*Java*

```
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyNonTransactionalTests {

    // ...

}
```

*Kotlin*

```
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest
import org.springframework.transaction.annotation.Propagation
import org.springframework.transaction.annotation.Transactional

@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyNonTransactionalTests {

    // ...

}
```

Data JPA tests may also inject a `TestEntityManager` bean, which provides an alternative to the standard JPA `EntityManager` that is specifically designed for tests.

**TIP**

`TestEntityManager` can also be auto-configured to any of your Spring-based test class by adding `@AutoConfigureTestEntityManager`. When doing so, make sure that your test is running in a transaction, for instance by adding `@Transactional` on your test class or method.

A `JdbcTemplate` is also available if you need that. The following example shows the `@DataJpaTest` annotation in use:

*Java*

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;

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

@DataJpaTest
class MyRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    void testExample() {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getEmployeeNumber()).isEqualTo("1234");
    }

}
```

*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.orm.jpa.DataJpaTest
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager

@DataJpaTest
class MyRepositoryTests(@Autowired val entityManager: TestEntityManager, @Autowired
val repository: UserRepository) {

    @Test
    fun testExample() {
        entityManager.persist(User("sboot", "1234"))
        val user = repository.findByUsername("sboot")
        assertThat(user?.username).isEqualTo("sboot")
        assertThat(user?.employeeNumber).isEqualTo("1234")
    }

}
```

In-memory embedded databases generally work well for tests, since they are fast and do not require any installation. If, however, you prefer to run tests against a real database you can use the `@AutoConfigureTestDatabase` annotation, as shown in the following example:

*Java*

```
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import
org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase.Replace;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
class MyRepositoryTests {

    // ...

}
```

*Kotlin*

```
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest

@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class MyRepositoryTests {

    // ...

}
```

#### Auto-configured JDBC Tests

`@JdbcTest` is similar to `@DataJpaTest` but is for tests that only require a `DataSource` and do not use Spring Data JDBC. By default, it configures an in-memory embedded database and a `JdbcTemplate`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@JdbcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

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

By default, JDBC 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:

*Java*

```
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyTransactionalTests {

}
```

*Kotlin*

```
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest
import org.springframework.transaction.annotation.Propagation
import org.springframework.transaction.annotation.Transactional

@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyTransactionalTests
```

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `@DataJpaTest`. (See "Auto-configured Data JPA Tests".)

#### Auto-configured Data JDBC Tests

`@DataJdbcTest` is similar to `@JdbcTest` but is for tests that use Spring Data JDBC repositories. By default, it configures an in-memory embedded database, a `JdbcTemplate`, and Spring Data JDBC repositories. Only `AbstractJdbcConfiguration` subclasses are scanned when the `@DataJdbcTest` annotation is used, regular `@Component` and `@ConfigurationProperties` beans are not scanned. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

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

By default, Data JDBC 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 test class as shown in the JDBC example.

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `@DataJpaTest`. (See "Auto-configured Data JPA Tests".)

#### Auto-configured Data R2DBC Tests

`@DataR2dbcTest` is similar to `@DataJdbcTest` but is for tests that use Spring Data R2DBC repositories. By default, it configures an in-memory embedded database, an `R2dbcEntityTemplate`, and Spring Data R2DBC repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned

when the `@DataR2dbcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

| TIP | A list of the auto-configurations that are enabled by `@DataR2dbcTest` can be found in the appendix. |
|---|---|

By default, Data R2DBC tests are not transactional.

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `@DataJpaTest`. (See "Auto-configured Data JPA Tests".)

#### Auto-configured jOOQ Tests

You can use `@JooqTest` in a similar fashion as `@JdbcTest` but for jOOQ-related tests. As jOOQ relies heavily on a Java-based schema that corresponds with the database schema, the existing `DataSource` is used. If you want to replace it with an in-memory database, you can use `@AutoConfigureTestDatabase` to override those settings. (For more about using jOOQ with Spring Boot, see "Using jOOQ".) Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@JooqTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

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

`@JooqTest` configures a `DSLContext`. The following example shows the `@JooqTest` annotation in use:

*Java*

```
import org.jooq.DSLContext;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;

@JooqTest
class MyJooqTests {

    @Autowired
    private DSLContext dslContext;

    // ...

}
```

*Kotlin*

```
import org.jooq.DSLContext
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.jooq.JooqTest

@JooqTest
class MyJooqTests(@Autowired val dslContext: DSLContext) {

    // ...

}
```

JOOQ tests are transactional and roll back at the end of each test by default. If that is not what you want, you can disable transaction management for a test or for the whole test class as shown in the JDBC example.

#### Auto-configured Data MongoDB Tests

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

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

The following class shows the `@DataMongoTest` annotation in use:

*Java*

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.data.mongodb.core.MongoTemplate;

@DataMongoTest
class MyDataMongoDbTests {

    @Autowired
    private MongoTemplate mongoTemplate;

    // ...

}
```

*Kotlin*

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest
import org.springframework.data.mongodb.core.MongoTemplate

@DataMongoTest
class MyDataMongoDbTests(@Autowired val mongoTemplate: MongoTemplate) {

    // ...

}
```

#### Auto-configured Data Neo4j Tests

You can use `@DataNeo4jTest` to test Neo4j applications. By default, it scans for `@Node` classes, and configures Spring Data Neo4j repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataNeo4jTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Neo4J with Spring Boot, see "Neo4j".)

> **TIP** A list of the auto-configuration settings that are enabled by `@DataNeo4jTest` can be found in the appendix.

The following example shows a typical setup for using Neo4J tests in Spring Boot:

*Java*

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;

@DataNeo4jTest
class MyDataNeo4jTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

*Kotlin*

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest

@DataNeo4jTest
class MyDataNeo4jTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

By default, Data Neo4j 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:

*Java*

```
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyDataNeo4jTests {

}
```

*Kotlin*

```
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest
import org.springframework.transaction.annotation.Propagation
import org.springframework.transaction.annotation.Transactional

@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyDataNeo4jTests
```

| **NOTE** | Transactional tests are not supported with reactive access. If you are using this style, you must configure `@DataNeo4jTest` tests as described above. |
|---|---|

#### Auto-configured Data Redis Tests

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

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

The following example shows the `@DataRedisTest` annotation in use:

*Java*

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest;

@DataRedisTest
class MyDataRedisTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

*Kotlin*

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest

@DataRedisTest
class MyDataRedisTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

#### Auto-configured Data LDAP Tests

You can use `@DataLdapTest` to test LDAP applications. By default, it configures an in-memory embedded LDAP (if available), configures an `LdapTemplate`, scans for `@Entry` classes, and configures Spring Data LDAP repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataLdapTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using LDAP with Spring Boot, see "LDAP".)

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

The following example shows the `@DataLdapTest` annotation in use:

*Java*

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.ldap.core.LdapTemplate;

@DataLdapTest
class MyDataLdapTests {

    @Autowired
    private LdapTemplate ldapTemplate;

    // ...

}
```

*Kotlin*

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest
import org.springframework.ldap.core.LdapTemplate

@DataLdapTest
class MyDataLdapTests(@Autowired val ldapTemplate: LdapTemplate) {

    // ...

}
```

In-memory embedded LDAP generally works well for tests, since it is fast and does not require any developer installation. If, however, you prefer to run tests against a real LDAP server, you should exclude the embedded LDAP auto-configuration, as shown in the following example:

*Java*

```
import
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;

@DataLdapTest(excludeAutoConfiguration = EmbeddedLdapAutoConfiguration.class)
class MyDataLdapTests {

    // ...

}
```

*Kotlin*

```
import
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest

@DataLdapTest(excludeAutoConfiguration = [EmbeddedLdapAutoConfiguration::class])
class MyDataLdapTests {

    // ...

}
```

#### Auto-configured REST Clients

You can use the `@RestClientTest` annotation to test REST clients. By default, it auto-configures Jackson, GSON, and Jsonb support, configures a `RestTemplateBuilder` and a `RestClient.Builder`, and adds support for `MockRestServiceServer`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@RestClientTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

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

The specific beans that you want to test should be specified by using the `value` or `components` attribute of `@RestClientTest`.

When using a `RestTemplateBuilder` in the beans under test and `RestTemplateBuilder.rootUri(String rootUri)` has been called when building the `RestTemplate`, then the root URI should be omitted from the `MockRestServiceServer` expectations 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.client.RestClientTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.client.MockRestServiceServer;

import static org.assertj.core.api.Assertions.assertThat;
import static
org.springframework.test.web.client.match.MockRestRequestMatchers.requestTo;
import static
org.springframework.test.web.client.response.MockRestResponseCreators.withSuccess;

@RestClientTest(RemoteVehicleDetailsService.class)
class MyRestTemplateServiceTests {

    @Autowired
    private RemoteVehicleDetailsService service;

    @Autowired
    private MockRestServiceServer server;

    @Test
    void getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() {

this.server.expect(requestTo("/greet/details")).andRespond(withSuccess("hello",
MediaType.TEXT_PLAIN));
        String greeting = this.service.callRestService();
        assertThat(greeting).isEqualTo("hello");
    }

}
```

*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.web.client.RestClientTest
import org.springframework.http.MediaType
import org.springframework.test.web.client.MockRestServiceServer
import org.springframework.test.web.client.match.MockRestRequestMatchers
import org.springframework.test.web.client.response.MockRestResponseCreators

@RestClientTest(RemoteVehicleDetailsService::class)
class MyRestTemplateServiceTests(
    @Autowired val service: RemoteVehicleDetailsService,
    @Autowired val server: MockRestServiceServer) {

    @Test
    fun getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() {
        server.expect(MockRestRequestMatchers.requestTo("/greet/details"))
            .andRespond(MockRestResponseCreators.withSuccess("hello",
MediaType.TEXT_PLAIN))
        val greeting = service.callRestService()
        assertThat(greeting).isEqualTo("hello")
    }

}
```

When using a `RestClient.Builder` in the beans under test, or when using a `RestTemplateBuilder` without calling `rootUri(String rootURI)`, the full URI must be used in the `MockRestServiceServer` expectations 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.client.RestClientTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.client.MockRestServiceServer;

import static org.assertj.core.api.Assertions.assertThat;
import static
org.springframework.test.web.client.match.MockRestRequestMatchers.requestTo;
import static
org.springframework.test.web.client.response.MockRestResponseCreators.withSuccess;

@RestClientTest(RemoteVehicleDetailsService.class)
class MyRestClientServiceTests {

    @Autowired
    private RemoteVehicleDetailsService service;

    @Autowired
    private MockRestServiceServer server;

    @Test
    void getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() {
        this.server.expect(requestTo("https://example.com/greet/details"))
            .andRespond(withSuccess("hello", MediaType.TEXT_PLAIN));
        String greeting = this.service.callRestService();
        assertThat(greeting).isEqualTo("hello");
    }

}
```

*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.web.client.RestClientTest
import org.springframework.http.MediaType
import org.springframework.test.web.client.MockRestServiceServer
import org.springframework.test.web.client.match.MockRestRequestMatchers
import org.springframework.test.web.client.response.MockRestResponseCreators

@RestClientTest(RemoteVehicleDetailsService::class)
class MyRestClientServiceTests(
    @Autowired val service: RemoteVehicleDetailsService,
    @Autowired val server: MockRestServiceServer) {

    @Test
    fun getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() {

server.expect(MockRestRequestMatchers.requestTo("https://example.com/greet/details"))
            .andRespond(MockRestResponseCreators.withSuccess("hello",
MediaType.TEXT_PLAIN))
        val greeting = service.callRestService()
        assertThat(greeting).isEqualTo("hello")
    }

}
```

### Auto-configured Spring REST Docs Tests

You can use the `@AutoConfigureRestDocs` annotation to use Spring REST Docs in your tests with Mock MVC, REST Assured, or WebTestClient. It removes the need for the JUnit extension in Spring REST Docs.

`@AutoConfigureRestDocs` can be used to override the default output directory (`target/generated-snippets` if you are using Maven or `build/generated-snippets` if you are using Gradle). It can also be used to configure the host, scheme, and port that appears in any documented URIs.

#### Auto-configured Spring REST Docs Tests With Mock MVC

`@AutoConfigureRestDocs` customizes the `MockMvc` bean to use Spring REST Docs when testing servlet-based web applications. You can inject it by using `@Autowired` and use it in your tests as you normally would when using Mock MVC and Spring REST Docs, 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.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(UserController.class)
@AutoConfigureRestDocs
class MyUserDocumentationTests {

    @Autowired
    private MockMvc mvc;

    @Test
    void listUsers() throws Exception {
        this.mvc.perform(get("/users").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk())
            .andDo(document("list-users"));
    }

}
```

*Kotlin*

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.http.MediaType
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentation
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
import org.springframework.test.web.servlet.result.MockMvcResultMatchers

@WebMvcTest(UserController::class)
@AutoConfigureRestDocs
class MyUserDocumentationTests(@Autowired val mvc: MockMvc) {

    @Test
    fun listUsers() {
        mvc.perform(MockMvcRequestBuilders.get("/users").accept(MediaType.TEXT_PLAIN))
            .andExpect(MockMvcResultMatchers.status().isOk)
            .andDo(MockMvcRestDocumentation.document("list-users"))
    }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, you can use a `RestDocsMockMvcConfigurationCustomizer` bean, as shown in the following example:

*Java*

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsMockMvcConfigurationCusto
mizer;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentationConfigurer;
import org.springframework.restdocs.templates.TemplateFormats;

@TestConfiguration(proxyBeanMethods = false)
public class MyRestDocsConfiguration implements RestDocsMockMvcConfigurationCustomizer
{

    @Override
    public void customize(MockMvcRestDocumentationConfigurer configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}
```

*Kotlin*

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsMockMvcConfigurationCusto
mizer
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentationConfigurer
import org.springframework.restdocs.templates.TemplateFormats

@TestConfiguration(proxyBeanMethods = false)
class MyRestDocsConfiguration : RestDocsMockMvcConfigurationCustomizer {

    override fun customize(configurer: MockMvcRestDocumentationConfigurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown())
    }

}
```

If you want to make use of Spring REST Docs support for a parameterized output directory, you can create a `RestDocumentationResultHandler` bean. The auto-configuration calls `alwaysDo` with this result handler, thereby causing each `MockMvc` call to automatically generate the default snippets. The following example shows a `RestDocumentationResultHandler` being defined:

*Java*

```
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentation;
import org.springframework.restdocs.mockmvc.RestDocumentationResultHandler;

@TestConfiguration(proxyBeanMethods = false)
public class MyResultHandlerConfiguration {

    @Bean
    public RestDocumentationResultHandler restDocumentation() {
        return MockMvcRestDocumentation.document("{method-name}");
    }

}
```

*Kotlin*

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.context.annotation.Bean
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentation
import org.springframework.restdocs.mockmvc.RestDocumentationResultHandler

@TestConfiguration(proxyBeanMethods = false)
class MyResultHandlerConfiguration {

    @Bean
    fun restDocumentation(): RestDocumentationResultHandler {
        return MockMvcRestDocumentation.document("{method-name}")
    }

}
```

**Auto-configured Spring REST Docs Tests With WebTestClient**

`@AutoConfigureRestDocs` can also be used with `WebTestClient` when testing reactive web applications. You can inject it by using `@Autowired` and use it in your tests as you normally would when using `@WebFluxTest` and Spring REST Docs, 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.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.test.web.reactive.server.WebTestClient;

import static
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation.document;

@WebFluxTest
@AutoConfigureRestDocs
class MyUsersDocumentationTests {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    void listUsers() {
        this.webTestClient
            .get().uri("/")
        .exchange()
        .expectStatus()
            .isOk()
        .expectBody()
            .consumeWith(document("list-users"));
    }

}
```

*Kotlin*

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest
import org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation
import org.springframework.test.web.reactive.server.WebTestClient

@WebFluxTest
@AutoConfigureRestDocs
class MyUsersDocumentationTests(@Autowired val webTestClient: WebTestClient) {

    @Test
    fun listUsers() {
        webTestClient
            .get().uri("/")
            .exchange()
            .expectStatus()
            .isOk
            .expectBody()
            .consumeWith(WebTestClientRestDocumentation.document("list-users"))
    }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, you can use a `RestDocsWebTestClientConfigurationCustomizer` bean, as shown in the following example:

*Java*

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsWebTestClientConfiguratio
nCustomizer;
import org.springframework.boot.test.context.TestConfiguration;
import
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentationConfigurer;

@TestConfiguration(proxyBeanMethods = false)
public class MyRestDocsConfiguration implements
RestDocsWebTestClientConfigurationCustomizer {

    @Override
    public void customize(WebTestClientRestDocumentationConfigurer configurer) {
        configurer.snippets().withEncoding("UTF-8");
    }

}
```

*Kotlin*

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsWebTestClientConfiguratio
nCustomizer
import org.springframework.boot.test.context.TestConfiguration
import
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentationConfigurer

@TestConfiguration(proxyBeanMethods = false)
class MyRestDocsConfiguration : RestDocsWebTestClientConfigurationCustomizer {

    override fun customize(configurer: WebTestClientRestDocumentationConfigurer) {
        configurer.snippets().withEncoding("UTF-8")
    }

}
```

If you want to make use of Spring REST Docs support for a parameterized output directory, you can use a `WebTestClientBuilderCustomizer` to configure a consumer for every entity exchange result. The following example shows such a `WebTestClientBuilderCustomizer` being defined:

*Java*

```
import org.springframework.boot.test.context.TestConfiguration;
import
org.springframework.boot.test.web.reactive.server.WebTestClientBuilderCustomizer;
import org.springframework.context.annotation.Bean;

import static
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation.document;

@TestConfiguration(proxyBeanMethods = false)
public class MyWebTestClientBuilderCustomizerConfiguration {

    @Bean
    public WebTestClientBuilderCustomizer restDocumentation() {
        return (builder) -> builder.entityExchangeResultConsumer(document("{method-
name}"));
    }

}
```

*Kotlin*

```
import org.springframework.boot.test.context.TestConfiguration
import
org.springframework.boot.test.web.reactive.server.WebTestClientBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation
import org.springframework.test.web.reactive.server.WebTestClient

@TestConfiguration(proxyBeanMethods = false)
class MyWebTestClientBuilderCustomizerConfiguration {

    @Bean
    fun restDocumentation(): WebTestClientBuilderCustomizer {
        return WebTestClientBuilderCustomizer { builder: WebTestClient.Builder ->
            builder.entityExchangeResultConsumer(
                WebTestClientRestDocumentation.document("{method-name}")
            )
        }
    }

}
```

#### Auto-configured Spring REST Docs Tests With REST Assured

`@AutoConfigureRestDocs` makes a `RequestSpecification` bean, preconfigured to use Spring REST Docs, available to your tests. You can inject it by using `@Autowired` and use it in your tests as you normally would when using REST Assured and Spring REST Docs, as shown in the following example:

*Java*

```
import io.restassured.specification.RequestSpecification;
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.server.LocalServerPort;

import static io.restassured.RestAssured.given;
import static org.hamcrest.Matchers.is;
import static
org.springframework.restdocs.restassured.RestAssuredRestDocumentation.document;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
class MyUserDocumentationTests {

    @Test
    void listUsers(@Autowired RequestSpecification documentationSpec, @LocalServerPort
int port) {
        given(documentationSpec)
            .filter(document("list-users"))
        .when()
            .port(port)
            .get("/")
        .then().assertThat()
            .statusCode(is(200));
    }

}
```

*Kotlin*

```
import io.restassured.RestAssured
import io.restassured.specification.RequestSpecification
import org.hamcrest.Matchers
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.boot.test.web.server.LocalServerPort
import org.springframework.restdocs.restassured.RestAssuredRestDocumentation

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
class MyUserDocumentationTests {

    @Test
    fun listUsers(@Autowired documentationSpec: RequestSpecification?,
@LocalServerPort port: Int) {
        RestAssured.given(documentationSpec)
            .filter(RestAssuredRestDocumentation.document("list-users"))
            .`when`()
            .port(port)["/"]
            .then().assertThat()
            .statusCode(Matchers.`is`(200))
    }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, a `RestDocsRestAssuredConfigurationCustomizer` bean can be used, as shown in the following example:

*Java*

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsRestAssuredConfigurationC
ustomizer;
import org.springframework.boot.test.context.TestConfiguration;
import
org.springframework.restdocs.restassured.RestAssuredRestDocumentationConfigurer;
import org.springframework.restdocs.templates.TemplateFormats;

@TestConfiguration(proxyBeanMethods = false)
public class MyRestDocsConfiguration implements
RestDocsRestAssuredConfigurationCustomizer {

    @Override
    public void customize(RestAssuredRestDocumentationConfigurer configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}
```

*Kotlin*

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsRestAssuredConfigurationC
ustomizer
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.restdocs.restassured.RestAssuredRestDocumentationConfigurer
import org.springframework.restdocs.templates.TemplateFormats

@TestConfiguration(proxyBeanMethods = false)
class MyRestDocsConfiguration : RestDocsRestAssuredConfigurationCustomizer {

    override fun customize(configurer: RestAssuredRestDocumentationConfigurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown())
    }

}
```

### Auto-configured Spring Web Services Tests

#### Auto-configured Spring Web Services Client Tests

You can use `@WebServiceClientTest` to test applications that call web services using the Spring Web Services project. By default, it configures a mock `WebServiceServer` bean and automatically customizes your `WebServiceTemplateBuilder`. (For more about using Web Services with Spring Boot, see "Web Services".)

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

The following example shows the `@WebServiceClientTest` annotation in use:

*Java*

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

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.webservices.client.WebServiceClientTest;
import org.springframework.ws.test.client.MockWebServiceServer;
import org.springframework.xml.transform.StringSource;

import static org.assertj.core.api.Assertions.assertThat;
import static org.springframework.ws.test.client.RequestMatchers.payload;
import static org.springframework.ws.test.client.ResponseCreators.withPayload;

@WebServiceClientTest(SomeWebService.class)
class MyWebServiceClientTests {

    @Autowired
    private MockWebServiceServer server;

    @Autowired
    private SomeWebService someWebService;

    @Test
    void mockServerCall() {
        this.server
            .expect(payload(new StringSource("<request/>")))
            .andRespond(withPayload(new
StringSource("<response><status>200</status></response>")));
        assertThat(this.someWebService.test())
            .extracting(Response::getStatus)
            .isEqualTo(200);
    }

}
```

*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.webservices.client.WebServiceClientTest
import org.springframework.ws.test.client.MockWebServiceServer
import org.springframework.ws.test.client.RequestMatchers
import org.springframework.ws.test.client.ResponseCreators
import org.springframework.xml.transform.StringSource

@WebServiceClientTest(SomeWebService::class)
class MyWebServiceClientTests(@Autowired val server: MockWebServiceServer, @Autowired
val someWebService: SomeWebService) {

    @Test
    fun mockServerCall() {
        server
            .expect(RequestMatchers.payload(StringSource("<request/>")))

.andRespond(ResponseCreators.withPayload(StringSource("<response><status>200</status><
/response>")))

assertThat(this.someWebService.test()).extracting(Response::status).isEqualTo(200)
    }

}
```

#### Auto-configured Spring Web Services Server Tests

You can use `@WebServiceServerTest` to test applications that implement web services using the Spring Web Services project. By default, it configures a `MockWebServiceClient` bean that can be used to call your web service endpoints. (For more about using Web Services with Spring Boot, see "Web Services".)

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

The following example shows the `@WebServiceServerTest` annotation in use:

*Java*

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

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.webservices.server.WebServiceServerTest;
import org.springframework.ws.test.server.MockWebServiceClient;
import org.springframework.ws.test.server.RequestCreators;
import org.springframework.ws.test.server.ResponseMatchers;
import org.springframework.xml.transform.StringSource;

@WebServiceServerTest(ExampleEndpoint.class)
class MyWebServiceServerTests {

    @Autowired
    private MockWebServiceClient client;

    @Test
    void mockServerCall() {
        this.client
            .sendRequest(RequestCreators.withPayload(new
StringSource("<ExampleRequest/>")))
            .andExpect(ResponseMatchers.payload(new
StringSource("<ExampleResponse>42</ExampleResponse>")));
    }

}
```

*Kotlin*

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.webservices.server.WebServiceServerTest
import org.springframework.ws.test.server.MockWebServiceClient
import org.springframework.ws.test.server.RequestCreators
import org.springframework.ws.test.server.ResponseMatchers
import org.springframework.xml.transform.StringSource

@WebServiceServerTest(ExampleEndpoint::class)
class MyWebServiceServerTests(@Autowired val client: MockWebServiceClient) {

    @Test
    fun mockServerCall() {
        client

.sendRequest(RequestCreators.withPayload(StringSource("<ExampleRequest/>")))

.andExpect(ResponseMatchers.payload(StringSource("<ExampleResponse>42</ExampleResponse
>")))
    }

}
```

### Additional Auto-configuration and Slicing

Each slice provides one or more `@AutoConfigure…` annotations that namely defines the auto-configurations that should be included as part of a slice. Additional auto-configurations can be added on a test-by-test basis by creating a custom `@AutoConfigure…` annotation or by adding `@ImportAutoConfiguration` to the test as shown in the following example:

*Java*

```
import org.springframework.boot.autoconfigure.ImportAutoConfiguration;
import
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration;
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;

@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration.class)
class MyJdbcTests {

}
```

*Kotlin*

```
import org.springframework.boot.autoconfigure.ImportAutoConfiguration
import org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest

@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration::class)
class MyJdbcTests
```

| | |
|---|---|
| **NOTE** | Make sure to not use the regular `@Import` annotation to import auto-configurations as they are handled in a specific way by Spring Boot. |

Alternatively, additional auto-configurations can be added for any use of a slice annotation by registering them in a file stored in `META-INF/spring` as shown in the following example:

*META-INF/spring/org.springframework.boot.test.autoconfigure.jdbc.JdbcTest.imports*

```
com.example.IntegrationAutoConfiguration
```

In this example, the `com.example.IntegrationAutoConfiguration` is enabled on every test annotated with `@JdbcTest`.

| | |
|---|---|
| **TIP** | You can use comments with `#` in this file. |
| **TIP** | A slice or `@AutoConfigure⋯` annotation can be customized this way as long as it is meta-annotated with `@ImportAutoConfiguration`. |

#### User Configuration and Slicing

If you structure your code in a sensible way, your `@SpringBootApplication` class is used by default as the configuration of your tests.

It then becomes important not to litter the application's main class with configuration settings that are specific to a particular area of its functionality.

Assume that you are using Spring Data MongoDB, you rely on the auto-configuration for it, and you have enabled auditing. You could define your `@SpringBootApplication` as follows:

*Java*

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.mongodb.config.EnableMongoAuditing;

@SpringBootApplication
@EnableMongoAuditing
public class MyApplication {

    // ...

}
```

*Kotlin*

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.data.mongodb.config.EnableMongoAuditing

@SpringBootApplication
@EnableMongoAuditing
class MyApplication {

    // ...

}
```

Because this class is the source configuration for the test, any slice test actually tries to enable Mongo auditing, which is definitely not what you want to do. A recommended approach is to move that area-specific configuration to a separate `@Configuration` class at the same level as your application, as shown in the following example:

*Java*

```
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.config.EnableMongoAuditing;

@Configuration(proxyBeanMethods = false)
@EnableMongoAuditing
public class MyMongoConfiguration {

    // ...

}
```

*Kotlin*

```
import org.springframework.context.annotation.Configuration
import org.springframework.data.mongodb.config.EnableMongoAuditing

@Configuration(proxyBeanMethods = false)
@EnableMongoAuditing
class MyMongoConfiguration {

    // ...

}
```

**NOTE** Depending on the complexity of your application, you may either have a single `@Configuration` class for your customizations or one class per domain area. The latter approach lets you enable it in one of your tests, if necessary, with the `@Import` annotation. See this how-to section for more details on when you might want to enable specific `@Configuration` classes for slice tests.

Test slices exclude `@Configuration` classes from scanning. For example, for a `@WebMvcTest`, the following configuration will not include the given `WebMvcConfigurer` bean in the application context loaded by the test slice:

*Java*

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration(proxyBeanMethods = false)
public class MyWebConfiguration {

    @Bean
    public WebMvcConfigurer testConfigurer() {
        return new WebMvcConfigurer() {
            // ...
        };
    }

}
```

*Kotlin*

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Configuration(proxyBeanMethods = false)
class MyWebConfiguration {

    @Bean
    fun testConfigurer(): WebMvcConfigurer {
        return object : WebMvcConfigurer {
            // ...
        }
    }

}
```

The configuration below will, however, cause the custom `WebMvcConfigurer` to be loaded by the test slice.

*Java*

```
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Component
public class MyWebMvcConfigurer implements WebMvcConfigurer {

    // ...

}
```

*Kotlin*

```
import org.springframework.stereotype.Component
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Component
class MyWebMvcConfigurer : WebMvcConfigurer {

    // ...

}
```

Another source of confusion is classpath scanning. Assume that, while you structured your code in a sensible way, you need to scan an additional package. Your application may resemble the following code:

*Java*

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@ComponentScan({ "com.example.app", "com.example.another" })
public class MyApplication {

    // ...

}
```

*Kotlin*

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.context.annotation.ComponentScan

@SpringBootApplication
@ComponentScan("com.example.app", "com.example.another")
class MyApplication {

    // ...

}
```

Doing so effectively overrides the default component scan directive with the side effect of scanning those two packages regardless of the slice that you chose. For instance, a `@DataJpaTest` seems to suddenly scan components and user configurations of your application. Again, moving the custom directive to a separate class is a good way to fix this issue.

| **TIP** | If this is not an option for you, you can create a `@SpringBootConfiguration` somewhere in the hierarchy of your test so that it is used instead. Alternatively, you can specify a source for your test, which disables the behavior of finding a default one. |
|---|---|

#### Using Spock to Test Spring Boot Applications

Spock 2.2 or later can be used to test a Spring Boot application. To do so, add a dependency on a `-groovy-4.0` version of Spock's `spock-spring` module to your application's build. `spock-spring` integrates Spring's test framework into Spock. See the documentation for Spock's Spring module for further details.

### 7.9.4. Testcontainers

The Testcontainers library provides a way to manage services running inside Docker containers. It integrates with JUnit, allowing you to write a test class that can start up a container before any of the tests run. Testcontainers is especially useful for writing integration tests that talk to a real backend service such as MySQL, MongoDB, Cassandra and others.

Testcontainers can be used in a Spring Boot test as follows:

*Java*

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

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

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>("neo4j:5");

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

}
```

*Kotlin*

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.testcontainers.containers.Neo4jContainer
import org.testcontainers.junit.jupiter.Container
import org.testcontainers.junit.jupiter.Testcontainers

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

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

    companion object {

        @Container
        val neo4j = Neo4jContainer("neo4j:5")

    }

}
```

This will start up a docker container running Neo4j (if Docker is running locally) before any of the tests are run. In most cases, you will need to configure the application to connect to the service running in the container.

#### Service Connections

A service connection is a connection to any remote service. Spring Boot's auto-configuration can consume the details of a service connection and use them to establish a connection to a remote service. When doing so, the connection details take precedence over any connection-related configuration properties.

When using Testcontainers, connection details can be automatically created for a service running in a container by annotating the container field in the test class.

*Java*

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Container
    @ServiceConnection
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>("neo4j:5");

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

}
```

*Kotlin*

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.testcontainers.service.connection.ServiceConnection
import org.testcontainers.containers.Neo4jContainer
import org.testcontainers.junit.jupiter.Container
import org.testcontainers.junit.jupiter.Testcontainers

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

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

    companion object {

        @Container
        @ServiceConnection
        val neo4j = Neo4jContainer("neo4j:5")

    }

}
```

Thanks to `@ServiceConnection`, the above configuration allows Neo4j-related beans in the application to communicate with Neo4j running inside the Testcontainers-managed Docker container. This is done by automatically defining a `Neo4jConnectionDetails` bean which is then used by the Neo4j auto-configuration, overriding any connection-related configuration properties.

**NOTE** You'll need to add the `spring-boot-testcontainers` module as a test dependency in order to use service connections with Testcontainers.

Service connection annotations are processed by `ContainerConnectionDetailsFactory` classes registered with `spring.factories`. A `ContainerConnectionDetailsFactory` can create a `ConnectionDetails` bean based on a specific `Container` subclass, or the Docker image name.

The following service connection factories are provided in the `spring-boot-testcontainers` jar:

| Connection Details | Matched on |
|---|---|
| `ActiveMQConnectionDetails` | Containers named "symptoma/activemq" |
| `CassandraConnectionDetails` | Containers of type `CassandraContainer` |
| `CouchbaseConnectionDetails` | Containers of type `CouchbaseContainer` |
| `ElasticsearchConnectionDetails` | Containers of type `ElasticsearchContainer` |

| Connection Details | Matched on |
| --- | --- |
| `FlywayConnectionDetails` | Containers of type `JdbcDatabaseContainer` |
| `JdbcConnectionDetails` | Containers of type `JdbcDatabaseContainer` |
| `KafkaConnectionDetails` | Containers of type `org.testcontainers.containers.KafkaContainer` or `RedpandaContainer` |
| `LiquibaseConnectionDetails` | Containers of type `JdbcDatabaseContainer` |
| `MongoConnectionDetails` | Containers of type `MongoDBContainer` |
| `Neo4jConnectionDetails` | Containers of type `Neo4jContainer` |
| `OtlpMetricsConnectionDetails` | Containers named "otel/opentelemetry-collector-contrib" |
| `OtlpTracingConnectionDetails` | Containers named "otel/opentelemetry-collector-contrib" |
| `PulsarConnectionDetails` | Containers of type `PulsarContainer` |
| `R2dbcConnectionDetails` | Containers of type `MariaDBContainer`, `MSSQLServerContainer`, `MySQLContainer`, `OracleContainer`, or `PostgreSQLContainer` |
| `RabbitConnectionDetails` | Containers of type `RabbitMQContainer` |
| `RedisConnectionDetails` | Containers named "redis" |
| `ZipkinConnectionDetails` | Containers named "openzipkin/zipkin" |

**TIP**

By default all applicable connection details beans will be created for a given `Container`. For example, a `PostgreSQLContainer` will create both `JdbcConnectionDetails` and `R2dbcConnectionDetails`.

If you want to create only a subset of the applicable types, you can use the `type` attribute of `@ServiceConnection`.

By default `Container.getDockerImageName().getRepository()` is used to obtain the name used to find connection details. The repository portion of the Docker image name ignores any registry and the version. This works as long as Spring Boot is able to get the instance of the `Container`, which is the case when using a `static` field like in the example above.

If you're using a `@Bean` method, Spring Boot won't call the bean method to get the Docker image name, because this would cause eager initialization issues. Instead, the return type of the bean method is used to find out which connection detail should be used. This works as long as you're using typed containers, e.g. `Neo4jContainer` or `RabbitMQContainer`. This stops working if you're using `GenericContainer`, e.g. with Redis, as shown in the following example:

*Java*

```
import org.testcontainers.containers.GenericContainer;

import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;
import org.springframework.context.annotation.Bean;

@TestConfiguration(proxyBeanMethods = false)
public class MyRedisConfiguration {

    @Bean
    @ServiceConnection(name = "redis")
    public GenericContainer<?> redisContainer() {
        return new GenericContainer<>("redis:7");
    }

}
```

*Kotlin*

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.service.connection.ServiceConnection
import org.springframework.context.annotation.Bean
import org.testcontainers.containers.GenericContainer

@TestConfiguration(proxyBeanMethods = false)
class MyRedisConfiguration {

    @Bean
    @ServiceConnection(name = "redis")
    fun redisContainer(): GenericContainer<*> {
        return GenericContainer("redis:7")
    }

}
```

Spring Boot can't tell from `GenericContainer` which container image is used, so the `name` attribute from `@ServiceConnection` must be used to provide that hint.

You can also can use the `name` attribute of `@ServiceConnection` to override which connection detail will be used, for example when using custom images. If you are using the Docker image `registry.mycompany.com/mirror/myredis`, you'd use `@ServiceConnection(name="redis")` to ensure `RedisConnectionDetails` are created.

#### Dynamic Properties

A slightly more verbose but also more flexible alternative to service connections is `@DynamicPropertySource`. A static `@DynamicPropertySource` method allows adding dynamic property

values to the Spring Environment.

*Java*

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>("neo4j:5");

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

    @DynamicPropertySource
    static void neo4jProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.neo4j.uri", neo4j::getBoltUrl);
    }

}
```

*Kotlin*

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.context.DynamicPropertyRegistry
import org.springframework.test.context.DynamicPropertySource
import org.testcontainers.containers.Neo4jContainer
import org.testcontainers.junit.jupiter.Container
import org.testcontainers.junit.jupiter.Testcontainers

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

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

    companion object {

        @Container
        val neo4j = Neo4jContainer("neo4j:5")

        @DynamicPropertySource
        fun neo4jProperties(registry: DynamicPropertyRegistry) {
            registry.add("spring.neo4j.uri") { neo4j.boltUrl }
        }

    }

}
```

The above configuration allows Neo4j-related beans in the application to communicate with Neo4j running inside the Testcontainers-managed Docker container.

### 7.9.5. Test Utilities

A few test utility classes that are generally useful when testing your application are packaged as part of `spring-boot`.

#### ConfigDataApplicationContextInitializer

`ConfigDataApplicationContextInitializer` is an `ApplicationContextInitializer` that you can apply to your tests to load Spring Boot `application.properties` files. You can use it when you do not need the full set of features provided by `@SpringBootTest`, as shown in the following example:

*Java*

```
import org.springframework.boot.test.context.ConfigDataApplicationContextInitializer;
import org.springframework.test.context.ContextConfiguration;

@ContextConfiguration(classes = Config.class, initializers =
ConfigDataApplicationContextInitializer.class)
class MyConfigFileTests {

    // ...

}
```

*Kotlin*

```
import org.springframework.boot.test.context.ConfigDataApplicationContextInitializer
import org.springframework.test.context.ContextConfiguration

@ContextConfiguration(classes = [Config::class], initializers =
[ConfigDataApplicationContextInitializer::class])
class MyConfigFileTests {

    // ...

}
```

**NOTE** Using `ConfigDataApplicationContextInitializer` alone does not provide support for `@Value("${…}")` injection. Its only job is to ensure that `application.properties` files are loaded into Spring's `Environment`. For `@Value` support, you need to either additionally configure a `PropertySourcesPlaceholderConfigurer` or use `@SpringBootTest`, which auto-configures one for you.

#### TestPropertyValues

`TestPropertyValues` lets you quickly add properties to a `ConfigurableEnvironment` or `ConfigurableApplicationContext`. You can call it with `key=value` strings, as follows:

*Java*

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

import org.springframework.boot.test.util.TestPropertyValues;
import org.springframework.mock.env.MockEnvironment;

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

class MyEnvironmentTests {

    @Test
    void testPropertySources() {
        MockEnvironment environment = new MockEnvironment();
        TestPropertyValues.of("org=Spring", "name=Boot").applyTo(environment);
        assertThat(environment.getProperty("name")).isEqualTo("Boot");
    }

}
```

*Kotlin*

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.boot.test.util.TestPropertyValues
import org.springframework.mock.env.MockEnvironment

class MyEnvironmentTests {

    @Test
    fun testPropertySources() {
        val environment = MockEnvironment()
        TestPropertyValues.of("org=Spring", "name=Boot").applyTo(environment)
        assertThat(environment.getProperty("name")).isEqualTo("Boot")
    }

}
```

#### OutputCapture

`OutputCapture` is a JUnit `Extension` that you can use to capture `System.out` and `System.err` output. To use it, add `@ExtendWith(OutputCaptureExtension.class)` and inject `CapturedOutput` as an argument to your test class constructor or test method as follows:

*Java*

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

import org.springframework.boot.test.system.CapturedOutput;
import org.springframework.boot.test.system.OutputCaptureExtension;

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

@ExtendWith(OutputCaptureExtension.class)
class MyOutputCaptureTests {

    @Test
    void testName(CapturedOutput output) {
        System.out.println("Hello World!");
        assertThat(output).contains("World");
    }

}
```

*Kotlin*

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.extension.ExtendWith
import org.springframework.boot.test.system.CapturedOutput
import org.springframework.boot.test.system.OutputCaptureExtension

@ExtendWith(OutputCaptureExtension::class)
class MyOutputCaptureTests {

    @Test
    fun testName(output: CapturedOutput?) {
        println("Hello World!")
        assertThat(output).contains("World")
    }

}
```

#### TestRestTemplate

`TestRestTemplate` is a convenience alternative to Spring's `RestTemplate` that is useful in integration tests. You can get a vanilla template or one that sends Basic HTTP authentication (with a username and password). In either case, the template is fault tolerant. This means that it behaves in a test-friendly way by not throwing exceptions on 4xx and 5xx errors. Instead, such errors can be detected through the returned `ResponseEntity` and its status code.

| TIP | Spring Framework 5.0 provides a new `WebTestClient` that works for WebFlux integration tests and both WebFlux and MVC end-to-end testing. It provides a fluent API for assertions, unlike `TestRestTemplate`. |
|---|---|

It is recommended, but not mandatory, to use the Apache HTTP Client (version 5.1 or better). If you have that on your classpath, the `TestRestTemplate` responds by configuring the client appropriately. If you do use Apache's HTTP client, some additional test-friendly features are enabled:

- Redirects are not followed (so you can assert the response location).
- Cookies are ignored (so the template is stateless).

`TestRestTemplate` can be instantiated directly in your integration tests, as shown in the following example:

*Java*

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

import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.ResponseEntity;

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

class MyTests {

    private final TestRestTemplate template = new TestRestTemplate();

    @Test
    void testRequest() {
        ResponseEntity<String> headers =
this.template.getForEntity("https://myhost.example.com/example", String.class);
        assertThat(headers.getHeaders().getLocation()).hasHost("other.example.com");
    }

}
```

*Kotlin*

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.boot.test.web.client.TestRestTemplate

class MyTests {

    private val template = TestRestTemplate()

    @Test
    fun testRequest() {
        val headers = template.getForEntity("https://myhost.example.com/example",
String::class.java)
        assertThat(headers.headers.location).hasHost("other.example.com")
    }

}
```

Alternatively, if you use the `@SpringBootTest` annotation with `WebEnvironment.RANDOM_PORT` or `WebEnvironment.DEFINED_PORT`, you can inject a fully configured `TestRestTemplate` and start using it. If necessary, additional customizations can be applied through the `RestTemplateBuilder` bean. Any URLs that do not specify a host and port automatically connect to the embedded server, as shown in the following example:

*Java*

```
import java.time.Duration;

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.context.TestConfiguration;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.http.HttpHeaders;

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

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

    @Autowired
    private TestRestTemplate template;

    @Test
    void testRequest() {
        HttpHeaders headers = this.template.getForEntity("/example",
String.class).getHeaders();
        assertThat(headers.getLocation()).hasHost("other.example.com");
    }

    @TestConfiguration(proxyBeanMethods = false)
    static class RestTemplateBuilderConfiguration {

        @Bean
        RestTemplateBuilder restTemplateBuilder() {
            return new RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1))
                .setReadTimeout(Duration.ofSeconds(1));
        }

    }

}
```

*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.context.TestConfiguration
import org.springframework.boot.test.web.client.TestRestTemplate
import org.springframework.boot.web.client.RestTemplateBuilder
import org.springframework.context.annotation.Bean
import java.time.Duration

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MySpringBootTests(@Autowired val template: TestRestTemplate) {

    @Test
    fun testRequest() {
        val headers = template.getForEntity("/example", String::class.java).headers
        assertThat(headers.location).hasHost("other.example.com")
    }

    @TestConfiguration(proxyBeanMethods = false)
    internal class RestTemplateBuilderConfiguration {

        @Bean
        fun restTemplateBuilder(): RestTemplateBuilder {
            return RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1))
                .setReadTimeout(Duration.ofSeconds(1))
        }

    }

}
```

## 7.10. Docker Compose Support

Docker Compose is a popular technology that can be used to define and manage multiple containers for services that your application needs. A `compose.yml` file is typically created next to your application which defines and configures service containers.

A typical workflow with Docker Compose is to run `docker compose up`, work on your application with it connecting to started services, then run `docker compose down` when you are finished.

The `spring-boot-docker-compose` module can be included in a project to provide support for working with containers using Docker Compose. Add the module dependency to your build, as shown in the following listings for Maven and Gradle:

*Maven*

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-docker-compose</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

*Gradle*

```
dependencies {
    developmentOnly("org.springframework.boot:spring-boot-docker-compose")
}
```

When this module is included as a dependency Spring Boot will do the following:

- Search for a `compose.yml` and other common compose filenames in your working directory
- Call `docker compose up` with the discovered `compose.yml`
- Create service connection beans for each supported container
- Call `docker compose stop` when the application is shutdown

If the Docker Compose services are already running when starting the application, Spring Boot will only create the service connection beans for each supported container. It will not call `docker compose up` again and it will not call `docker compose stop` when the application is shutdown.

**TIP**

Repackaged archives do not contain Spring Boot's Docker Compose by default. If you want to use this support, you need to include it. When using the Maven plugin, set the `excludeDockerCompose` property to `false`. When using the Gradle plugin, configure the task's classpath to include the `developmentOnly` configuration.

### 7.10.1. Prerequisites

You need to have the `docker` and `docker compose` (or `docker-compose`) CLI applications on your path. The minimum supported Docker Compose version is 2.2.0.

### 7.10.2. Service Connections

A service connection is a connection to any remote service. Spring Boot's auto-configuration can consume the details of a service connection and use them to establish a connection to a remote service. When doing so, the connection details take precedence over any connection-related configuration properties.

When using Spring Boot's Docker Compose support, service connections are established to the port mapped by the container.