NOTE: Docker compose is usually used in such a way that the ports inside the container are mapped to ephemeral ports on your computer. For example, a Postgres server may run inside the container using port 5432 but be mapped to a totally different port locally. The service connection will always discover and use the locally mapped port.

Service connections are established by using the image name of the container. The following service connections are currently supported:

Connection Details	Matched on
`ActiveMQConnectionDetails`	Containers named "symptoma/activemq"
`CassandraConnectionDetails`	Containers named "cassandra"
`ElasticsearchConnectionDetails`	Containers named "elasticsearch"
`JdbcConnectionDetails`	Containers named "gvenzl/oracle-free", "gvenzl/oracle-xe", "mariadb", "mssql/server", "mysql", or "postgres"
`MongoConnectionDetails`	Containers named "mongo"
`Neo4jConnectionDetails`	Containers named "neo4j"
`OtlpMetricsConnectionDetails`	Containers named "otel/opentelemetry-collector-contrib"
`OtlpTracingConnectionDetails`	Containers named "otel/opentelemetry-collector-contrib"
`PulsarConnectionDetails`	Containers named "apachepulsar/pulsar"
`R2dbcConnectionDetails`	Containers named "gvenzl/oracle-free", "gvenzl/oracle-xe", "mariadb", "mssql/server", "mysql", or "postgres"
`RabbitConnectionDetails`	Containers named "rabbitmq"
`RedisConnectionDetails`	Containers named "redis"
`ZipkinConnectionDetails`	Containers named "openzipkin/zipkin".

7.10.3. Custom Images

Sometimes you may need to use your own version of an image to provide a service. You can use any custom image as long as it behaves in the same way as the standard image. Specifically, any environment variables that the standard image supports must also be used in your custom image.

If your image uses a different name, you can use a label in your `compose.yml` file so that Spring Boot can provide a service connection. Use a label named `org.springframework.boot.service-connection` to provide the service name.

For example:

```
services:
  redis:
    image: 'mycompany/mycustomredis:7.0'
    ports:
      - '6379'
    labels:
      org.springframework.boot.service-connection: redis
```

7.10.4. Skipping Specific Containers

If you have a container image defined in your `compose.yml` that you don't want connected to your application you can use a label to ignore it. Any container with labeled with `org.springframework.boot.ignore` will be ignored by Spring Boot.

For example:

```
services:
  redis:
    image: 'redis:7.0'
    ports:
      - '6379'
    labels:
      org.springframework.boot.ignore: true
```

7.10.5. Using a Specific Compose File

If your compose file is not in the same directory as your application, or if it's named differently, you can use `spring.docker.compose.file` in your `application.properties` or `application.yaml` to point to a different file. Properties can be defined as an exact path or a path that's relative to your application.

For example:

Properties

```
spring.docker.compose.file=../my-compose.yml
```

Yaml

```
spring:
  docker:
    compose:
      file: "../my-compose.yml"
```

7.10.6. Waiting for Container Readiness

Containers started by Docker Compose may take some time to become fully ready. The recommended way of checking for readiness is to add a `healthcheck` section under the service definition in your `compose.yml` file.

Since it's not uncommon for `healthcheck` configuration to be omitted from `compose.yml` files, Spring Boot also checks directly for service readiness. By default, a container is considered ready when a TCP/IP connection can be established to its mapped port.

You can disable this on a per-container basis by adding a `org.springframework.boot.readiness-check.tcp.disable` label in your `compose.yml` file.

For example:

```
services:
  redis:
    image: 'redis:7.0'
    ports:
      - '6379'
    labels:
      org.springframework.boot.readiness-check.tcp.disable: true
```

You can also change timeout values in your `application.properties` or `application.yaml` file:

Properties

```
spring.docker.compose.readiness.tcp.connect-timeout=10s
spring.docker.compose.readiness.tcp.read-timeout=5s
```

Yaml

```
spring:
  docker:
    compose:
      readiness:
        tcp:
          connect-timeout: 10s
          read-timeout: 5s
```

The overall timeout can be configured using `spring.docker.compose.readiness.timeout`.

7.10.7. Controlling the Docker Compose Lifecycle

By default Spring Boot calls `docker compose up` when your application starts and `docker compose stop` when it's shut down. If you prefer to have different lifecycle management you can use the `spring.docker.compose.lifecycle-management` property.

The following values are supported:

- `none` - Do not start or stop Docker Compose
- `start-only` - Start Docker Compose when the application starts and leave it running
- `start-and-stop` - Start Docker Compose when the application starts and stop it when the JVM exits

In addition you can use the `spring.docker.compose.start.command` property to change whether `docker compose up` or `docker compose start` is used. The `spring.docker.compose.stop.command` allows you to configure if `docker compose down` or `docker compose stop` is used.

The following example shows how lifecycle management can be configured:

Properties

```
spring.docker.compose.lifecycle-management=start-and-stop
spring.docker.compose.start.command=start
spring.docker.compose.stop.command=down
spring.docker.compose.stop.timeout=1m
```

Yaml

```
spring:
  docker:
    compose:
      lifecycle-management: start-and-stop
      start:
        command: start
      stop:
        command: down
        timeout: 1m
```

7.10.8. Activating Docker Compose Profiles

Docker Compose profiles are similar to Spring profiles in that they let you adjust your Docker Compose configuration for specific environments. If you want to activate a specific Docker Compose profile you can use the `spring.docker.compose.profiles.active` property in your `application.properties` or `application.yaml` file:

Properties

```
spring.docker.compose.profiles.active=myprofile
```

Yaml

```
spring:
  docker:
    compose:
      profiles:
        active: "myprofile"
```

7.10.9. Using Docker Compose in Tests

By default, Spring Boot's Docker Compose support is disabled when running tests.

To enable Docker Compose support in tests, set `spring.docker.compose.skip.in-tests` to `false`.

When using Gradle, you also need to change the configuration of the `spring-boot-docker-compose` dependency from `developmentOnly` to `testAndDevelopmentOnly`:

Gradle

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

7.11. Testcontainers Support

As well as using Testcontainers for integration testing, it's also possible to use them at development time. The next sections will provide more details about that.

7.11.1. Using Testcontainers at Development Time

This approach allows developers to quickly start containers for the services that the application depends on, removing the need to manually provision things like database servers. Using Testcontainers in this way provides functionality similar to Docker Compose, except that your container configuration is in Java rather than YAML.

To use Testcontainers at development time you need to launch your application using your "test" classpath rather than "main". This will allow you to access all declared test dependencies and give you a natural place to write your test configuration.

To create a test launchable version of your application you should create an "Application" class in the `src/test` directory. For example, if your main application is in `src/main/java/com/example/MyApplication.java`, you should create `src/test/java/com/example/TestMyApplication.java`

The `TestMyApplication` class can use the `SpringApplication.from(…)` method to launch the real application:

Java

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

public class TestMyApplication {

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

}
```

Kotlin

```
import org.springframework.boot.fromApplication

fun main(args: Array<String>) {
    fromApplication<MyApplication>().run(*args)
}
```

You'll also need to define the `Container` instances that you want to start along with your application. To do this, you need to make sure that the `spring-boot-testcontainers` module has been added as a `test` dependency. Once that has been done, you can create a `@TestConfiguration` class that declares `@Bean` methods for the containers you want to start.

You can also annotate your `@Bean` methods with `@ServiceConnection` in order to create `ConnectionDetails` beans. See the service connections section for details of the supported technologies.

A typical Testcontainers configuration would look like this:

Java

```
import org.testcontainers.containers.Neo4jContainer;

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 MyContainersConfiguration {

    @Bean
    @ServiceConnection
    public Neo4jContainer<?> neo4jContainer() {
        return new Neo4jContainer<>("neo4j:5");
    }

}
```

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.Neo4jContainer

@TestConfiguration(proxyBeanMethods = false)
class MyContainersConfiguration {

    @Bean
    @ServiceConnection
    fun neo4jContainer(): Neo4jContainer<*> {
        return Neo4jContainer("neo4j:5")
    }

}
```

NOTE The lifecycle of `Container` beans is automatically managed by Spring Boot. Containers will be started and stopped automatically.

TIP You can use the `spring.testcontainers.beans.startup` property to change how containers are started. By default `sequential` startup is used, but you may also choose `parallel` if you wish to start multiple containers in parallel.

Once you have defined your test configuration, you can use the `with(…)` method to attach it to your test launcher:

Java

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

public class TestMyApplication {

    public static void main(String[] args) {

SpringApplication.from(MyApplication::main).with(MyContainersConfiguration.class).run(
args);
    }

}
```

Kotlin

```kotlin
import org.springframework.boot.fromApplication
import org.springframework.boot.with

fun main(args: Array<String>) {
    fromApplication<MyApplication>().with(MyContainersConfiguration::class).run(*args)
}
```

You can now launch `TestMyApplication` as you would any regular Java `main` method application to start your application and the containers that it needs to run.

TIP You can use the Maven goal `spring-boot:test-run` or the Gradle task `bootTestRun` to do this from the command line.

Contributing Dynamic Properties at Development Time

If you want to contribute dynamic properties at development time from your `Container` `@Bean` methods, you can do so by injecting a `DynamicPropertyRegistry`. This works in a similar way to the `@DynamicPropertySource` annotation that you can use in your tests. It allows you to add properties that will become available once your container has started.

A typical configuration would look like this:

Java

```java
import org.testcontainers.containers.MongoDBContainer;

import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.test.context.DynamicPropertyRegistry;

@TestConfiguration(proxyBeanMethods = false)
public class MyContainersConfiguration {

    @Bean
    public MongoDBContainer mongoDbContainer(DynamicPropertyRegistry properties) {
        MongoDBContainer container = new MongoDBContainer("mongo:5.0");
        properties.add("spring.data.mongodb.host", container::getHost);
        properties.add("spring.data.mongodb.port", container::getFirstMappedPort);
        return container;
    }

}
```

Kotlin

```kotlin
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.context.annotation.Bean
import org.springframework.test.context.DynamicPropertyRegistry
import org.testcontainers.containers.MongoDBContainer

@TestConfiguration(proxyBeanMethods = false)
class MyContainersConfiguration {

    @Bean
    fun monogDbContainer(properties: DynamicPropertyRegistry): MongoDBContainer {
        var container = MongoDBContainer("mongo:5.0")
        properties.add("spring.data.mongodb.host", container::getHost)
        properties.add("spring.data.mongodb.port", container::getFirstMappedPort)
        return container
    }

}
```

NOTE Using a `@ServiceConnection` is recommended whenever possible, however, dynamic properties can be a useful fallback for technologies that don't yet have `@ServiceConnection` support.

Importing Testcontainer Declaration Classes

A common pattern when using Testcontainers is to declare `Container` instances as static fields. Often these fields are defined directly on the test class. They can also be declared on a parent class or on an interface that the test implements.

For example, the following `MyContainers` interface declares `mongo` and `neo4j` containers:

```
import org.testcontainers.containers.MongoDBContainer;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;

import org.springframework.boot.testcontainers.service.connection.ServiceConnection;

public interface MyContainers {

    @Container
    @ServiceConnection
    MongoDBContainer mongoContainer = new MongoDBContainer("mongo:5.0");

    @Container
    @ServiceConnection
    Neo4jContainer<?> neo4jContainer = new Neo4jContainer<>("neo4j:5");

}
```

If you already have containers defined in this way, or you just prefer this style, you can import these declaration classes rather than defining your containers as `@Bean` methods. To do so, add the `@ImportTestcontainers` annotation to your test configuration class:

Java

```
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.context.ImportTestcontainers;

@TestConfiguration(proxyBeanMethods = false)
@ImportTestcontainers(MyContainers.class)
public class MyContainersConfiguration {

}
```

Kotlin

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.context.ImportTestcontainers

@TestConfiguration(proxyBeanMethods = false)
@ImportTestcontainers(MyContainers::class)
class MyContainersConfiguration
```

TIP If you don’t intend to use the service connections feature but want to use `@DynamicPropertySource` instead, remove the `@ServiceConnection` annotation from the `Container` fields. You can also add `@DynamicPropertySource` annotated methods to your declaration class.

Using DevTools with Testcontainers at Development Time

When using devtools, you can annotate beans and bean methods with `@RestartScope`. Such beans won't be recreated when the devtools restart the application. This is especially useful for Testcontainer `Container` beans, as they keep their state despite the application restart.

Java

```java
import org.testcontainers.containers.MongoDBContainer;

import org.springframework.boot.devtools.restart.RestartScope;
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 MyContainersConfiguration {

    @Bean
    @RestartScope
    @ServiceConnection
    public MongoDBContainer mongoDbContainer() {
        return new MongoDBContainer("mongo:5.0");
    }

}
```

Kotlin

```kotlin
import org.springframework.boot.devtools.restart.RestartScope
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.MongoDBContainer

@TestConfiguration(proxyBeanMethods = false)
class MyContainersConfiguration {

    @Bean
    @RestartScope
    @ServiceConnection
    fun monogDbContainer(): MongoDBContainer {
        return MongoDBContainer("mongo:5.0")
    }

}
```

WARNING If you're using Gradle and want to use this feature, you need to change the configuration of the `spring-boot-devtools` dependency from `developmentOnly` to `testAndDevelopmentOnly`. With the default scope of `developmentOnly`, the `bootTestRun` task will not pick up changes in your code, as the devtools are not active.

7.12. Creating Your Own Auto-configuration

If you work in a company that develops shared libraries, or if you work on an open-source or commercial library, you might want to develop your own auto-configuration. Auto-configuration classes can be bundled in external jars and still be picked up by Spring Boot.

Auto-configuration can be associated to a "starter" that provides the auto-configuration code as well as the typical libraries that you would use with it. We first cover what you need to know to build your own auto-configuration and then we move on to the typical steps required to create a custom starter.

7.12.1. Understanding Auto-configured Beans

Classes that implement auto-configuration are annotated with `@AutoConfiguration`. This annotation itself is meta-annotated with `@Configuration`, making auto-configurations standard `@Configuration` classes. Additional `@Conditional` annotations are used to constrain when the auto-configuration should apply. Usually, auto-configuration classes use `@ConditionalOnClass` and `@ConditionalOnMissingBean` annotations. This ensures that auto-configuration applies only when relevant classes are found and when you have not declared your own `@Configuration`.

You can browse the source code of `spring-boot-autoconfigure` to see the `@AutoConfiguration` classes that Spring provides (see the `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file).

7.12.2. Locating Auto-configuration Candidates

Spring Boot checks for the presence of a `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file within your published jar. The file should list your configuration classes, with one class name per line, as shown in the following example:

```
com.mycorp.libx.autoconfigure.LibXAutoConfiguration
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

TIP You can add comments to the imports file using the `#` character.

NOTE

Auto-configurations must be loaded *only* by being named in the imports file. Make sure that they are defined in a specific package space and that they are never the target of component scanning. Furthermore, auto-configuration classes should not enable component scanning to find additional components. Specific `@Import` annotations should be used instead.

If your configuration needs to be applied in a specific order, you can use the `before`, `beforeName`, `after` and `afterName` attributes on the `@AutoConfiguration` annotation or the dedicated `@AutoConfigureBefore` and `@AutoConfigureAfter` annotations. For example, if you provide web-specific configuration, your class may need to be applied after `WebMvcAutoConfiguration`.

If you want to order certain auto-configurations that should not have any direct knowledge of each other, you can also use `@AutoConfigureOrder`. That annotation has the same semantic as the regular `@Order` annotation but provides a dedicated order for auto-configuration classes.

As with standard `@Configuration` classes, the order in which auto-configuration classes are applied only affects the order in which their beans are defined. The order in which those beans are subsequently created is unaffected and is determined by each bean's dependencies and any `@DependsOn` relationships.

7.12.3. Condition Annotations

You almost always want to include one or more `@Conditional` annotations on your auto-configuration class. The `@ConditionalOnMissingBean` annotation is one common example that is used to allow developers to override auto-configuration if they are not happy with your defaults.

Spring Boot includes a number of `@Conditional` annotations that you can reuse in your own code by annotating `@Configuration` classes or individual `@Bean` methods. These annotations include:

- Class Conditions
- Bean Conditions
- Property Conditions
- Resource Conditions
- Web Application Conditions
- SpEL Expression Conditions

Class Conditions

The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations let `@Configuration` classes be included based on the presence or absence of specific classes. Due to the fact that annotation metadata is parsed by using ASM, you can use the `value` attribute to refer to the real class, even though that class might not actually appear on the running application classpath. You can also use the `name` attribute if you prefer to specify the class name by using a `String` value.

This mechanism does not apply the same way to `@Bean` methods where typically the return type is the target of the condition: before the condition on the method applies, the JVM will have loaded the class and potentially processed method references which will fail if the class is not present.

To handle this scenario, a separate `@Configuration` class can be used to isolate the condition, as shown in the following example:

Java

```java
import org.springframework.boot.autoconfigure.AutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@AutoConfiguration
// Some conditions ...
public class MyAutoConfiguration {

    // Auto-configured beans ...

    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(SomeService.class)
    public static class SomeServiceConfiguration {

        @Bean
        @ConditionalOnMissingBean
        public SomeService someService() {
            return new SomeService();
        }

    }

}
```

Kotlin

```kotlin
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
// Some conditions ...
class MyAutoConfiguration {

    // Auto-configured beans ...
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(SomeService::class)
    class SomeServiceConfiguration {

        @Bean
        @ConditionalOnMissingBean
        fun someService(): SomeService {
            return SomeService()
        }

    }

}
```

TIP If you use `@ConditionalOnClass` or `@ConditionalOnMissingClass` as a part of a meta-annotation to compose your own composed annotations, you must use `name` as referring to the class in such a case is not handled.

Bean Conditions

The `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations let a bean be included based on the presence or absence of specific beans. You can use the `value` attribute to specify beans by type or `name` to specify beans by name. The `search` attribute lets you limit the `ApplicationContext` hierarchy that should be considered when searching for beans.

When placed on a `@Bean` method, the target type defaults to the return type of the method, as shown in the following example:

Java

```java
import org.springframework.boot.autoconfigure.AutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;

@AutoConfiguration
public class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public SomeService someService() {
        return new SomeService();
    }

}
```

Kotlin

```kotlin
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    fun someService(): SomeService {
        return SomeService()
    }

}
```

In the preceding example, the `someService` bean is going to be created if no bean of type `SomeService` is already contained in the `ApplicationContext`.

TIP You need to be very careful about the order in which bean definitions are added, as these conditions are evaluated based on what has been processed so far. For this reason, we recommend using only `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations on auto-configuration classes (since these are guaranteed to load after any user-defined bean definitions have been added).

NOTE `@ConditionalOnBean` and `@ConditionalOnMissingBean` do not prevent `@Configuration` classes from being created. The only difference between using these conditions at the class level and marking each contained `@Bean` method with the annotation is that the former prevents registration of the `@Configuration` class as a bean if the condition does not match.

TIP When declaring a @Bean method, provide as much type information as possible in the method's return type. For example, if your bean's concrete class implements an interface the bean method's return type should be the concrete class and not the interface. Providing as much type information as possible in @Bean methods is particularly important when using bean conditions as their evaluation can only rely upon to type information that is available in the method signature.

Property Conditions

The @ConditionalOnProperty annotation lets configuration be included based on a Spring Environment property. Use the prefix and name attributes to specify the property that should be checked. By default, any property that exists and is not equal to false is matched. You can also create more advanced checks by using the havingValue and matchIfMissing attributes.

If multiple names are given in the name attribute, all of the properties have to pass the test for the condition to match.

Resource Conditions

The @ConditionalOnResource annotation lets configuration be included only when a specific resource is present. Resources can be specified by using the usual Spring conventions, as shown in the following example: file:/home/user/test.dat.

Web Application Conditions

The @ConditionalOnWebApplication and @ConditionalOnNotWebApplication annotations let configuration be included depending on whether the application is a web application. A servlet-based web application is any application that uses a Spring WebApplicationContext, defines a session scope, or has a ConfigurableWebEnvironment. A reactive web application is any application that uses a ReactiveWebApplicationContext, or has a ConfigurableReactiveWebEnvironment.

The @ConditionalOnWarDeployment and @ConditionalOnNotWarDeployment annotations let configuration be included depending on whether the application is a traditional WAR application that is deployed to a servlet container. This condition will not match for applications that are run with an embedded web server.

SpEL Expression Conditions

The @ConditionalOnExpression annotation lets configuration be included based on the result of a SpEL expression.

NOTE Referencing a bean in the expression will cause that bean to be initialized very early in context refresh processing. As a result, the bean won't be eligible for post-processing (such as configuration properties binding) and its state may be incomplete.

7.12.4. Testing your Auto-configuration

An auto-configuration can be affected by many factors: user configuration (@Bean definition and

`Environment` customization), condition evaluation (presence of a particular library), and others. Concretely, each test should create a well defined `ApplicationContext` that represents a combination of those customizations. `ApplicationContextRunner` provides a great way to achieve that.

WARNING `ApplicationContextRunner` doesn't work when running the tests in a native image.

`ApplicationContextRunner` is usually defined as a field of the test class to gather the base, common configuration. The following example makes sure that `MyServiceAutoConfiguration` is always invoked:

Java

```
private final ApplicationContextRunner contextRunner = new ApplicationContextRunner()
    .withConfiguration(AutoConfigurations.of(MyServiceAutoConfiguration.class));
```

Kotlin

```
val contextRunner = ApplicationContextRunner()
    .withConfiguration(AutoConfigurations.of(MyServiceAutoConfiguration::class.java))
```

TIP If multiple auto-configurations have to be defined, there is no need to order their declarations as they are invoked in the exact same order as when running the application.

Each test can use the runner to represent a particular use case. For instance, the sample below invokes a user configuration (`UserConfiguration`) and checks that the auto-configuration backs off properly. Invoking `run` provides a callback context that can be used with `AssertJ`.

Java

```
@Test
void defaultServiceBacksOff() {
    this.contextRunner.withUserConfiguration(UserConfiguration.class).run((context) ->
{
        assertThat(context).hasSingleBean(MyService.class);

assertThat(context).getBean("myCustomService").isSameAs(context.getBean(MyService.clas
s));
    });
}

@Configuration(proxyBeanMethods = false)
static class UserConfiguration {

    @Bean
    MyService myCustomService() {
        return new MyService("mine");
    }

}
```

Kotlin

```
@Test
fun defaultServiceBacksOff() {
    contextRunner.withUserConfiguration(UserConfiguration::class.java)
        .run { context: AssertableApplicationContext ->
            assertThat(context).hasSingleBean(MyService::class.java)
            assertThat(context).getBean("myCustomService")
                .isSameAs(context.getBean(MyService::class.java))
        }
}

@Configuration(proxyBeanMethods = false)
internal class UserConfiguration {

    @Bean
    fun myCustomService(): MyService {
        return MyService("mine")
    }

}
```

It is also possible to easily customize the `Environment`, as shown in the following example:

Java

```
@Test
void serviceNameCanBeConfigured() {
    this.contextRunner.withPropertyValues("user.name=test123").run((context) -> {
        assertThat(context).hasSingleBean(MyService.class);
        assertThat(context.getBean(MyService.class).getName()).isEqualTo("test123");
    });
}
```

Kotlin

```
@Test
fun serviceNameCanBeConfigured() {
    contextRunner.withPropertyValues("user.name=test123").run { context:
AssertableApplicationContext ->
        assertThat(context).hasSingleBean(MyService::class.java)
        assertThat(context.getBean(MyService::class.java).name).isEqualTo("test123")
    }
}
```

The runner can also be used to display the `ConditionEvaluationReport`. The report can be printed at `INFO` or `DEBUG` level. The following example shows how to use the `ConditionEvaluationReportLoggingListener` to print the report in auto-configuration tests.

Java

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

import
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListene
r;
import org.springframework.boot.logging.LogLevel;
import org.springframework.boot.test.context.runner.ApplicationContextRunner;

class MyConditionEvaluationReportingTests {

    @Test
    void autoConfigTest() {
        new ApplicationContextRunner()

.withInitializer(ConditionEvaluationReportLoggingListener.forLogLevel(LogLevel.INFO))
            .run((context) -> {
                // Test something...
            });
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListene
r
import org.springframework.boot.logging.LogLevel
import org.springframework.boot.test.context.assertj.AssertableApplicationContext
import org.springframework.boot.test.context.runner.ApplicationContextRunner

class MyConditionEvaluationReportingTests {

    @Test
    fun autoConfigTest() {
        ApplicationContextRunner()

.withInitializer(ConditionEvaluationReportLoggingListener.forLogLevel(LogLevel.INFO))
            .run { context: AssertableApplicationContext? -> }
    }

}
```

Simulating a Web Context

If you need to test an auto-configuration that only operates in a servlet or reactive web application context, use the `WebApplicationContextRunner` or `ReactiveWebApplicationContextRunner` respectively.

Overriding the Classpath

It is also possible to test what happens when a particular class and/or package is not present at runtime. Spring Boot ships with a `FilteredClassLoader` that can easily be used by the runner. In the following example, we assert that if `MyService` is not present, the auto-configuration is properly disabled:

Java

```
@Test
void serviceIsIgnoredIfLibraryIsNotPresent() {
    this.contextRunner.withClassLoader(new FilteredClassLoader(MyService.class))
        .run((context) -> assertThat(context).doesNotHaveBean("myService"));
}
```

Kotlin

```
@Test
fun serviceIsIgnoredIfLibraryIsNotPresent() {
    contextRunner.withClassLoader(FilteredClassLoader(MyService::class.java))
        .run { context: AssertableApplicationContext? ->
            assertThat(context).doesNotHaveBean("myService")
        }
}
```

7.12.5. Creating Your Own Starter

A typical Spring Boot starter contains code to auto-configure and customize the infrastructure of a given technology, let's call that "acme". To make it easily extensible, a number of configuration keys in a dedicated namespace can be exposed to the environment. Finally, a single "starter" dependency is provided to help users get started as easily as possible.

Concretely, a custom starter can contain the following:

- The `autoconfigure` module that contains the auto-configuration code for "acme".
- The `starter` module that provides a dependency to the `autoconfigure` module as well as "acme" and any additional dependencies that are typically useful. In a nutshell, adding the starter should provide everything needed to start using that library.

This separation in two modules is in no way necessary. If "acme" has several flavors, options or optional features, then it is better to separate the auto-configuration as you can clearly express the fact some features are optional. Besides, you have the ability to craft a starter that provides an opinion about those optional dependencies. At the same time, others can rely only on the `autoconfigure` module and craft their own starter with different opinions.

If the auto-configuration is relatively straightforward and does not have optional features, merging the two modules in the starter is definitely an option.

Naming

You should make sure to provide a proper namespace for your starter. Do not start your module names with `spring-boot`, even if you use a different Maven `groupId`. We may offer official support for the thing you auto-configure in the future.

As a rule of thumb, you should name a combined module after the starter. For example, assume that you are creating a starter for "acme" and that you name the auto-configure module `acme-spring-boot` and the starter `acme-spring-boot-starter`. If you only have one module that combines the two, name it `acme-spring-boot-starter`.

Configuration keys

If your starter provides configuration keys, use a unique namespace for them. In particular, do not include your keys in the namespaces that Spring Boot uses (such as `server`, `management`, `spring`, and so on). If you use the same namespace, we may modify these namespaces in the future in ways that break your modules. As a rule of thumb, prefix all your keys with a namespace that you own (for example `acme`).

Make sure that configuration keys are documented by adding field javadoc for each property, as shown in the following example:

Java

```java
import java.time.Duration;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("acme")
public class AcmeProperties {

    /**
     * Whether to check the location of acme resources.
     */
    private boolean checkLocation = true;

    /**
     * Timeout for establishing a connection to the acme server.
     */
    private Duration loginTimeout = Duration.ofSeconds(3);

    public boolean isCheckLocation() {
        return this.checkLocation;
    }

    public void setCheckLocation(boolean checkLocation) {
        this.checkLocation = checkLocation;
    }

    public Duration getLoginTimeout() {
        return this.loginTimeout;
    }

    public void setLoginTimeout(Duration loginTimeout) {
        this.loginTimeout = loginTimeout;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import java.time.Duration

@ConfigurationProperties("acme")
class AcmeProperties(

    /**
     * Whether to check the location of acme resources.
     */
    var isCheckLocation: Boolean = true,

    /**
     * Timeout for establishing a connection to the acme server.
     */
    var loginTimeout:Duration = Duration.ofSeconds(3))
```

NOTE You should only use plain text with `@ConfigurationProperties` field Javadoc, since they are not processed before being added to the JSON.

Here are some rules we follow internally to make sure descriptions are consistent:

- Do not start the description by "The" or "A".
- For `boolean` types, start the description with "Whether" or "Enable".
- For collection-based types, start the description with "Comma-separated list"
- Use `java.time.Duration` rather than `long` and describe the default unit if it differs from milliseconds, such as "If a duration suffix is not specified, seconds will be used".
- Do not provide the default value in the description unless it has to be determined at runtime.

Make sure to trigger meta-data generation so that IDE assistance is available for your keys as well. You may want to review the generated metadata (`META-INF/spring-configuration-metadata.json`) to make sure your keys are properly documented. Using your own starter in a compatible IDE is also a good idea to validate that quality of the metadata.

The “autoconfigure” Module

The `autoconfigure` module contains everything that is necessary to get started with the library. It may also contain configuration key definitions (such as `@ConfigurationProperties`) and any callback interface that can be used to further customize how the components are initialized.

TIP You should mark the dependencies to the library as optional so that you can include the `autoconfigure` module in your projects more easily. If you do it that way, the library is not provided and, by default, Spring Boot backs off.

Spring Boot uses an annotation processor to collect the conditions on auto-configurations in a metadata file (`META-INF/spring-autoconfigure-metadata.properties`). If that file is present, it is used to eagerly filter auto-configurations that do not match, which will improve startup time.

When building with Maven, it is recommended to add the following dependency in a module that contains auto-configurations:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure-processor</artifactId>
    <optional>true</optional>
</dependency>
```

If you have defined auto-configurations directly in your application, make sure to configure the `spring-boot-maven-plugin` to prevent the `repackage` goal from adding the dependency into the uber jar:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.springframework.boot</groupId>
                            <artifactId>spring-boot-autoconfigure-
processor</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

With Gradle, the dependency should be declared in the `annotationProcessor` configuration, as shown in the following example:

```
dependencies {
    annotationProcessor "org.springframework.boot:spring-boot-autoconfigure-processor"
}
```

Starter Module

The starter is really an empty jar. Its only purpose is to provide the necessary dependencies to work with the library. You can think of it as an opinionated view of what is required to get started.

Do not make assumptions about the project in which your starter is added. If the library you are auto-configuring typically requires other starters, mention them as well. Providing a proper set of *default* dependencies may be hard if the number of optional dependencies is high, as you should avoid including dependencies that are unnecessary for a typical usage of the library. In other words, you should not include optional dependencies.

NOTE Either way, your starter must reference the core Spring Boot starter (`spring-boot-starter`) directly or indirectly (there is no need to add it if your starter relies on another starter). If a project is created with only your custom starter, Spring Boot's core features will be honoured by the presence of the core starter.

7.13. Kotlin Support

Kotlin is a statically-typed language targeting the JVM (and other platforms) which allows writing concise and elegant code while providing interoperability with existing libraries written in Java.

Spring Boot provides Kotlin support by leveraging the support in other Spring projects such as Spring Framework, Spring Data, and Reactor. See the Spring Framework Kotlin support documentation for more information.

The easiest way to start with Spring Boot and Kotlin is to follow this comprehensive tutorial. You can create new Kotlin projects by using start.spring.io. Feel free to join the #spring channel of Kotlin Slack or ask a question with the `spring` and `kotlin` tags on Stack Overflow if you need support.

7.13.1. Requirements

Spring Boot requires at least Kotlin 1.7.x and manages a suitable Kotlin version through dependency management. To use Kotlin, `org.jetbrains.kotlin:kotlin-stdlib` and `org.jetbrains.kotlin:kotlin-reflect` must be present on the classpath. The `kotlin-stdlib` variants `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` can also be used.

Since Kotlin classes are final by default, you are likely to want to configure kotlin-spring plugin in order to automatically open Spring-annotated classes so that they can be proxied.

Jackson's Kotlin module is required for serializing / deserializing JSON data in Kotlin. It is automatically registered when found on the classpath. A warning message is logged if Jackson and Kotlin are present but the Jackson Kotlin module is not.

TIP These dependencies and plugins are provided by default if one bootstraps a Kotlin project on start.spring.io.

7.13.2. Null-safety

One of Kotlin's key features is null-safety. It deals with `null` values at compile time rather than deferring the problem to runtime and encountering a `NullPointerException`. This helps to eliminate a common source of bugs without paying the cost of wrappers like `Optional`. Kotlin also allows using functional constructs with nullable values as described in this comprehensive guide to null-safety in Kotlin.

Although Java does not allow one to express null-safety in its type system, Spring Framework, Spring Data, and Reactor now provide null-safety of their API through tooling-friendly annotations. By default, types from Java APIs used in Kotlin are recognized as platform types for which null-checks are relaxed. Kotlin's support for JSR 305 annotations combined with nullability annotations provide null-safety for the related Spring API in Kotlin.

The JSR 305 checks can be configured by adding the `-Xjsr305` compiler flag with the following options: `-Xjsr305={strict|warn|ignore}`. The default behavior is the same as `-Xjsr305=warn`. The `strict` value is required to have null-safety taken in account in Kotlin types inferred from Spring API but should be used with the knowledge that Spring API nullability declaration could evolve even between minor releases and more checks may be added in the future).

WARNING Generic type arguments, varargs and array elements nullability are not yet supported. See SPR-15942 for up-to-date information. Also be aware that Spring Boot's own API is not yet annotated.

7.13.3. Kotlin API

runApplication

Spring Boot provides an idiomatic way to run an application with `runApplication<MyApplication>(*args)` as shown in the following example:

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

@SpringBootApplication
class MyApplication

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

This is a drop-in replacement for `SpringApplication.run(MyApplication::class.java, *args)`. It also allows customization of the application as shown in the following example:

```
runApplication<MyApplication>(*args) {
    setBannerMode(OFF)
}
```

Extensions

Kotlin extensions provide the ability to extend existing classes with additional functionality. The Spring Boot Kotlin API makes use of these extensions to add new Kotlin specific conveniences to existing APIs.

`TestRestTemplate` extensions, similar to those provided by Spring Framework for `RestOperations` in

Spring Framework, are provided. Among other things, the extensions make it possible to take advantage of Kotlin reified type parameters.

7.13.4. Dependency management

In order to avoid mixing different versions of Kotlin dependencies on the classpath, Spring Boot imports the Kotlin BOM.

With Maven, the Kotlin version can be customized by setting the `kotlin.version` property and plugin management is provided for `kotlin-maven-plugin`. With Gradle, the Spring Boot plugin automatically aligns the `kotlin.version` with the version of the Kotlin plugin.

Spring Boot also manages the version of Coroutines dependencies by importing the Kotlin Coroutines BOM. The version can be customized by setting the `kotlin-coroutines.version` property.

TIP `org.jetbrains.kotlinx:kotlinx-coroutines-reactor` dependency is provided by default if one bootstraps a Kotlin project with at least one reactive dependency on start.spring.io.

7.13.5. @ConfigurationProperties

`@ConfigurationProperties` when used in combination with constructor binding supports classes with immutable `val` properties as shown in the following example:

```
@ConfigurationProperties("example.kotlin")
data class KotlinExampleProperties(
        val name: String,
        val description: String,
        val myService: MyService) {

    data class MyService(
            val apiToken: String,
            val uri: URI
    )
}
```

TIP To generate your own metadata using the annotation processor, `kapt` should be configured with the `spring-boot-configuration-processor` dependency. Note that some features (such as detecting the default value or deprecated items) are not working due to limitations in the model kapt provides.

7.13.6. Testing

While it is possible to use JUnit 4 to test Kotlin code, JUnit 5 is provided by default and is recommended. JUnit 5 enables a test class to be instantiated once and reused for all of the class's tests. This makes it possible to use `@BeforeAll` and `@AfterAll` annotations on non-static methods, which is a good fit for Kotlin.

To mock Kotlin classes, MockK is recommended. If you need the `MockK` equivalent of the Mockito specific `@MockBean` and `@SpyBean` annotations, you can use SpringMockK which provides similar `@MockkBean` and `@SpykBean` annotations.

7.13.7. Resources

Further reading

- Kotlin language reference
- Kotlin Slack (with a dedicated #spring channel)
- Stack Overflow with `spring` and `kotlin` tags
- Try Kotlin in your browser
- Kotlin blog
- Awesome Kotlin
- Tutorial: building web applications with Spring Boot and Kotlin
- Developing Spring Boot applications with Kotlin
- A Geospatial Messenger with Kotlin, Spring Boot and PostgreSQL
- Introducing Kotlin support in Spring Framework 5.0
- Spring Framework 5 Kotlin APIs, the functional way

Examples

- spring-boot-kotlin-demo: regular Spring Boot + Spring Data JPA project
- mixit: Spring Boot 2 + WebFlux + Reactive Spring Data MongoDB
- spring-kotlin-fullstack: WebFlux Kotlin fullstack example with Kotlin2js for frontend instead of JavaScript or TypeScript
- spring-petclinic-kotlin: Kotlin version of the Spring PetClinic Sample Application
- spring-kotlin-deepdive: a step by step migration for Boot 1.0 + Java to Boot 2.0 + Kotlin
- spring-boot-coroutines-demo: Coroutines sample project

7.14. SSL

Spring Boot provides the ability to configure SSL trust material that can be applied to several types of connections in order to support secure communications. Configuration properties with the prefix `spring.ssl.bundle` can be used to specify named sets of trust material and associated information.

7.14.1. Configuring SSL With Java KeyStore Files

Configuration properties with the prefix `spring.ssl.bundle.jks` can be used to configure bundles of trust material created with the Java `keytool` utility and stored in Java KeyStore files in the JKS or PKCS12 format. Each bundle has a user-provided name that can be used to reference the bundle.

When used to secure an embedded web server, a `keystore` is typically configured with a Java

KeyStore containing a certificate and private key as shown in this example:

Properties

```
spring.ssl.bundle.jks.mybundle.key.alias=application
spring.ssl.bundle.jks.mybundle.keystore.location=classpath:application.p12
spring.ssl.bundle.jks.mybundle.keystore.password=secret
spring.ssl.bundle.jks.mybundle.keystore.type=PKCS12
```

Yaml

```
spring:
  ssl:
    bundle:
      jks:
        mybundle:
          key:
            alias: "application"
          keystore:
            location: "classpath:application.p12"
            password: "secret"
            type: "PKCS12"
```

When used to secure a client-side connection, a `truststore` is typically configured with a Java KeyStore containing the server certificate as shown in this example:

Properties

```
spring.ssl.bundle.jks.mybundle.truststore.location=classpath:server.p12
spring.ssl.bundle.jks.mybundle.truststore.password=secret
```

Yaml

```
spring:
  ssl:
    bundle:
      jks:
        mybundle:
          truststore:
            location: "classpath:server.p12"
            password: "secret"
```

See JksSslBundleProperties for the full set of supported properties.

7.14.2. Configuring SSL With PEM-encoded Certificates

Configuration properties with the prefix `spring.ssl.bundle.pem` can be used to configure bundles of trust material in the form of PEM-encoded text. Each bundle has a user-provided name that can be used to reference the bundle.

When used to secure an embedded web server, a `keystore` is typically configured with a certificate and private key as shown in this example:

Properties

```
spring.ssl.bundle.pem.mybundle.keystore.certificate=classpath:application.crt
spring.ssl.bundle.pem.mybundle.keystore.private-key=classpath:application.key
```

Yaml

```
spring:
  ssl:
    bundle:
      pem:
        mybundle:
          keystore:
            certificate: "classpath:application.crt"
            private-key: "classpath:application.key"
```

When used to secure a client-side connection, a `truststore` is typically configured with the server certificate as shown in this example:

Properties

```
spring.ssl.bundle.pem.mybundle.truststore.certificate=classpath:server.crt
```

Yaml

```
spring:
  ssl:
    bundle:
      pem:
        mybundle:
          truststore:
            certificate: "classpath:server.crt"
```

TIP

PEM content can be used directly for both the `certificate` and `private-key` properties. If the property values contains `BEGIN` and `END` markers then they will be treated as PEM content rather than a resource location.

The following example shows how a truststore certificate can be defined:

Properties

```
spring.ssl.bundle.pem.mybundle.truststore.certificate=\
-----BEGIN CERTIFICATE-----\n\
MIID1zCCAr+gAwIBAgIUNM5QQv8IzVQsgSmmdPQNaqyzWs4wDQYJKoZIhvcNAQEL\n\
BQAwezELMAkGA1UEBhMCWFgxEjAQBgNVBAgMCVN0YXRlTmFtZTERMA8GA1UEBwwI\n\
...\n\
V0IJjcmYjEZbTvpjFKznvaFiOUv+8L7jHQ1/Yf+9c3C8gSjdUfv88m17pqYXd+Ds\n\
HEmfmNNjht130UyjNCITmLVXyy5p35vWmdf95U3uEbJSnNVtXH8qRmN9oK9mUpDb\n\
ngX6JBJI7fw7tXoqWSLHNiBODM88fUlQSho8\n\
-----END CERTIFICATE-----\n
```

Yaml

```
spring:
  ssl:
    bundle:
      pem:
        mybundle:
          truststore:
            certificate: |
              -----BEGIN CERTIFICATE-----

MIID1zCCAr+gAwIBAgIUNM5QQv8IzVQsgSmmdPQNaqyzWs4wDQYJKoZIhvcNAQEL

BQAwezELMAkGA1UEBhMCWFgxEjAQBgNVBAgMCVN0YXRlTmFtZTERMA8GA1UEBwwI
              ...

V0IJjcmYjEZbTvpjFKznvaFiOUv+8L7jHQ1/Yf+9c3C8gSjdUfv88m17pqYXd+Ds

HEmfmNNjht130UyjNCITmLVXyy5p35vWmdf95U3uEbJSnNVtXH8qRmN9oK9mUpDb
              ngX6JBJI7fw7tXoqWSLHNiBODM88fUlQSho8
              -----END CERTIFICATE-----
```

See PemSslBundleProperties for the full set of supported properties.

7.14.3. Applying SSL Bundles

Once configured using properties, SSL bundles can be referred to by name in configuration properties for various types of connections that are auto-configured by Spring Boot. See the sections on embedded web servers, data technologies, and REST clients for further information.

7.14.4. Using SSL Bundles

Spring Boot auto-configures a bean of type `SslBundles` that provides access to each of the named bundles configured using the `spring.ssl.bundle` properties.

An `SslBundle` can be retrieved from the auto-configured `SslBundles` bean and used to create objects that are used to configure SSL connectivity in client libraries. The `SslBundle` provides a layered approach of obtaining these SSL objects:

- `getStores()` provides access to the key store and trust store `java.security.KeyStore` instances as well as any required key store password.
- `getManagers()` provides access to the `java.net.ssl.KeyManagerFactory` and `java.net.ssl.TrustManagerFactory` instances as well as the `java.net.ssl.KeyManager` and `java.net.ssl.TrustManager` arrays that they create.
- `createSslContext()` provides a convenient way to obtain a new `java.net.ssl.SSLContext` instance.

In addition, the `SslBundle` provides details about the key being used, the protocol to use and any option that should be applied to the SSL engine.

The following example shows retrieving an `SslBundle` and using it to create an `SSLContext`:

Java

```java
import javax.net.ssl.SSLContext;

import org.springframework.boot.ssl.SslBundle;
import org.springframework.boot.ssl.SslBundles;
import org.springframework.stereotype.Component;

@Component
public class MyComponent {

    public MyComponent(SslBundles sslBundles) {
        SslBundle sslBundle = sslBundles.getBundle("mybundle");
        SSLContext sslContext = sslBundle.createSslContext();
        // do something with the created sslContext
    }

}
```

Kotlin

```
import org.springframework.boot.ssl.SslBundles
import org.springframework.stereotype.Component

@Component
class MyComponent(sslBundles: SslBundles) {

    init {
        val sslBundle = sslBundles.getBundle("mybundle")
        val sslContext = sslBundle.createSslContext()
        // do something with the created sslContext
    }

}
```

7.14.5. Reloading SSL bundles

SSL bundles can be reloaded when the key material changes. The component consuming the bundle has to be compatible with reloadable SSL bundles. Currently the following components are compatible:

- Tomcat web server
- Netty web server

To enable reloading, you need to opt-in via a configuration property as shown in this example:

Properties

```
spring.ssl.bundle.pem.mybundle.reload-on-update=true
spring.ssl.bundle.pem.mybundle.keystore.certificate=file:/some/directory/application.c
rt
spring.ssl.bundle.pem.mybundle.keystore.private-
key=file:/some/directory/application.key
```

Yaml

```
spring:
  ssl:
    bundle:
      pem:
        mybundle:
          reload-on-update: true
          keystore:
            certificate: "file:/some/directory/application.crt"
            private-key: "file:/some/directory/application.key"
```

A file watcher is then watching the files and if they change, the SSL bundle will be reloaded. This in turn triggers a reload in the consuming component, e.g. Tomcat rotates the certificates in the SSL enabled connectors.

You can configure the quiet period (to make sure that there are no more changes) of the file watcher with the `spring.ssl.bundle.watch.file.quiet-period` property.

7.15. What to Read Next

If you want to learn more about any of the classes discussed in this section, see the Spring Boot API documentation or you can browse the source code directly. If you have specific questions, see the how-to section.

If you are comfortable with Spring Boot's core features, you can continue on and read about production-ready features.

Chapter 8. Web

Spring Boot is well suited for web application development. You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the `spring-boot-starter-web` module to get up and running quickly. You can also choose to build reactive web applications by using the `spring-boot-starter-webflux` module.

If you have not yet developed a Spring Boot web application, you can follow the "Hello World!" example in the *Getting started* section.

8.1. Servlet Web Applications

If you want to build servlet-based web applications, you can take advantage of Spring Boot's auto-configuration for Spring MVC or Jersey.

8.1.1. The “Spring Web MVC Framework”

The Spring Web MVC framework (often referred to as “Spring MVC”) is a rich “model view controller” web framework. Spring MVC lets you create special `@Controller` or `@RestController` beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using `@RequestMapping` annotations.

The following code shows a typical `@RestController` that serves JSON data:

Java

```
import java.util.List;

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class MyRestController {

    private final UserRepository userRepository;

    private final CustomerRepository customerRepository;

    public MyRestController(UserRepository userRepository, CustomerRepository
customerRepository) {
        this.userRepository = userRepository;
        this.customerRepository = customerRepository;
    }

    @GetMapping("/{userId}")
    public User getUser(@PathVariable Long userId) {
        return this.userRepository.findById(userId).get();
    }

    @GetMapping("/{userId}/customers")
    public List<Customer> getUserCustomers(@PathVariable Long userId) {
        return
this.userRepository.findById(userId).map(this.customerRepository::findByUser).get();
    }

    @DeleteMapping("/{userId}")
    public void deleteUser(@PathVariable Long userId) {
        this.userRepository.deleteById(userId);
    }

}
```

Kotlin

```kotlin
import org.springframework.web.bind.annotation.DeleteMapping
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController


@RestController
@RequestMapping("/users")
class MyRestController(private val userRepository: UserRepository, private val
customerRepository: CustomerRepository) {

    @GetMapping("/{userId}")
    fun getUser(@PathVariable userId: Long): User {
        return userRepository.findById(userId).get()
    }

    @GetMapping("/{userId}/customers")
    fun getUserCustomers(@PathVariable userId: Long): List<Customer> {
        return
userRepository.findById(userId).map(customerRepository::findByUser).get()
    }

    @DeleteMapping("/{userId}")
    fun deleteUser(@PathVariable userId: Long) {
        userRepository.deleteById(userId)
    }

}
```

“WebMvc.fn”, the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:

Java

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import org.springframework.web.servlet.function.RequestPredicate;
import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.ServerResponse;

import static org.springframework.web.servlet.function.RequestPredicates.accept;
import static org.springframework.web.servlet.function.RouterFunctions.route;

@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {

    private static final RequestPredicate ACCEPT_JSON =
accept(MediaType.APPLICATION_JSON);

    @Bean
    public RouterFunction<ServerResponse> routerFunction(MyUserHandler userHandler) {
        return route()
                .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
                .GET("/{user}/customers", ACCEPT_JSON, userHandler::getUserCustomers)
                .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
                .build();
    }

}
```

Kotlin

```kotlin
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.MediaType
import org.springframework.web.servlet.function.RequestPredicates.accept
import org.springframework.web.servlet.function.RouterFunction
import org.springframework.web.servlet.function.RouterFunctions
import org.springframework.web.servlet.function.ServerResponse

@Configuration(proxyBeanMethods = false)
class MyRoutingConfiguration {

    @Bean
    fun routerFunction(userHandler: MyUserHandler): RouterFunction<ServerResponse> {
        return RouterFunctions.route()
            .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
            .GET("/{user}/customers", ACCEPT_JSON, userHandler::getUserCustomers)
            .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
            .build()
    }

    companion object {
        private val ACCEPT_JSON = accept(MediaType.APPLICATION_JSON)
    }

}
```

Java

```java
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.function.ServerRequest;
import org.springframework.web.servlet.function.ServerResponse;

@Component
public class MyUserHandler {

    public ServerResponse getUser(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

    public ServerResponse getUserCustomers(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

    public ServerResponse deleteUser(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

}
```

Kotlin

```kotlin
import org.springframework.stereotype.Component
import org.springframework.web.servlet.function.ServerRequest
import org.springframework.web.servlet.function.ServerResponse

@Component
class MyUserHandler {

    fun getUser(request: ServerRequest?): ServerResponse {
        return ServerResponse.ok().build()
    }

    fun getUserCustomers(request: ServerRequest?): ServerResponse {
        return ServerResponse.ok().build()
    }

    fun deleteUser(request: ServerRequest?): ServerResponse {
        return ServerResponse.ok().build()
    }

}
```

Spring MVC is part of the core Spring Framework, and detailed information is available in the reference documentation. There are also several guides that cover Spring MVC available at spring.io/guides.

TIP You can define as many `RouterFunction` beans as you like to modularize the definition of the router. Beans can be ordered if you need to apply a precedence.

Spring MVC Auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications. It replaces the need for `@EnableWebMvc` and the two cannot be used together. In addition to Spring MVC's defaults, the auto-configuration provides the following features:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered later in this document).
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
- Support for `HttpMessageConverters` (covered later in this document).
- Automatic registration of `MessageCodesResolver` (covered later in this document).
- Static `index.html` support.
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered later in this document).

If you want to keep those Spring Boot MVC customizations and make more MVC customizations (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`.

If you want to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, and still keep the Spring Boot MVC customizations, you can declare a bean of type `WebMvcRegistrations` and use it to provide custom instances of those components. The custom instances will be subject to further initialization and configuration by Spring MVC. To participate in, and if desired, override that subsequent processing, a `WebMvcConfigurer` should be used.

If you do not want to use the auto-configuration and want to take complete control of Spring MVC, add your own `@Configuration` annotated with `@EnableWebMvc`. Alternatively, add your own `@Configuration`-annotated `DelegatingWebMvcConfiguration` as described in the Javadoc of `@EnableWebMvc`.

Spring MVC Conversion Service

Spring MVC uses a different `ConversionService` to the one used to convert values from your `application.properties` or `application.yaml` file. It means that `Period`, `Duration` and `DataSize` converters are not available and that `@DurationUnit` and `@DataSizeUnit` annotations will be ignored.

If you want to customize the `ConversionService` used by Spring MVC, you can provide a `WebMvcConfigurer` bean with an `addFormatters` method. From this method you can register any converter that you like, or you can delegate to the static methods available on `ApplicationConversionService`.

Conversion can also be customized using the `spring.mvc.format.*` configuration properties. When not configured, the following defaults are used:

Property	`DateTimeFormatter`
`spring.mvc.format.date`	`ofLocalizedDate(FormatStyle.SHORT)`
`spring.mvc.format.time`	`ofLocalizedTime(FormatStyle.SHORT)`
`spring.mvc.format.date-time`	`ofLocalizedDateTime(FormatStyle.SHORT)`

HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box. For example, objects can be automatically converted to JSON (by using the Jackson library) or XML (by using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in `UTF-8`.

Any `HttpMessageConverter` bean that is present in the context is added to the list of converters. You can also override default converters in the same way.

If you need to add or customize converters, you can use Spring Boot's `HttpMessageConverters` class, as shown in the following listing:

Java

```java
import org.springframework.boot.autoconfigure.http.HttpMessageConverters;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.converter.HttpMessageConverter;

@Configuration(proxyBeanMethods = false)
public class MyHttpMessageConvertersConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = new AdditionalHttpMessageConverter();
        HttpMessageConverter<?> another = new AnotherHttpMessageConverter();
        return new HttpMessageConverters(additional, another);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.http.HttpMessageConverters
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.converter.HttpMessageConverter

@Configuration(proxyBeanMethods = false)
class MyHttpMessageConvertersConfiguration {

    @Bean
    fun customConverters(): HttpMessageConverters {
        val additional: HttpMessageConverter<*> = AdditionalHttpMessageConverter()
        val another: HttpMessageConverter<*> = AnotherHttpMessageConverter()
        return HttpMessageConverters(additional, another)
    }

}
```

For further control, you can also sub-class `HttpMessageConverters` and override its `postProcessConverters` and/or `postProcessPartConverters` methods. This can be useful when you want to re-order or remove some of the converters that Spring MVC configures by default.

MessageCodesResolver

Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: `MessageCodesResolver`. If you set the `spring.mvc.message-codes-resolver-format` property `PREFIX_ERROR_CODE` or `POSTFIX_ERROR_CODE`, Spring Boot creates one for you (see the enumeration in `DefaultMessageCodesResolver.Format`).

Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath or from the root of the `ServletContext`. It uses the `ResourceHttpRequestHandler` from Spring MVC so that you can modify that behavior by adding your own `WebMvcConfigurer` and overriding the `addResourceHandlers` method.

In a stand-alone web application, the default servlet from the container is not enabled. It can be enabled using the `server.servlet.register-default-servlet` property.

The default servlet acts as a fallback, serving content from the root of the `ServletContext` if Spring decides not to handle it. Most of the time, this does not happen (unless you modify the default MVC configuration), because Spring can always handle requests through the `DispatcherServlet`.

By default, resources are mapped on `/**`, but you can tune that with the `spring.mvc.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

Properties

```
spring.mvc.static-path-pattern=/resources/**
```

Yaml

```
spring:
  mvc:
    static-path-pattern: "/resources/**"
```

You can also customize the static resource locations by using the `spring.web.resources.static-locations` property (replacing the default values with a list of directory locations). The root servlet context path, `"/"`, is automatically added as a location as well.

In addition to the “standard” static resource locations mentioned earlier, a special case is made for Webjars content. By default, any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format. The path can be customized with the `spring.mvc.webjars-path-pattern` property.

TIP Do not use the `src/main/webapp` directory if your application is packaged as a jar. Although this directory is a common standard, it works **only** with war packaging, and it is silently ignored by most build tools if you generate a jar.

Spring Boot also supports the advanced resource handling features provided by Spring MVC, allowing use cases such as cache-busting static resources or using version agnostic URLs for Webjars.

To use version agnostic URLs for Webjars, add the `webjars-locator-core` dependency. Then declare your Webjar. Using jQuery as an example, adding `"/webjars/jquery/jquery.min.js"` results in `"/webjars/jquery/x.y.z/jquery.min.js"` where `x.y.z` is the Webjar version.

NOTE If you use JBoss, you need to declare the `webjars-locator-jboss-vfs` dependency instead of the `webjars-locator-core`. Otherwise, all Webjars resolve as a `404`.

To use cache busting, the following configuration configures a cache busting solution for all static resources, effectively adding a content hash, such as `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`, in URLs:

Properties

```
spring.web.resources.chain.strategy.content.enabled=true
spring.web.resources.chain.strategy.content.paths=/**
```

Yaml

```
spring:
  web:
    resources:
      chain:
        strategy:
          content:
            enabled: true
            paths: "/**"
```

NOTE Links to resources are rewritten in templates at runtime, thanks to a `ResourceUrlEncodingFilter` that is auto-configured for Thymeleaf and FreeMarker. You should manually declare this filter when using JSPs. Other template engines are currently not automatically supported but can be with custom template macros/helpers and the use of the `ResourceUrlProvider`.

When loading resources dynamically with, for example, a JavaScript module loader, renaming files is not an option. That is why other strategies are also supported and can be combined. A "fixed" strategy adds a static version string in the URL without changing the file name, as shown in the following example:

Properties

```
spring.web.resources.chain.strategy.content.enabled=true
spring.web.resources.chain.strategy.content.paths=/**
spring.web.resources.chain.strategy.fixed.enabled=true
spring.web.resources.chain.strategy.fixed.paths=/js/lib/
spring.web.resources.chain.strategy.fixed.version=v12
```

Yaml

```
spring:
  web:
    resources:
      chain:
        strategy:
          content:
            enabled: true
            paths: "/**"
          fixed:
            enabled: true
            paths: "/js/lib/"
            version: "v12"
```

With this configuration, JavaScript modules located under `"/js/lib/"` use a fixed versioning strategy (`"/v12/js/lib/mymodule.js"`), while other resources still use the content one (`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`).

See `WebProperties.Resources` for more supported options.

TIP This feature has been thoroughly described in a dedicated blog post and in Spring Framework's reference documentation.

Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

This only acts as a fallback for actual index routes defined by the application. The ordering is defined by the order of `HandlerMapping` beans which is by default the following:

`RouterFunctionMapping`	Endpoints declared with `RouterFunction` beans
`RequestMappingHandlerMapping`	Endpoints declared in `@Controller` beans
`WelcomePageHandlerMapping`	The welcome page support

Custom Favicon

As with other static resources, Spring Boot checks for a `favicon.ico` in the configured static content locations. If such a file is present, it is automatically used as the favicon of the application.

Path Matching and Content Negotiation

Spring MVC can map incoming HTTP requests to handlers by looking at the request path and matching it to the mappings defined in your application (for example, `@GetMapping` annotations on Controller methods).

Spring Boot chooses to disable suffix pattern matching by default, which means that requests like `"GET /projects/spring-boot.json"` will not be matched to `@GetMapping("/projects/spring-boot")` mappings. This is considered as a best practice for Spring MVC applications. This feature was mainly useful in the past for HTTP clients which did not send proper "Accept" request headers; we needed to make sure to send the correct Content Type to the client. Nowadays, Content Negotiation is much more reliable.

There are other ways to deal with HTTP clients that do not consistently send proper "Accept" request headers. Instead of using suffix matching, we can use a query parameter to ensure that requests like `"GET /projects/spring-boot?format=json"` will be mapped to `@GetMapping("/projects/spring-boot")`:

Properties

```
spring.mvc.contentnegotiation.favor-parameter=true
```

Yaml

```
spring:
  mvc:
    contentnegotiation:
      favor-parameter: true
```

Or if you prefer to use a different parameter name:

Properties

```
spring.mvc.contentnegotiation.favor-parameter=true
spring.mvc.contentnegotiation.parameter-name=myparam
```

Yaml

```
spring:
  mvc:
    contentnegotiation:
      favor-parameter: true
      parameter-name: "myparam"
```

Most standard media types are supported out-of-the-box, but you can also define new ones:

Properties

```
spring.mvc.contentnegotiation.media-types.markdown=text/markdown
```

Yaml

```
spring:
  mvc:
    contentnegotiation:
      media-types:
        markdown: "text/markdown"
```

As of Spring Framework 5.3, Spring MVC supports two strategies for matching request paths to controllers. By default, Spring Boot uses the `PathPatternParser` strategy. `PathPatternParser` is an optimized implementation but comes with some restrictions compared to the `AntPathMatcher` strategy. `PathPatternParser` restricts usage of some path pattern variants. It is also incompatible with configuring the `DispatcherServlet` with a path prefix (`spring.mvc.servlet.path`).

The strategy can be configured using the `spring.mvc.pathmatch.matching-strategy` configuration property, as shown in the following example:

Properties

```
spring.mvc.pathmatch.matching-strategy=ant-path-matcher
```

Yaml

```
spring:
  mvc:
    pathmatch:
      matching-strategy: "ant-path-matcher"
```

By default, Spring MVC will send a 404 Not Found error response if a handler is not found for a request. To have a `NoHandlerFoundException` thrown instead, set configprop:spring.mvc.throw-exception-if-no-handler-found to `true`. Note that, by default, the serving of static content is mapped to `/**` and will, therefore, provide a handler for all requests. For a `NoHandlerFoundException` to be thrown, you must also set `spring.mvc.static-path-pattern` to a more specific value such as `/resources/**` or set `spring.web.resources.add-mappings` to `false` to disable serving of static content entirely.

ConfigurableWebBindingInitializer

Spring MVC uses a `WebBindingInitializer` to initialize a `WebDataBinder` for a particular request. If you create your own `ConfigurableWebBindingInitializer` `@Bean`, Spring Boot automatically configures Spring MVC to use it.

Template Engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

TIP If possible, JSPs should be avoided. There are several known limitations when using them with embedded servlet containers.

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

TIP Depending on how you run your application, your IDE may order the classpath differently. Running your application in the IDE from its main method results in a different ordering than when you run your application by using Maven or Gradle or from its packaged jar. This can cause Spring Boot to fail to find the expected template. If you have this problem, you can reorder the classpath in the IDE to place the module's classes and resources first.

Error Handling

By default, Spring Boot provides an `/error` mapping that handles all errors in a sensible way, and it is registered as a "global" error page in the servlet container. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a "whitelabel" error view that renders the same data in HTML format (to customize it, add a `View` that resolves to `error`).

There are a number of `server.error` properties that can be set if you want to customize the default error handling behavior. See the "Server Properties" section of the Appendix.

To replace the default behavior completely, you can implement `ErrorController` and register a bean definition of that type or add a bean of type `ErrorAttributes` to use the existing mechanism but replace the contents.

TIP The `BasicErrorController` can be used as a base class for a custom `ErrorController`. This is particularly useful if you want to add a handler for a new content type (the default is to handle `text/html` specifically and provide a fallback for everything else). To do so, extend `BasicErrorController`, add a public method with a `@RequestMapping` that has a `produces` attribute, and create a bean of your new type.

As of Spring Framework 6.0, RFC 7807 Problem Details is supported. Spring MVC can produce custom error messages with the `application/problem+json` media type, like:

```
{
  "type": "https://example.org/problems/unknown-project",
  "title": "Unknown project",
  "status": 404,
  "detail": "No project found for id 'spring-unknown'",
  "instance": "/projects/spring-unknown"
}
```

This support can be enabled by setting `spring.mvc.problemdetails.enabled` to `true`.

You can also define a class annotated with `@ControllerAdvice` to customize the JSON document to return for a particular controller and/or exception type, as shown in the following example: