // class body...
}
```

① Setting the locations attribute to a list of XML files.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from "/app-config.xml" and
// "/test-config.xml" in the root of the classpath
@ContextConfiguration("/app-config.xml", "/test-config.xml") ①
class MyTest {
    // class body...
}
```

① Setting the locations attribute to a list of XML files.

`@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:

Java

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-config.xml"}) ①
class MyTest {
    // class body...
}
```

① Specifying XML files without using the `location` attribute.

Kotlin

```
@ExtendWith(SpringExtension::class)
@ContextConfiguration("/app-config.xml", "/test-config.xml") ①
class MyTest {
    // class body...
}
```

① Specifying XML files without using the `location` attribute.

If you omit both the `locations` and the `value` attributes from the `@ContextConfiguration` annotation, the TestContext framework tries 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"`. The following example shows how to do so:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTest-context.xml"
@ContextConfiguration ①
class MyTest {
    // class body...
}
```

① Loading configuration from the default location.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTest-context.xml"
@ContextConfiguration ①
class MyTest {
    // class body...
}
```

① Loading configuration from the default location.

Context Configuration with Groovy Scripts

To load an `ApplicationContext` for your tests by using Groovy scripts that use the Groovy Bean Definition DSL, you can 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.

The following example shows how to specify Groovy configuration files:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from "/AppConfig.groovy" and
// "/TestConfig.groovy" in the root of the classpath
@ContextConfiguration({"/AppConfig.groovy", "/TestConfig.Groovy"}) ①
class MyTest {
    // class body...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from "/AppConfig.groovy" and
// "/TestConfig.groovy" in the root of the classpath
@ContextConfiguration("/AppConfig.groovy", "/TestConfig.Groovy") ①
class MyTest {
    // class body...
}
```

① Specifying the location of Groovy configuration files.

If you omit both the `locations` and `value` attributes from the `@ContextConfiguration` annotation, the TestContext framework tries 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 loads your application context from `"classpath:com/example/MyTestContext.groovy"`. The following example shows how to use the default:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTestContext.groovy"
@ContextConfiguration ①
class MyTest {
    // class body...
}
```

① Loading configuration from the default location.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTestContext.groovy"
@ContextConfiguration ①
class MyTest {
    // class body...
}
```

① Loading configuration from the default location.

Declaring XML configuration and Groovy scripts simultaneously

You can declare both XML configuration files and Groovy scripts simultaneously by using the `locations` or `value` attribute of `@ContextConfiguration`. If the path to a configured resource location ends with `.xml`, it is loaded by using an `XmlBeanDefinitionReader`. Otherwise, it is loaded by using a `GroovyBeanDefinitionReader`.

The following listing shows how to combine both in an integration test:

Java

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

Kotlin

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

Context Configuration with Component Classes

To load an `ApplicationContext` for your tests by using component classes (see Java-based container configuration), you can annotate your test class with `@ContextConfiguration` and configure the `classes` attribute with an array that contains references to component classes. The following example shows how to do so:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from AppConfig and TestConfig
@ContextConfiguration(classes = {AppConfig.class, TestConfig.class}) ①
class MyTest {
    // class body...
}
```

① Specifying component classes.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from AppConfig and TestConfig
@ContextConfiguration(classes = [AppConfig::class, TestConfig::class]) ①
class MyTest {
    // class body...
}
```

① Specifying component classes.

Component Classes

The term “component class” can refer to any of the following:

- A class annotated with `@Configuration`.
- A component (that is, a class annotated with `@Component`, `@Service`, `@Repository`, or other stereotype annotations).
- A JSR-330 compliant class that is annotated with `jakarta.inject` annotations.
- Any class that contains `@Bean`-methods.
- Any other class that is intended to be registered as a Spring component (i.e., a Spring bean in the `ApplicationContext`), potentially taking advantage of automatic autowiring of a single constructor without the use of Spring annotations.

See the javadoc of `@Configuration` and `@Bean` for further information regarding the configuration and semantics of component classes, paying special attention to the discussion of `@Bean` Lite Mode.

If you omit the `classes` attribute from the `@ContextConfiguration` annotation, the TestContext framework tries to detect the presence of default configuration classes. Specifically, `AnnotationConfigContextLoader` and `AnnotationConfigWebContextLoader` detect all `static` nested classes of the test class that meet the requirements for configuration class implementations, as specified in the `@Configuration` javadoc. 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. In the following example, the `OrderServiceTest` class declares a `static` nested configuration class named `Config` that is automatically used to load the `ApplicationContext` for the test class:

Java

```
@SpringJUnitConfig ①
// ApplicationContext will be loaded from the
// static nested Config class
class OrderServiceTest {

    @Configuration
    static class Config {

        // this bean will be injected into the OrderServiceTest class
        @Bean
        OrderService orderService() {
            OrderService orderService = new OrderServiceImpl();
            // set properties, etc.
            return orderService;
        }
    }

    @Autowired
    OrderService orderService;

    @Test
    void testOrderService() {
        // test the orderService
    }

}
```

① Loading configuration information from the nested `Config` class.

Kotlin

```
@SpringJUnitConfig ①
// ApplicationContext will be loaded from the nested Config class
class OrderServiceTest {

    @Autowired
    lateinit var orderService: OrderService

    @Configuration
    class Config {

        // this bean will be injected into the OrderServiceTest class
        @Bean
        fun orderService(): OrderService {
            // set properties, etc.
            return OrderServiceImpl()
        }
    }

    @Test
    fun testOrderService() {
        // test the orderService
    }
}
```

① Loading configuration information from the nested `Config` class.

Mixing XML, Groovy Scripts, and Component Classes

It may sometimes be desirable to mix XML configuration files, Groovy scripts, and component classes (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 (such as Spring Boot) provide first-class support for loading an `ApplicationContext` from different types of resources simultaneously (for example, 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 for each 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` through `@ContextConfiguration`, and, with the standard testing support in the TestContext framework, you have the following options.

If you want to use resource locations (for example, XML or Groovy) and `@Configuration` classes to configure your tests, you must pick one as the entry point, and that one must include or import the

other. For example, in XML or Groovy scripts, you can include `@Configuration` classes by using component scanning or defining 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 define either a set of XML or Groovy resource locations or a set of `@Configuration` classes from which your production `ApplicationContext` is loaded, 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 by 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 are then 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 (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. The following example shows how to use initializers:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from TestConfig
// and initialized by TestAppCtxInitializer
@ContextConfiguration(
    classes = TestConfig.class,
    initializers = TestAppCtxInitializer.class) ①
class MyTest {
    // class body...
}
```

① Specifying configuration by using a configuration class and an initializer.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from TestConfig
// and initialized by TestAppCtxInitializer
@ContextConfiguration(
        classes = [TestConfig::class],
        initializers = [TestAppCtxInitializer::class]) ①
class MyTest {
    // class body...
}
```

① Specifying configuration by using a configuration class and an initializer.

You can also omit the declaration of XML configuration files, Groovy scripts, or component 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. The following example shows how to do so:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be initialized by EntireAppInitializer
// which presumably registers beans in the context
@ContextConfiguration(initializers = EntireAppInitializer.class) ①
class MyTest {
    // class body...
}
```

① Specifying configuration by using only an initializer.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be initialized by EntireAppInitializer
// which presumably registers beans in the context
@ContextConfiguration(initializers = [EntireAppInitializer::class]) ①
class MyTest {
    // class body...
}
```

① Specifying configuration by using only an initializer.

Context Configuration Inheritance

`@ContextConfiguration` supports boolean `inheritLocations` and `inheritInitializers` attributes that denote whether resource locations or component 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 component classes as well as the context initializers declared by any superclasses. Specifically, the resource locations or component 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 are added to the set of initializers defined by test superclasses. Thus, subclasses have the option of extending the resource locations, component classes, or context initializers.

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

As of Spring Framework 5.3, test configuration may also be inherited from enclosing classes. See `@Nested` test class configuration for details.

In the next example, which uses XML resource locations, the `ApplicationContext` for `ExtendedTest` is loaded from `base-config.xml` and `extended-config.xml`, in that order. Beans defined in `extended-`

`config.xml` can, therefore, override (that is, replace) those defined in `base-config.xml`. The following example shows how one class can extend another and use both its own configuration file and the superclass's configuration file:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from "/base-config.xml"
// in the root of the classpath
@ContextConfiguration("/base-config.xml") ①
class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from "/base-config.xml" and
// "/extended-config.xml" in the root of the classpath
@ContextConfiguration("/extended-config.xml") ②
class ExtendedTest extends BaseTest {
    // class body...
}
```

① Configuration file defined in the superclass.

② Configuration file defined in the subclass.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from "/base-config.xml"
// in the root of the classpath
@ContextConfiguration("/base-config.xml") ①
open class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from "/base-config.xml" and
// "/extended-config.xml" in the root of the classpath
@ContextConfiguration("/extended-config.xml") ②
class ExtendedTest : BaseTest() {
    // class body...
}
```

① Configuration file defined in the superclass.

② Configuration file defined in the subclass.

Similarly, in the next example, which uses component classes, the `ApplicationContext` for `ExtendedTest` is loaded from the `BaseConfig` and `ExtendedConfig` classes, in that order. Beans defined in `ExtendedConfig` can, therefore, override (that is, replace) those defined in `BaseConfig`. The following example shows how one class can extend another and use both its own configuration class and the superclass's configuration class:

Java

```
// ApplicationContext will be loaded from BaseConfig
@SpringJUnitConfig(BaseConfig.class) ①
class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from BaseConfig and ExtendedConfig
@SpringJUnitConfig(ExtendedConfig.class) ②
class ExtendedTest extends BaseTest {
    // class body...
}
```

① Configuration class defined in the superclass.

② Configuration class defined in the subclass.

Kotlin

```
// ApplicationContext will be loaded from BaseConfig
@SpringJUnitConfig(BaseConfig::class) ①
open class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from BaseConfig and ExtendedConfig
@SpringJUnitConfig(ExtendedConfig::class) ②
class ExtendedTest : BaseTest() {
    // class body...
}
```

① Configuration class defined in the superclass.

② Configuration class defined in the subclass.

In the next example, which uses context initializers, the `ApplicationContext` for `ExtendedTest` is initialized by 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. The following example shows how one class can extend another and use both its own initializer and the superclass's initializer:

Java

```
// ApplicationContext will be initialized by BaseInitializer
@SpringJUnitConfig(initializers = BaseInitializer.class) ①
class BaseTest {
    // class body...
}

// ApplicationContext will be initialized by BaseInitializer
// and ExtendedInitializer
@SpringJUnitConfig(initializers = ExtendedInitializer.class) ②
class ExtendedTest extends BaseTest {
    // class body...
}
```

① Initializer defined in the superclass.

② Initializer defined in the subclass.

Kotlin

```
// ApplicationContext will be initialized by BaseInitializer
@SpringJUnitConfig(initializers = [BaseInitializer::class]) ①
open class BaseTest {
    // class body...
}

// ApplicationContext will be initialized by BaseInitializer
// and ExtendedInitializer
@SpringJUnitConfig(initializers = [ExtendedInitializer::class]) ②
class ExtendedTest : BaseTest() {
    // class body...
}
```

① Initializer defined in the superclass.

② Initializer defined in the subclass.

Context Configuration with Environment Profiles

The Spring Framework has first-class support for the notion of environments and profiles (AKA "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.

You can use `@ActiveProfiles` with any implementation of the `SmartContextLoader` SPI, but `@ActiveProfiles` is not supported with implementations of the older `ContextLoader` SPI.

Consider two 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>
```

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from "classpath:/app-config.xml"
@ContextConfiguration("/app-config.xml")
@ActiveProfiles("dev")
class TransferServiceTest {

    @Autowired
    TransferService transferService;

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

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from "classpath:/app-config.xml"
@ContextConfiguration("/app-config.xml")
@ActiveProfiles("dev")
class TransferServiceTest {

    @Autowired
    lateinit var transferService: TransferService

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

When `TransferServiceTest` is run, its `ApplicationContext` is loaded from the `app-config.xml` configuration file in the root of the classpath. If you inspect `app-config.xml`, you can see 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, in the `dev` profile, and in 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 is created and populated with test data, and the `accountRepository` bean is wired with a reference to the development `DataSource`. That is 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 included only when no other profile is specifically activated. You can use this 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 with `@Configuration` classes instead of XML:

Java

```
@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();
    }
}
```

Kotlin

```
@Configuration
@Profile("dev")
class StandaloneDataConfig {

    @Bean
    fun dataSource(): DataSource {
        return EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("classpath:com/bank/config/sql/schema.sql")
                .addScript("classpath:com/bank/config/sql/test-data.sql")
                .build()
    }
}
```

Java

```
@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");
    }
}
```

Kotlin

```
@Configuration
@Profile("production")
class JndiDataConfig {

    @Bean(destroyMethod = "")
    fun dataSource(): DataSource {
        val ctx = InitialContext()
        return ctx.lookup("java:comp/env/jdbc/datasource") as DataSource
    }
}
```

Java

```
@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();
    }
}
```

Kotlin

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

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

Java

```
@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();
    }
}
```

Kotlin

```
@Configuration
class TransferServiceConfig {

    @Autowired
    lateinit var dataSource: DataSource

    @Bean
    fun transferService(): TransferService {
        return DefaultTransferService(accountRepository(), feePolicy())
    }

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }

    @Bean
    fun feePolicy(): FeePolicy {
        return ZeroFeePolicy()
    }
}
```

Java

```
@SpringJUnitConfig({
        TransferServiceConfig.class,
        StandaloneDataConfig.class,
        JndiDataConfig.class,
        DefaultDataConfig.class})
@ActiveProfiles("dev")
class TransferServiceTest {

    @Autowired
    TransferService transferService;

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

Kotlin

```
@SpringJUnitConfig(
        TransferServiceConfig::class,
        StandaloneDataConfig::class,
        JndiDataConfig::class,
        DefaultDataConfig::class)
@ActiveProfiles("dev")
class TransferServiceTest {

    @Autowired
    lateinit var transferService: TransferService

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

In this variation, we have split the XML configuration into four independent `@Configuration` classes:

- `TransferServiceConfig`: Acquires a `dataSource` through dependency injection by 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 by using 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, you can declare `@ActiveProfiles` once on a base class, and subclasses 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`:

As of Spring Framework 5.3, test configuration may also be inherited from enclosing classes. See `@Nested` test class configuration for details.

Java

```
@SpringJUnitConfig({
        TransferServiceConfig.class,
        StandaloneDataConfig.class,
        JndiDataConfig.class,
        DefaultDataConfig.class})
@ActiveProfiles("dev")
abstract class AbstractIntegrationTest {
}
```

Kotlin

```
@SpringJUnitConfig(
        TransferServiceConfig::class,
        StandaloneDataConfig::class,
        JndiDataConfig::class,
        DefaultDataConfig::class)
@ActiveProfiles("dev")
abstract class AbstractIntegrationTest {
}
```

Java

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

    @Autowired
    TransferService transferService;

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

Kotlin

```
// "dev" profile inherited from superclass
class TransferServiceTest : AbstractIntegrationTest() {

    @Autowired
    lateinit var transferService: TransferService

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

`@ActiveProfiles` also supports an `inheritProfiles` attribute that can be used to disable the inheritance of active profiles, as the following example shows:

Java

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

Kotlin

```
// "dev" profile overridden with "production"
@ActiveProfiles("production", inheritProfiles = false)
class ProductionTransferServiceTest : 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 run on a continuous integration build server.
- The presence of certain environment variables.
- The presence of custom class-level annotations.
- Other concerns.

To resolve active bean definition profiles programmatically, you can implement a custom `ActiveProfilesResolver` and register it by using the `resolver` attribute of `@ActiveProfiles`. For further information, see the corresponding javadoc. The following example demonstrates how to implement and register a custom `OperatingSystemActiveProfilesResolver`:

Java

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

Kotlin

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

Java

```
public class OperatingSystemActiveProfilesResolver implements ActiveProfilesResolver {

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

Kotlin

```
class OperatingSystemActiveProfilesResolver : ActiveProfilesResolver {

    override fun resolve(testClass: Class<*>): Array<String> {
        val profile: String = ...
        // determine the value of profile based on the operating system
        return arrayOf(profile)
    }
}
```

Context Configuration with Test Property Sources

The Spring Framework has first-class support for the notion of an environment with a hierarchy of property sources, and you can configure integration tests with test-specific property sources. In contrast to the `@PropertySource` annotation used on `@Configuration` classes, you can declare the

`@TestPropertySource` annotation on a test class to declare resource locations for test properties files or inlined properties. These test property sources are added to the set of `PropertySources` in the `Environment` for the `ApplicationContext` loaded for the annotated integration test.

You can use `@TestPropertySource` 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 through the `getPropertySourceLocations()` and `getPropertySourceProperties()` methods in `MergedContextConfiguration`.

Declaring Test Property Sources

You can configure test properties files by using the `locations` or `value` attribute of `@TestPropertySource`.

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 is interpreted as a Spring `Resource`. A plain path (for example, `"test.properties"`) is 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 resource (for example: `"/org/example/test.xml"`). A path that references a URL (for example, a path prefixed with `classpath:`, `file:`, or `http:`) is loaded by using the specified resource protocol. Resource location wildcards (such as `*/.properties`) are not permitted: Each location must evaluate to exactly one `.properties` or `.xml` resource.

The following example uses a test properties file:

Java

```
@ContextConfiguration
@TestPropertySource("/test.properties") ①
class MyIntegrationTests {
    // class body...
}
```

① Specifying a properties file with an absolute path.

Kotlin

```
@ContextConfiguration
@TestPropertySource("/test.properties") ①
class MyIntegrationTests {
    // class body...
}
```

① Specifying a properties file with an absolute path.

You can configure inlined properties in the form of key-value pairs by using the `properties` attribute of `@TestPropertySource`, as shown in the next example. All key-value pairs are 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`

The following example sets two inlined properties:

Java

```
@ContextConfiguration
@TestPropertySource(properties = {"timezone = GMT", "port: 4242"}) ①
class MyIntegrationTests {
    // class body...
}
```

① Setting two properties by using two variations of the key-value syntax.

Kotlin

```
@ContextConfiguration
@TestPropertySource(properties = ["timezone = GMT", "port: 4242"]) ①
class MyIntegrationTests {
    // class body...
}
```

① Setting two properties by using two variations of the key-value syntax.

As of Spring Framework 5.2, `@TestPropertySource` can be used as *repeatable annotation*. That means that you can have multiple declarations of `@TestPropertySource` on a single test class, with the `locations` and `properties` from later `@TestPropertySource` annotations overriding those from previous `@TestPropertySource` annotations.

In addition, you may declare multiple composed annotations on a test class that are each meta-annotated with `@TestPropertySource`, and all of those `@TestPropertySource` declarations will contribute to your test property sources.

Directly present `@TestPropertySource` annotations always take precedence over meta-present `@TestPropertySource` annotations. In other words, `locations` and `properties` from a directly present `@TestPropertySource` annotation will override the `locations` and `properties` from a `@TestPropertySource` annotation used as a meta-annotation.

Default Properties File Detection

If `@TestPropertySource` is declared as an empty annotation (that is, without explicit values for the `locations` or `properties` attributes), an attempt is 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` is thrown.

Precedence

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

In the next example, the `timezone` and `port` properties and any properties defined in `"/test.properties"` 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 are overridden by the inlined properties declared by using the `properties` attribute. The following example shows how to specify properties both in a file and inline:

Java

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

Kotlin

```
@ContextConfiguration
@TestPropertySource("/test.properties",
        properties = ["timezone = GMT", "port: 4242"]
)
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 shadow (that is, 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.

As of Spring Framework 5.3, test configuration may also be inherited from enclosing classes. See `@Nested` test class configuration for details.

In the next example, the `ApplicationContext` for `BaseTest` is loaded by using only the `base.properties` file as a test property source. In contrast, the `ApplicationContext` for `ExtendedTest` is loaded by using the `base.properties` and `extended.properties` files as test property source locations. The following example shows how to define properties in both a subclass and its superclass by using `properties` files:

Java

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

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

Kotlin

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

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

In the next example, the `ApplicationContext` for `BaseTest` is loaded by using only the inlined `key1` property. In contrast, the `ApplicationContext` for `ExtendedTest` is loaded by using the inlined `key1` and `key2` properties. The following example shows how to define properties in both a subclass and its superclass by using inline properties:

Java

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

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

Kotlin

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

@TestPropertySource(properties = ["key2 = value2"])
@ContextConfiguration
class ExtendedTest : BaseTest() {
    // ...
}
```

Context Configuration with Dynamic Property Sources

As of Spring Framework 5.2.5, the TestContext framework provides support for *dynamic* properties via the `@DynamicPropertySource` annotation. This annotation can be used in integration tests that need to add properties with dynamic values to the set of `PropertySources` in the `Environment` for the `ApplicationContext` loaded for the integration test.

The `@DynamicPropertySource` annotation and its supporting infrastructure were originally designed to allow properties from Testcontainers based tests to be exposed easily to Spring integration tests. However, this feature may also be used with any form of external resource whose lifecycle is maintained outside the test's `ApplicationContext`.

In contrast to the `@TestPropertySource` annotation that is applied at the class level, `@DynamicPropertySource` must be applied to a `static` method that accepts a single `DynamicPropertyRegistry` argument which is used to add *name-value* pairs to the `Environment`. Values

are dynamic and provided via a `Supplier` which is only invoked when the property is resolved. Typically, method references are used to supply values, as can be seen in the following example which uses the Testcontainers project to manage a Redis container outside of the Spring `ApplicationContext`. The IP address and port of the managed Redis container are made available to components within the test's `ApplicationContext` via the `redis.host` and `redis.port` properties. These properties can be accessed via Spring's `Environment` abstraction or injected directly into Spring-managed components – for example, via `@Value("${redis.host}")` and `@Value("${redis.port}")`, respectively.

If you use `@DynamicPropertySource` in a base class and discover that tests in subclasses fail because the dynamic properties change between subclasses, you may need to annotate your base class with `@DirtiesContext` to ensure that each subclass gets its own `ApplicationContext` with the correct dynamic properties.

Java

```
@SpringJUnitConfig(/* ... */)
@Testcontainers
class ExampleIntegrationTests {

    @Container
    static RedisContainer redis = new RedisContainer();

    @DynamicPropertySource
    static void redisProperties(DynamicPropertyRegistry registry) {
        registry.add("redis.host", redis::getContainerIpAddress);
        registry.add("redis.port", redis::getMappedPort);
    }

    // tests ...

}
```

Kotlin

```
@SpringJUnitConfig(/* ... */)
@Testcontainers
class ExampleIntegrationTests {

    companion object {

        @Container
        @JvmStatic
        val redis: RedisContainer = RedisContainer()

        @DynamicPropertySource
        @JvmStatic
        fun redisProperties(registry: DynamicPropertyRegistry) {
            registry.add("redis.host", redis::getContainerIpAddress)
            registry.add("redis.port", redis::getMappedPort)
        }
    }

    // tests ...

}
```

Precedence

Dynamic properties have higher precedence than those loaded from `@TestPropertySource`, the operating system's environment, Java system properties, or property sources added by the application declaratively by using `@PropertySource` or programmatically. Thus, dynamic properties can be used to selectively override properties loaded via `@TestPropertySource`, system property sources, and application property sources.

Loading a `WebApplicationContext`

To instruct the TestContext framework to load a `WebApplicationContext` instead of a standard `ApplicationContext`, you can 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` is set to `src/main/webapp`. This is interpreted as a path relative to the root of your JVM (normally the path to your project). If you are familiar with the directory structure of a web application in a Maven project, you know that `src/main/webapp` is the default location for the root of your WAR. If you need to override this default, you can provide an alternate path to the `@WebAppConfiguration` annotation (for example, `@WebAppConfiguration("src/test/webapp")`). If you wish to reference a base resource path from the classpath instead of the file system, you can use Spring's `classpath:` prefix.

Note that Spring's testing support for `WebApplicationContext` implementations is on par with its support for standard `ApplicationContext` implementations. When testing with a

`WebApplicationContext`, you are free to declare XML configuration files, Groovy scripts, or `@Configuration` classes by using `@ContextConfiguration`. You are also free to use any other test annotations, such as `@ActiveProfiles`, `@TestExecutionListeners`, `@Sql`, `@Rollback`, and others.

The remaining examples in this section show some of the various configuration options for loading a `WebApplicationContext`. The following example shows the TestContext framework's support for convention over configuration:

Java

```
@ExtendWith(SpringExtension.class)

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

// detects "WacTests-context.xml" in the same package
// or static nested @Configuration classes
@ContextConfiguration
class WacTests {
    //...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)

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

// detects "WacTests-context.xml" in the same package
// or static nested @Configuration classes
@ContextConfiguration
class WacTests {
    //...
}
```

If you annotate a test class with `@WebAppConfiguration` without specifying a resource base path, the resource path effectively defaults to `file:src/main/webapp`. Similarly, if you declare `@ContextConfiguration` without specifying resource `locations`, component `classes`, or context `initializers`, Spring tries to detect the presence of your configuration by using conventions (that is, `WacTests-context.xml` in the same package as the `WacTests` class or static nested `@Configuration` classes).

The following example shows how to explicitly declare a resource base path with `@WebAppConfiguration` and an XML resource location with `@ContextConfiguration`:

Java

```
@ExtendWith(SpringExtension.class)

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

// classpath resource
@ContextConfiguration("/spring/test-servlet-config.xml")
class WacTests {
    //...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)

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

// classpath resource
@ContextConfiguration("/spring/test-servlet-config.xml")
class WacTests {
    //...
}
```

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.

The following example shows that we can override the default resource semantics for both annotations by specifying a Spring resource prefix:

Java

```
@ExtendWith(SpringExtension.class)

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

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

Kotlin

```
@ExtendWith(SpringExtension::class)

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

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

Contrast the comments in this example with the previous example.

Working with Web Mocks

To provide comprehensive web testing support, the TestContext framework has a `ServletTestExecutionListener` that is enabled by default. When testing against a `WebApplicationContext`, this `TestExecutionListener` sets up default thread-local state by using Spring Web's `RequestContextHolder` before each test method and creates a `MockHttpServletRequest`, a `MockHttpServletResponse`, and a `ServletWebRequest` based on the base resource path configured with `@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 shows 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`.

Java

```
@SpringJUnitWebConfig
class WacTests {

    @Autowired
    WebApplicationContext wac; // cached

    @Autowired
    MockServletContext servletContext; // cached

    @Autowired
    MockHttpSession session;

    @Autowired
    MockHttpServletRequest request;

    @Autowired
    MockHttpServletResponse response;

    @Autowired
    ServletWebRequest webRequest;

    //...
}
```

Kotlin

```
@SpringJUnitWebConfig
class WacTests {

    @Autowired
    lateinit var wac: WebApplicationContext // cached

    @Autowired
    lateinit var servletContext: MockServletContext // cached

    @Autowired
    lateinit var session: MockHttpSession

    @Autowired
    lateinit var request: MockHttpServletRequest

    @Autowired
    lateinit var response: MockHttpServletResponse

    @Autowired
    lateinit var webRequest: ServletWebRequest

    //...
}
```

Context Caching

Once the TestContext framework loads an `ApplicationContext` (or `WebApplicationContext`) for a test, that context is 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`) – this includes `@DynamicPropertySource` methods as well as various features from Spring Boot's testing support such as `@MockBean` and `@SpyBean`.
- `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 loads the corresponding `ApplicationContext` and stores 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` is 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 run in separate processes, the static cache is cleared between each test execution, which effectively disables 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 cannot cache application contexts between test classes, and the build process runs significantly more slowly as a result.

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. You can configure the maximum size from the command line or a build script by setting a JVM system property named `spring.test.context.cache.maxSize`. As an alternative, you can set the same property via the `SpringProperties` mechanism.

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 run, it is often beneficial to know exactly how many contexts have been loaded and cached. To view the statistics for the underlying context cache, you can 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 running the next test that requires the same application context. Note that support for the `@DirtiesContext` annotation is provided by the `DirtiesContextBeforeModesTestExecutionListener` and the `DirtiesContextTestExecutionListener`, which are enabled by default.

ApplicationContext lifecycle and console logging

When you need to debug a test executed with the Spring TestContext Framework, it can be useful to analyze the console output (that is, output to the `SYSOUT` and `SYSERR` streams). Some build tools and IDEs are able to associate console output with a given test; however, some console output cannot be easily associated with a given test.

With regard to console logging triggered by the Spring Framework itself or by components registered in the `ApplicationContext`, it is important to understand the lifecycle of an `ApplicationContext` that has been loaded by the Spring TestContext Framework within a test suite.

The `ApplicationContext` for a test is typically loaded when an instance of the test class is being prepared — for example, to perform dependency injection into `@Autowired` fields of the test instance. This means that any console logging triggered during the initialization of the `ApplicationContext` typically cannot be associated with an individual test method. However, if the context is closed immediately before the execution of a test method according to `@DirtiesContext` semantics, a new instance of the context will be loaded just prior to execution of the test method. In the latter scenario, an IDE or build tool may potentially associate console logging with the individual test method.

The `ApplicationContext` for a test can be closed via one of the following scenarios.

- The context is closed according to `@DirtiesContext` semantics.
- The context is closed because it has been automatically evicted from the cache according to the LRU eviction policy.
- The context is closed via a JVM shutdown hook when the JVM for the test suite terminates.

If the context is closed according to `@DirtiesContext` semantics after a particular test method, an IDE or build tool may potentially associate console logging with the individual test method. If the context is closed according to `@DirtiesContext` semantics after a test class, any console logging triggered during the shutdown of the `ApplicationContext` cannot be associated with an individual test method. Similarly, any console logging triggered during the shutdown phase via a JVM shutdown hook cannot be associated with an individual test method.

When a Spring `ApplicationContext` is closed via a JVM shutdown hook, callbacks executed during the shutdown phase are executed on a thread named `SpringContextShutdownHook`. So, if you wish to disable console logging triggered when the `ApplicationContext` is closed via a JVM shutdown hook, you may be able to register a custom filter with your logging framework that allows you to ignore any logging initiated by that thread.

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` instances. For example, if you are developing a Spring MVC web application, you typically have a root `WebApplicationContext` loaded by Spring's `ContextLoaderListener` and a child `WebApplicationContext` loaded by 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.

You can write integration tests that use context hierarchies by declaring context configuration with 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, you can also 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 (that is, XML configuration files or component classes) must be consistent. Otherwise, it is perfectly acceptable to have different levels in a context hierarchy configured using different resource types.

The remaining JUnit Jupiter based examples in this section show 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 that consists of two levels, one for the root `WebApplicationContext` (loaded by using the `TestAppConfig` `@Configuration` class) and one for the dispatcher servlet `WebApplicationContext` (loaded by using the `WebConfig` `@Configuration` class). The `WebApplicationContext` that is autowired into the test instance is the one for the child context (that is, the lowest context in the hierarchy). The following listing shows this configuration scenario:

Java

```
@ExtendWith(SpringExtension.class)
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = TestAppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
class ControllerIntegrationTests {

    @Autowired
    WebApplicationContext wac;

    // ...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
@WebAppConfiguration
@ContextHierarchy(
    ContextConfiguration(classes = [TestAppConfig::class]),
    ContextConfiguration(classes = [WebConfig::class]))
class ControllerIntegrationTests {

    @Autowired
    lateinit var wac: WebApplicationContext

    // ...
}
```

Class hierarchy with implicit parent context

The test classes in this example 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 follow the standard semantics for `@ContextConfiguration`. `SoapWebServiceTests` and `RestWebServiceTests` both extend `AbstractWebTests` and define a context hierarchy by using `@ContextHierarchy`. The result is that three application contexts are loaded (one for each declaration of `@ContextConfiguration`), and the application context loaded based on the configuration in `AbstractWebTests` is set as the parent context for each of the contexts loaded for the concrete subclasses. The following listing shows this configuration scenario:

Java

```
@ExtendWith(SpringExtension.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 {}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
@WebAppConfiguration
@ContextConfiguration("file:src/main/webapp/WEB-INF/applicationContext.xml")
abstract class AbstractWebTests

@ContextHierarchy(ContextConfiguration("/spring/soap-ws-config.xml"))
class SoapWebServiceTests : AbstractWebTests()

@ContextHierarchy(ContextConfiguration("/spring/rest-ws-config.xml"))
class RestWebServiceTests : AbstractWebTests()
```

Class hierarchy with merged context hierarchy configuration

The classes in this example show 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, by ensuring that the names declared in the `name` attribute in `@ContextConfiguration` are both `child`. The result is that three application contexts are 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` is set as the parent context for the contexts loaded from `/user-config.xml` and `{"/user-config.xml", "/order-config.xml"}`. The following listing shows this configuration scenario:

Java

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

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

Kotlin

```
@ExtendWith(SpringExtension::class)
@ContextHierarchy(
    ContextConfiguration(name = "parent", locations = ["/app-config.xml"]),
    ContextConfiguration(name = "child", locations = ["/user-config.xml"]))
open class BaseTests {}

@ContextHierarchy(
    ContextConfiguration(name = "child", locations = ["/order-config.xml"])
)
class ExtendedTests : 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` is loaded only from `/test-user-config.xml` and has its parent set to the context loaded from `/app-config.xml`. The following listing shows this configuration scenario:

Java

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

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

Kotlin

```
@ExtendWith(SpringExtension::class)
@ContextHierarchy(
    ContextConfiguration(name = "parent", locations = ["/app-config.xml"]),
    ContextConfiguration(name = "child", locations = ["/user-config.xml"]))
open class BaseTests {}

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

Dirtying a context within a context hierarchy

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

3.5.7. 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` or related annotations. 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. If you are using JUnit Jupiter you may also optionally use constructor injection (see Dependency Injection with `SpringExtension`). For consistency with Spring's annotation-based injection support, you may also use Spring's `@Autowired` annotation or the `@Inject` annotation from JSR-330 for field and setter injection.

For testing frameworks other than JUnit Jupiter, the TestContext framework does not participate in instantiation of the test class. Thus, the use of `@Autowired` or `@Inject` for constructors has no effect for test classes.

Although field injection is discouraged in production code, field injection is actually quite natural in test code. The rationale for the difference is that you will never instantiate your test class directly. Consequently, there is no need to be able to invoke a `public` constructor or setter method on your test class.

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`. You can 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",`

`TitleRepository.class)`.

If you do not want dependency injection applied to your test instances, 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 Jupiter. The same DI techniques can be used in conjunction with any supported testing framework.

The following examples make calls to static assertion methods, such as `assertNotNull()`, but without prepending the call with `Assertions`. 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 Jupiter based implementation of the test class that uses `@Autowired` for field injection:

Java

```
@ExtendWith(SpringExtension.class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
class HibernateTitleRepositoryTests {

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

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

Kotlin

```
@ExtendWith(SpringExtension::class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    @Autowired
    lateinit var titleRepository: HibernateTitleRepository

    @Test
    fun findById() {
        val title = titleRepository.findById(10)
        assertNotNull(title)
    }
}
```

Alternatively, you can configure the class to use `@Autowired` for setter injection, as follows:

Java

```
@ExtendWith(SpringExtension.class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
class HibernateTitleRepositoryTests {

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

    @Autowired
    void setTitleRepository(HibernateTitleRepository titleRepository) {
        this.titleRepository = titleRepository;
    }

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

Kotlin

```
@ExtendWith(SpringExtension::class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    lateinit var titleRepository: HibernateTitleRepository

    @Autowired
    fun setTitleRepository(titleRepository: HibernateTitleRepository) {
        this.titleRepository = titleRepository
    }

    @Test
    fun findById() {
        val title = titleRepository.findById(10)
        assertNotNull(title)
    }
}
```

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

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

    <!-- this bean will be injected into the HibernateTitleRepositoryTests class -->
    <bean id="titleRepository"
class="com.foo.repository.hibernate.HibernateTitleRepository">
        <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):

Java

```
// ...

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

// ...
```

Kotlin

```
// ...

    @Autowired
    override fun setDataSource(@Qualifier("myDataSource") dataSource:
DataSource) {
        super.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 can effectively also point to a specific bean by name there (as shown earlier, assuming that `myDataSource` is the bean `id`).

3.5.8. Testing Request- and Session-scoped Beans

Spring has supported Request- and session-scoped beans since the early years, and you can 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` (with dependency injection).
- Perform assertions against the mocks.

The next code snippet shows 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 by using SpEL expressions that retrieve the username and password from the current HTTP request. In our test, we want to configure these request parameters through the mock managed by the TestContext framework. The following listing shows the configuration for this use case:

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` (that is, 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` (that is, the one in which we just set parameters). We can then perform assertions against the results based on the known inputs for the username and password. The following listing shows how to do so:

Java

```
@SpringJUnitWebConfig
class RequestScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpServletRequest request;

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

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

Kotlin

```
@SpringJUnitWebConfig
class RequestScopedBeanTests {

    @Autowired lateinit var userService: UserService
    @Autowired lateinit var request: MockHttpServletRequest

    @Test
    fun requestScope() {
        request.setParameter("user", "enigma")
        request.setParameter("pswd", "\$pr!ng")

        val results = userService.loginUser()
        // assert results
    }
}
```

The following code snippet is similar to the one we saw earlier 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 by using a SpEL expression that retrieves the theme from the current HTTP session. In our test, we need to configure a theme in the mock session managed by the TestContext framework. The following example shows how to do so:

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. The following example shows how to do so:

Java

```
@SpringJUnitWebConfig
class SessionScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpSession session;

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

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

Kotlin

```
@SpringJUnitWebConfig
class SessionScopedBeanTests {

    @Autowired lateinit var userService: UserService
    @Autowired lateinit var session: MockHttpSession

    @Test
    fun sessionScope() {
        session.setAttribute("theme", "blue")

        val results = userService.processUserPreferences()
        // assert results
    }
}
```

3.5.9. 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 with `@ContextConfiguration` semantics (further details are provided later). In addition, you must declare Spring's `@Transactional` annotation either at the class or the method level for your tests.

Test-managed Transactions

Test-managed transactions are transactions that are managed declaratively by using the `TransactionalTestExecutionListener` or programmatically by using `TestTransaction` (described later). You should not confuse such transactions with Spring-managed transactions (those managed directly by Spring within the `ApplicationContext` loaded for tests) or application-managed transactions (those managed programmatically within application code that is invoked by tests). Spring-managed and application-managed transactions typically participate in test-managed transactions. However, you should use caution 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).

Preemptive timeouts and test-managed transactions

Caution must be taken when using any form of preemptive timeouts from a testing framework in conjunction with Spring's test-managed transactions.

Specifically, Spring's testing support binds transaction state to the current thread (via a `java.lang.ThreadLocal` variable) *before* the current test method is invoked. If a testing framework invokes the current test method in a new thread in order to support a preemptive timeout, any actions performed within the current test method will *not* be invoked within the test-managed transaction. Consequently, the result of any such actions will not be rolled back with the test-managed transaction. On the contrary, such actions will be committed to the persistent store — for example, a relational database — even though the test-managed transaction is properly rolled back by Spring.

Situations in which this can occur include but are not limited to the following.

- JUnit 4's `@Test(timeout = …)` support and `TimeOut` rule
- JUnit Jupiter's `assertTimeoutPreemptively(…)` methods in the `org.junit.jupiter.api.Assertions` class
- TestNG's `@Test(timeOut = …)` support

Enabling and Disabling Transactions

Annotating a test method with `@Transactional` causes the test to be run within a transaction that is, by default, automatically rolled back after completion of the test. If a test class is annotated with `@Transactional`, each test method within that class hierarchy runs within a transaction. Test methods that are not annotated with `@Transactional` (at the class or method level) are not run within a transaction. Note that `@Transactional` is not supported on test lifecycle methods — for example, methods annotated with JUnit Jupiter's `@BeforeAll`, `@BeforeEach`, etc. Furthermore, tests that are annotated with `@Transactional` but have the `propagation` attribute set to `NOT_SUPPORTED` or `NEVER` are not run within a transaction.

Table 1. `@Transactional` attribute support

Attribute	Supported for test-managed transactions
`value` and `transactionManager`	yes
`propagation`	only `Propagation.NOT_SUPPORTED` and `Propagation.NEVER` are supported
`isolation`	no
`timeout`	no
`readOnly`	no
`rollbackFor` and `rollbackForClassName`	no: use `TestTransaction.flagForRollback()` instead
`noRollbackFor` and `noRollbackForClassName`	no: use `TestTransaction.flagForCommit()` instead

Method-level lifecycle methods — for example, methods annotated with JUnit Jupiter's `@BeforeEach` or `@AfterEach` — are run within a test-managed transaction. On the other hand, suite-level and class-level lifecycle methods — for example, methods annotated with JUnit Jupiter's `@BeforeAll` or `@AfterAll` and methods annotated with TestNG's `@BeforeSuite`, `@AfterSuite`, `@BeforeClass`, or `@AfterClass` — are *not* run within a test-managed transaction.

If you need to run code in a suite-level or class-level lifecycle method within a transaction, you may wish to inject a corresponding `PlatformTransactionManager` into your test class and then use that with a `TransactionTemplate` for programmatic transaction management.

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`:

Java

```
@SpringJUnitConfig(TestConfig.class)
@Transactional
class HibernateUserRepositoryTests {

    @Autowired
    HibernateUserRepository repository;

    @Autowired
    SessionFactory sessionFactory;

    JdbcTemplate jdbcTemplate;

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

    @Test
    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);
    }

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

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

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
@Transactional
class HibernateUserRepositoryTests {

    @Autowired
    lateinit var repository: HibernateUserRepository

    @Autowired
    lateinit var sessionFactory: SessionFactory

    lateinit var jdbcTemplate: JdbcTemplate

    @Autowired
    fun setDataSource(dataSource: DataSource) {
        this.jdbcTemplate = JdbcTemplate(dataSource)
    }

    @Test
    fun createUser() {
        // track initial state in test database:
        val count = countRowsInTable("user")

        val user = User()
        repository.save(user)

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

    private fun countRowsInTable(tableName: String): Int {
        return JdbcTestUtils.countRowsInTable(jdbcTemplate, tableName)
    }

    private fun assertNumUsers(expected: Int) {
        assertEquals("Number of rows in the [user] table.", expected,
countRowsInTable("user"))
    }
}
```

As explained in Transaction Rollback and Commit Behavior, there is no need to clean up the database after the `createUser()` method runs, since any changes made to the database are automatically rolled back by the `TransactionalTestExecutionListener`.

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

You can interact with test-managed transactions programmatically by using the static methods in `TestTransaction`. For example, you can use `TestTransaction` 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`. See the javadoc for `TestTransaction` for further details.

Java

```
@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"));
    }
}
```

Kotlin

```
@ContextConfiguration(classes = [TestConfig::class])
class ProgrammaticTransactionManagementTests :
AbstractTransactionalJUnit4SpringContextTests() {

    @Test
    fun 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 fun assertNumUsers(expected: Int) {
        assertEquals("Number of rows in the [user] table.", expected,
countRowsInTable("user"))
    }
}
```

Running Code Outside of a Transaction

Occasionally, you may need to run certain code before or after a transactional test method but outside the transactional context — for example, to verify the initial database state prior to running your test or to verify expected transactional commit behavior after your test runs (if the test was configured to commit the transaction). `TransactionalTestExecutionListener` supports the `@BeforeTransaction` and `@AfterTransaction` annotations for exactly such scenarios. You can 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 runs 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 run within a transaction. In addition, methods annotated with `@BeforeTransaction` or `@AfterTransaction` are not run 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. If there are multiple instances of `PlatformTransactionManager` within the test's `ApplicationContext`, you can declare a qualifier by using `@Transactional("myTxMgr")` or `@Transactional(transactionManager = "myTxMgr")`, or `TransactionManagementConfigurer` can be implemented by an `@Configuration` class. Consult the javadoc 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 Jupiter based example displays a fictitious integration testing scenario that highlights all transaction-related annotations. The example is not intended to demonstrate best practices but rather to demonstrate how these annotations can be used. See the annotation support section for further information and configuration examples. Transaction management for `@Sql` contains an additional example that uses `@Sql` for declarative SQL script execution with default transaction rollback semantics. The following example shows the relevant annotations:

Java

```
@SpringJUnitConfig
@Transactional(transactionManager = "txMgr")
@Commit
class FictitiousTransactionalTest {

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

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

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

    @AfterEach
    void tearDownWithinTransaction() {
        // run "tear down" logic within the transaction
    }

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

}
```

Kotlin

```
@SpringJUnitConfig
@Transactional(transactionManager = "txMgr")
@Commit
class FictitiousTransactionalTest {

    @BeforeTransaction
    fun verifyInitialDatabaseState() {
        // logic to verify the initial state before a transaction is started
    }

    @BeforeEach
    fun setUpTestDataWithinTransaction() {
        // set up test data within the transaction
    }

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

    @AfterEach
    fun tearDownWithinTransaction() {
        // run "tear down" logic within the transaction
    }

    @AfterTransaction
    fun 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 run that code. Failing to flush the underlying unit of work can produce false positives: Your test passes, but the same code throws an exception in a live, production environment. Note that this applies to any ORM framework that maintains an in-memory unit of work. 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:

Java

```
// ...

@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();
}

// ...
```

Kotlin

```
// ...

@Autowired
lateinit var sessionFactory: SessionFactory

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

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

// ...
```

The following example shows matching methods for JPA:

Java

```
// ...

@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();
}

// ...
```

Kotlin

```
// ...

@PersistenceContext
lateinit var entityManager:EntityManager

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

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

// ...
```

3.5.10. Executing SQL Scripts

When writing integration tests against a relational database, it is often beneficial to run 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 run 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 run, `ScriptUtils` may suit your needs better than some of the other alternatives described later. See the javadoc for individual methods in `ScriptUtils` for further details.

`ResourceDatabasePopulator` provides an object-based API for programmatically populating, initializing, or cleaning up a database by 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 running the scripts. Each of the configuration options has a reasonable default value. See the javadoc for details on default values. To run the scripts configured in a `ResourceDatabasePopulator`, you can invoke either the `populate(Connection)` method to run the populator against a `java.sql.Connection` or the `execute(DataSource)` method to run 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 run the scripts against a `DataSource`:

Java

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

Kotlin

```
@Test
fun databaseTest() {
    val populator = ResourceDatabasePopulator()
    populator.addScripts(
            ClassPathResource("test-schema.sql"),
            ClassPathResource("test-data.sql"))
    populator.setSeparator("@@")
    populator.execute(dataSource)
    // run code that uses the test schema and data
}
```

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

Executing SQL scripts declaratively with @Sql

In addition to the aforementioned mechanisms for running SQL scripts programmatically, you can declaratively configure SQL scripts in the Spring TestContext Framework. Specifically, you can declare the `@Sql` annotation on a test class or test method to configure individual SQL statements or the resource paths to SQL scripts that should be run against a given database before or after an integration test method. Support for `@Sql` is provided by the `SqlScriptsTestExecutionListener`, which is enabled by default.

Method-level `@Sql` declarations override class-level declarations by default. As of Spring Framework 5.2, however, this behavior may be configured per test class or per test method via `@SqlMergeMode`. See Merging and Overriding Configuration with `@SqlMergeMode` for further details.

Path Resource Semantics

Each path is interpreted as a Spring `Resource`. A plain path (for example, `"schema.sql"`) is 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 resource (for example, `"/org/example/schema.sql"`). A path that references a URL (for example, a path prefixed with `classpath:`, `file:`, `http:`) is loaded by using the specified resource protocol.

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

Java

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

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

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

Kotlin

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

    @Test
    fun emptySchemaTest() {
        // run code that uses the test schema without any test data
    }

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

Default Script Detection

If no SQL scripts or statements are specified, an attempt is made to detect a `default` script, depending on where `@Sql` is declared. If a default cannot be detected, an `IllegalStateException` is 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 you need to configure multiple sets of SQL scripts for a given test class or test method but with different syntax configuration, different error handling rules, or different execution phases per set, you can declare multiple instances of `@Sql`. With Java 8, you can use `@Sql` as a repeatable annotation. Otherwise, you can use the `@SqlGroup` annotation as an explicit container for declaring multiple instances of `@Sql`.

The following example shows how to use `@Sql` as a repeatable annotation with Java 8:

Java

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

Kotlin

```
// Repeatable annotations with non-SOURCE retention are not yet supported by Kotlin
```

In the scenario presented in the preceding example, the `test-schema.sql` script uses a different syntax for single-line comments.

The following example is identical to the preceding example, except that the `@Sql` declarations are grouped together within `@SqlGroup`. With Java 8 and above, the use of `@SqlGroup` is optional, but you may need to use `@SqlGroup` for compatibility with other JVM languages such as Kotlin.

Java

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

Kotlin

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

Script Execution Phases

By default, SQL scripts are run before the corresponding test method. However, if you need to run a particular set of scripts after the test method (for example, to clean up database state), you can use the `executionPhase` attribute in `@Sql`, as the following example shows:

Java

```
@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
)
void userTest() {
    // run code that needs the test data to be committed
    // to the database outside of the test's transaction
}
```

Kotlin

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

Note that `ISOLATED` and `AFTER_TEST_METHOD` are statically imported from `Sql.TransactionMode` and `Sql.ExecutionPhase`, respectively.

Script Configuration with `@SqlConfig`

You can configure script parsing and error handling by using 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 by using 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 javadoc 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), `{}` (for arrays), or `DEFAULT` (for enumerations). This approach lets local declarations of `@SqlConfig` 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. See the javadoc of individual attributes in `@Sql` and `@SqlConfig` for details.

Transaction management for `@Sql`

By default, the `SqlScriptsTestExecutionListener` infers the desired transaction semantics for scripts configured by using `@Sql`. Specifically, SQL scripts are run 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 can specify explicit names by setting the `dataSource` and `transactionManager` attributes of `@SqlConfig`. Furthermore, you can control the transaction propagation behavior by setting the `transactionMode` attribute of `@SqlConfig` (for example, whether scripts should be run 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 javadoc for `@SqlConfig` and `SqlScriptsTestExecutionListener` provide detailed information, and the following example shows a typical testing scenario that uses JUnit Jupiter and transactional tests with `@Sql`:

Java

```
@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);
        // run 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.");
    }
}
```

Kotlin

```
@SpringJUnitConfig(TestDatabaseConfig::class)
@Transactional
class TransactionalSqlScriptsTests @Autowired constructor(dataSource: DataSource) {

    val jdbcTemplate: JdbcTemplate = JdbcTemplate(dataSource)

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

    fun countRowsInTable(tableName: String): Int {
        return JdbcTestUtils.countRowsInTable(jdbcTemplate, tableName)
    }

    fun assertNumUsers(expected: Int) {
        assertEquals(expected, countRowsInTable("user"),
                "Number of rows in the [user] table.")
    }
}
```

Note that there is no need to clean up the database after the `usersTest()` method is run, since any changes made to the database (either within the test method or within the `/test-data.sql` script) are automatically rolled back by the `TransactionalTestExecutionListener` (see transaction management for details).

Merging and Overriding Configuration with `@SqlMergeMode`

As of Spring Framework 5.2, it is possible to merge method-level `@Sql` declarations with class-level declarations. For example, this allows you to provide the configuration for a database schema or some common test data once per test class and then provide additional, use case specific test data per test method. To enable `@Sql` merging, annotate either your test class or test method with `@SqlMergeMode(MERGE)`. To disable merging for a specific test method (or specific test subclass), you can switch back to the default mode via `@SqlMergeMode(OVERRIDE)`. Consult the `@SqlMergeMode` annotation documentation section for examples and further details.

3.5.11. Parallel Test Execution

Spring Framework 5.0 introduced 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 run in parallel without any changes to test code or configuration.

For details on how to set up parallel test execution, see 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 fail intermittently or seemingly randomly. The Spring Team therefore provides the following general guidelines for when not to run tests in parallel.

Do not run tests in parallel if the tests:

- Use Spring Framework's `@DirtiesContext` support.
- Use Spring Boot's `@MockBean` or `@SpyBean` support.
- Use JUnit 4's `@FixMethodOrder` support or any testing framework feature that is designed to ensure that test methods run in a particular order. Note, however, that this does not apply if entire test classes are run in parallel.
- Change the state of shared services or systems such as a database, message broker, filesystem, and others. This applies to both embedded 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 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 javadoc 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 need to verify that it is suitable for parallel test execution.

3.5.12. TestContext Framework Support Classes

This section describes the various classes that support the Spring TestContext Framework.

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 want to use the Spring TestContext Framework with an alternative runner (such as JUnit 4's `Parameterized` runner) or third-party runners (such as the `MockitoJUnitRunner`), you can, optionally, use Spring's support for JUnit rules instead.

The following code listing shows the minimal requirements for configuring a test class to run with the custom Spring `Runner`:

Java

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

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

Kotlin

```
@RunWith(SpringRunner::class)
@TestExecutionListeners
class SimpleTest {

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

In the preceding example, `@TestExecutionListeners` is configured with an empty list, to disable the default listeners, which otherwise would require an `ApplicationContext` to be configured through `@ContextConfiguration`.

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 of being independent of any `org.junit.runner.Runner` implementation and can, therefore, be combined with existing alternative runners (such as JUnit 4's `Parameterized`) or third-party runners (such as the `MockitoJUnitRunner`).

To support the full functionality of the TestContext framework, you must combine a

`SpringClassRule` with a `SpringMethodRule`. The following example shows the proper way to declare these rules in an integration test:

Java

```
// 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() {
        // test logic...
    }
}
```

Kotlin

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

    @Rule
    val springMethodRule = SpringMethodRule()

    @Test
    fun testMethod() {
        // test logic...
    }

    companion object {
        @ClassRule
        val springClassRule = SpringClassRule()
    }
}
```

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 you can use 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 you can use to run SQL statements to query the database. You can use such queries to confirm database state both before and after running 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 that delegate to methods in `JdbcTestUtils` by using the aforementioned `jdbcTemplate`. Furthermore, `AbstractTransactionalJUnit4SpringContextTests` provides an `executeSqlScript(..)` method for running 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)`, you 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 provides 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 `SpringExtension` for further details.
- Powerful support for conditional test execution based on SpEL expressions, environment variables, system properties, and so on. 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 shows how to configure a test class to use the `SpringExtension` in conjunction with `@ContextConfiguration`:

Java

```
// 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() {
        // test logic...
    }
}
```

Kotlin

```
// 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
    fun testMethod() {
        // test logic...
    }
}
```

Since you can also use annotations in JUnit 5 as meta-annotations, Spring provides the `@SpringJUnitConfig` and `@SpringJUnitWebConfig` composed annotations to simplify the configuration of the test `ApplicationContext` and JUnit Jupiter.

The following example uses `@SpringJUnitConfig` to reduce the amount of configuration used in the previous example:

Java

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

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

Kotlin

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

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

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

Java

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

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

Kotlin

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

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

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

Dependency Injection with `SpringExtension`

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

methods.

Specifically, `SpringExtension` can inject dependencies from the test's `ApplicationContext` into test constructors and methods that are annotated with `@BeforeAll`, `@AfterAll`, `@BeforeEach`, `@AfterEach`, `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and others.

Constructor Injection

If a specific 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 injects the value for that specific parameter with the corresponding bean or value from the test's `ApplicationContext`.

Spring can also be configured to autowire all arguments for a test class constructor if the constructor is considered to be *autowirable*. A constructor is considered to be autowirable if one of the following conditions is met (in order of precedence).

- The constructor is annotated with `@Autowired`.
- `@TestConstructor` is present or meta-present on the test class with the `autowireMode` attribute set to `ALL`.
- The default *test constructor autowire mode* has been changed to `ALL`.

See `@TestConstructor` for details on the use of `@TestConstructor` and how to change the global *test constructor autowire mode*.

If the constructor for a test class is considered to be *autowirable*, Spring assumes the responsibility for resolving arguments for all parameters in the constructor. Consequently, no other `ParameterResolver` registered with JUnit Jupiter can resolve parameters for such a constructor.

Constructor injection for test classes must not be used in conjunction with JUnit Jupiter's `@TestInstance(PER_CLASS)` support if `@DirtiesContext` is used to close the test's `ApplicationContext` before or after test methods.

The reason is that `@TestInstance(PER_CLASS)` instructs JUnit Jupiter to cache the test instance between test method invocations. Consequently, the test instance will retain references to beans that were originally injected from an `ApplicationContext` that has been subsequently closed. Since the constructor for the test class will only be invoked once in such scenarios, dependency injection will not occur again, and subsequent tests will interact with beans from the closed `ApplicationContext` which may result in errors.

To use `@DirtiesContext` with "before test method" or "after test method" modes in conjunction with `@TestInstance(PER_CLASS)`, one must configure dependencies from Spring to be supplied via field or setter injection so that they can be re-injected between test method invocations.

In the following example, Spring injects the `OrderService` bean from the `ApplicationContext` loaded

from `TestConfig.class` into the `OrderServiceIntegrationTests` constructor.

Java

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

    private final OrderService orderService;

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

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

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
class OrderServiceIntegrationTests @Autowired constructor(private val orderService:
OrderService){
    // tests that use the injected OrderService
}
```

Note that this feature lets test dependencies be `final` and therefore immutable.

If the `spring.test.constructor.autowire.mode` property is to `all` (see `@TestConstructor`), we can omit the declaration of `@Autowired` on the constructor in the previous example, resulting in the following.

Java

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

    private final OrderService orderService;

    OrderServiceIntegrationTests(OrderService orderService) {
        this.orderService = orderService;
    }

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

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
class OrderServiceIntegrationTests(val 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 injects the value for that specific parameter with the corresponding bean from the test's `ApplicationContext`.

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

Java

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

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

Kotlin

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

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

Due to the robustness of the `ParameterResolver` support in JUnit Jupiter, you can also 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 shows how to have both Spring and JUnit Jupiter inject dependencies into the `placeOrderRepeatedly()` test method simultaneously.

Java

```
@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
    }
}
```

Kotlin

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

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

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

Note that the use of `@RepeatedTest` from JUnit Jupiter lets the test method gain access to the `RepetitionInfo`.

`@Nested` test class configuration

The *Spring TestContext Framework* has supported the use of test-related annotations on `@Nested` test classes in JUnit Jupiter since Spring Framework 5.0; however, until Spring Framework 5.3 class-level test configuration annotations were not *inherited* from enclosing classes like they are from superclasses.

Spring Framework 5.3 introduces first-class support for inheriting test class configuration from enclosing classes, and such configuration will be inherited by default. To change from the default `INHERIT` mode to `OVERRIDE` mode, you may annotate an individual `@Nested` test class with `@NestedTestConfiguration(EnclosingConfiguration.OVERRIDE)`. An explicit `@NestedTestConfiguration` declaration will apply to the annotated test class as well as any of its subclasses and nested classes. Thus, you may annotate a top-level test class with `@NestedTestConfiguration`, and that will apply to all of its nested test classes recursively.

In order to allow development teams to change the default to `OVERRIDE` – for example, for compatibility with Spring Framework 5.0 through 5.2 – the default mode can be changed globally via a JVM system property or a `spring.properties` file in the root of the classpath. See the "Changing

the default enclosing configuration inheritance mode" note for details.

Although the following "Hello World" example is very simplistic, it shows how to declare common configuration on a top-level class that is inherited by its `@Nested` test classes. In this particular example, only the `TestConfig` configuration class is inherited. Each nested test class provides its own set of active profiles, resulting in a distinct `ApplicationContext` for each nested test class (see Context Caching for details). Consult the list of supported annotations to see which annotations can be inherited in `@Nested` test classes.

Java

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

    @Nested
    @ActiveProfiles("lang_en")
    class EnglishGreetings {

        @Test
        void hello(@Autowired GreetingService service) {
            assertThat(service.greetWorld()).isEqualTo("Hello World");
        }
    }

    @Nested
    @ActiveProfiles("lang_de")
    class GermanGreetings {

        @Test
        void hello(@Autowired GreetingService service) {
            assertThat(service.greetWorld()).isEqualTo("Hallo Welt");
        }
    }
}
```

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
class GreetingServiceTests {

    @Nested
    @ActiveProfiles("lang_en")
    inner class EnglishGreetings {

        @Test
        fun hello(@Autowired service:GreetingService) {
            assertThat(service.greetWorld()).isEqualTo("Hello World")
        }
    }

    @Nested
    @ActiveProfiles("lang_de")
    inner class GermanGreetings {

        @Test
        fun hello(@Autowired service:GreetingService) {
            assertThat(service.greetWorld()).isEqualTo("Hallo Welt")
        }
    }
}
```

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 you can use 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 you can use to run SQL statements to query the database. You can use such queries to confirm database state both before and after running 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 that delegate to methods in `JdbcTestUtils` by using the aforementioned `jdbcTemplate`. Furthermore, `AbstractTransactionalTestNGSpringContextTests` provides an `executeSqlScript(..)` method for running 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. WebTestClient

`WebTestClient` is an HTTP client designed for testing server applications. It wraps Spring's WebClient and uses it to perform requests but exposes a testing facade for verifying responses. `WebTestClient` can be used to perform end-to-end HTTP tests. It can also be used to test Spring MVC and Spring WebFlux applications without a running server via mock server request and response objects.

Kotlin users: See this section related to use of the `WebTestClient`.

3.6.1. Setup

To set up a `WebTestClient` you need to choose a server setup to bind to. This can be one of several mock server setup choices or a connection to a live server.

Bind to Controller

This setup allows you to test specific controller(s) via mock request and response objects, without a running server.

For WebFlux applications, use the following which loads infrastructure equivalent to the WebFlux Java config, registers the given controller(s), and creates a WebHandler chain to handle requests:

Java

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

Kotlin

```
val client = WebTestClient.bindToController(TestController()).build()
```

For Spring MVC, use the following which delegates to the StandaloneMockMvcBuilder to load infrastructure equivalent to the WebMvc Java config, registers the given controller(s), and creates an instance of MockMvc to handle requests:

Java

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

Kotlin

```
val client = MockMvcWebTestClient.bindToController(TestController()).build()
```

Bind to `ApplicationContext`

This setup allows you to load Spring configuration with Spring MVC or Spring WebFlux infrastructure and controller declarations and use it to handle requests via mock request and response objects, without a running server.

For WebFlux, use the following where the Spring `ApplicationContext` is passed to WebHttpHandlerBuilder to create the WebHandler chain to handle requests:

Java

```
@SpringJUnitConfig(WebConfig.class) ①
class MyTests {

    WebTestClient client;

    @BeforeEach
    void setUp(ApplicationContext context) {  ②
        client = WebTestClient.bindToApplicationContext(context).build(); ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

Kotlin

```
@SpringJUnitConfig(WebConfig::class) ①
class MyTests {

    lateinit var client: WebTestClient

    @BeforeEach
    fun setUp(context: ApplicationContext) { ②
        client = WebTestClient.bindToApplicationContext(context).build() ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

For Spring MVC, use the following where the Spring `ApplicationContext` is passed to MockMvcBuilders.webAppContextSetup to create a MockMvc instance to handle requests:

Java

```
@ExtendWith(SpringExtension.class)
@WebAppConfiguration("classpath:META-INF/web-resources") ①
@ContextHierarchy({
    @ContextConfiguration(classes = RootConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
class MyTests {

    @Autowired
    WebApplicationContext wac; ②

    WebTestClient client;

    @BeforeEach
    void setUp() {
        client = MockMvcWebTestClient.bindToApplicationContext(this.wac).build(); ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

Kotlin

```
@ExtendWith(SpringExtension.class)
@WebAppConfiguration("classpath:META-INF/web-resources") ①
@ContextHierarchy({
    @ContextConfiguration(classes = RootConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
class MyTests {

    @Autowired
    lateinit var wac: WebApplicationContext; ②

    lateinit var client: WebTestClient

    @BeforeEach
    fun setUp() { ②
        client = MockMvcWebTestClient.bindToApplicationContext(wac).build() ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

Bind to Router Function

This setup allows you to test functional endpoints via mock request and response objects, without a running server.

For WebFlux, use the following which delegates to `RouterFunctions.toWebHandler` to create a server setup to handle requests:

Java

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

Kotlin

```
val route: RouterFunction<*> = ...
val client = WebTestClient.bindToRouterFunction(route).build()
```

For Spring MVC there are currently no options to test WebMvc functional endpoints.

Bind to Server

This setup connects to a running server to perform full, end-to-end HTTP tests:

Java

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

Kotlin

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

Client Config

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

Java

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

Kotlin

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

3.6.2. Writing Tests

`WebTestClient` provides an API identical to WebClient up to the point of performing a request by using `exchange()`. See the WebClient documentation for examples on how to prepare a request with any content including form data, multipart data, and more.

After the call to `exchange()`, `WebTestClient` diverges from the `WebClient` and instead continues with a workflow to verify responses.

To assert the response status and headers, use the following:

Java

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

Kotlin

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

If you would like for all expectations to be asserted even if one of them fails, you can use `expectAll(..)` instead of multiple chained `expect*(..)` calls. This feature is similar to the *soft assertions* support in AssertJ and the `assertAll()` support in JUnit Jupiter.

Java

```
client.get().uri("/persons/1")
    .accept(MediaType.APPLICATION_JSON)
    .exchange()
    .expectAll(
        spec -> spec.expectStatus().isOk(),
        spec -> spec.expectHeader().contentType(MediaType.APPLICATION_JSON)
    );
```

You can then choose to decode the response body through one of the following:

- `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 an empty body.

And perform assertions on the resulting higher level Object(s):

Java

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

Kotlin

```
import org.springframework.test.web.reactive.server.expectBodyList

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

If the built-in assertions are insufficient, you can consume the object instead and perform any other assertions:

Java

```
import org.springframework.test.web.reactive.server.expectBody

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

Kotlin

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

Or you can exit the workflow and obtain an `EntityExchangeResult`:

Java

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

Kotlin

```
import org.springframework.test.web.reactive.server.expectBody

val result = client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk
        .expectBody<Person>()
        .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 is not expected to have content, you can assert that as follows:

Java

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

Kotlin

```
client.post().uri("/persons")
        .bodyValue(person)
        .exchange()
        .expectStatus().isCreated()
        .expectBody().isEmpty()
```

If you want to ignore the response content, the following releases the content without any assertions:

Java

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

Kotlin

```
client.get().uri("/persons/123")
        .exchange()
        .expectStatus().isNotFound
        .expectBody<Unit>()
```

JSON Content

You can use `expectBody()` without a target type to perform assertions on the raw content rather than through higher level Object(s).

To verify the full JSON content with JSONAssert:

Java

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

Kotlin

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

To verify JSON content with JSONPath:

Java

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

Kotlin

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

Streaming Responses

To test potentially infinite streams such as `"text/event-stream"` or `"application/x-ndjson"`, start by verifying the response status and headers, and then obtain a `FluxExchangeResult`:

Java

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

Kotlin

```
import org.springframework.test.web.reactive.server.returnResult

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

Now you're ready to consume the response stream with `StepVerifier` from `reactor-test`:

Java

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

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

Kotlin

```
val eventFlux = result.getResponseBody()

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

MockMvc Assertions

`WebTestClient` is an HTTP client and as such it can only verify what is in the client response including status, headers, and body.

When testing a Spring MVC application with a MockMvc server setup, you have the extra choice to perform further assertions on the server response. To do that start by obtaining an `ExchangeResult` after asserting the body:

Java

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

// For a response without a body
EntityExchangeResult<Void> result = client.get().uri("/path")
        .exchange()
        .expectBody().isEmpty();
```

Kotlin

```
// For a response with a body
val result = client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody(Person.class)
        .returnResult();

// For a response without a body
val result = client.get().uri("/path")
        .exchange()
        .expectBody().isEmpty();
```

Then switch to MockMvc server response assertions:

Java

```
MockMvcWebTestClient.resultActionsFor(result)
        .andExpect(model().attribute("integer", 3))
        .andExpect(model().attribute("string", "a string value"));
```

Kotlin

```
MockMvcWebTestClient.resultActionsFor(result)
        .andExpect(model().attribute("integer", 3))
        .andExpect(model().attribute("string", "a string value"));
```

3.7. MockMvc

The Spring MVC Test framework, also known as MockMvc, provides support for testing Spring MVC applications. It performs full Spring MVC request handling but via mock request and response objects instead of a running server.

MockMvc can be used on its own to perform requests and verify responses. It can also be used through the WebTestClient where MockMvc is plugged in as the server to handle requests with. The advantage of `WebTestClient` is the option to work with higher level objects instead of raw data as well as the ability to switch to full, end-to-end HTTP tests against a live server and use the same test API.

3.7.1. Overview

You can write plain unit tests for Spring MVC by instantiating a controller, injecting it with dependencies, and calling its methods. However such tests do not verify request mappings, data binding, message conversion, type conversion, validation, and nor do they involve any of the supporting `@InitBinder`, `@ModelAttribute`, or `@ExceptionHandler` methods.

The Spring MVC Test framework, also known as `MockMvc`, aims to provide more complete testing for Spring MVC controllers without a running server. It does that by invoking the `DispacherServlet` and passing "mock" implementations of the Servlet API from the `spring-test` module which replicates the full Spring MVC request handling without a running server.

MockMvc is a server side test framework that lets you verify most of the functionality of a Spring MVC application using lightweight and targeted tests. You can use it on its own to perform requests and to verify responses, or you can also use it through the WebTestClient API with MockMvc plugged in as the server to handle requests with.

Static Imports

When using MockMvc directly to perform requests, you'll need static imports for:

- `MockMvcBuilders.*`
- `MockMvcRequestBuilders.*`
- `MockMvcResultMatchers.*`
- `MockMvcResultHandlers.*`

An easy way to remember that is search for `MockMvc*`. If using Eclipse be sure to also add the above as "favorite static members" in the Eclipse preferences.

When using MockMvc through the WebTestClient you do not need static imports. The `WebTestClient` provides a fluent API without static imports.

Setup Choices

MockMvc can be setup in one of two ways. One is to point directly to the controllers you want to test and programmatically configure Spring MVC infrastructure. The second is to point to Spring configuration with Spring MVC and controller infrastructure in it.

To set up MockMvc for testing a specific controller, use the following:

Java

```
class MyWebTests {

    MockMvc mockMvc;

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

    // ...

}
```

Kotlin

```
class MyWebTests {

    lateinit var mockMvc : MockMvc

    @BeforeEach
    fun setup() {
        mockMvc = MockMvcBuilders.standaloneSetup(AccountController()).build()
    }

    // ...

}
```

Or you can also use this setup when testing through the WebTestClient which delegates to the same builder as shown above.

To set up MockMvc through Spring configuration, use the following:

Java

```
@SpringJUnitWebConfig(locations = "my-servlet-context.xml")
class MyWebTests {

    MockMvc mockMvc;

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

    // ...

}
```

Kotlin

```
@SpringJUnitWebConfig(locations = ["my-servlet-context.xml"])
class MyWebTests {

    lateinit var mockMvc: MockMvc

    @BeforeEach
    fun setup(wac: WebApplicationContext) {
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build()
    }

    // ...

}
```

Or you can also use this setup when testing through the WebTestClient which delegates to the same builder as shown above.

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 to remain focused on testing the web layer. The following example declares 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 to set up and verify your expectations, as the

following example shows:

Java

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

    @Autowired
    AccountService accountService;

    MockMvc mockMvc;

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

    // ...

}
```

Kotlin

```
@SpringJUnitWebConfig(locations = ["test-servlet-context.xml"])
class AccountTests {

    @Autowired
    lateinit var accountService: AccountService

    lateinit mockMvc: MockMvc

    @BeforeEach
    fun setup(wac: WebApplicationContext) {
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build()
    }

    // ...

}
```

The `standaloneSetup`, on the other hand, is a little closer to a unit test. It tests one controller at a time. You can manually inject the controller with mock dependencies, and it does not 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.

As with most "integration versus unit testing" debates, 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 can write all your 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:

Java

```
// 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();
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

In addition, third-party frameworks (and applications) can pre-package setup instructions, such as those in a `MockMvcConfigurer`. The Spring Framework has one such built-in implementation that helps to save and re-use the HTTP session across requests. You can use it as follows:

Java

```
// static import of SharedHttpSessionConfigurer.sharedHttpSession

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

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

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

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

Performing Requests

This section shows how to use MockMvc on its own to perform requests and verify responses. If using MockMvc through the `WebTestClient` please see the corresponding section on Writing Tests

instead.

To perform requests that use any HTTP method, as the following example shows:

Java

```
// static import of MockMvcRequestBuilders.*

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

Kotlin

```
import org.springframework.test.web.servlet.post

mockMvc.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. Rather, you have to set it up to be similar to the following example:

Java

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

Kotlin

```
import org.springframework.test.web.servlet.multipart

mockMvc.multipart("/doc") {
    file("a1", "ABC".toByteArray(charset("UTF8")))
}
```

You can specify query parameters in URI template style, as the following example shows:

Java

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

Kotlin

```
mockMvc.get("/hotels?thing={thing}", "somewhere")
```

You can also add Servlet request parameters that represent either query or form parameters, as the following example shows:

Java

```
mockMvc.perform(get("/hotels").param("thing", "somewhere"));
```

Kotlin

```
import org.springframework.test.web.servlet.get

mockMvc.get("/hotels") {
    param("thing", "somewhere")
}
```

If application code relies on Servlet request parameters and does not check the query string explicitly (as is most often the case), it does not matter which option you use. Keep in mind, however, that query parameters provided with the URI template are decoded while request parameters provided through the `param(…)` method are expected to already be decoded.

In most cases, it is preferable to leave the context path and the Servlet path out of 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 work, as the following example shows:

Java

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

Kotlin

```
import org.springframework.test.web.servlet.get

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

In the preceding example, it would be cumbersome to set the `contextPath` and `servletPath` with every performed request. Instead, you can set up default request properties, as the following example shows:

Java

```
class MyWebTests {

    MockMvc mockMvc;

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

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

The preceding properties 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 do not matter, since they must be specified on every request.

Defining Expectations

You can define expectations by appending one or more `andExpect(..)` calls after performing a request, as the following example shows. As soon as one expectation fails, no other expectations will be asserted.

Java

```
// static import of MockMvcRequestBuilders.* and MockMvcResultMatchers.*

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

Kotlin

```
import org.springframework.test.web.servlet.get

mockMvc.get("/accounts/1").andExpect {
    status().isOk()
}
```

You can define multiple expectations by appending `andExpectAll(..)` after performing a request, as the following example shows. In contrast to `andExpect(..)`, `andExpectAll(..)` guarantees that all supplied expectations will be asserted and that all failures will be tracked and reported.

Java

```
// static import of MockMvcRequestBuilders.* and MockMvcResultMatchers.*

mockMvc.perform(get("/accounts/1")).andExpectAll(
    status().isOk(),
    content().contentType("application/json;charset=UTF-8"));
```

`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 let you 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 let you inspect Servlet specific aspects, such as request and session attributes.

The following test asserts that binding or validation failed:

Java

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

Kotlin

```
import org.springframework.test.web.servlet.post

mockMvc.post("/persons").andExpect {
    status().isOk()
    model {
        attributeHasErrors("person")
    }
}
```

Many times, when writing tests, it is useful to dump the results of the performed request. You can do so as follows, where `print()` is a static import from `MockMvcResultHandlers`:

Java

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

Kotlin

```
import org.springframework.test.web.servlet.post

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

As long as request processing does not cause an unhandled exception, the `print()` method prints all the available result data to `System.out`. There is also 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)` prints the result data to `System.err`, while invoking `print(myWriter)` prints the result data to a custom writer. If you want to have the result data logged instead of printed, you can invoke the `log()` method, which logs 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, as the following example shows:

Java

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

Kotlin

```
var mvcResult = mockMvc.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, as the following example shows:

Java

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

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

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

When a JSON response content contains hypermedia links created with Spring HATEOAS, you can verify the resulting links by using JsonPath expressions, as the following example shows:

Java

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

Kotlin

```
mockMvc.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, you can verify the resulting links by using XPath expressions:

Java

```
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"));
```

Kotlin

```
val ns = mapOf("ns" to "http://www.w3.org/2005/Atom")
mockMvc.get("/handle") {
    accept(MediaType.APPLICATION_XML)
}.andExpect {
    xpath("/person/ns:link[@rel='self']/@href", ns) {
        string("http://localhost:8080/people")
    }
}
```

Async Requests

This section shows how to use MockMvc on its own to test asynchronous request handling. If using MockMvc through the WebTestClient, there is nothing special to do to make asynchronous requests work as the `WebTestClient` automatically does what is described in this section.

Servlet asynchronous requests, supported in Spring MVC, work by exiting the Servlet container thread and allowing the application to compute the response asynchronously, after which an async dispatch is made to complete processing on a Servlet container thread.

In Spring MVC Test, async requests can be tested by asserting the produced async value first, then manually performing the async dispatch, and finally verifying the response. Below is an example test for controller methods that return `DeferredResult`, `Callable`, or reactive type such as Reactor `Mono`:

Java

```
// static import of MockMvcRequestBuilders.* and MockMvcResultMatchers.*

@Test
void test() throws Exception {
    MvcResult mvcResult = this.mockMvc.perform(get("/path"))
            .andExpect(status().isOk()) ①
            .andExpect(request().asyncStarted()) ②
            .andExpect(request().asyncResult("body")) ③
            .andReturn();

    this.mockMvc.perform(asyncDispatch(mvcResult)) ④
            .andExpect(status().isOk()) ⑤
            .andExpect(content().string("body"));
}
```

① Check response status is still unchanged

② Async processing must have started

③ Wait and assert the async result

④ Manually perform an ASYNC dispatch (as there is no running container)

⑤ Verify the final response

Kotlin

```
@Test
fun test() {
    var mvcResult = mockMvc.get("/path").andExpect {
        status().isOk() ①
        request { asyncStarted() } ②
        // TODO Remove unused generic parameter
        request { asyncResult<Nothing>("body") } ③
    }.andReturn()


    mockMvc.perform(asyncDispatch(mvcResult)) ④
            .andExpect {
                status().isOk() ⑤
                content().string("body")
            }
}
```

① Check response status is still unchanged

② Async processing must have started

③ Wait and assert the async result

④ Manually perform an ASYNC dispatch (as there is no running container)

⑤ Verify the final response

Streaming Responses

The best way to test streaming responses such as Server-Sent Events is through the WebTestClient which can be used as a test client to connect to a `MockMvc` instance to perform tests on Spring MVC controllers without a running server. For example:

Java

```
WebTestClient client = MockMvcWebTestClient.bindToController(new
SseController()).build();

FluxExchangeResult<Person> exchangeResult = client.get()
        .uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectHeader().contentType("text/event-stream")
        .returnResult(Person.class);

// Use StepVerifier from Project Reactor to test the streaming response

StepVerifier.create(exchangeResult.getResponseBody())
        .expectNext(new Person("N0"), new Person("N1"), new Person("N2"))
        .expectNextCount(4)
        .consumeNextWith(person -> assertThat(person.getName()).endsWith("7"))
        .thenCancel()
        .verify();
```

`WebTestClient` can also connect to a live server and perform full end-to-end integration tests. This is also supported in Spring Boot where you can test a running server.

Filter Registrations

When setting up a `MockMvc` instance, you can register one or more Servlet `Filter` instances, as the following example shows:

Java

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

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

Registered filters are invoked through the `MockFilterChain` from `spring-test`, and the last filter delegates to the `DispatcherServlet`.

MockMvc vs End-to-End Tests

MockMVc is built on Servlet API mock implementations from the `spring-test` module and does not rely on a running container. Therefore, there are some differences when compared to full end-to-end integration tests with an actual client and a live server running.

The easiest way to think about this is by starting with a blank `MockHttpServletRequest`. Whatever you add to it is what the request becomes. 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 that, if you use JSPs, you can verify the JSP page to which the request was forwarded, but no HTML is rendered. In other words, the JSP is not invoked. Note, however, that all other rendering technologies that do not rely on forwarding, such as Thymeleaf and Freemarker, render HTML to the response body as expected. The same is true for rendering JSON, XML, and other formats through `@ResponseBody` methods.

Alternatively, you may consider the full end-to-end integration testing support from Spring Boot with `@SpringBootTest`. See the Spring Boot Reference Guide.

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 are testing the web layer only through the `DispatcherServlet` but with actual Spring configuration, as you might test the data access layer in isolation from the layers above it. Also, you can use the stand-alone 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 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, and other details. That means that it is easier to write expectations, since the server is not an opaque box, as it is when testing it through an actual HTTP client. This is generally an advantage of classic unit testing: It is easier to write, reason about, and debug but does not replace the need for full integration tests. At the same time, it is 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 Examples

The framework's own tests include many sample tests intended to show how to use MockMvc on its own or through the WebTestClient. Browse these examples for further ideas.

3.7.2. HtmlUnit Integration

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

- Easily test HTML pages by using tools such as HtmlUnit, WebDriver, and 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 (for example, Thymeleaf, FreeMarker, and others), 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`, as follows:

Java

```
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"));
```

Kotlin

```
@Test
fun test() {
    mockMvc.post("/messages/") {
        param("summary", "Spring Rocks")
        param("text", "In case you didn't know, Spring Rocks!")
    }.andExpect {
        status().is3xxRedirection()
        redirectedUrl("/messages/123")
    }
}
```

What if we want to test the form view that lets us 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 produce the correct request to create a new message? A naive attempt might resemble the following:

Java

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

Kotlin

```
mockMvc.get("/messages/form").andExpect {
    xpath("//input[@name='summary']") { exists() }
    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 continues to pass, even though the HTML form is out of synch with the controller. To resolve this we can combine our two tests, as follows:

Java

```
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"));
```

Kotlin

```
val summaryParamName = "summary";
val textParamName = "text";
mockMvc.get("/messages/form").andExpect {
    xpath("//input[@name='$summaryParamName']") { exists() }
    xpath("//textarea[@name='$textParamName']") { exists() }
}
mockMvc.post("/messages/") {
    param(summaryParamName, "Spring Rocks")
    param(textParamName, "In case you didn't know, Spring Rocks!")
}.andExpect {
    status().is3xxRedirection()
    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 as we take more factors into account: Are the fields the correct type? Are the fields enabled? And so on.
- 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, we still cannot account for some things. 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 can potentially use additional resources that impact the behavior of the page, such as JavaScript validation.

Integration Testing to the Rescue?

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

- Does our page display a notification to the user to indicate 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 need to ensure our database contains the proper messages. 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 such items as auto-generated ids, timestamps, and others 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 that run 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

You have a number of options when you want 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 want 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 need to use HtmlUnit 2.18 or higher.

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

Java

```
WebClient webClient;

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

Kotlin

```
lateinit var webClient: WebClient

@BeforeEach
fun setup(context: WebApplicationContext) {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build()
}
```

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

This ensures that any URL that references `localhost` as the server is directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL is requested by using a network connection, as normal. This lets us 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:

Java

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

Kotlin

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

The default context path is "". Alternatively, we can specify the context path, as described 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, as the following example shows:

Java

```
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();
```

Kotlin

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

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

Java

```
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!");
```

Kotlin

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

The preceding code 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 also test the behavior of JavaScript within our pages.

See 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 in the following example:

Java

```
WebClient webClient;

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

Kotlin

```
lateinit var webClient: WebClient

@BeforeEach
fun setup(context: WebApplicationContext) {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build()
}
```

We can also specify additional configuration options, as the following example shows:

Java

```
WebClient webClient;

@BeforeEach
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();
}
```

Kotlin

```
lateinit var webClient: WebClient

@BeforeEach
fun 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:

Java

```
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();
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

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, see 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 use 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 lets us easily organize our code. To better show how it works, we explore an example in this section.

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”, we might have something that resembles the following repeated in multiple places within our tests:

Java

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

Kotlin

```
val 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. This violates the DRY principle, so we should ideally extract this code into its own method, as follows:

Java

```
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);
}
```

Kotlin

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

fun setSummary(currentPage:HtmlPage , summary: String) {
    val summaryInput = currentPage.getHtmlElementById("summary")
    summaryInput.setValueAttribute(summary)
}
```

Doing so 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, as the following example shows:

Java

```
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());
    }
}
```

Kotlin

```
    class CreateMessagePage(private val currentPage: HtmlPage) {

        val summaryInput: HtmlTextInput = currentPage.getHtmlElementById("summary")

        val submit: HtmlSubmitInput = currentPage.getHtmlElementById("submit")

        fun <T> createMessage(summary: String, text: String): T {
            setSummary(summary)

            val result = submit.click()
            val error = at(result)

            return (if (error) CreateMessagePage(result) else ViewMessagePage(result))
as T
        }

        fun setSummary(summary: String) {
            summaryInput.setValueAttribute(summary)
        }

        fun at(page: HtmlPage): Boolean {
            return "Create Message" == page.getTitleText()
        }
    }
}
```

Formerly, this pattern was known as the Page Object Pattern. While we can certainly do this with HtmlUnit, WebDriver provides some tools that we 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 by using the `MockMvcHtmlUnitDriverBuilder` as the following example shows:

Java

```
WebDriver driver;

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

Kotlin

```
lateinit var driver: WebDriver

@BeforeEach
fun setup(context: WebApplicationContext) {
    driver = MockMvcHtmlUnitDriverBuilder
            .webAppContextSetup(context)
            .build()
}
```

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

The preceding example ensures that any URL that references `localhost` as the server is directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL is requested by using a network connection, as normal. This lets us 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:

Java

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

Kotlin

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

We can then fill out the form and submit it to create a message, as follows:

Java

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

Kotlin

```
val 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. Consider the following `CreateMessagePage` implementation:

Java

```
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 `CreateMessagePage` extends the `AbstractPage`. We do not 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, and other features, we can place this logic in a shared location.

② We have a member variable for each of the parts of the HTML page in which we are interested. These are of type `WebElement`. WebDriver's `PageFactory` lets us remove a lot of code from the HtmlUnit version of `CreateMessagePage` by automatically resolving each `WebElement`. The `PageFactory#initElements(WebDriver,Class<T>)` method automatically resolves 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 shows how to use the `@FindBy` annotation to look up our submit button with a `css` selector (**input[type=submit]**).

Kotlin

```
class CreateMessagePage(private val driver: WebDriver) : AbstractPage(driver) { ①

    ②
    private lateinit var summary: WebElement
    private lateinit var text: WebElement

    ③
    @FindBy(css = "input[type=submit]")
    private lateinit var submit: WebElement

    fun <T> createMessage(resultPage: Class<T>, summary: String, details: String): T {
        this.summary.sendKeys(summary)
        text.sendKeys(details)
        submit.click()
        return PageFactory.initElements(driver, resultPage)
    }
    companion object {
        fun to(driver: WebDriver): CreateMessagePage {
            driver.get("http://localhost:9990/mail/messages/form")
            return PageFactory.initElements(driver, CreateMessagePage::class.java)
        }
    }
}
```

① `CreateMessagePage` extends the `AbstractPage`. We do not 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, and other features, we can place this logic in a shared location.

② We have a member variable for each of the parts of the HTML page in which we are interested. These are of type `WebElement`. WebDriver's `PageFactory` lets us remove a lot of code from the HtmlUnit version of `CreateMessagePage` by automatically resolving each `WebElement`. The `PageFactory#initElements(WebDriver,Class<T>)` method automatically resolves 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 shows how to use the `@FindBy` annotation to look up our submit button with a `css` selector (**input[type=submit]**).

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

Java

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

Kotlin

```
assertThat(viewMessagePage.message).isEqualTo(expectedMessage)
assertThat(viewMessagePage.success).isEqualTo("Successfully created a new message")
```

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

Java

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

Kotlin

```
fun getMessage() = Message(getId(), getCreated(), getSummary(), getText())
```

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

Lastly, we must not forget to close the `WebDriver` instance when the test is complete, as follows:

Java

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

Kotlin

```
@AfterEach
fun destroy() {
    if (driver != null) {
        driver.close()
    }
}
```

For additional information on using WebDriver, see 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, as follows:

Java

```
WebDriver driver;

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

Kotlin

```
lateinit var driver: WebDriver

@BeforeEach
fun setup(context: WebApplicationContext) {
    driver = MockMvcHtmlUnitDriverBuilder
            .webAppContextSetup(context)
            .build()
}
```

We can also specify additional configuration options, as follows:

Java

```
WebDriver driver;

@BeforeEach
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();
}
```

Kotlin

```
lateinit var driver: WebDriver

@BeforeEach
fun 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:

Java

```
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();
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

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, see Setup Choices.

MockMvc and Geb

In the previous section, we saw how to use MockMvc with WebDriver. In this section, we 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, see Advanced `MockMvcHtmlUnitDriverBuilder`.

This ensures that any URL referencing `localhost` as the server is directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL is requested by using a network connection as normal. This lets us 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, as follows:

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

Any unrecognized method calls or property accesses or references that are not found are 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 using 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. Consider 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() }
    }
}
```

Our `CreateMessagePage` extends `Page`. We do not go over the details of `Page`, but, in summary, it contains common functionality for all of our pages. We define a URL in which this page can be found. This lets us 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 in which we are interested.

Finally, we can verify that a new message was created successfully, as follows:

```
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, see The Book of Geb user's manual.

3.8. Testing Client Applications

You can use client-side tests 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 (that is, without running a server). The following example shows how to do so:

Java

```
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();
```

Kotlin

```
val restTemplate = RestTemplate()

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

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

mockServer.verify()
```

In the preceding 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 can define additional expected requests and stub responses as needed. When we define expected requests and stub responses, 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. The following example uses `ignoreExpectOrder`:

Java

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

Kotlin

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

Even with unordered requests by default, each request is allowed to run once only. The `expect` method provides an overloaded variant that accepts an `ExpectedCount` argument that specifies a count range (for example, `once`, `manyTimes`, `max`, `min`, `between`, and so on). The following example uses

`times`:

Java

```
RestTemplate restTemplate = new RestTemplate();

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

// ...

mockServer.verify();
```

Kotlin

```
val restTemplate = RestTemplate()

val mockServer = MockRestServiceServer.bindTo(restTemplate).build()
mockServer.expect(times(2), requestTo("/something")).andRespond(withSuccess())
mockServer.expect(times(3), requestTo("/somewhere")).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 applies only to the first of any expected request. For example if "/something" is expected two times followed by "/somewhere" three times, then there should be a request to "/something" before there is a request to "/somewhere", but, aside from that subsequent "/something" and "/somewhere", 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 you can configure into a `RestTemplate` to bind it to a `MockMvc` instance. That allows processing requests using actual server-side logic but without running a server. The following example shows how to do so:

Java

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

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

Kotlin

```
val mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build()
restTemplate = RestTemplate(MockMvcClientHttpRequestFactory(mockMvc))

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

3.8.1. Static Imports

As with server-side tests, the fluent API for client-side tests requires a few static imports. Those are easy to find by searching for `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 (such IntelliJ) may not require any additional configuration. Check for the support for code completion on static members.

3.8.2. Further Examples of Client-side REST Tests

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

Chapter 4. Further Resources

See the following resources for more information about testing:

- JUnit: “A programmer-friendly testing framework for Java”. Used by the Spring Framework in its test suite and supported in the Spring TestContext Framework.
- TestNG: A testing framework inspired by JUnit with added support for test groups, data-driven testing, distributed testing, and other features. Supported in the Spring TestContext Framework
- AssertJ: “Fluent assertions for Java”, including support for Java 8 lambdas, streams, and other features.
- 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. Used by the Spring Framework in its test suite.
- 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.”
- JMock: Library that supports test-driven development of Java code with mock objects.
- DbUnit: JUnit extension (also usable with Ant and Maven) that is targeted at database-driven projects and, among other things, puts your database into a known state between test runs.
- Testcontainers: Java library that supports JUnit tests, providing lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container.
- The Grinder: Java load testing framework.
- SpringMockK: Support for Spring Boot integration tests written in Kotlin using MockK instead of Mockito.