Profile-specific Configuration

The `<springProfile>` tag lets you optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the `<configuration>` element. Use the `name` attribute to specify which profile accepts the configuration. The `<springProfile>` tag can contain a profile name (for example `staging`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example `production & (eu-central | eu-west)`. Check the Spring Framework reference guide for more details. The following listing shows three sample profiles:

```
<springProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
</springProfile>

<springProfile name="dev | staging">
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active
-->
</springProfile>

<springProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is not active -->
</springProfile>
```

Environment Properties

The `<springProperty>` tag lets you expose properties from the Spring `Environment` for use within Logback. Doing so can be useful if you want to access values from your `application.properties` file in your Logback configuration. The tag works in a similar way to Logback's standard `<property>` tag. However, rather than specifying a direct `value`, you specify the `source` of the property (from the `Environment`). If you need to store the property somewhere other than in `local` scope, you can use the `scope` attribute. If you need a fallback value (in case the property is not set in the `Environment`), you can use the `defaultValue` attribute. The following example shows how to expose properties for use within Logback:

```
<springProperty scope="context" name="fluentHost" source="myapp.fluentd.host"
        defaultValue="localhost"/>
<appender name="FLUENT" class="ch.qos.logback.more.appenders.DataFluentAppender">
    <remoteHost>${fluentHost}</remoteHost>
    ...
</appender>
```

NOTE	The `source` must be specified in kebab case (such as `my.property-name`). However, properties can be added to the `Environment` by using the relaxed rules.

7.4.10. Log4j2 Extensions

Spring Boot includes a number of extensions to Log4j2 that can help with advanced configuration.

You can use these extensions in any `log4j2-spring.xml` configuration file.

NOTE	Because the standard `log4j2.xml` configuration file is loaded too early, you cannot use extensions in it. You need to either use `log4j2-spring.xml` or define a `logging.config` property.

NOTE	The extensions supersede the Spring Boot support provided by Log4J. You should make sure not to include the `org.apache.logging.log4j:log4j-spring-boot` module in your build.

Profile-specific Configuration

The `<SpringProfile>` tag lets you optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the `<Configuration>` element. Use the `name` attribute to specify which profile accepts the configuration. The `<SpringProfile>` tag can contain a profile name (for example `staging`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example `production & (eu-central | eu-west)`. Check the Spring Framework reference guide for more details. The following listing shows three sample profiles:

```
<SpringProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
</SpringProfile>

<SpringProfile name="dev | staging">
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active
-->
</SpringProfile>

<SpringProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is not active -->
</SpringProfile>
```

Environment Properties Lookup

If you want to refer to properties from your Spring `Environment` within your Log4j2 configuration you can use `spring:` prefixed lookups. Doing so can be useful if you want to access values from your `application.properties` file in your Log4j2 configuration.

The following example shows how to set a Log4j2 property named `applicationName` that reads `spring.application.name` from the Spring `Environment`:

```
<Properties>
    <Property name="applicationName">${spring:spring.application.name}</Property>
</Properties>
```

NOTE	The lookup key should be specified in kebab case (such as `my.property-name`).

Log4j2 System Properties

Log4j2 supports a number of System Properties that can be used to configure various items. For example, the `log4j2.skipJansi` system property can be used to configure if the `ConsoleAppender` will try to use a Jansi output stream on Windows.

All system properties that are loaded after the Log4j2 initialization can be obtained from the Spring `Environment`. For example, you could add `log4j2.skipJansi=false` to your `application.properties` file to have the `ConsoleAppender` use Jansi on Windows.

NOTE	The Spring `Environment` is only considered when system properties and OS environment variables do not contain the value being loaded.
WARNING	System properties that are loaded during early Log4j2 initialization cannot reference the Spring `Environment`. For example, the property Log4j2 uses to allow the default Log4j2 implementation to be chosen is used before the Spring Environment is available.

7.5. Internationalization

Spring Boot supports localized messages so that your application can cater to users of different language preferences. By default, Spring Boot looks for the presence of a `messages` resource bundle at the root of the classpath.

NOTE	The auto-configuration applies when the default properties file for the configured resource bundle is available (`messages.properties` by default). If your resource bundle contains only language-specific properties files, you are required to add the default. If no properties file is found that matches any of the configured base names, there will be no auto-configured `MessageSource`.

The basename of the resource bundle as well as several other attributes can be configured using the `spring.messages` namespace, as shown in the following example:

Properties

```
spring.messages.basename=messages,config.i18n.messages
spring.messages.fallback-to-system-locale=false
```

Yaml

```
spring:
  messages:
    basename: "messages,config.i18n.messages"
    fallback-to-system-locale: false
```

TIP	`spring.messages.basename` supports comma-separated list of locations, either a package qualifier or a resource resolved from the classpath root.

See `MessageSourceProperties` for more supported options.

7.6. Aspect-Oriented Programming

Spring Boot provides auto-configuration for aspect-oriented programming (AOP). You can learn more about AOP with Spring in the Spring Framework reference documentation.

By default, Spring Boot's auto-configuration configures Spring AOP to use CGLib proxies. To use JDK proxies instead, set `configprop:spring.aop.proxy-target-class` to `false`.

If AspectJ is on the classpath, Spring Boot's auto-configuration will automatically enable AspectJ auto proxy such that `@EnableAspectJAutoProxy` is not required.

7.7. JSON

Spring Boot provides integration with three JSON mapping libraries:

- Gson
- Jackson
- JSON-B

Jackson is the preferred and default library.

7.7.1. Jackson

Auto-configuration for Jackson is provided and Jackson is part of `spring-boot-starter-json`. When Jackson is on the classpath an `ObjectMapper` bean is automatically configured. Several configuration properties are provided for customizing the configuration of the `ObjectMapper`.

Custom Serializers and Deserializers

If you use Jackson to serialize and deserialize JSON data, you might want to write your own `JsonSerializer` and `JsonDeserializer` classes. Custom serializers are usually registered with Jackson through a module, but Spring Boot provides an alternative `@JsonComponent` annotation that makes it easier to directly register Spring Beans.

You can use the `@JsonComponent` annotation directly on `JsonSerializer`, `JsonDeserializer` or `KeyDeserializer` implementations. You can also use it on classes that contain serializers/deserializers as inner classes, as shown in the following example:

Java

```
import java.io.IOException;

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.ObjectCodec;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonDeserializer;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.JsonSerializer;
import com.fasterxml.jackson.databind.SerializerProvider;

import org.springframework.boot.jackson.JsonComponent;

@JsonComponent
public class MyJsonComponent {

    public static class Serializer extends JsonSerializer<MyObject> {

        @Override
        public void serialize(MyObject value, JsonGenerator jgen, SerializerProvider
serializers) throws IOException {
            jgen.writeStartObject();
            jgen.writeStringField("name", value.getName());
            jgen.writeNumberField("age", value.getAge());
            jgen.writeEndObject();
        }

    }

    public static class Deserializer extends JsonDeserializer<MyObject> {

        @Override
        public MyObject deserialize(JsonParser jsonParser, DeserializationContext
ctxt) throws IOException {
            ObjectCodec codec = jsonParser.getCodec();
            JsonNode tree = codec.readTree(jsonParser);
            String name = tree.get("name").textValue();
            int age = tree.get("age").intValue();
            return new MyObject(name, age);
        }

    }

}
```

Kotlin

```
import com.fasterxml.jackson.core.JsonGenerator
import com.fasterxml.jackson.core.JsonParser
import com.fasterxml.jackson.core.JsonProcessingException
import com.fasterxml.jackson.databind.DeserializationContext
import com.fasterxml.jackson.databind.JsonDeserializer
import com.fasterxml.jackson.databind.JsonNode
import com.fasterxml.jackson.databind.JsonSerializer
import com.fasterxml.jackson.databind.SerializerProvider
import org.springframework.boot.jackson.JsonComponent
import java.io.IOException

@JsonComponent
class MyJsonComponent {

    class Serializer : JsonSerializer<MyObject>() {
        @Throws(IOException::class)
        override fun serialize(value: MyObject, jgen: JsonGenerator, serializers:
SerializerProvider) {
            jgen.writeStartObject()
            jgen.writeStringField("name", value.name)
            jgen.writeNumberField("age", value.age)
            jgen.writeEndObject()
        }
    }

    class Deserializer : JsonDeserializer<MyObject>() {
        @Throws(IOException::class, JsonProcessingException::class)
        override fun deserialize(jsonParser: JsonParser, ctxt:
DeserializationContext): MyObject {
            val codec = jsonParser.codec
            val tree = codec.readTree<JsonNode>(jsonParser)
            val name = tree["name"].textValue()
            val age = tree["age"].intValue()
            return MyObject(name, age)
        }
    }

}
```

All `@JsonComponent` beans in the `ApplicationContext` are automatically registered with Jackson. Because `@JsonComponent` is meta-annotated with `@Component`, the usual component-scanning rules apply.

Spring Boot also provides `JsonObjectSerializer` and `JsonObjectDeserializer` base classes that provide useful alternatives to the standard Jackson versions when serializing objects. See `JsonObjectSerializer` and `JsonObjectDeserializer` in the Javadoc for details.

The example above can be rewritten to use `JsonObjectSerializer`/`JsonObjectDeserializer` as follows:

Java

```
import java.io.IOException;

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.ObjectCodec;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.SerializerProvider;

import org.springframework.boot.jackson.JsonComponent;
import org.springframework.boot.jackson.JsonObjectDeserializer;
import org.springframework.boot.jackson.JsonObjectSerializer;

@JsonComponent
public class MyJsonComponent {

    public static class Serializer extends JsonObjectSerializer<MyObject> {

        @Override
        protected void serializeObject(MyObject value, JsonGenerator jgen,
SerializerProvider provider)
                throws IOException {
            jgen.writeStringField("name", value.getName());
            jgen.writeNumberField("age", value.getAge());
        }

    }

    public static class Deserializer extends JsonObjectDeserializer<MyObject> {

        @Override
        protected MyObject deserializeObject(JsonParser jsonParser,
DeserializationContext context, ObjectCodec codec,
                JsonNode tree) throws IOException {
            String name = nullSafeValue(tree.get("name"), String.class);
            int age = nullSafeValue(tree.get("age"), Integer.class);
            return new MyObject(name, age);
        }

    }

}
```

Kotlin

```
`object`

import com.fasterxml.jackson.core.JsonGenerator
import com.fasterxml.jackson.core.JsonParser
import com.fasterxml.jackson.core.ObjectCodec
import com.fasterxml.jackson.databind.DeserializationContext
import com.fasterxml.jackson.databind.JsonNode
import com.fasterxml.jackson.databind.SerializerProvider
import org.springframework.boot.jackson.JsonComponent
import org.springframework.boot.jackson.JsonObjectDeserializer
import org.springframework.boot.jackson.JsonObjectSerializer
import java.io.IOException

@JsonComponent
class MyJsonComponent {

    class Serializer : JsonObjectSerializer<MyObject>() {
        @Throws(IOException::class)
        override fun serializeObject(value: MyObject, jgen: JsonGenerator, provider:
SerializerProvider) {
            jgen.writeStringField("name", value.name)
            jgen.writeNumberField("age", value.age)
        }
    }

    class Deserializer : JsonObjectDeserializer<MyObject>() {
        @Throws(IOException::class)
        override fun deserializeObject(jsonParser: JsonParser, context:
DeserializationContext,
                codec: ObjectCodec, tree: JsonNode): MyObject {
            val name = nullSafeValue(tree["name"], String::class.java)
            val age = nullSafeValue(tree["age"], Int::class.java)
            return MyObject(name, age)
        }
    }

}
```

Mixins

Jackson has support for mixins that can be used to mix additional annotations into those already declared on a target class. Spring Boot's Jackson auto-configuration will scan your application's packages for classes annotated with `@JsonMixin` and register them with the auto-configured `ObjectMapper`. The registration is performed by Spring Boot's `JsonMixinModule`.

7.7.2. Gson

Auto-configuration for Gson is provided. When Gson is on the classpath a `Gson` bean is automatically

configured. Several `spring.gson.*` configuration properties are provided for customizing the configuration. To take more control, one or more `GsonBuilderCustomizer` beans can be used.

7.7.3. JSON-B

Auto-configuration for JSON-B is provided. When the JSON-B API and an implementation are on the classpath a `Jsonb` bean will be automatically configured. The preferred JSON-B implementation is Eclipse Yasson for which dependency management is provided.

7.8. Task Execution and Scheduling

In the absence of an `Executor` bean in the context, Spring Boot auto-configures an `AsyncTaskExecutor`. When virtual threads are enabled (using Java 21+ and `spring.threads.virtual.enabled` set to `true`) this will be a `SimpleAsyncTaskExecutor` that uses virtual threads. Otherwise, it will be a `ThreadPoolTaskExecutor` with sensible defaults. In either case, the auto-configured executor will be automatically used for:

- asynchronous task execution (`@EnableAsync`)
- Spring for GraphQL's asynchronous handling of `Callable` return values from controller methods
- Spring MVC's asynchronous request processing
- Spring WebFlux's blocking execution support

TIP

If you have defined a custom `Executor` in the context, both regular task execution (that is `@EnableAsync`) and Spring for GraphQL will use it. However, the Spring MVC and Spring WebFlux support will only use it if it is an `AsyncTaskExecutor` implementation (named `applicationTaskExecutor`). Depending on your target arrangement, you could change your `Executor` into an `AsyncTaskExecutor` or define both an `AsyncTaskExecutor` and an `AsyncConfigurer` wrapping your custom `Executor`.

The auto-configured `ThreadPoolTaskExecutorBuilder` allows you to easily create instances that reproduce what the auto-configuration does by default.

When a `ThreadPoolTaskExecutor` is auto-configured, the thread pool uses 8 core threads that can grow and shrink according to the load. Those default settings can be fine-tuned using the `spring.task.execution` namespace, as shown in the following example:

Properties

```
spring.task.execution.pool.max-size=16
spring.task.execution.pool.queue-capacity=100
spring.task.execution.pool.keep-alive=10s
```

Yaml

```
spring:
  task:
    execution:
      pool:
        max-size: 16
        queue-capacity: 100
        keep-alive: "10s"
```

This changes the thread pool to use a bounded queue so that when the queue is full (100 tasks), the thread pool increases to maximum 16 threads. Shrinking of the pool is more aggressive as threads are reclaimed when they are idle for 10 seconds (rather than 60 seconds by default).

A scheduler can also be auto-configured if it needs to be associated with scheduled task execution (using `@EnableScheduling` for instance).

If virtual threads are enabled (using Java 21+ and `spring.threads.virtual.enabled` set to `true`) this will be a `SimpleAsyncTaskScheduler` that uses virtual threads. This `SimpleAsyncTaskScheduler` will ignore any pooling related properties.

If virtual threads are not enabled, it will be a `ThreadPoolTaskScheduler` with sensible defaults. The `ThreadPoolTaskScheduler` uses one thread by default and its settings can be fine-tuned using the `spring.task.scheduling` namespace, as shown in the following example:

Properties

```
spring.task.scheduling.thread-name-prefix=scheduling-
spring.task.scheduling.pool.size=2
```

Yaml

```
spring:
  task:
    scheduling:
      thread-name-prefix: "scheduling-"
      pool:
        size: 2
```

A `ThreadPoolTaskExecutorBuilder` bean, a `SimpleAsyncTaskExecutorBuilder` bean, a `ThreadPoolTaskSchedulerBuilder` bean and a `SimpleAsyncTaskSchedulerBuilder` are made available in the context if a custom executor or scheduler needs to be created. The `SimpleAsyncTaskExecutorBuilder` and `SimpleAsyncTaskSchedulerBuilder` beans are auto-configured to use virtual threads if they are enabled (using Java 21+ and `spring.threads.virtual.enabled` set to `true`).

7.9. Testing

Spring Boot provides a number of utilities and annotations to help when testing your application. Test support is provided by two modules: `spring-boot-test` contains core items, and `spring-boot-test-autoconfigure` supports auto-configuration for tests.

Most developers use the `spring-boot-starter-test` "Starter", which imports both Spring Boot test modules as well as JUnit Jupiter, AssertJ, Hamcrest, and a number of other useful libraries.

TIP

If you have tests that use JUnit 4, JUnit 5's vintage engine can be used to run them. To use the vintage engine, add a dependency on `junit-vintage-engine`, as shown in the following example:

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

`hamcrest-core` is excluded in favor of `org.hamcrest:hamcrest` that is part of `spring-boot-starter-test`.

7.9.1. Test Scope Dependencies

The `spring-boot-starter-test` "Starter" (in the `test` `scope`) contains the following provided libraries:

- JUnit 5: The de-facto standard for unit testing Java applications.
- Spring Test & Spring Boot Test: Utilities and integration test support for Spring Boot applications.
- AssertJ: A fluent assertion library.
- Hamcrest: A library of matcher objects (also known as constraints or predicates).
- Mockito: A Java mocking framework.
- JSONassert: An assertion library for JSON.
- JsonPath: XPath for JSON.
- Awaitility: A library for testing asynchronous systems.

We generally find these common libraries to be useful when writing tests. If these libraries do not suit your needs, you can add additional test dependencies of your own.

7.9.2. Testing Spring Applications

One of the major advantages of dependency injection is that it should make your code easier to unit test. You can instantiate objects by using the `new` operator without even involving Spring. You can also use *mock objects* instead of real dependencies.

Often, you need to move beyond unit testing and start integration testing (with a Spring `ApplicationContext`). It is useful to be able to perform integration testing without requiring deployment of your application or needing to connect to other infrastructure.

The Spring Framework includes a dedicated test module for such integration testing. You can declare a dependency directly to `org.springframework:spring-test` or use the `spring-boot-starter-test` "Starter" to pull it in transitively.

If you have not used the `spring-test` module before, you should start by reading the relevant section of the Spring Framework reference documentation.

7.9.3. Testing Spring Boot Applications

A Spring Boot application is a Spring `ApplicationContext`, so nothing very special has to be done to test it beyond what you would normally do with a vanilla Spring context.

NOTE	External properties, logging, and other features of Spring Boot are installed in the context by default only if you use `SpringApplication` to create it.

Spring Boot provides a `@SpringBootTest` annotation, which can be used as an alternative to the standard `spring-test` `@ContextConfiguration` annotation when you need Spring Boot features. The annotation works by creating the `ApplicationContext` used in your tests through `SpringApplication`. In addition to `@SpringBootTest` a number of other annotations are also provided for testing more specific slices of an application.

TIP	If you are using JUnit 4, do not forget to also add `@RunWith(SpringRunner.class)` to your test, otherwise the annotations will be ignored. If you are using JUnit 5, there is no need to add the equivalent `@ExtendWith(SpringExtension.class)` as `@SpringBootTest` and the other `@…Test` annotations are already annotated with it.

By default, `@SpringBootTest` will not start a server. You can use the `webEnvironment` attribute of `@SpringBootTest` to further refine how your tests run:

- `MOCK`(Default) : Loads a web `ApplicationContext` and provides a mock web environment. Embedded servers are not started when using this annotation. If a web environment is not available on your classpath, this mode transparently falls back to creating a regular non-web `ApplicationContext`. It can be used in conjunction with `@AutoConfigureMockMvc` or `@AutoConfigureWebTestClient` for mock-based testing of your web application.
- `RANDOM_PORT`: Loads a `WebServerApplicationContext` and provides a real web environment. Embedded servers are started and listen on a random port.
- `DEFINED_PORT`: Loads a `WebServerApplicationContext` and provides a real web environment. Embedded servers are started and listen on a defined port (from your `application.properties`)

or on the default port of `8080`.

- `NONE`: Loads an `ApplicationContext` by using `SpringApplication` but does not provide *any* web environment (mock or otherwise).

NOTE	If your test is `@Transactional`, it rolls back the transaction at the end of each test method by default. However, as using this arrangement with either `RANDOM_PORT` or `DEFINED_PORT` implicitly provides a real servlet environment, the HTTP client and server run in separate threads and, thus, in separate transactions. Any transaction initiated on the server does not roll back in this case.

NOTE	`@SpringBootTest` with `webEnvironment = WebEnvironment.RANDOM_PORT` will also start the management server on a separate random port if your application uses a different port for the management server.

Detecting Web Application Type

If Spring MVC is available, a regular MVC-based application context is configured. If you have only Spring WebFlux, we will detect that and configure a WebFlux-based application context instead.

If both are present, Spring MVC takes precedence. If you want to test a reactive web application in this scenario, you must set the `spring.main.web-application-type` property:

Java

```
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(properties = "spring.main.web-application-type=reactive")
class MyWebFluxTests {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.context.SpringBootTest

@SpringBootTest(properties = ["spring.main.web-application-type=reactive"])
class MyWebFluxTests {

    // ...

}
```

Detecting Test Configuration

If you are familiar with the Spring Test Framework, you may be used to using `@ContextConfiguration(classes=…)` in order to specify which Spring `@Configuration` to load.

Alternatively, you might have often used nested `@Configuration` classes within your test.

When testing Spring Boot applications, this is often not required. Spring Boot's `@*Test` annotations search for your primary configuration automatically whenever you do not explicitly define one.

The search algorithm works up from the package that contains the test until it finds a class annotated with `@SpringBootApplication` or `@SpringBootConfiguration`. As long as you structured your code in a sensible way, your main configuration is usually found.

If you use a test annotation to test a more specific slice of your application, you should avoid adding configuration settings that are specific to a particular area on the main method's application class.

NOTE

The underlying component scan configuration of `@SpringBootApplication` defines exclude filters that are used to make sure slicing works as expected. If you are using an explicit `@ComponentScan` directive on your `@SpringBootApplication`-annotated class, be aware that those filters will be disabled. If you are using slicing, you should define them again.

If you want to customize the primary configuration, you can use a nested `@TestConfiguration` class. Unlike a nested `@Configuration` class, which would be used instead of your application's primary configuration, a nested `@TestConfiguration` class is used in addition to your application's primary configuration.

NOTE

Spring's test framework caches application contexts between tests. Therefore, as long as your tests share the same configuration (no matter how it is discovered), the potentially time-consuming process of loading the context happens only once.

Using the Test Configuration Main Method

Typically the test configuration discovered by `@SpringBootTest` will be your main `@SpringBootApplication`. In most well structured applications, this configuration class will also include the `main` method used to launch the application.

For example, the following is a very common code pattern for a typical Spring Boot application:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import
org.springframework.boot.docs.using.structuringyourcode.locatingthemainclass.MyApplica
tion
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

In the example above, the `main` method doesn't do anything other than delegate to `SpringApplication.run`. It is, however, possible to have a more complex `main` method that applies customizations before calling `SpringApplication.run`.

For example, here is an application that changes the banner mode and sets additional profiles:

Java

```
import org.springframework.boot.Banner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(MyApplication.class);
        application.setBannerMode(Banner.Mode.OFF);
        application.setAdditionalProfiles("myprofile");
        application.run(args);
    }

}
```

Kotlin

```
import org.springframework.boot.Banner
import org.springframework.boot.runApplication
import org.springframework.boot.autoconfigure.SpringBootApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args) {
        setBannerMode(Banner.Mode.OFF)
        setAdditionalProfiles("myprofile")
    }
}
```

Since customizations in the `main` method can affect the resulting `ApplicationContext`, it's possible that you might also want to use the `main` method to create the `ApplicationContext` used in your tests. By default, `@SpringBootTest` will not call your `main` method, and instead the class itself is used directly to create the `ApplicationContext`

If you want to change this behavior, you can change the `useMainMethod` attribute of `@SpringBootTest` to `UseMainMethod.ALWAYS` or `UseMainMethod.WHEN_AVAILABLE`. When set to `ALWAYS`, the test will fail if no `main` method can be found. When set to `WHEN_AVAILABLE` the `main` method will be used if it is available, otherwise the standard loading mechanism will be used.

For example, the following test will invoke the `main` method of `MyApplication` in order to create the `ApplicationContext`. If the main method sets additional profiles then those will be active when the `ApplicationContext` starts.

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.UseMainMethod;

@SpringBootTest(useMainMethod = UseMainMethod.ALWAYS)
class MyApplicationTests {

    @Test
    void exampleTest() {
        // ...
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.UseMainMethod

@SpringBootTest(useMainMethod = UseMainMethod.ALWAYS)
class MyApplicationTests {

    @Test
    fun exampleTest() {
        // ...
    }

}
```

Excluding Test Configuration

If your application uses component scanning (for example, if you use `@SpringBootApplication` or `@ComponentScan`), you may find top-level configuration classes that you created only for specific tests accidentally get picked up everywhere.

As we have seen earlier, `@TestConfiguration` can be used on an inner class of a test to customize the primary configuration. `@TestConfiguration` can also be used on a top-level class. Doing so indicates that the class should not be picked up by scanning. You can then import the class explicitly where it is required, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Import;

@SpringBootTest
@Import(MyTestsConfiguration.class)
class MyTests {

    @Test
    void exampleTest() {
        // ...
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.context.annotation.Import

@SpringBootTest
@Import(MyTestsConfiguration::class)
class MyTests {

    @Test
    fun exampleTest() {
        // ...
    }

}
```

NOTE	If you directly use `@ComponentScan` (that is, not through `@SpringBootApplication`) you need to register the `TypeExcludeFilter` with it. See the Javadoc for details.
NOTE	An imported `@TestConfiguration` is processed earlier than an inner-class `@TestConfiguration` and an imported `@TestConfiguration` will be processed before any configuration found through component scanning. Generally speaking, this difference in ordering has no noticeable effect but it is something to be aware of if you're relying on bean overriding.

Using Application Arguments

If your application expects arguments, you can have `@SpringBootTest` inject them using the `args` attribute.

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.test.context.SpringBootTest;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(args = "--app.test=one")
class MyApplicationArgumentTests {

    @Test
    void applicationArgumentsPopulated(@Autowired ApplicationArguments args) {
        assertThat(args.getOptionNames()).containsOnly("app.test");
        assertThat(args.getOptionValues("app.test")).containsOnly("one");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.ApplicationArguments
import org.springframework.boot.test.context.SpringBootTest

@SpringBootTest(args = ["--app.test=one"])
class MyApplicationArgumentTests {

    @Test
    fun applicationArgumentsPopulated(@Autowired args: ApplicationArguments) {
        assertThat(args.optionNames).containsOnly("app.test")
        assertThat(args.getOptionValues("app.test")).containsOnly("one")
    }

}
```

Testing With a Mock Environment

By default, `@SpringBootTest` does not start the server but instead sets up a mock environment for testing web endpoints.

With Spring MVC, we can query our web endpoints using `MockMvc` or `WebTestClient`, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.reactive.server.WebTestClient;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
class MyMockMvcTests {

    @Test
    void testWithMockMvc(@Autowired MockMvc mvc) throws Exception {

mvc.perform(get("/")).andExpect(status().isOk()).andExpect(content().string("Hello
World"));
    }

    // If Spring WebFlux is on the classpath, you can drive MVC tests with a
WebTestClient
    @Test
    void testWithWebTestClient(@Autowired WebTestClient webClient) {
        webClient
                .get().uri("/")
                .exchange()
                .expectStatus().isOk()
                .expectBody(String.class).isEqualTo("Hello World");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
import org.springframework.test.web.servlet.result.MockMvcResultMatchers

@SpringBootTest
@AutoConfigureMockMvc
class MyMockMvcTests {

    @Test
    fun testWithMockMvc(@Autowired mvc: MockMvc) {

mvc.perform(MockMvcRequestBuilders.get("/")).andExpect(MockMvcResultMatchers.status().
isOk)
            .andExpect(MockMvcResultMatchers.content().string("Hello World"))
    }

    // If Spring WebFlux is on the classpath, you can drive MVC tests with a
WebTestClient

    @Test
    fun testWithWebTestClient(@Autowired webClient: WebTestClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Hello World")
    }

}
```

TIP If you want to focus only on the web layer and not start a complete `ApplicationContext`, consider using `@WebMvcTest` instead.

With Spring WebFlux endpoints, you can use `WebTestClient` as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestClient;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.reactive.server.WebTestClient;

@SpringBootTest
@AutoConfigureWebTestClient
class MyMockWebTestClientTests {

    @Test
    void exampleTest(@Autowired WebTestClient webClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello World");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestClient
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody

@SpringBootTest
@AutoConfigureWebTestClient
class MyMockWebTestClientTests {

    @Test
    fun exampleTest(@Autowired webClient: WebTestClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Hello World")
    }

}
```

Testing within a mocked environment is usually faster than running with a full servlet container. However, since mocking occurs at the Spring MVC layer, code that relies on lower-level servlet container behavior cannot be directly tested with MockMvc.

TIP

For example, Spring Boot's error handling is based on the "error page" support provided by the servlet container. This means that, whilst you can test your MVC layer throws and handles exceptions as expected, you cannot directly test that a specific custom error page is rendered. If you need to test these lower-level concerns, you can start a fully running server as described in the next section.

Testing With a Running Server

If you need to start a full running server, we recommend that you use random ports. If you use `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`, an available port is picked at random each time your test runs.

The `@LocalServerPort` annotation can be used to inject the actual port used into your test. For convenience, tests that need to make REST calls to the started server can additionally `@Autowire` a `WebTestClient`, which resolves relative links to the running server and comes with a dedicated API for verifying responses, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.test.web.reactive.server.WebTestClient;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortWebTestClientTests {

    @Test
    void exampleTest(@Autowired WebTestClient webClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello World");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortWebTestClientTests {

    @Test
    fun exampleTest(@Autowired webClient: WebTestClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Hello World")
    }

}
```

TIP `WebTestClient` can also used with a mock environment, removing the need for a running server, by annotating your test class with `@AutoConfigureWebTestClient`.

This setup requires `spring-webflux` on the classpath. If you can not or will not add webflux, Spring Boot also provides a `TestRestTemplate` facility:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortTestRestTemplateTests {

    @Test
    void exampleTest(@Autowired TestRestTemplate restTemplate) {
        String body = restTemplate.getForObject("/", String.class);
        assertThat(body).isEqualTo("Hello World");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.boot.test.web.client.TestRestTemplate

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortTestRestTemplateTests {

    @Test
    fun exampleTest(@Autowired restTemplate: TestRestTemplate) {
        val body = restTemplate.getForObject("/", String::class.java)
        assertThat(body).isEqualTo("Hello World")
    }

}
```

Customizing WebTestClient

To customize the `WebTestClient` bean, configure a `WebTestClientBuilderCustomizer` bean. Any such beans are called with the `WebTestClient.Builder` that is used to create the `WebTestClient`.

Using JMX

As the test context framework caches context, JMX is disabled by default to prevent identical components to register on the same domain. If such test needs access to an `MBeanServer`, consider marking it dirty as well:

Java

```
import javax.management.MBeanServer;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.DirtiesContext;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(properties = "spring.jmx.enabled=true")
@DirtiesContext
class MyJmxTests {

    @Autowired
    private MBeanServer mBeanServer;

    @Test
    void exampleTest() {
        assertThat(this.mBeanServer.getDomains()).contains("java.lang");
        // ...
    }

}
```

Kotlin

```
import javax.management.MBeanServer

import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.annotation.DirtiesContext

@SpringBootTest(properties = ["spring.jmx.enabled=true"])
@DirtiesContext
class MyJmxTests(@Autowired val mBeanServer: MBeanServer) {

    @Test
    fun exampleTest() {
        assertThat(mBeanServer.domains).contains("java.lang")
        // ...
    }

}
```

Using Observations

If you annotate a sliced test with `@AutoConfigureObservability`, it auto-configures an `ObservationRegistry`.

Using Metrics

Regardless of your classpath, meter registries, except the in-memory backed, are not auto-configured when using `@SpringBootTest`.

If you need to export metrics to a different backend as part of an integration test, annotate it with `@AutoConfigureObservability`.

If you annotate a sliced test with `@AutoConfigureObservability`, it auto-configures an in-memory `MeterRegistry`. Data exporting in sliced tests is not supported with the `@AutoConfigureObservability` annotation.

Using Tracing

Regardless of your classpath, tracing components which are reporting data are not auto-configured when using `@SpringBootTest`.

If you need those components as part of an integration test, annotate the test with `@AutoConfigureObservability`.

If you have created your own reporting components (e.g. a custom `SpanExporter` or `SpanHandler`) and you don't want them to be active in tests, you can use the `@ConditionalOnEnabledTracing` annotation to disable them.

If you annotate a sliced test with `@AutoConfigureObservability`, it auto-configures a no-op `Tracer`. Data exporting in sliced tests is not supported with the `@AutoConfigureObservability` annotation.

Mocking and Spying Beans

When running tests, it is sometimes necessary to mock certain components within your application context. For example, you may have a facade over some remote service that is unavailable during development. Mocking can also be useful when you want to simulate failures that might be hard to trigger in a real environment.

Spring Boot includes a `@MockBean` annotation that can be used to define a Mockito mock for a bean inside your `ApplicationContext`. You can use the annotation to add new beans or replace a single existing bean definition. The annotation can be used directly on test classes, on fields within your test, or on `@Configuration` classes and fields. When used on a field, the instance of the created mock is also injected. Mock beans are automatically reset after each test method.

NOTE

If your test uses one of Spring Boot's test annotations (such as `@SpringBootTest`), this feature is automatically enabled. To use this feature with a different arrangement, listeners must be explicitly added, as shown in the following example:

Java

```
import
org.springframework.boot.test.mock.mockito.MockitoTestExecutionListener;
import
org.springframework.boot.test.mock.mockito.ResetMocksTestExecutionListen
er;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.TestExecutionListeners;

@ContextConfiguration(classes = MyConfig.class)
@TestExecutionListeners({ MockitoTestExecutionListener.class,
ResetMocksTestExecutionListener.class })
class MyTests {

    // ...

}
```

Kotlin

```
import
org.springframework.boot.test.mock.mockito.MockitoTestExecutionListener
import
org.springframework.boot.test.mock.mockito.ResetMocksTestExecutionListen
er
import org.springframework.test.context.ContextConfiguration
import org.springframework.test.context.TestExecutionListeners

@ContextConfiguration(classes = [MyConfig::class])
@TestExecutionListeners(
    MockitoTestExecutionListener::class,
    ResetMocksTestExecutionListener::class
)
class MyTests {

    // ...

}
```

The following example replaces an existing `RemoteService` bean with a mock implementation:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.given;

@SpringBootTest
class MyTests {

    @Autowired
    private Reverser reverser;

    @MockBean
    private RemoteService remoteService;

    @Test
    void exampleTest() {
        given(this.remoteService.getValue()).willReturn("spring");
        String reverse = this.reverser.getReverseValue(); // Calls injected
RemoteService
        assertThat(reverse).isEqualTo("gnirps");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.mock.mockito.MockBean

@SpringBootTest
class MyTests(@Autowired val reverser: Reverser, @MockBean val remoteService:
RemoteService) {

    @Test
    fun exampleTest() {
        given(remoteService.value).willReturn("spring")
        val reverse = reverser.reverseValue // Calls injected RemoteService
        assertThat(reverse).isEqualTo("gnirps")
    }

}
```

NOTE `@MockBean` cannot be used to mock the behavior of a bean that is exercised during application context refresh. By the time the test is executed, the application context refresh has completed and it is too late to configure the mocked behavior. We recommend using a `@Bean` method to create and configure the mock in this situation.

Additionally, you can use `@SpyBean` to wrap any existing bean with a Mockito `spy`. See the Javadoc for full details.

NOTE While Spring's test framework caches application contexts between tests and reuses a context for tests sharing the same configuration, the use of `@MockBean` or `@SpyBean` influences the cache key, which will most likely increase the number of contexts.

TIP If you are using `@SpyBean` to spy on a bean with `@Cacheable` methods that refer to parameters by name, your application must be compiled with `-parameters`. This ensures that the parameter names are available to the caching infrastructure once the bean has been spied upon.

TIP When you are using `@SpyBean` to spy on a bean that is proxied by Spring, you may need to remove Spring's proxy in some situations, for example when setting expectations using `given` or `when`. Use `AopTestUtils.getTargetObject(yourProxiedSpy)` to do so.

Auto-configured Tests

Spring Boot's auto-configuration system works well for applications but can sometimes be a little too much for tests. It often helps to load only the parts of the configuration that are required to test a "slice" of your application. For example, you might want to test that Spring MVC controllers are

mapping URLs correctly, and you do not want to involve database calls in those tests, or you might want to test JPA entities, and you are not interested in the web layer when those tests run.

The `spring-boot-test-autoconfigure` module includes a number of annotations that can be used to automatically configure such "slices". Each of them works in a similar way, providing a `@…Test` annotation that loads the `ApplicationContext` and one or more `@AutoConfigure…` annotations that can be used to customize auto-configuration settings.

NOTE	Each slice restricts component scan to appropriate components and loads a very restricted set of auto-configuration classes. If you need to exclude one of them, most `@…Test` annotations provide an `excludeAutoConfiguration` attribute. Alternatively, you can use `@ImportAutoConfiguration#exclude`.
NOTE	Including multiple "slices" by using several `@…Test` annotations in one test is not supported. If you need multiple "slices", pick one of the `@…Test` annotations and include the `@AutoConfigure…` annotations of the other "slices" by hand.
TIP	It is also possible to use the `@AutoConfigure…` annotations with the standard `@SpringBootTest` annotation. You can use this combination if you are not interested in "slicing" your application but you want some of the auto-configured test beans.

Auto-configured JSON Tests

To test that object JSON serialization and deserialization is working as expected, you can use the `@JsonTest` annotation. `@JsonTest` auto-configures the available supported JSON mapper, which can be one of the following libraries:

- Jackson `ObjectMapper`, any `@JsonComponent` beans and any Jackson `Module`s
- `Gson`
- `Jsonb`

TIP	A list of the auto-configurations that are enabled by `@JsonTest` can be found in the appendix.

If you need to configure elements of the auto-configuration, you can use the `@AutoConfigureJsonTesters` annotation.

Spring Boot includes AssertJ-based helpers that work with the JSONAssert and JsonPath libraries to check that JSON appears as expected. The `JacksonTester`, `GsonTester`, `JsonbTester`, and `BasicJsonTester` classes can be used for Jackson, Gson, Jsonb, and Strings respectively. Any helper fields on the test class can be `@Autowired` when using `@JsonTest`. The following example shows a test class for Jackson:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.json.JsonTest;
import org.springframework.boot.test.json.JacksonTester;

import static org.assertj.core.api.Assertions.assertThat;

@JsonTest
class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    void serialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda", "Civic");
        // Assert against a `.json` file in the same package as the test
        assertThat(this.json.write(details)).isEqualToJson("expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");

assertThat(this.json.write(details)).extractingJsonPathStringValue("@.make").isEqualTo
("Honda");
    }

    @Test
    void deserialize() throws Exception {
        String content = "{\"make\":\"Ford\",\"model\":\"Focus\"}";
        assertThat(this.json.parse(content)).isEqualTo(new VehicleDetails("Ford",
"Focus"));
        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.json.JsonTest
import org.springframework.boot.test.json.JacksonTester

@JsonTest
class MyJsonTests(@Autowired val json: JacksonTester<VehicleDetails>) {

    @Test
    fun serialize() {
        val details = VehicleDetails("Honda", "Civic")
        // Assert against a `.json` file in the same package as the test
        assertThat(json.write(details)).isEqualToJson("expected.json")
        // Or use JSON path based assertions
        assertThat(json.write(details)).hasJsonPathStringValue("@.make")

assertThat(json.write(details)).extractingJsonPathStringValue("@.make").isEqualTo("Hon
da")
    }

    @Test
    fun deserialize() {
        val content = "{\"make\":\"Ford\",\"model\":\"Focus\"}"
        assertThat(json.parse(content)).isEqualTo(VehicleDetails("Ford", "Focus"))
        assertThat(json.parseObject(content).make).isEqualTo("Ford")
    }

}
```

NOTE JSON helper classes can also be used directly in standard unit tests. To do so, call the `initFields` method of the helper in your `@Before` method if you do not use `@JsonTest`.

If you use Spring Boot's AssertJ-based helpers to assert on a number value at a given JSON path, you might not be able to use `isEqualTo` depending on the type. Instead, you can use AssertJ's `satisfies` to assert that the value matches the given condition. For instance, the following example asserts that the actual number is a float value close to `0.15` within an offset of `0.01`.

Java

```
@Test
void someTest() throws Exception {
    SomeObject value = new SomeObject(0.152f);

assertThat(this.json.write(value)).extractingJsonPathNumberValue("@.test.numberValue")
        .satisfies((number) -> assertThat(number.floatValue()).isCloseTo(0.15f,
within(0.01f)));
}
```

Kotlin

```
@Test
fun someTest() {
    val value = SomeObject(0.152f)
    assertThat(json.write(value)).extractingJsonPathNumberValue("@.test.numberValue")
        .satisfies(ThrowingConsumer { number ->
            assertThat(number.toFloat()).isCloseTo(0.15f, within(0.01f))
        })
}
```

Auto-configured Spring MVC Tests

To test whether Spring MVC controllers are working as expected, use the `@WebMvcTest` annotation. `@WebMvcTest` auto-configures the Spring MVC infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `Filter`, `HandlerInterceptor`, `WebMvcConfigurer`, `WebMvcRegistrations`, and `HandlerMethodArgumentResolver`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@WebMvcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP	A list of the auto-configuration settings that are enabled by `@WebMvcTest` can be found in the appendix.

TIP	If you need to register extra components, such as the Jackson `Module`, you can import additional configuration classes by using `@Import` on your test.

Often, `@WebMvcTest` is limited to a single controller and is used in combination with `@MockBean` to provide mock implementations for required collaborators.

`@WebMvcTest` also auto-configures `MockMvc`. Mock MVC offers a powerful way to quickly test MVC controllers without needing to start a full HTTP server.

TIP	You can also auto-configure `MockMvc` in a non-`@WebMvcTest` (such as `@SpringBootTest`) by annotating it with `@AutoConfigureMockMvc`. The following example uses `MockMvc`:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import static org.mockito.BDDMockito.given;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(UserVehicleController.class)
class MyControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));
        this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk())
            .andExpect(content().string("Honda Civic"));
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.boot.test.mock.mockito.MockBean
import org.springframework.http.MediaType
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
import org.springframework.test.web.servlet.result.MockMvcResultMatchers

@WebMvcTest(UserVehicleController::class)
class MyControllerTests(@Autowired val mvc: MockMvc) {

    @MockBean
    lateinit var userVehicleService: UserVehicleService

    @Test
    fun testExample() {
        given(userVehicleService.getVehicleDetails("sboot"))
            .willReturn(VehicleDetails("Honda", "Civic"))

mvc.perform(MockMvcRequestBuilders.get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
            .andExpect(MockMvcResultMatchers.status().isOk)
            .andExpect(MockMvcResultMatchers.content().string("Honda Civic"))
    }

}
```

TIP If you need to configure elements of the auto-configuration (for example, when servlet filters should be applied) you can use attributes in the `@AutoConfigureMockMvc` annotation.

If you use HtmlUnit and Selenium, auto-configuration also provides an HtmlUnit `WebClient` bean and/or a Selenium `WebDriver` bean. The following example uses HtmlUnit:

Java

```
import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.HtmlPage;
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.given;

@WebMvcTest(UserVehicleController.class)
class MyHtmlUnitTests {

    @Autowired
    private WebClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot")).willReturn(new
VehicleDetails("Honda", "Civic"));
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
        assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
    }

}
```

Kotlin

```
import com.gargoylesoftware.htmlunit.WebClient
import com.gargoylesoftware.htmlunit.html.HtmlPage
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.boot.test.mock.mockito.MockBean

@WebMvcTest(UserVehicleController::class)
class MyHtmlUnitTests(@Autowired val webClient: WebClient) {

    @MockBean
    lateinit var userVehicleService: UserVehicleService

    @Test
    fun testExample() {

given(userVehicleService.getVehicleDetails("sboot")).willReturn(VehicleDetails("Honda"
, "Civic"))
        val page = webClient.getPage<HtmlPage>("/sboot/vehicle.html")
        assertThat(page.body.textContent).isEqualTo("Honda Civic")
    }

}
```

NOTE	By default, Spring Boot puts `WebDriver` beans in a special "scope" to ensure that the driver exits after each test and that a new instance is injected. If you do not want this behavior, you can add `@Scope("singleton")` to your `WebDriver` `@Bean` definition.

WARNING	The `webDriver` scope created by Spring Boot will replace any user defined scope of the same name. If you define your own `webDriver` scope you may find it stops working when you use `@WebMvcTest`.

If you have Spring Security on the classpath, `@WebMvcTest` will also scan `WebSecurityConfigurer` beans. Instead of disabling security completely for such tests, you can use Spring Security's test support. More details on how to use Spring Security's `MockMvc` support can be found in this *Testing With Spring Security* how-to section.

TIP	Sometimes writing Spring MVC tests is not enough; Spring Boot can help you run full end-to-end tests with an actual server.

Auto-configured Spring WebFlux Tests

To test that Spring WebFlux controllers are working as expected, you can use the `@WebFluxTest` annotation. `@WebFluxTest` auto-configures the Spring WebFlux infrastructure and limits scanned

beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `WebFilter`, and `WebFluxConfigurer`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@WebFluxTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP A list of the auto-configurations that are enabled by `@WebFluxTest` can be found in the appendix.

TIP If you need to register extra components, such as Jackson `Module`, you can import additional configuration classes using `@Import` on your test.

Often, `@WebFluxTest` is limited to a single controller and used in combination with the `@MockBean` annotation to provide mock implementations for required collaborators.

`@WebFluxTest` also auto-configures `WebTestClient`, which offers a powerful way to quickly test WebFlux controllers without needing to start a full HTTP server.

TIP You can also auto-configure `WebTestClient` in a non-`@WebFluxTest` (such as `@SpringBootTest`) by annotating it with `@AutoConfigureWebTestClient`. The following example shows a class that uses both `@WebFluxTest` and a `WebTestClient`:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.reactive.server.WebTestClient;

import static org.mockito.BDDMockito.given;

@WebFluxTest(UserVehicleController.class)
class MyControllerTests {

    @Autowired
    private WebTestClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));

this.webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN).exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Honda Civic");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest
import org.springframework.boot.test.mock.mockito.MockBean
import org.springframework.http.MediaType
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody

@WebFluxTest(UserVehicleController::class)
class MyControllerTests(@Autowired val webClient: WebTestClient) {

    @MockBean
    lateinit var userVehicleService: UserVehicleService

    @Test
    fun testExample() {
        given(userVehicleService.getVehicleDetails("sboot"))
            .willReturn(VehicleDetails("Honda", "Civic"))
        webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN).exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Honda Civic")
    }

}
```

TIP This setup is only supported by WebFlux applications as using `WebTestClient` in a mocked web application only works with WebFlux at the moment.

NOTE `@WebFluxTest` cannot detect routes registered through the functional web framework. For testing `RouterFunction` beans in the context, consider importing your `RouterFunction` yourself by using `@Import` or by using `@SpringBootTest`.

NOTE `@WebFluxTest` cannot detect custom security configuration registered as a `@Bean` of type `SecurityWebFilterChain`. To include that in your test, you will need to import the configuration that registers the bean by using `@Import` or by using `@SpringBootTest`.

TIP Sometimes writing Spring WebFlux tests is not enough; Spring Boot can help you run full end-to-end tests with an actual server.

Auto-configured Spring GraphQL Tests

Spring GraphQL offers a dedicated testing support module; you'll need to add it to your project:

Maven

```
<dependencies>
  <dependency>
    <groupId>org.springframework.graphql</groupId>
    <artifactId>spring-graphql-test</artifactId>
    <scope>test</scope>
  </dependency>
  <!-- Unless already present in the compile scope -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Gradle

```
dependencies {
  testImplementation("org.springframework.graphql:spring-graphql-test")
  // Unless already present in the implementation configuration
  testImplementation("org.springframework.boot:spring-boot-starter-webflux")
}
```

This testing module ships the GraphQlTester. The tester is heavily used in test, so be sure to become familiar with using it. There are `GraphQlTester` variants and Spring Boot will auto-configure them depending on the type of tests:

- the `ExecutionGraphQlServiceTester` performs tests on the server side, without a client nor a transport
- the `HttpGraphQlTester` performs tests with a client that connects to a server, with or without a live server

Spring Boot helps you to test your Spring GraphQL Controllers with the `@GraphQlTest` annotation. `@GraphQlTest` auto-configures the Spring GraphQL infrastructure, without any transport nor server being involved. This limits scanned beans to `@Controller`, `RuntimeWiringConfigurer`, `JsonComponent`, `Converter`, `GenericConverter`, `DataFetcherExceptionResolver`, `Instrumentation` and `GraphQlSourceBuilderCustomizer`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@GraphQlTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

> **TIP** A list of the auto-configurations that are enabled by `@GraphQlTest` can be found in the appendix.

Often, `@GraphQlTest` is limited to a set of controllers and used in combination with the `@MockBean` annotation to provide mock implementations for required collaborators.

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.docs.web.graphql.runtimewiring.GreetingController;
import org.springframework.boot.test.autoconfigure.graphql.GraphQlTest;
import org.springframework.graphql.test.tester.GraphQlTester;

@GraphQlTest(GreetingController.class)
class GreetingControllerTests {

    @Autowired
    private GraphQlTester graphQlTester;

    @Test
    void shouldGreetWithSpecificName() {
        this.graphQlTester.document("{ greeting(name: \"Alice\") } ")
            .execute()
            .path("greeting")
            .entity(String.class)
            .isEqualTo("Hello, Alice!");
    }

    @Test
    void shouldGreetWithDefaultName() {
        this.graphQlTester.document("{ greeting } ")
            .execute()
            .path("greeting")
            .entity(String.class)
            .isEqualTo("Hello, Spring!");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.docs.web.graphql.runtimewiring.GreetingController
import org.springframework.boot.test.autoconfigure.graphql.GraphQlTest
import org.springframework.graphql.test.tester.GraphQlTester

@GraphQlTest(GreetingController::class)
internal class GreetingControllerTests {

    @Autowired
    lateinit var graphQlTester: GraphQlTester

    @Test
    fun shouldGreetWithSpecificName() {
        graphQlTester.document("{ greeting(name: \"Alice\") }
").execute().path("greeting").entity(String::class.java)
                .isEqualTo("Hello, Alice!")
    }

    @Test
    fun shouldGreetWithDefaultName() {
        graphQlTester.document("{ greeting }
").execute().path("greeting").entity(String::class.java)
                .isEqualTo("Hello, Spring!")
    }

}
```

`@SpringBootTest` tests are full integration tests and involve the entire application. When using a random or defined port, a live server is configured and an `HttpGraphQlTester` bean is contributed automatically so you can use it to test your server. When a MOCK environment is configured, you can also request an `HttpGraphQlTester` bean by annotating your test class with `@AutoConfigureHttpGraphQlTester`:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.graphql.tester.AutoConfigureHttpGraphQlTes
ter;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.graphql.test.tester.HttpGraphQlTester;

@AutoConfigureHttpGraphQlTester
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
class GraphQlIntegrationTests {

    @Test
    void shouldGreetWithSpecificName(@Autowired HttpGraphQlTester graphQlTester) {
        HttpGraphQlTester authenticatedTester = graphQlTester.mutate()
            .webTestClient((client) -> client.defaultHeaders((headers) ->
headers.setBasicAuth("admin", "ilovespring")))
            .build();
        authenticatedTester.document("{ greeting(name: \"Alice\") } ")
            .execute()
            .path("greeting")
            .entity(String.class)
            .isEqualTo("Hello, Alice!");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.graphql.tester.AutoConfigureHttpGraphQlTes
ter
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.graphql.test.tester.HttpGraphQlTester
import org.springframework.http.HttpHeaders
import org.springframework.test.web.reactive.server.WebTestClient

@AutoConfigureHttpGraphQlTester
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
class GraphQlIntegrationTests {

    @Test
    fun shouldGreetWithSpecificName(@Autowired graphQlTester: HttpGraphQlTester) {
        val authenticatedTester = graphQlTester.mutate()
            .webTestClient { client: WebTestClient.Builder ->
                client.defaultHeaders { headers: HttpHeaders ->
                    headers.setBasicAuth("admin", "ilovespring")
                }
            }.build()
        authenticatedTester.document("{ greeting(name: \"Alice\") } ").execute()
            .path("greeting").entity(String::class.java).isEqualTo("Hello, Alice!")
    }
}
```

Auto-configured Data Cassandra Tests

You can use `@DataCassandraTest` to test Cassandra applications. By default, it configures a `CassandraTemplate`, scans for `@Table` classes, and configures Spring Data Cassandra repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataCassandraTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Cassandra with Spring Boot, see "Cassandra".)

> **TIP** A list of the auto-configuration settings that are enabled by `@DataCassandraTest` can be found in the appendix.

The following example shows a typical setup for using Cassandra tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.cassandra.DataCassandraTest;

@DataCassandraTest
class MyDataCassandraTests {

    @Autowired
    private SomeRepository repository;

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.cassandra.DataCassandraTest

@DataCassandraTest
class MyDataCassandraTests(@Autowired val repository: SomeRepository)
```

Auto-configured Data Couchbase Tests

You can use `@DataCouchbaseTest` to test Couchbase applications. By default, it configures a `CouchbaseTemplate` or `ReactiveCouchbaseTemplate`, scans for `@Document` classes, and configures Spring Data Couchbase repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataCouchbaseTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Couchbase with Spring Boot, see "Couchbase", earlier in this chapter.)

TIP	A list of the auto-configuration settings that are enabled by `@DataCouchbaseTest` can be found in the appendix.

The following example shows a typical setup for using Couchbase tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.couchbase.DataCouchbaseTest;

@DataCouchbaseTest
class MyDataCouchbaseTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.couchbase.DataCouchbaseTest

@DataCouchbaseTest
class MyDataCouchbaseTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

Auto-configured Data Elasticsearch Tests

You can use `@DataElasticsearchTest` to test Elasticsearch applications. By default, it configures an `ElasticsearchRestTemplate`, scans for `@Document` classes, and configures Spring Data Elasticsearch repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataElasticsearchTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Elasticsearch with Spring Boot, see "Elasticsearch", earlier in this chapter.)

TIP	A list of the auto-configuration settings that are enabled by `@DataElasticsearchTest` can be found in the appendix.

The following example shows a typical setup for using Elasticsearch tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.data.elasticsearch.DataElasticsearchTest;

@DataElasticsearchTest
class MyDataElasticsearchTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.data.elasticsearch.DataElasticsearchTest

@DataElasticsearchTest
class MyDataElasticsearchTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

Auto-configured Data JPA Tests

You can use the `@DataJpaTest` annotation to test JPA applications. By default, it scans for `@Entity` classes and configures Spring Data JPA repositories. If an embedded database is available on the classpath, it configures one as well. SQL queries are logged by default by setting the `spring.jpa.show-sql` property to `true`. This can be disabled using the `showSql` attribute of the annotation.

Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataJpaTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP A list of the auto-configuration settings that are enabled by `@DataJpaTest` can be found in the appendix.

By default, data JPA tests are transactional and roll back at the end of each test. See the relevant section in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class as follows: