Part IV. Spring Boot features

This section dives into the details of Spring Boot. Here you can learn about the key features that you may want to use and customize. If you have not already done so, you might want to read the "Part II, “Getting Started”" and "Part III, “Using Spring Boot”" sections, so that you have a good grounding of the basics.

23. SpringApplication

The `SpringApplication` class provides a convenient way to bootstrap a Spring application that is started from a `main()` method. In many situations, you can delegate to the static `SpringApplication.run` method, as shown in the following example:

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

When your application starts, you should see something similar to the following output:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v2.1.0.BUILD-SNAPSHOT

2013-07-31 00:08:16.117  INFO 56603 --- [           main] o.s.b.s.app.SampleApplication            :
 Starting SampleApplication v0.1.0 on mycomputer with PID 56603 (/apps/myapp.jar started by pwebb)
2013-07-31 00:08:16.166  INFO 56603 --- [           main]
 ationConfigServletWebServerApplicationContext : Refreshing
 org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@6e5a8246:
 startup date [Wed Jul 31 00:08:16 PDT 2013]; root of context hierarchy
2014-03-04 13:09:54.912  INFO 41370 --- [           main] .t.TomcatServletWebServerFactory : Server
 initialized with port: 8080
2014-03-04 13:09:56.501  INFO 41370 --- [           main] o.s.b.s.app.SampleApplication            :
 Started SampleApplication in 2.992 seconds (JVM running for 3.658)
```

By default, `INFO` logging messages are shown, including some relevant startup details, such as the user that launched the application. If you need a log level other than `INFO`, you can set it, as described in Section 26.4, “Log Levels”,

23.1 Startup Failure

If your application fails to start, registered `FailureAnalyzers` get a chance to provide a dedicated error message and a concrete action to fix the problem. For instance, if you start a web application on port `8080` and that port is already in use, you should see something similar to the following message:

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Embedded servlet container failed to start. Port 8080 was already in use.

Action:

Identify and stop the process that's listening on port 8080 or configure this application to listen on
 another port.
```

> **Note**
>
> Spring Boot provides numerous `FailureAnalyzer` implementations, and you can add your own.

If no failure analyzers are able to handle the exception, you can still display the full conditions report to better understand what went wrong. To do

so, you need to enable the `debug` property or enable `DEBUG` logging for `org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener`.

For instance, if you are running your application by using `java -jar`, you can enable the `debug` property as follows:

```
$ java -jar myproject-0.0.1-SNAPSHOT.jar --debug
```

23.2 Customizing the Banner

The banner that is printed on start up can be changed by adding a `banner.txt` file to your classpath or by setting the `spring.banner.location` property to the location of such a file. If the file has an encoding other than UTF-8, you can set `spring.banner.charset`. In addition to a text file, you can also add a `banner.gif`, `banner.jpg`, or `banner.png` image file to your classpath or set the `spring.banner.image.location` property. Images are converted into an ASCII art representation and printed above any text banner.

Inside your `banner.txt` file, you can use any of the following placeholders:

Table 23.1. Banner variables

Variable	Description
`${application.version}`	The version number of your application, as declared in `MANIFEST.MF`. For example, `Implementation-Version: 1.0` is printed as `1.0`.
`${application.formatted-version}`	The version number of your application, as declared in `MANIFEST.MF` and formatted for display (surrounded with brackets and prefixed with `v`). For example `(v1.0)`.
`${spring-boot.version}`	The Spring Boot version that you are using. For example `2.1.0.BUILD-SNAPSHOT`.
`${spring-boot.formatted-version}`	The Spring Boot version that you are using, formatted for display (surrounded with brackets and prefixed with `v`). For example `(v2.1.0.BUILD-SNAPSHOT)`.
`${Ansi.NAME}` (or `${AnsiColor.NAME}`, `${AnsiBackground.NAME}`, `${AnsiStyle.NAME}`)	Where `NAME` is the name of an ANSI escape code. See `AnsiPropertySource` for details.
`${application.title}`	The title of your application, as declared in `MANIFEST.MF`. For example `Implementation-Title: MyApp` is printed as `MyApp`.

Tip

The `SpringApplication.setBanner(…)` method can be used if you want to generate a banner programmatically. Use the `org.springframework.boot.Banner` interface and implement your own `printBanner()` method.

You can also use the `spring.main.banner-mode` property to determine if the banner has to be printed on `System.out` (`console`), sent to the configured logger (`log`), or not produced at all (`off`).

The printed banner is registered as a singleton bean under the following name: `springBootBanner`.

> **Note**
>
> YAML maps `off` to `false`, so be sure to add quotes if you want to disable the banner in your application, as shown in the following example:

```
spring:
 main:
  banner-mode: "off"
```

23.3 Customizing SpringApplication

If the `SpringApplication` defaults are not to your taste, you can instead create a local instance and customize it. For example, to turn off the banner, you could write:

```
public static void main(String[] args) {
 SpringApplication app = new SpringApplication(MySpringConfiguration.class);
 app.setBannerMode(Banner.Mode.OFF);
 app.run(args);
}
```

> **Note**
>
> The constructor arguments passed to `SpringApplication` are configuration sources for Spring beans. In most cases, these are references to `@Configuration` classes, but they could also be references to XML configuration or to packages that should be scanned.

It is also possible to configure the `SpringApplication` by using an `application.properties` file. See *Chapter 24, Externalized Configuration* for details.

For a complete list of the configuration options, see the `SpringApplication` Javadoc.

23.4 Fluent Builder API

If you need to build an `ApplicationContext` hierarchy (multiple contexts with a parent/child relationship) or if you prefer using a “fluent” builder API, you can use the `SpringApplicationBuilder`.

The `SpringApplicationBuilder` lets you chain together multiple method calls and includes `parent` and `child` methods that let you create a hierarchy, as shown in the following example:

```
new SpringApplicationBuilder()
  .sources(Parent.class)
  .child(Application.class)
  .bannerMode(Banner.Mode.OFF)
  .run(args);
```

> **Note**
>
> There are some restrictions when creating an `ApplicationContext` hierarchy. For example, Web components **must** be contained within the child context, and the same `Environment` is

used for both parent and child contexts. See the `SpringApplicationBuilder` Javadoc for full details.

23.5 Application Events and Listeners

In addition to the usual Spring Framework events, such as `ContextRefreshedEvent`, a `SpringApplication` sends some additional application events.

Note

Some events are actually triggered before the `ApplicationContext` is created, so you cannot register a listener on those as a `@Bean`. You can register them with the `SpringApplication.addListeners(…)` method or the `SpringApplicationBuilder.listeners(…)` method.

If you want those listeners to be registered automatically, regardless of the way the application is created, you can add a `META-INF/spring.factories` file to your project and reference your listener(s) by using the `org.springframework.context.ApplicationListener` key, as shown in the following example:

```
org.springframework.context.ApplicationListener=com.example.project.MyListener
```

Application events are sent in the following order, as your application runs:

1. An `ApplicationStartingEvent` is sent at the start of a run but before any processing, except for the registration of listeners and initializers.
2. An `ApplicationEnvironmentPreparedEvent` is sent when the `Environment` to be used in the context is known but before the context is created.
3. An `ApplicationPreparedEvent` is sent just before the refresh is started but after bean definitions have been loaded.
4. An `ApplicationStartedEvent` is sent after the context has been refreshed but before any application and command-line runners have been called.
5. An `ApplicationReadyEvent` is sent after any application and command-line runners have been called. It indicates that the application is ready to service requests.
6. An `ApplicationFailedEvent` is sent if there is an exception on startup.

Tip

You often need not use application events, but it can be handy to know that they exist. Internally, Spring Boot uses events to handle a variety of tasks.

Application events are sent by using Spring Framework's event publishing mechanism. Part of this mechanism ensures that an event published to the listeners in a child context is also published to the listeners in any ancestor contexts. As a result of this, if your application uses a hierarchy of `SpringApplication` instances, a listener may receive multiple instances of the same type of application event.

To allow your listener to distinguish between an event for its context and an event for a descendant context, it should request that its application context is injected and then compare the injected context with the context of the event. The context can be injected by implementing `ApplicationContextAware` or, if the listener is a bean, by using `@Autowired`.

23.6 Web Environment

A `SpringApplication` attempts to create the right type of `ApplicationContext` on your behalf. The algorithm used to determine a `WebApplicationType` is fairly simple:

- If Spring MVC is present, an `AnnotationConfigServletWebServerApplicationContext` is used
- If Spring MVC is not present and Spring WebFlux is present, an `AnnotationConfigReactiveWebServerApplicationContext` is used
- Otherwise, `AnnotationConfigApplicationContext` is used

This means that if you are using Spring MVC and the new `WebClient` from Spring WebFlux in the same application, Spring MVC will be used by default. You can override that easily by calling `setWebApplicationType(WebApplicationType)`.

It is also possible to take complete control of the `ApplicationContext` type that is used by calling `setApplicationContextClass(…)`.

> **Tip**
>
> It is often desirable to call `setWebApplicationType(WebApplicationType.NONE)` when using `SpringApplication` within a JUnit test.

23.7 Accessing Application Arguments

If you need to access the application arguments that were passed to `SpringApplication.run(…)`, you can inject a `org.springframework.boot.ApplicationArguments` bean. The `ApplicationArguments` interface provides access to both the raw `String[]` arguments as well as parsed `option` and `non-option` arguments, as shown in the following example:

```
import org.springframework.boot.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.stereotype.*;

@Component
public class MyBean {

 @Autowired
 public MyBean(ApplicationArguments args) {
  boolean debug = args.containsOption("debug");
  List<String> files = args.getNonOptionArgs();
  // if run with "--debug logfile.txt" debug=true, files=["logfile.txt"]
 }

}
```

> **Tip**
>
> Spring Boot also registers a `CommandLinePropertySource` with the Spring `Environment`. This lets you also inject single application arguments by using the `@Value` annotation.

23.8 Using the ApplicationRunner or CommandLineRunner

If you need to run some specific code once the `SpringApplication` has started, you can implement the `ApplicationRunner` or `CommandLineRunner` interfaces. Both interfaces work in the same way and offer a single `run` method, which is called just before `SpringApplication.run(…)` completes.

The `CommandLineRunner` interfaces provides access to application arguments as a simple string array, whereas the `ApplicationRunner` uses the `ApplicationArguments` interface discussed earlier. The following example shows a `CommandLineRunner` with a `run` method:

```
import org.springframework.boot.*;
import org.springframework.stereotype.*;

@Component
public class MyBean implements CommandLineRunner {

 public void run(String... args) {
  // Do something...
 }

}
```

If several `CommandLineRunner` or `ApplicationRunner` beans are defined that must be called in a specific order, you can additionally implement the `org.springframework.core.Ordered` interface or use the `org.springframework.core.annotation.Order` annotation.

23.9 Application Exit

Each `SpringApplication` registers a shutdown hook with the JVM to ensure that the `ApplicationContext` closes gracefully on exit. All the standard Spring lifecycle callbacks (such as the `DisposableBean` interface or the `@PreDestroy` annotation) can be used.

In addition, beans may implement the `org.springframework.boot.ExitCodeGenerator` interface if they wish to return a specific exit code when `SpringApplication.exit()` is called. This exit code can then be passed to `System.exit()` to return it as a status code, as shown in the following example:

```
@SpringBootApplication
public class ExitCodeApplication {

 @Bean
 public ExitCodeGenerator exitCodeGenerator() {
  return () -> 42;
 }

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

}
```

Also, the `ExitCodeGenerator` interface may be implemented by exceptions. When such an exception is encountered, Spring Boot returns the exit code provided by the implemented `getExitCode()` method.

23.10 Admin Features

It is possible to enable admin-related features for the application by specifying the `spring.application.admin.enabled` property. This exposes the `SpringApplicationAdminMXBean` on the platform `MBeanServer`. You could use this feature to administer your Spring Boot application remotely. This feature could also be useful for any service wrapper implementation.

> **Tip**
>
> If you want to know on which HTTP port the application is running, get the property with a key of `local.server.port`.

> **Caution**
>
> Take care when enabling this feature, as the MBean exposes a method to shutdown the application.

24. Externalized Configuration

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use properties files, YAML files, environment variables, and command-line arguments to externalize configuration. Property values can be injected directly into your beans by using the `@Value` annotation, accessed through Spring's `Environment` abstraction, or be bound to structured objects through `@ConfigurationProperties`.

Spring Boot uses a very particular `PropertySource` order that is designed to allow sensible overriding of values. Properties are considered in the following order:

1. Devtools global settings properties on your home directory (`~/.spring-boot-devtools.properties` when devtools is active).
2. `@TestPropertySource` annotations on your tests.
3. `@SpringBootTest#properties` annotation attribute on your tests.
4. Command line arguments.
5. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
6. `ServletConfig` init parameters.
7. `ServletContext` init parameters.
8. JNDI attributes from `java:comp/env`.
9. Java System properties (`System.getProperties()`).
10. OS environment variables.
11. A `RandomValuePropertySource` that has properties only in `random.*`.
12. Profile-specific application properties outside of your packaged jar (`application-{profile}.properties` and YAML variants).
13. Profile-specific application properties packaged inside your jar (`application-{profile}.properties` and YAML variants).
14. Application properties outside of your packaged jar (`application.properties` and YAML variants).
15. Application properties packaged inside your jar (`application.properties` and YAML variants).
16. `@PropertySource` annotations on your `@Configuration` classes.
17. Default properties (specified by setting `SpringApplication.setDefaultProperties`).

To provide a concrete example, suppose you develop a `@Component` that uses a `name` property, as shown in the following example:

```
import org.springframework.stereotype.*;
import org.springframework.beans.factory.annotation.*;
```

```
@Component
public class MyBean {

    @Value("${name}")
    private String name;

    // ...

}
```

On your application classpath (for example, inside your jar) you can have an `application.properties` file that provides a sensible default property value for `name`. When running in a new environment, an `application.properties` file can be provided outside of your jar that overrides the `name`. For one-off testing, you can launch with a specific command line switch (for example, `java -jar app.jar --name="Spring"`).

> **Tip**
>
> The `SPRING_APPLICATION_JSON` properties can be supplied on the command line with an environment variable. For example, you could use the following line in a UN*X shell:
>
> ```
> $ SPRING_APPLICATION_JSON='{"acme":{"name":"test"}}' java -jar myapp.jar
> ```
>
> In the preceding example, you end up with `acme.name=test` in the Spring `Environment`. You can also supply the JSON as `spring.application.json` in a System property, as shown in the following example:
>
> ```
> $ java -Dspring.application.json='{"name":"test"}' -jar myapp.jar
> ```
>
> You can also supply the JSON by using a command line argument, as shown in the following example:
>
> ```
> $ java -jar myapp.jar --spring.application.json='{"name":"test"}'
> ```
>
> You can also supply the JSON as a JNDI variable, as follows: `java:comp/env/spring.application.json`.

24.1 Configuring Random Values

The `RandomValuePropertySource` is useful for injecting random values (for example, into secrets or test cases). It can produce integers, longs, uuids, or strings, as shown in the following example:

```
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.uuid=${random.uuid}
my.number.less.than.ten=${random.int(10)}
my.number.in.range=${random.int[1024,65536]}
```

The `random.int*` syntax is `OPEN value (,max) CLOSE` where the `OPEN,CLOSE` are any character and `value,max` are integers. If `max` is provided, then `value` is the minimum value and `max` is the maximum value (exclusive).

24.2 Accessing Command Line Properties

By default, `SpringApplication` converts any command line option arguments (that is, arguments starting with `--`, such as `--server.port=9000`) to a `property` and adds them to the Spring

`Environment`. As mentioned previously, command line properties always take precedence over other property sources.

If you do not want command line properties to be added to the `Environment`, you can disable them by using `SpringApplication.setAddCommandLineProperties(false)`.

24.3 Application Property Files

`SpringApplication` loads properties from `application.properties` files in the following locations and adds them to the Spring `Environment`:

1. A `/config` subdirectory of the current directory
2. The current directory
3. A classpath `/config` package
4. The classpath root

The list is ordered by precedence (properties defined in locations higher in the list override those defined in lower locations).

> **Note**
>
> You can also use YAML ('.yml') files as an alternative to '.properties'.

If you do not like `application.properties` as the configuration file name, you can switch to another file name by specifying a `spring.config.name` environment property. You can also refer to an explicit location by using the `spring.config.location` environment property (which is a comma-separated list of directory locations or file paths). The following example shows how to specify a different file name:

```
$ java -jar myproject.jar --spring.config.name=myproject
```

The following example shows how to specify two locations:

```
$ java -jar myproject.jar --spring.config.location=classpath:/default.properties,classpath:/
override.properties
```

> **Warning**
>
> `spring.config.name` and `spring.config.location` are used very early to determine which files have to be loaded, so they must be defined as an environment property (typically an OS environment variable, a system property, or a command-line argument).

If `spring.config.location` contains directories (as opposed to files), they should end in `/` (and, at runtime, be appended with the names generated from `spring.config.name` before being loaded, including profile-specific file names). Files specified in `spring.config.location` are used as-is, with no support for profile-specific variants, and are overridden by any profile-specific properties.

Config locations are searched in reverse order. By default, the configured locations are `classpath:/,classpath:/config/,file:./,file:./config/`. The resulting search order is the following:

1. `file:./config/`
2. `file:./`

3. `classpath:/config/`

4. `classpath:/`

When custom config locations are configured by using `spring.config.location`, they replace the default locations. For example, if `spring.config.location` is configured with the value `classpath:/custom-config/,file:./custom-config/`, the search order becomes the following:

1. `file:./custom-config/`

2. `classpath:custom-config/`

Alternatively, when custom config locations are configured by using `spring.config.additional-location`, they are used in addition to the default locations. Additional locations are searched before the default locations. For example, if additional locations of `classpath:/custom-config/,file:./custom-config/` are configured, the search order becomes the following:

1. `file:./custom-config/`

2. `classpath:custom-config/`

3. `file:./config/`

4. `file:./`

5. `classpath:/config/`

6. `classpath:/`

This search ordering lets you specify default values in one configuration file and then selectively override those values in another. You can provide default values for your application in `application.properties` (or whatever other basename you choose with `spring.config.name`) in one of the default locations. These default values can then be overridden at runtime with a different file located in one of the custom locations.

> **Note**
>
> If you use environment variables rather than system properties, most operating systems disallow period-separated key names, but you can use underscores instead (for example, `SPRING_CONFIG_NAME` instead of `spring.config.name`).

> **Note**
>
> If your application runs in a container, then JNDI properties (in `java:comp/env`) or servlet context initialization parameters can be used instead of, or as well as, environment variables or system properties.

24.4 Profile-specific Properties

In addition to `application.properties` files, profile-specific properties can also be defined by using the following naming convention: `application-{profile}.properties`. The `Environment` has a set of default profiles (by default, `[default]`) that are used if no active profiles are set. In other words, if no profiles are explicitly activated, then properties from `application-default.properties` are loaded.

Profile-specific properties are loaded from the same locations as standard `application.properties`, with profile-specific files always overriding the non-specific ones, whether or not the profile-specific files are inside or outside your packaged jar.

If several profiles are specified, a last-wins strategy applies. For example, profiles specified by the `spring.profiles.active` property are added after those configured through the `SpringApplication` API and therefore take precedence.

> **Note**
>
> If you have specified any files in `spring.config.location`, profile-specific variants of those files are not considered. Use directories in `spring.config.location` if you want to also use profile-specific properties.

24.5 Placeholders in Properties

The values in `application.properties` are filtered through the existing `Environment` when they are used, so you can refer back to previously defined values (for example, from System properties).

```
app.name=MyApp
app.description=${app.name} is a Spring Boot application
```

> **Tip**
>
> You can also use this technique to create “short” variants of existing Spring Boot properties. See the *Section 75.4, “Use ‘Short’ Command Line Arguments”* how-to for details.

24.6 Using YAML Instead of Properties

YAML is a superset of JSON and, as such, is a convenient format for specifying hierarchical configuration data. The `SpringApplication` class automatically supports YAML as an alternative to properties whenever you have the SnakeYAML library on your classpath.

> **Note**
>
> If you use “Starters”, SnakeYAML is automatically provided by `spring-boot-starter`.

Loading YAML

Spring Framework provides two convenient classes that can be used to load YAML documents. The `YamlPropertiesFactoryBean` loads YAML as `Properties` and the `YamlMapFactoryBean` loads YAML as a `Map`.

For example, consider the following YAML document:

```
environments:
 dev:
  url: http://dev.example.com
  name: Developer Setup
 prod:
  url: http://another.example.com
  name: My Cool App
```

The preceding example would be transformed into the following properties:

```
environments.dev.url=http://dev.example.com
environments.dev.name=Developer Setup
environments.prod.url=http://another.example.com
environments.prod.name=My Cool App
```

YAML lists are represented as property keys with `[index]` dereferencers. For example, consider the following YAML:

```
my:
servers:
 - dev.example.com
 - another.example.com
```

The preceding example would be transformed into these properties:

```
my.servers[0]=dev.example.com
my.servers[1]=another.example.com
```

To bind to properties like that by using Spring Boot's `Binder` utilities (which is what `@ConfigurationProperties` does), you need to have a property in the target bean of type `java.util.List` (or `Set`) and you either need to provide a setter or initialize it with a mutable value. For example, the following example binds to the properties shown previously:

```
@ConfigurationProperties(prefix="my")
public class Config {

 private List<String> servers = new ArrayList<String>();

 public List<String> getServers() {
  return this.servers;
 }
}
```

Exposing YAML as Properties in the Spring Environment

The `YamlPropertySourceLoader` class can be used to expose YAML as a `PropertySource` in the Spring `Environment`. Doing so lets you use the `@Value` annotation with placeholders syntax to access YAML properties.

Multi-profile YAML Documents

You can specify multiple profile-specific YAML documents in a single file by using a `spring.profiles` key to indicate when the document applies, as shown in the following example:

```
server:
 address: 192.168.1.100
---
spring:
 profiles: development
server:
 address: 127.0.0.1
---
spring:
 profiles: production & eu-central
server:
 address: 192.168.1.120
```

In the preceding example, if the `development` profile is active, the `server.address` property is `127.0.0.1`. Similarly, if the `production` **and** `eu-central` profiles are active, the `server.address` property is `192.168.1.120`. If the `development`, `production` and `eu-central` profiles are **not** enabled, then the value for the property is `192.168.1.100`.

> **Note**
>
> `spring.profiles` can therefore contain a simple profile name (for example `production`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example `production & (eu-central | eu-west)`. Check the reference guide for more details.

If none are explicitly active when the application context starts, the default profiles are activated. So, in the following YAML, we set a value for `spring.security.user.password` that is available **only** in the "default" profile:

```
server:
  port: 8000
---
spring:
  profiles: default
  security:
    user:
      password: weak
```

Whereas, in the following example, the password is always set because it is not attached to any profile, and it would have to be explicitly reset in all other profiles as necessary:

```
server:
  port: 8000
spring:
  security:
    user:
      password: weak
```

Spring profiles designated by using the `spring.profiles` element may optionally be negated by using the `!` character. If both negated and non-negated profiles are specified for a single document, at least one non-negated profile must match, and no negated profiles may match.

YAML Shortcomings

YAML files cannot be loaded by using the `@PropertySource` annotation. So, in the case that you need to load values that way, you need to use a properties file.

24.7 Type-safe Configuration Properties

Using the `@Value("${property}")` annotation to inject configuration properties can sometimes be cumbersome, especially if you are working with multiple properties or your data is hierarchical in nature. Spring Boot provides an alternative method of working with properties that lets strongly typed beans govern and validate the configuration of your application, as shown in the following example:

```
package com.example;

import java.net.InetAddress;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

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

@ConfigurationProperties("acme")
public class AcmeProperties {

 private boolean enabled;
```

```
 private InetAddress remoteAddress;

 private final Security security = new Security();

 public boolean isEnabled() { ... }

 public void setEnabled(boolean enabled) { ... }

 public InetAddress getRemoteAddress() { ... }

 public void setRemoteAddress(InetAddress remoteAddress) { ... }

 public Security getSecurity() { ... }

 public static class Security {

  private String username;

  private String password;

  private List<String> roles = new ArrayList<>(Collections.singleton("USER"));

  public String getUsername() { ... }

  public void setUsername(String username) { ... }

  public String getPassword() { ... }

  public void setPassword(String password) { ... }

  public List<String> getRoles() { ... }

  public void setRoles(List<String> roles) { ... }

 }
}
```

The preceding POJO defines the following properties:

- `acme.enabled`, with a value of `false` by default.
- `acme.remote-address`, with a type that can be coerced from `String`.
- `acme.security.username`, with a nested "security" object whose name is determined by the name of the property. In particular, the return type is not used at all there and could have been `SecurityProperties`.
- `acme.security.password`.
- `acme.security.roles`, with a collection of `String`.

> **Note**
>
> Getters and setters are usually mandatory, since binding is through standard Java Beans property descriptors, just like in Spring MVC. A setter may be omitted in the following cases:
>
> - Maps, as long as they are initialized, need a getter but not necessarily a setter, since they can be mutated by the binder.
> - Collections and arrays can be accessed either through an index (typically with YAML) or by using a single comma-separated value (properties). In the latter case, a setter is mandatory. We recommend to always add a setter for such types. If you initialize a collection, make sure it is not immutable (as in the preceding example).

- If nested POJO properties are initialized (like the `Security` field in the preceding example), a setter is not required. If you want the binder to create the instance on the fly by using its default constructor, you need a setter.

Some people use Project Lombok to add getters and setters automatically. Make sure that Lombok does not generate any particular constructor for such a type, as it is used automatically by the container to instantiate the object.

Finally, only standard Java Bean properties are considered and binding on static properties is not supported.

Tip

See also the differences between `@Value` and `@ConfigurationProperties`.

You also need to list the properties classes to register in the `@EnableConfigurationProperties` annotation, as shown in the following example:

```
@Configuration
@EnableConfigurationProperties(AcmeProperties.class)
public class MyConfiguration {
}
```

Note

When the `@ConfigurationProperties` bean is registered that way, the bean has a conventional name: `<prefix>-<fqn>`, where `<prefix>` is the environment key prefix specified in the `@ConfigurationProperties` annotation and `<fqn>` is the fully qualified name of the bean. If the annotation does not provide any prefix, only the fully qualified name of the bean is used.

The bean name in the example above is `acme-com.example.AcmeProperties`.

Even if the preceding configuration creates a regular bean for `AcmeProperties`, we recommend that `@ConfigurationProperties` only deal with the environment and, in particular, does not inject other beans from the context. Having said that, the `@EnableConfigurationProperties` annotation is *also* automatically applied to your project so that any *existing* bean annotated with `@ConfigurationProperties` is configured from the `Environment`. You could shortcut `MyConfiguration` by making sure `AcmeProperties` is already a bean, as shown in the following example:

```
@Component
@ConfigurationProperties(prefix="acme")
public class AcmeProperties {

 // ... see the preceding example

}
```

This style of configuration works particularly well with the `SpringApplication` external YAML configuration, as shown in the following example:

```
# application.yml

acme:
 remote-address: 192.168.1.1
```

```
 security:
  username: admin
  roles:
    - USER
    - ADMIN

# additional configuration as required
```

To work with `@ConfigurationProperties` beans, you can inject them in the same way as any other bean, as shown in the following example:

```
@Service
public class MyService {

 private final AcmeProperties properties;

 @Autowired
 public MyService(AcmeProperties properties) {
     this.properties = properties;
 }

  //...

 @PostConstruct
 public void openConnection() {
  Server server = new Server(this.properties.getRemoteAddress());
  // ...
 }

}
```

> **Tip**
>
> Using `@ConfigurationProperties` also lets you generate metadata files that can be used by IDEs to offer auto-completion for your own keys. See the Appendix B, *Configuration Metadata* appendix for details.

Third-party Configuration

As well as using `@ConfigurationProperties` to annotate a class, you can also use it on public `@Bean` methods. Doing so can be particularly useful when you want to bind properties to third-party components that are outside of your control.

To configure a bean from the `Environment` properties, add `@ConfigurationProperties` to its bean registration, as shown in the following example:

```
@ConfigurationProperties(prefix = "another")
@Bean
public AnotherComponent anotherComponent() {
 ...
}
```

Any property defined with the `another` prefix is mapped onto that `AnotherComponent` bean in manner similar to the preceding `AcmeProperties` example.

Relaxed Binding

Spring Boot uses some relaxed rules for binding `Environment` properties to `@ConfigurationProperties` beans, so there does not need to be an exact match between the `Environment` property name and the bean property name. Common examples where this is useful

include dash-separated environment properties (for example, `context-path` binds to `contextPath`), and capitalized environment properties (for example, `PORT` binds to `port`).

For example, consider the following `@ConfigurationProperties` class:

```
@ConfigurationProperties(prefix="acme.my-project.person")
public class OwnerProperties {

 private String firstName;

 public String getFirstName() {
  return this.firstName;
 }

 public void setFirstName(String firstName) {
  this.firstName = firstName;
 }

}
```

In the preceding example, the following properties names can all be used:

Table 24.1. relaxed binding

Property	Note
`acme.my-project.person.first-name`	Kebab case, which is recommended for use in `.properties` and `.yml` files.
`acme.myProject.person.firstName`	Standard camel case syntax.
`acme.my_project.person.first_name`	Underscore notation, which is an alternative format for use in `.properties` and `.yml` files.
`ACME_MYPROJECT_PERSON_FIRSTNAME`	Upper case format, which is recommended when using system environment variables.

Note

The `prefix` value for the annotation *must* be in kebab case (lowercase and separated by `-`, such as `acme.my-project.person`).

Table 24.2. relaxed binding rules per property source

Property Source	Simple	List
Properties Files	Camel case, kebab case, or underscore notation	Standard list syntax using `[ ]` or comma-separated values
YAML Files	Camel case, kebab case, or underscore notation	Standard YAML list syntax or comma-separated values
Environment Variables	Upper case format with underscore as the delimiter. _ should not be used within a property name	Numeric values surrounded by underscores, such as `MY_ACME_1_OTHER = my.acme[1].other`

Property Source	Simple	List
System properties	Camel case, kebab case, or underscore notation	Standard list syntax using `[ ]` or comma-separated values

> **Tip**
>
> We recommend that, when possible, properties are stored in lower-case kebab format, such as `my.property-name=acme`.

Merging Complex Types

When lists are configured in more than one place, overriding works by replacing the entire list.

For example, assume a `MyPojo` object with `name` and `description` attributes that are `null` by default. The following example exposes a list of `MyPojo` objects from `AcmeProperties`:

```
@ConfigurationProperties("acme")
public class AcmeProperties {

 private final List<MyPojo> list = new ArrayList<>();

 public List<MyPojo> getList() {
  return this.list;
 }

}
```

Consider the following configuration:

```
acme:
  list:
    - name: my name
      description: my description
---
spring:
  profiles: dev
acme:
  list:
    - name: my another name
```

If the `dev` profile is not active, `AcmeProperties.list` contains one `MyPojo` entry, as previously defined. If the `dev` profile is enabled, however, the `list` *still* contains only one entry (with a name of `my another name` and a description of `null`). This configuration *does not* add a second `MyPojo` instance to the list, and it does not merge the items.

When a `List` is specified in multiple profiles, the one with the highest priority (and only that one) is used. Consider the following example:

```
acme:
  list:
    - name: my name
      description: my description
    - name: another name
      description: another description
---
spring:
  profiles: dev
acme:
  list:
    - name: my another name
```

In the preceding example, if the `dev` profile is active, `AcmeProperties.list` contains *one* `MyPojo` entry (with a name of `my another name` and a description of `null`). For YAML, both comma-separated lists and YAML lists can be used for completely overriding the contents of the list.

For `Map` properties, you can bind with property values drawn from multiple sources. However, for the same property in multiple sources, the one with the highest priority is used. The following example exposes a `Map<String, MyPojo>` from `AcmeProperties`:

```
@ConfigurationProperties("acme")
public class AcmeProperties {

 private final Map<String, MyPojo> map = new HashMap<>();

 public Map<String, MyPojo> getMap() {
  return this.map;
 }

}
```

Consider the following configuration:

```
acme:
  map:
    key1:
      name: my name 1
      description: my description 1
---
spring:
  profiles: dev
acme:
  map:
    key1:
      name: dev name 1
    key2:
      name: dev name 2
      description: dev description 2
```

If the `dev` profile is not active, `AcmeProperties.map` contains one entry with key `key1` (with a name of `my name 1` and a description of `my description 1`). If the `dev` profile is enabled, however, `map` contains two entries with keys `key1` (with a name of `dev name 1` and a description of `my description 1`) and `key2` (with a name of `dev name 2` and a description of `dev description 2`).

> **Note**
>
> The preceding merging rules apply to properties from all property sources and not just YAML files.

Properties Conversion

Spring Boot attempts to coerce the external application properties to the right type when it binds to the `@ConfigurationProperties` beans. If you need custom type conversion, you can provide a `ConversionService` bean (with a bean named `conversionService`) or custom property editors (through a `CustomEditorConfigurer` bean) or custom `Converters` (with bean definitions annotated as `@ConfigurationPropertiesBinding`).

> **Note**
>
> As this bean is requested very early during the application lifecycle, make sure to limit the dependencies that your `ConversionService` is using. Typically, any dependency that you

> require may not be fully initialized at creation time. You may want to rename your custom `ConversionService` if it is not required for configuration keys coercion and only rely on custom converters qualified with `@ConfigurationPropertiesBinding`.

Converting durations

Spring Boot has dedicated support for expressing durations. If you expose a `java.time.Duration` property, the following formats in application properties are available:

- A regular `long` representation (using milliseconds as the default unit unless a `@DurationUnit` has been specified)
- The standard ISO-8601 format used by `java.util.Duration`
- A more readable format where the value and the unit are coupled (e.g. `10s` means 10 seconds)

Consider the following example:

```
@ConfigurationProperties("app.system")
public class AppSystemProperties {

 @DurationUnit(ChronoUnit.SECONDS)
 private Duration sessionTimeout = Duration.ofSeconds(30);

 private Duration readTimeout = Duration.ofMillis(1000);

 public Duration getSessionTimeout() {
  return this.sessionTimeout;
 }

 public void setSessionTimeout(Duration sessionTimeout) {
  this.sessionTimeout = sessionTimeout;
 }

 public Duration getReadTimeout() {
  return this.readTimeout;
 }

 public void setReadTimeout(Duration readTimeout) {
  this.readTimeout = readTimeout;
 }

}
```

To specify a session timeout of 30 seconds, `30`, `PT30S` and `30s` are all equivalent. A read timeout of 500ms can be specified in any of the following form: `500`, `PT0.5S` and `500ms`.

You can also use any of the supported unit. These are:

- `ns` for nanoseconds
- `us` for microseconds
- `ms` for milliseconds
- `s` for seconds
- `m` for minutes
- `h` for hours

- `d` for days

The default unit is milliseconds and can be overridden using `@DurationUnit` as illustrated in the sample above.

> **Tip**
>
> If you are upgrading from a previous version that is simply using `Long` to express the duration, make sure to define the unit (using `@DurationUnit`) if it isn't milliseconds alongside the switch to `Duration`. Doing so gives a transparent upgrade path while supporting a much richer format.

@ConfigurationProperties Validation

Spring Boot attempts to validate `@ConfigurationProperties` classes whenever they are annotated with Spring's `@Validated` annotation. You can use JSR-303 `javax.validation` constraint annotations directly on your configuration class. To do so, ensure that a compliant JSR-303 implementation is on your classpath and then add constraint annotations to your fields, as shown in the following example:

```
@ConfigurationProperties(prefix="acme")
@Validated
public class AcmeProperties {

 @NotNull
 private InetAddress remoteAddress;

 // ... getters and setters

}
```

> **Tip**
>
> You can also trigger validation by annotating the `@Bean` method that creates the configuration properties with `@Validated`.

Although nested properties will also be validated when bound, it's good practice to also annotate the associated field as `@Valid`. This ensure that validation is triggered even if no nested properties are found. The following example builds on the preceding `AcmeProperties` example:

```
@ConfigurationProperties(prefix="acme")
@Validated
public class AcmeProperties {

 @NotNull
 private InetAddress remoteAddress;

 @Valid
 private final Security security = new Security();

 // ... getters and setters

 public static class Security {

  @NotEmpty
  public String username;

  // ... getters and setters

 }
```

```
}
```

You can also add a custom Spring `Validator` by creating a bean definition called `configurationPropertiesValidator`. The `@Bean` method should be declared `static`. The configuration properties validator is created very early in the application's lifecycle, and declaring the `@Bean` method as static lets the bean be created without having to instantiate the `@Configuration` class. Doing so avoids any problems that may be caused by early instantiation. There is a property validation sample that shows how to set things up.

> **Tip**
>
> The `spring-boot-actuator` module includes an endpoint that exposes all `@ConfigurationProperties` beans. Point your web browser to `/actuator/configprops` or use the equivalent JMX endpoint. See the "Production ready features" section for details.

@ConfigurationProperties vs. @Value

The `@Value` annotation is a core container feature, and it does not provide the same features as type-safe configuration properties. The following table summarizes the features that are supported by `@ConfigurationProperties` and `@Value`:

Feature	`@ConfigurationProperties`	`@Value`
Relaxed binding	Yes	No
Meta-data support	Yes	No
`SpEL` evaluation	No	Yes

If you define a set of configuration keys for your own components, we recommend you group them in a POJO annotated with `@ConfigurationProperties`. You should also be aware that, since `@Value` does not support relaxed binding, it is not a good candidate if you need to provide the value by using environment variables.

Finally, while you can write a `SpEL` expression in `@Value`, such expressions are not processed from application property files.

25. Profiles

Spring Profiles provide a way to segregate parts of your application configuration and make it be available only in certain environments. Any `@Component` or `@Configuration` can be marked with `@Profile` to limit when it is loaded, as shown in the following example:

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

 // ...

}
```

You can use a `spring.profiles.active` `Environment` property to specify which profiles are active. You can specify the property in any of the ways described earlier in this chapter. For example, you could include it in your `application.properties`, as shown in the following example:

```
spring.profiles.active=dev,hsqldb
```

You could also specify it on the command line by using the following switch: `--spring.profiles.active=dev,hsqldb`.

25.1 Adding Active Profiles

The `spring.profiles.active` property follows the same ordering rules as other properties: The highest `PropertySource` wins. This means that you can specify active profiles in `application.properties` and then **replace** them by using the command line switch.

Sometimes, it is useful to have profile-specific properties that **add** to the active profiles rather than replace them. The `spring.profiles.include` property can be used to unconditionally add active profiles. The `SpringApplication` entry point also has a Java API for setting additional profiles (that is, on top of those activated by the `spring.profiles.active` property). See the `setAdditionalProfiles()` method in SpringApplication.

For example, when an application with the following properties is run by using the switch, `--spring.profiles.active=prod`, the `proddb` and `prodmq` profiles are also activated:

```
---
my.property: fromyamlfile
---
spring.profiles: prod
spring.profiles.include:
  - proddb
  - prodmq
```

> **Note**
>
> Remember that the `spring.profiles` property can be defined in a YAML document to determine when this particular document is included in the configuration. See Section 75.7, “Change Configuration Depending on the Environment” for more details.

25.2 Programmatically Setting Profiles

You can programmatically set active profiles by calling `SpringApplication.setAdditionalProfiles(…)` before your application runs. It is also possible to activate profiles by using Spring's `ConfigurableEnvironment` interface.

25.3 Profile-specific Configuration Files

Profile-specific variants of both `application.properties` (or `application.yml`) and files referenced through `@ConfigurationProperties` are considered as files and loaded. See "Section 24.4, “Profile-specific Properties”" for details.

26. Logging

Spring Boot uses Commons Logging for all internal logging but leaves the underlying log implementation open. Default configurations are provided for Java Util Logging, Log4J2, and Logback. In each case, loggers are pre-configured to use console output with optional file output also available.

By default, if you use the “Starters”, Logback is used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J all work correctly.

> **Tip**
>
> There are a lot of logging frameworks available for Java. Do not worry if the above list seems confusing. Generally, you do not need to change your logging dependencies and the Spring Boot defaults work just fine.

26.1 Log Format

The default log output from Spring Boot resembles the following example:

```
2014-03-05 10:57:51.112  INFO 45469 --- [           main] org.apache.catalina.core.StandardEngine  :
 Starting Servlet Engine: Apache Tomcat/7.0.52
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       :
 Initializing Spring embedded WebApplicationContext
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.s.web.context.ContextLoader            :
 Root WebApplicationContext: initialization completed in 1358 ms
2014-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean        :
 Mapping servlet: 'dispatcherServlet' to [/]
2014-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  :
 Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
```

The following items are output:

- Date and Time: Millisecond precision and easily sortable.
- Log Level: `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE`.
- Process ID.
- A `---` separator to distinguish the start of actual log messages.
- Thread name: Enclosed in square brackets (may be truncated for console output).
- Logger name: This is usually the source class name (often abbreviated).
- The log message.

> **Note**
>
> Logback does not have a `FATAL` level. It is mapped to `ERROR`.

26.2 Console Output

The default log configuration echoes messages to the console as they are written. By default, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged. You can also enable a “debug” mode by starting your application with a `--debug` flag.

```
$ java -jar myapp.jar --debug
```

Note

You can also specify `debug=true` in your `application.properties`.

When the debug mode is enabled, a selection of core loggers (embedded container, Hibernate, and Spring Boot) are configured to output more information. Enabling the debug mode does *not* configure your application to log all messages with `DEBUG` level.

Alternatively, you can enable a “trace” mode by starting your application with a `--trace` flag (or `trace=true` in your `application.properties`). Doing so enables trace logging for a selection of core loggers (embedded container, Hibernate schema generation, and the whole Spring portfolio).

Color-coded Output

If your terminal supports ANSI, color output is used to aid readability. You can set `spring.output.ansi.enabled` to a supported value to override the auto detection.

Color coding is configured by using the `%clr` conversion word. In its simplest form, the converter colors the output according to the log level, as shown in the following example:

```
%clr(%5p)
```

The following table describes the mapping of log levels to colors:

Level	Color
`FATAL`	Red
`ERROR`	Red
`WARN`	Yellow
`INFO`	Green
`DEBUG`	Green
`TRACE`	Green

Alternatively, you can specify the color or style that should be used by providing it as an option to the conversion. For example, to make the text yellow, use the following setting:

```
%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){yellow}
```

The following colors and styles are supported:

- `blue`
- `cyan`
- `faint`
- `green`
- `magenta`

- `red`
- `yellow`

26.3 File Output

By default, Spring Boot logs only to the console and does not write log files. If you want to write log files in addition to the console output, you need to set a `logging.file` or `logging.path` property (for example, in your `application.properties`).

The following table shows how the `logging.*` properties can be used together:

Table 26.1. Logging properties

`logging.file`	`logging.path`	Example	Description
(none)	*(none)*		Console only logging.
Specific file	*(none)*	`my.log`	Writes to the specified log file. Names can be an exact location or relative to the current directory.
(none)	Specific directory	`/var/log`	Writes `spring.log` to the specified directory. Names can be an exact location or relative to the current directory.

Log files rotate when they reach 10 MB and, as with console output, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged by default. Size limits can be changed using the `logging.file.max-size` property. Previously rotated files are archived indefinitely unless the `logging.file.max-history` property has been set.

> **Note**
>
> The logging system is initialized early in the application lifecycle. Consequently, logging properties are not found in property files loaded through `@PropertySource` annotations.

> **Tip**
>
> Logging properties are independent of the actual logging infrastructure. As a result, specific configuration keys (such as `logback.configurationFile` for Logback) are not managed by spring Boot.

26.4 Log Levels

All the supported logging systems can have the logger levels set in the Spring `Environment` (for example, in `application.properties`) by using `logging.level.<logger-name>=<level>` where `level` is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, or OFF. The `root` logger can be configured by using `logging.level.root`.

The following example shows potential logging settings in `application.properties`:

```
logging.level.root=WARN
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

26.5 Custom Log Configuration

The various logging systems can be activated by including the appropriate libraries on the classpath and can be further customized by providing a suitable configuration file in the root of the classpath or in a location specified by the following Spring `Environment` property: `logging.config`.

You can force Spring Boot to use a particular logging system by using the `org.springframework.boot.logging.LoggingSystem` system property. The value should be the fully qualified class name of a `LoggingSystem` implementation. You can also disable Spring Boot's logging configuration entirely by using a value of `none`.

> **Note**
>
> Since logging is initialized **before** the `ApplicationContext` is created, it is not possible to control logging from `@PropertySources` in Spring `@Configuration` files. The only way to change the logging system or disable it entirely is via System properties.

Depending on your logging system, the following files are loaded:

Logging System	Customization
Logback	`logback-spring.xml`, `logback-spring.groovy`, `logback.xml`, or `logback.groovy`
Log4j2	`log4j2-spring.xml` or `log4j2.xml`
JDK (Java Util Logging)	`logging.properties`

> **Note**
>
> When possible, we recommend that you use the `-spring` variants for your logging configuration (for example, `logback-spring.xml` rather than `logback.xml`). If you use standard configuration locations, Spring cannot completely control log initialization.

> **Warning**
>
> There are known classloading issues with Java Util Logging that cause problems when running from an 'executable jar'. We recommend that you avoid it when running from an 'executable jar' if at all possible.

To help with the customization, some other properties are transferred from the Spring `Environment` to System properties, as described in the following table:

Spring Environment	System Property	Comments
`logging.exception-conversion-word`	`LOG_EXCEPTION_CONVERSION_WORD`	The conversion word used when logging exceptions.
`logging.file`	`LOG_FILE`	If defined, it is used in the default log configuration.

Spring Environment	System Property	Comments
`logging.file.max-size`	`LOG_FILE_MAX_SIZE`	Maximum log file size (if LOG_FILE enabled). (Only supported with the default Logback setup.)
`logging.file.max-history`	`LOG_FILE_MAX_HISTORY`	Maximum number of archive log files to keep (if LOG_FILE enabled). (Only supported with the default Logback setup.)
`logging.path`	`LOG_PATH`	If defined, it is used in the default log configuration.
`logging.pattern.console`	`CONSOLE_LOG_PATTERN`	The log pattern to use on the console (stdout). (Only supported with the default Logback setup.)
`logging.pattern.dateformat`	`LOG_DATEFORMAT_PATTERN`	Appender pattern for log date format. (Only supported with the default Logback setup.)
`logging.pattern.file`	`FILE_LOG_PATTERN`	The log pattern to use in a file (if `LOG_FILE` is enabled). (Only supported with the default Logback setup.)
`logging.pattern.level`	`LOG_LEVEL_PATTERN`	The format to use when rendering the log level (default `%5p`). (Only supported with the default Logback setup.)
`PID`	`PID`	The current process ID (discovered if possible and when not already defined as an OS environment variable).

All the supported logging systems can consult System properties when parsing their configuration files. See the default configurations in `spring-boot.jar` for examples:

- Logback
- Log4j 2
- Java Util logging

> **Tip**
>
> If you want to use a placeholder in a logging property, you should use Spring Boot's syntax and not the syntax of the underlying framework. Notably, if you use Logback, you should use `:` as the delimiter between a property name and its default value and not use `:-`.

Tip

You can add MDC and other ad-hoc content to log lines by overriding only the `LOG_LEVEL_PATTERN` (or `logging.pattern.level` with Logback). For example, if you use `logging.pattern.level=user:%X{user} %5p`, then the default log format contains an MDC entry for "user", if it exists, as shown in the following example.

```
2015-09-30 12:30:04.031 user:someone INFO 22174 --- [  nio-8080-exec-0] demo.Controller
Handling authenticated request
```

26.6 Logback Extensions

Spring Boot includes a number of extensions to Logback that can help with advanced configuration. You can use these extensions in your `logback-spring.xml` configuration file.

Note

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

Warning

The extensions cannot be used with Logback's configuration scanning. If you attempt to do so, making changes to the configuration file results in an error similar to one of the following being logged:

```
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProperty],
 current ElementPath is [[configuration][springProperty]]
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProfile],
 current ElementPath is [[configuration][springProfile]]
```

Profile-specific Configuration

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

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

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

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

Environment Properties

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

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

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

27. Developing Web Applications

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.

27.1 The “Spring Web MVC Framework”

The Spring Web MVC framework (often referred to as simply “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:

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

 @RequestMapping(value="/{user}", method=RequestMethod.GET)
 public User getUser(@PathVariable Long user) {
  // ...
 }

 @RequestMapping(value="/{user}/customers", method=RequestMethod.GET)
 List<Customer> getUserCustomers(@PathVariable Long user) {
  // ...
 }

 @RequestMapping(value="/{user}", method=RequestMethod.DELETE)
 public User deleteUser(@PathVariable Long user) {
  // ...
 }

}
```

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.

Spring MVC Auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications.

The auto-configuration adds the following features on top of Spring’s defaults:

- 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.
- Custom `Favicon` support (covered later in this document).
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered later in this document).

If you want to keep Spring Boot MVC features and you want to add additional MVC configuration (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`. If you wish to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, you can declare a `WebMvcRegistrationsAdapter` instance to provide such components.

If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`.

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`.

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

```
import org.springframework.boot.autoconfigure.web.HttpMessageConverters;
import org.springframework.context.annotation.*;
import org.springframework.http.converter.*;

@Configuration
public class MyConfiguration {

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

}
```

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.

Custom JSON Serializers and Deserializers

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

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

```
import java.io.*;
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import org.springframework.boot.jackson.*;

@JsonComponent
public class Example {

 public static class Serializer extends JsonSerializer<SomeObject> {
  // ...
 }

 public static class Deserializer extends JsonDeserializer<SomeObject> {
  // ...
 }

}
```

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

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

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 also enabled and 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:

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

You can also customize the static resource locations by using the `spring.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. Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format.

> **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:

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.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:

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

Custom Favicon

Spring Boot looks for a `favicon.ico` in the configured static content locations and the root of the classpath (in that order). 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"` won't 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 don't 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")`:

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

# We can change the parameter name, which is "format" by default:
# spring.mvc.contentnegotiation.parameter-name=myparam

# We can also register additional file extensions/media types with:
spring.mvc.contentnegotiation.media-types.markdown=text/markdown
```

If you understand the caveats and would still like your application to use suffix pattern matching, the following configuration is required:

```
spring.mvc.contentnegotiation.favor-path-extension=true

# You can also restrict that feature to known extensions only
# spring.mvc.pathmatch.use-registered-suffix-pattern=true

# We can also register additional file extensions/media types with:
# spring.mvc.contentnegotiation.media-types.adoc=text/asciidoc
```

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, IntelliJ IDEA orders 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 templates on the classpath. If you have this problem, you can reorder the classpath in the IDE to place the module's classes and resources first. Alternatively, you can configure the template prefix to search every `templates` directory on the classpath, as follows: `classpath*:/templates/`.

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`). 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.

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:

```
@ControllerAdvice(basePackageClasses = AcmeController.class)
public class AcmeControllerAdvice extends ResponseEntityExceptionHandler {

 @ExceptionHandler(YourException.class)
 @ResponseBody
 ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) {
  HttpStatus status = getStatus(request);
  return new ResponseEntity<>(new CustomErrorType(status.value(), ex.getMessage()), status);
 }

 private HttpStatus getStatus(HttpServletRequest request) {
  Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.status_code");
  if (statusCode == null) {
   return HttpStatus.INTERNAL_SERVER_ERROR;
  }
  return HttpStatus.valueOf(statusCode);
 }

}
```

In the preceding example, if `YourException` is thrown by a controller defined in the same package as `AcmeController`, a JSON representation of the `CustomErrorType` POJO is used instead of the `ErrorAttributes` representation.

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` folder. Error pages can either be static HTML (that is, added under any of the static resource folders) or be built by using templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

To map all `5xx` errors by using a FreeMarker template, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.ftl
             +- <other templates>
```

For more complex mappings, you can also add beans that implement the `ErrorViewResolver` interface, as shown in the following example:

```
public class MyErrorViewResolver implements ErrorViewResolver {

 @Override
 public ModelAndView resolveErrorView(HttpServletRequest request,
   HttpStatus status, Map<String, Object> model) {
```

```
  // Use the request or status to optionally return a ModelAndView
  return ...
 }

}
```

You can also use regular Spring MVC features such as `@ExceptionHandler` methods and `@ControllerAdvice`. The `ErrorController` then picks up any unhandled exceptions.

Mapping Error Pages outside of Spring MVC

For applications that do not use Spring MVC, you can use the `ErrorPageRegistrar` interface to directly register `ErrorPages`. This abstraction works directly with the underlying embedded servlet container and works even if you do not have a Spring MVC `DispatcherServlet`.

```
@Bean
public ErrorPageRegistrar errorPageRegistrar(){
 return new MyErrorPageRegistrar();
}

// ...

private static class MyErrorPageRegistrar implements ErrorPageRegistrar {

 @Override
 public void registerErrorPages(ErrorPageRegistry registry) {
  registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
 }

}
```

> **Note**
>
> If you register an `ErrorPage` with a path that ends up being handled by a `Filter` (as is common with some non-Spring web frameworks, like Jersey and Wicket), then the `Filter` has to be explicitly registered as an `ERROR` dispatcher, as shown in the following example:

```
@Bean
public FilterRegistrationBean myFilter() {
 FilterRegistrationBean registration = new FilterRegistrationBean();
 registration.setFilter(new MyFilter());
 ...
 registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.class));
 return registration;
}
```

Note that the default `FilterRegistrationBean` does not include the `ERROR` dispatcher type.

CAUTION:When deployed to a servlet container, Spring Boot uses its error page filter to forward a request with an error status to the appropriate error page. The request can only be forwarded to the correct error page if the response has not already been committed. By default, WebSphere Application Server 8.0 and later commits the response upon successful completion of a servlet's service method. You should disable this behavior by setting `com.ibm.ws.webcontainer.invokeFlushAfterService` to `false`.

Spring HATEOAS

If you develop a RESTful API that makes use of hypermedia, Spring Boot provides auto-configuration for Spring HATEOAS that works well with most applications. The auto-configuration replaces the need to use `@EnableHypermediaSupport` and registers a number of beans to ease building

hypermedia-based applications, including a `LinkDiscoverers` (for client side support) and an `ObjectMapper` configured to correctly marshal responses into the desired representation. The `ObjectMapper` is customized by setting the various `spring.jackson.*` properties or, if one exists, by a `Jackson2ObjectMapperBuilder` bean.

You can take control of Spring HATEOAS's configuration by using `@EnableHypermediaSupport`. Note that doing so disables the `ObjectMapper` customization described earlier.

CORS Support

Cross-origin resource sharing (CORS) is a W3C specification implemented by most browsers that lets you specify in a flexible way what kind of cross-domain requests are authorized, instead of using some less secure and less powerful approaches such as IFRAME or JSONP.

As of version 4.2, Spring MVC supports CORS. Using controller method CORS configuration with `@CrossOrigin` annotations in your Spring Boot application does not require any specific configuration. Global CORS configuration can be defined by registering a `WebMvcConfigurer` bean with a customized `addCorsMappings(CorsRegistry)` method, as shown in the following example:

```
@Configuration
public class MyConfiguration {

 @Bean
 public WebMvcConfigurer corsConfigurer() {
  return new WebMvcConfigurer() {
   @Override
   public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**");
   }
  };
 }
}
```

27.2 The "Spring WebFlux Framework"

Spring WebFlux is the new reactive web framework introduced in Spring Framework 5.0. Unlike Spring MVC, it does not require the Servlet API, is fully asynchronous and non-blocking, and implements the Reactive Streams specification through the Reactor project.

Spring WebFlux comes in two flavors: functional and annotation-based. The annotation-based one is quite close to the Spring MVC model, as shown in the following example:

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

 @GetMapping("/{user}")
 public Mono<User> getUser(@PathVariable Long user) {
  // ...
 }

 @GetMapping("/{user}/customers")
 public Flux<Customer> getUserCustomers(@PathVariable Long user) {
  // ...
 }

 @DeleteMapping("/{user}")
 public Mono<User> deleteUser(@PathVariable Long user) {
  // ...
 }

}
```

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

```
@Configuration
public class RoutingConfiguration {

 @Bean
 public RouterFunction<ServerResponse> monoRouterFunction(UserHandler userHandler) {
  return route(GET("/{user}").and(accept(APPLICATION_JSON)), userHandler::getUser)
    .andRoute(GET("/{user}/customers").and(accept(APPLICATION_JSON)), userHandler::getUserCustomers)
    .andRoute(DELETE("/{user}").and(accept(APPLICATION_JSON)), userHandler::deleteUser);
 }

}

@Component
public class UserHandler {

 public Mono<ServerResponse> getUser(ServerRequest request) {
  // ...
 }

 public Mono<ServerResponse> getUserCustomers(ServerRequest request) {
  // ...
 }

 public Mono<ServerResponse> deleteUser(ServerRequest request) {
  // ...
 }
}
```

WebFlux is part of the Spring Framework and detailed information is available in its reference documentation.

> **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.

To get started, add the `spring-boot-starter-webflux` module to your application.

> **Note**
>
> Adding both `spring-boot-starter-web` and `spring-boot-starter-webflux` modules in your application results in Spring Boot auto-configuring Spring MVC, not WebFlux. This behavior has been chosen because many Spring developers add `spring-boot-starter-webflux` to their Spring MVC application to use the reactive `WebClient`. You can still enforce your choice by setting the chosen application type to `SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`.

Spring WebFlux Auto-configuration

Spring Boot provides auto-configuration for Spring WebFlux that works well with most applications.

The auto-configuration adds the following features on top of Spring’s defaults:

- Configuring codecs for `HttpMessageReader` and `HttpMessageWriter` instances (described later in this document).
- Support for serving static resources, including support for WebJars (described later in this document).

If you want to keep Spring Boot WebFlux features and you want to add additional WebFlux configuration, you can add your own `@Configuration` class of type `WebFluxConfigurer` but **without** `@EnableWebFlux`.

If you want to take complete control of Spring WebFlux, you can add your own `@Configuration` annotated with `@EnableWebFlux`.

HTTP Codecs with HttpMessageReaders and HttpMessageWriters

Spring WebFlux uses the `HttpMessageReader` and `HttpMessageWriter` interfaces to convert HTTP requests and responses. They are configured with `CodecConfigurer` to have sensible defaults by looking at the libraries available in your classpath.

Spring Boot applies further customization by using `CodecCustomizer` instances. For example, `spring.jackson.*` configuration keys are applied to the Jackson codec.

If you need to add or customize codecs, you can create a custom `CodecCustomizer` component, as shown in the following example:

```
import org.springframework.boot.web.codec.CodecCustomizer;

@Configuration
public class MyConfiguration {

 @Bean
 public CodecCustomizer myCodecCustomizer() {
  return codecConfigurer -> {
   // ...
  }
 }

}
```

You can also leverage Boot's custom JSON serializers and deserializers.

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. It uses the `ResourceWebHandler` from Spring WebFlux so that you can modify that behavior by adding your own `WebFluxConfigurer` and overriding the `addResourceHandlers` method.

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

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

You can also customize the static resource locations by using `spring.resources.static-locations`. Doing so replaces the default values with a list of directory locations. If you do so, the default welcome page detection switches to your custom locations. So, if there is an `index.html` in any of your locations on startup, it is the home page of the application.

In addition to the "standard" static resource locations listed earlier, a special case is made for Webjars content. Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format.

> **Tip**
>
> Spring WebFlux applications do not strictly depend on the Servlet API, so they cannot be deployed as war files and do not use the `src/main/webapp` directory.

Template Engines

As well as REST web services, you can also use Spring WebFlux to serve dynamic HTML content. Spring WebFlux supports a variety of templating technologies, including Thymeleaf, FreeMarker, and Mustache.

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

- FreeMarker
- Thymeleaf
- Mustache

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

Error Handling

Spring Boot provides a `WebExceptionHandler` that handles all errors in a sensible way. Its position in the processing order is immediately before the handlers provided by WebFlux, which are considered last. 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 handler that renders the same data in HTML format. You can also provide your own HTML templates to display errors (see the next section).

The first step to customizing this feature often involves using the existing mechanism but replacing or augmenting the error contents. For that, you can add a bean of type `ErrorAttributes`.

To change the error handling behavior, you can implement `ErrorWebExceptionHandler` and register a bean definition of that type. Because a `WebExceptionHandler` is quite low-level, Spring Boot also provides a convenient `AbstractErrorWebExceptionHandler` to let you handle errors in a WebFlux functional way, as shown in the following example:

```
public class CustomErrorWebExceptionHandler extends AbstractErrorWebExceptionHandler {

 // Define constructor here

 @Override
 protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes errorAttributes) {

  return RouterFunctions
    .route(aPredicate, aHandler)
    .andRoute(anotherPredicate, anotherHandler);
 }

}
```

For a more complete picture, you can also subclass `DefaultErrorWebExceptionHandler` directly and override specific methods.

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` folder. Error pages can either be static HTML (that is, added under any of the static resource folders) or built with templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

To map all `5xx` errors by using a Mustache template, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.mustache
             +- <other templates>
```

Web Filters

Spring WebFlux provides a `WebFilter` interface that can be implemented to filter HTTP request-response exchanges. `WebFilter` beans found in the application context will be automatically used to filter each exchange.

Where the order of the filters is important they can implement `Ordered` or be annotated with `@Order`. Spring Boot auto-configuration may configure web filters for you. When it does so, the orders shown in the following table will be used:

Web Filter	Order
`MetricsWebFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`WebFilterChainProxy` (Spring Security)	`-100`
`HttpTraceWebFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

27.3 JAX-RS and Jersey

If you prefer the JAX-RS programming model for REST endpoints, you can use one of the available implementations instead of Spring MVC. Jersey and Apache CXF work quite well out of the box. CXF requires you to register its `Servlet` or `Filter` as a `@Bean` in your application context. Jersey has some native Spring support, so we also provide auto-configuration support for it in Spring Boot, together with a starter.

To get started with Jersey, include the `spring-boot-starter-jersey` as a dependency and then you need one `@Bean` of type `ResourceConfig` in which you register all the endpoints, as shown in the following example:

```
@Component
public class JerseyConfig extends ResourceConfig {

 public JerseyConfig() {
  register(Endpoint.class);
 }

}
```

Warning

Jersey's support for scanning executable archives is rather limited. For example, it cannot scan for endpoints in a package found in `WEB-INF/classes` when running an executable war file. To avoid this limitation, the `packages` method should not be used, and endpoints should be registered individually by using the `register` method, as shown in the preceding example.

For more advanced customizations, you can also register an arbitrary number of beans that implement `ResourceConfigCustomizer`.

All the registered endpoints should be `@Components` with HTTP resource annotations (`@GET` and others), as shown in the following example:

```
@Component
@Path("/hello")
public class Endpoint {

 @GET
 public String message() {
  return "Hello";
 }

}
```

Since the `Endpoint` is a Spring `@Component`, its lifecycle is managed by Spring and you can use the `@Autowired` annotation to inject dependencies and use the `@Value` annotation to inject external configuration. By default, the Jersey servlet is registered and mapped to `/*`. You can change the mapping by adding `@ApplicationPath` to your `ResourceConfig`.

By default, Jersey is set up as a Servlet in a `@Bean` of type `ServletRegistrationBean` named `jerseyServletRegistration`. By default, the servlet is initialized lazily, but you can customize that behavior by setting `spring.jersey.servlet.load-on-startup`. You can disable or override that bean by creating one of your own with the same name. You can also use a filter instead of a servlet by setting `spring.jersey.type=filter` (in which case, the `@Bean` to replace or override is `jerseyFilterRegistration`). The filter has an `@Order`, which you can set with `spring.jersey.filter.order`. Both the servlet and the filter registrations can be given init parameters by using `spring.jersey.init.*` to specify a map of properties.

There is a Jersey sample so that you can see how to set things up.

27.4 Embedded Servlet Container Support

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. Most developers use the appropriate "Starter" to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port `8080`.

> **Warning**
>
> If you choose to use Tomcat on CentOS, be aware that, by default, a temporary directory is used to store compiled JSPs, file uploads, and so on. This directory may be deleted by `tmpwatch` while your application is running, leading to failures. To avoid this behavior, you may want to customize your `tmpwatch` configuration such that `tomcat.*` directories are not deleted or configure `server.tomcat.basedir` such that embedded Tomcat uses a different location.

Servlets, Filters, and listeners

When using an embedded servlet container, you can register servlets, filters, and all the listeners (such as `HttpSessionListener`) from the Servlet spec, either by using Spring beans or by scanning for Servlet components.

Registering Servlets, Filters, and Listeners as Spring Beans

Any `Servlet`, `Filter`, or servlet `*Listener` instance that is a Spring bean is registered with the embedded container. This can be particularly convenient if you want to refer to a value from your `application.properties` during configuration.

By default, if the context contains only a single Servlet, it is mapped to `/`. In the case of multiple servlet beans, the bean name is used as a path prefix. Filters map to `/*`.

If convention-based mapping is not flexible enough, you can use the `ServletRegistrationBean`, `FilterRegistrationBean`, and `ServletListenerRegistrationBean` classes for complete control.

Spring Boot ships with many auto-configurations that may define Filter beans. Here are a few examples of Filters and their respective order (lower order value means higher precedence):

Servlet Filter	Order
`OrderedCharacterEncodingFilter`	`Ordered.HIGHEST_PRECEDENCE`
`WebMvcMetricsFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`ErrorPageFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`HttpTraceFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

It is usually safe to leave Filter beans unordered.

If a specific order is required, you should avoid configuring a Filter that reads the request body at `Ordered.HIGHEST_PRECEDENCE`, since it might go against the character encoding configuration of your application. If a Servlet filter wraps the request, it should be configured with an order that is less than or equal to `FilterRegistrationBean.REQUEST_WRAPPER_FILTER_MAX_ORDER`.

Servlet Context Initialization

Embedded servlet containers do not directly execute the Servlet 3.0+ `javax.servlet.ServletContainerInitializer` interface or Spring's `org.springframework.web.WebApplicationInitializer` interface. This is an intentional

design decision intended to reduce the risk that third party libraries designed to run inside a war may break Spring Boot applications.

If you need to perform servlet context initialization in a Spring Boot application, you should register a bean that implements the `org.springframework.boot.web.servlet.ServletContextInitializer` interface. The single `onStartup` method provides access to the `ServletContext` and, if necessary, can easily be used as an adapter to an existing `WebApplicationInitializer`.

Scanning for Servlets, Filters, and listeners

When using an embedded container, automatic registration of classes annotated with `@WebServlet`, `@WebFilter`, and `@WebListener` can be enabled by using `@ServletComponentScan`.

> **Tip**
>
> `@ServletComponentScan` has no effect in a standalone container, where the container's built-in discovery mechanisms are used instead.

The ServletWebServerApplicationContext

Under the hood, Spring Boot uses a different type of `ApplicationContext` for embedded servlet container support. The `ServletWebServerApplicationContext` is a special type of `WebApplicationContext` that bootstraps itself by searching for a single `ServletWebServerFactory` bean. Usually a `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` has been auto-configured.

> **Note**
>
> You usually do not need to be aware of these implementation classes. Most applications are auto-configured, and the appropriate `ApplicationContext` and `ServletWebServerFactory` are created on your behalf.

Customizing Embedded Servlet Containers

Common servlet container settings can be configured by using Spring `Environment` properties. Usually, you would define the properties in your `application.properties` file.

Common server settings include:

- Network settings: Listen port for incoming HTTP requests (`server.port`), interface address to bind to `server.address`, and so on.
- Session settings: Whether the session is persistent (`server.servlet.session.persistence`), session timeout (`server.servlet.session.timeout`), location of session data (`server.servlet.session.store-dir`), and session-cookie configuration (`server.servlet.session.cookie.*`).
- Error management: Location of the error page (`server.error.path`) and so on.
- SSL

- HTTP compression

Spring Boot tries as much as possible to expose common settings, but this is not always possible. For those cases, dedicated namespaces offer server-specific customizations (see `server.tomcat` and `server.undertow`). For instance, access logs can be configured with specific features of the embedded servlet container.

> **Tip**
>
> See the `ServerProperties` class for a complete list.

Programmatic Customization

If you need to programmatically configure your embedded servlet container, you can register a Spring bean that implements the `WebServerFactoryCustomizer` interface. `WebServerFactoryCustomizer` provides access to the `ConfigurableServletWebServerFactory`, which includes numerous customization setter methods. The following example shows programmatically setting the port:

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements
 WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

 @Override
 public void customize(ConfigurableServletWebServerFactory server) {
  server.setPort(9000);
 }

}
```

> **Note**
>
> `TomcatServletWebServerFactory`, `JettyServletWebServerFactory` and `UndertowServletWebServerFactory` are dedicated variants of `ConfigurableServletWebServerFactory` that have additional customization setter methods for Tomcat, Jetty and Undertow respectively.

Customizing ConfigurableServletWebServerFactory Directly

If the preceding customization techniques are too limited, you can register the `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` bean yourself.

```
@Bean
public ConfigurableServletWebServerFactory webServerFactory() {
 TomcatServletWebServerFactory factory = new TomcatServletWebServerFactory();
 factory.setPort(9000);
 factory.setSessionTimeout(10, TimeUnit.MINUTES);
 factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND, "/notfound.html"));
 return factory;
}
```

Setters are provided for many configuration options. Several protected method “hooks” are also provided should you need to do something more exotic. See the source code documentation for details.

JSP Limitations

When running a Spring Boot application that uses an embedded servlet container (and is packaged as an executable archive), there are some limitations in the JSP support.

- With Jetty and Tomcat, it should work if you use war packaging. An executable war will work when launched with `java -jar`, and will also be deployable to any standard container. JSPs are not supported when using an executable jar.
- Undertow does not support JSPs.
- Creating a custom `error.jsp` page does not override the default view for error handling. Custom error pages should be used instead.

There is a JSP sample so that you can see how to set things up.

28. Security

If Spring Security is on the classpath, then web applications are secured by default. Spring Boot relies on Spring Security's content-negotiation strategy to determine whether to use `httpBasic` or `formLogin`. To add method-level security to a web application, you can also add `@EnableGlobalMethodSecurity` with your desired settings. Additional information can be found in the Spring Security Reference Guide.

The default `UserDetailsService` has a single user. The user name is `user`, and the password is random and is printed at INFO level when the application starts, as shown in the following example:

```
Using generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35
```

> **Note**
>
> If you fine-tune your logging configuration, ensure that the `org.springframework.boot.autoconfigure.security` category is set to log `INFO`-level messages. Otherwise, the default password is not printed.

You can change the username and password by providing a `spring.security.user.name` and `spring.security.user.password`.

The basic features you get by default in a web application are:

- A `UserDetailsService` (or `ReactiveUserDetailsService` in case of a WebFlux application) bean with in-memory store and a single user with a generated password (see `SecurityProperties.User` for the properties of the user).
- Form-based login or HTTP Basic security (depending on Content-Type) for the entire application (including actuator endpoints if actuator is on the classpath).
- A `DefaultAuthenticationEventPublisher` for publishing authentication events.

You can provide a different `AuthenticationEventPublisher` by adding a bean for it.

28.1 MVC Security

The default security configuration is implemented in `SecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `SecurityAutoConfiguration` imports `SpringBootWebSecurityConfiguration` for web security and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications. To switch off the default web application security configuration completely, you can add a bean of type `WebSecurityConfigurerAdapter` (doing so does not disable the `UserDetailsService` configuration or Actuator's security).

To also switch off the `UserDetailsService` configuration, you can add a bean of type `UserDetailsService`, `AuthenticationProvider`, or `AuthenticationManager`. There are several secure applications in the Spring Boot samples to get you started with common use cases.

Access rules can be overridden by adding a custom `WebSecurityConfigurerAdapter`. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `RequestMatcher` that is based on

the `management.endpoints.web.base-path` property. `PathRequest` can be used to create a `RequestMatcher` for resources in commonly used locations.

28.2 WebFlux Security

Similar to Spring MVC applications, you can secure your WebFlux applications by adding the `spring-boot-starter-security` dependency. The default security configuration is implemented in `ReactiveSecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `ReactiveSecurityAutoConfiguration` imports `WebFluxSecurityConfiguration` for web security and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications. To switch off the default web application security configuration completely, you can add a bean of type `WebFilterChainProxy` (doing so does not disable the `UserDetailsService` configuration or Actuator's security).

To also switch off the `UserDetailsService` configuration, you can add a bean of type `ReactiveUserDetailsService` or `ReactiveAuthenticationManager`.

Access rules can be configured by adding a custom `SecurityWebFilterChain`. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `ServerWebExchangeMatcher` that is based on the `management.endpoints.web.base-path` property.

`PathRequest` can be used to create a `ServerWebExchangeMatcher` for resources in commonly used locations.

For example, you can customize your security configuration by adding something like:

```
@Bean
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
 return http
  .authorizeExchange()
   .matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
   .pathMatchers("/foo", "/bar")
    .authenticated().and()
   .formLogin().and()
  .build();
}
```

28.3 OAuth2

OAuth2 is a widely used authorization framework that is supported by Spring.

Client

If you have `spring-security-oauth2-client` on your classpath, you can take advantage of some auto-configuration to make it easy to set up an OAuth2 Client. This configuration makes use of the properties under `OAuth2ClientProperties`. The same properties are applicable for both servlet and reactive applications.

You can register multiple OAuth2 clients and providers under the `spring.security.oauth2.client` prefix, as shown in the following example:

```
spring.security.oauth2.client.registration.my-client-1.client-id=abcd
spring.security.oauth2.client.registration.my-client-1.client-secret=password
spring.security.oauth2.client.registration.my-client-1.client-name=Client for user scope
spring.security.oauth2.client.registration.my-client-1.provider=my-oauth-provider
```

```
spring.security.oauth2.client.registration.my-client-1.scope=user
spring.security.oauth2.client.registration.my-client-1.redirect-uri-template=http://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-1.client-authentication-method=basic
spring.security.oauth2.client.registration.my-client-1.authorization-grant-type=authorization_code

spring.security.oauth2.client.registration.my-client-2.client-id=abcd
spring.security.oauth2.client.registration.my-client-2.client-secret=password
spring.security.oauth2.client.registration.my-client-2.client-name=Client for email scope
spring.security.oauth2.client.registration.my-client-2.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-2.scope=email
spring.security.oauth2.client.registration.my-client-2.redirect-uri-template=http://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-2.client-authentication-method=basic
spring.security.oauth2.client.registration.my-client-2.authorization-grant-type=authorization_code

spring.security.oauth2.client.provider.my-oauth-provider.authorization-uri=http://my-auth-server/oauth/
authorize
spring.security.oauth2.client.provider.my-oauth-provider.token-uri=http://my-auth-server/oauth/token
spring.security.oauth2.client.provider.my-oauth-provider.user-info-uri=http://my-auth-server/userinfo
spring.security.oauth2.client.provider.my-oauth-provider.jwk-set-uri=http://my-auth-server/token_keys
spring.security.oauth2.client.provider.my-oauth-provider.user-name-attribute=name
```

By default, Spring Security's `OAuth2LoginAuthenticationFilter` only processes URLs matching `/login/oauth2/code/*`. If you want to customize the `redirect-uri-template` to use a different pattern, you need to provide configuration to process that custom pattern. For example, for servlet applications, you can add your own `WebSecurityConfigurerAdapter` that resembles the following:

```
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  http
   .authorizeRequests()
    .anyRequest().authenticated()
    .and()
   .oauth2Login()
    .redirectionEndpoint()
     .baseUri("/custom-callback");
 }
}
```

For common OAuth2 and OpenID providers, including Google, Github, Facebook, and Okta, we provide a set of provider defaults (`google`, `github`, `facebook`, and `okta`, respectively).

If you do not need to customize these providers, you can set the `provider` attribute to the one for which you need to infer defaults. Also, if the ID of your client matches the default supported provider, Spring Boot infers that as well.

In other words, the two configurations in the following example use the Google provider:

```
spring.security.oauth2.client.registration.my-client.client-id=abcd
spring.security.oauth2.client.registration.my-client.client-secret=password
spring.security.oauth2.client.registration.my-client.provider=google

spring.security.oauth2.client.registration.google.client-id=abcd
spring.security.oauth2.client.registration.google.client-secret=password
```

Server

Currently, Spring Security does not provide support for implementing an OAuth 2.0 Authorization Server or Resource Server. However, this functionality is available from the Spring Security OAuth project, which will eventually be superseded by Spring Security completely. Until then, you can use the `spring-security-oauth2-autoconfigure` module to easily set up an OAuth 2.0 server; see its documentation for instructions.

28.4 Actuator Security

For security purposes, all actuators other than `/health` and `/info` are disabled by default. The `management.endpoints.web.exposure.include` property can be used to enable the actuators.

If Spring Security is on the classpath and no other WebSecurityConfigurerAdapter is present, all actuators other than `/health` and `/info` are secured by Spring Boot auto-config. If you define a custom `WebSecurityConfigurerAdapter`, Spring Boot auto-config will back off and you will be in full control of actuator access rules.

> **Note**
>
> Before setting the `management.endpoints.web.exposure.include`, ensure that the exposed actuators do not contain sensitive information and/or are secured by placing them behind a firewall or by something like Spring Security.

Cross Site Request Forgery Protection

Since Spring Boot relies on Spring Security's defaults, CSRF protection is turned on by default. This means that the actuator endpoints that require a `POST` (shutdown and loggers endpoints), `PUT` or `DELETE` will get a 403 forbidden error when the default security configuration is in use.

> **Note**
>
> We recommend disabling CSRF protection completely only if you are creating a service that is used by non-browser clients.

Additional information about CSRF protection can be found in the Spring Security Reference Guide.

29. Working with SQL Databases

The Spring Framework provides extensive support for working with SQL databases, from direct JDBC access using `JdbcTemplate` to complete “object relational mapping” technologies such as Hibernate. Spring Data provides an additional level of functionality: creating `Repository` implementations directly from interfaces and using conventions to generate queries from your method names.

29.1 Configure a DataSource

Java’s `javax.sql.DataSource` interface provides a standard method of working with database connections. Traditionally, a 'DataSource' uses a `URL` along with some credentials to establish a database connection.

> **Tip**
>
> See the “How-to” section for more advanced examples, typically to take full control over the configuration of the DataSource.

Embedded Database Support

It is often convenient to develop applications by using an in-memory embedded database. Obviously, in-memory databases do not provide persistent storage. You need to populate your database when your application starts and be prepared to throw away data when your application ends.

> **Tip**
>
> The “How-to” section includes a section on how to initialize a database.

Spring Boot can auto-configure embedded H2, HSQL, and Derby databases. You need not provide any connection URLs. You need only include a build dependency to the embedded database that you want to use.

> **Note**
>
> If you are using this feature in your tests, you may notice that the same database is reused by your whole test suite regardless of the number of application contexts that you use. If you want to make sure that each context has a separate embedded database, you should set `spring.datasource.generate-unique-name` to `true`.

For example, the typical POM dependencies would be as follows:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
 <groupId>org.hsqldb</groupId>
 <artifactId>hsqldb</artifactId>
 <scope>runtime</scope>
</dependency>
```

Note

You need a dependency on `spring-jdbc` for an embedded database to be auto-configured. In this example, it is pulled in transitively through `spring-boot-starter-data-jpa`.

Tip

If, for whatever reason, you do configure the connection URL for an embedded database, take care to ensure that the database's automatic shutdown is disabled. If you use H2, you should use `DB_CLOSE_ON_EXIT=FALSE` to do so. If you use HSQLDB, you should ensure that `shutdown=true` is not used. Disabling the database's automatic shutdown lets Spring Boot control when the database is closed, thereby ensuring that it happens once access to the database is no longer needed.

Connection to a Production Database

Production database connections can also be auto-configured by using a pooling `DataSource`. Spring Boot uses the following algorithm for choosing a specific implementation:

1. We prefer HikariCP for its performance and concurrency. If HikariCP is available, we always choose it.
2. Otherwise, if the Tomcat pooling `DataSource` is available, we use it.
3. If neither HikariCP nor the Tomcat pooling datasource are available and if Commons DBCP2 is available, we use it.

If you use the `spring-boot-starter-jdbc` or `spring-boot-starter-data-jpa` "starters", you automatically get a dependency to `HikariCP`.

Note

You can bypass that algorithm completely and specify the connection pool to use by setting the `spring.datasource.type` property. This is especially important if you run your application in a Tomcat container, as `tomcat-jdbc` is provided by default.

Tip

Additional connection pools can always be configured manually. If you define your own `DataSource` bean, auto-configuration does not occur.

DataSource configuration is controlled by external configuration properties in `spring.datasource.*`. For example, you might declare the following section in `application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Note

You should at least specify the URL by setting the `spring.datasource.url` property. Otherwise, Spring Boot tries to auto-configure an embedded database.

> **Tip**
>
> You often do not need to specify the `driver-class-name`, since Spring Boot can deduce it for most databases from the `url`.

> **Note**
>
> For a pooling `DataSource` to be created, we need to be able to verify that a valid `Driver` class is available, so we check for that before doing anything. In other words, if you set `spring.datasource.driver-class-name=com.mysql.jdbc.Driver`, then that class has to be loadable.

See `DataSourceProperties` for more of the supported options. These are the standard options that work regardless of the actual implementation. It is also possible to fine-tune implementation-specific settings by using their respective prefix (`spring.datasource.hikari.*`, `spring.datasource.tomcat.*`, and `spring.datasource.dbcp2.*`). Refer to the documentation of the connection pool implementation you are using for more details.

For instance, if you use the Tomcat connection pool, you could customize many additional settings, as shown in the following example:

```
# Number of ms to wait before throwing an exception if no connection is available.
spring.datasource.tomcat.max-wait=10000

# Maximum number of active connections that can be allocated from this pool at the same time.
spring.datasource.tomcat.max-active=50

# Validate the connection before borrowing it from the pool.
spring.datasource.tomcat.test-on-borrow=true
```

Connection to a JNDI DataSource

If you deploy your Spring Boot application to an Application Server, you might want to configure and manage your DataSource by using your Application Server's built-in features and access it by using JNDI.

The `spring.datasource.jndi-name` property can be used as an alternative to the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties to access the `DataSource` from a specific JNDI location. For example, the following section in `application.properties` shows how you can access a JBoss AS defined `DataSource`:

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

29.2 Using JdbcTemplate

Spring's `JdbcTemplate` and `NamedParameterJdbcTemplate` classes are auto-configured, and you can `@Autowire` them directly into your own beans, as shown in the following example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

 private final JdbcTemplate jdbcTemplate;
```

```
@Autowired
public MyBean(JdbcTemplate jdbcTemplate) {
 this.jdbcTemplate = jdbcTemplate;
}

// ...

}
```

You can customize some properties of the template by using the `spring.jdbc.template.*` properties, as shown in the following example:

```
spring.jdbc.template.max-rows=500
```

> **Note**
>
> The `NamedParameterJdbcTemplate` reuses the same `JdbcTemplate` instance behind the scenes. If more than one `JdbcTemplate` is defined and no primary candidate exists, the `NamedParameterJdbcTemplate` is not auto-configured.

29.3 JPA and “Spring Data”

The Java Persistence API is a standard technology that lets you “map” objects to relational databases. The `spring-boot-starter-data-jpa` POM provides a quick way to get started. It provides the following key dependencies:

- Hibernate: One of the most popular JPA implementations.
- Spring Data JPA: Makes it easy to implement JPA-based repositories.
- Spring ORMs: Core ORM support from the Spring Framework.

> **Tip**
>
> We do not go into too many details of JPA or Spring Data here. You can follow the “Accessing Data with JPA” guide from spring.io and read the Spring Data JPA and Hibernate reference documentation.

Entity Classes

Traditionally, JPA “Entity” classes are specified in a `persistence.xml` file. With Spring Boot, this file is not necessary and “Entity Scanning” is used instead. By default, all packages below your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) are searched.

Any classes annotated with `@Entity`, `@Embeddable`, or `@MappedSuperclass` are considered. A typical entity class resembles the following example:

```
package com.example.myapp.domain;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class City implements Serializable {

 @Id
```

```
 @GeneratedValue
 private Long id;

 @Column(nullable = false)
 private String name;

 @Column(nullable = false)
 private String state;

 // ... additional members, often include @OneToMany mappings

 protected City() {
  // no-args constructor required by JPA spec
  // this one is protected since it shouldn't be used directly
 }

 public City(String name, String state) {
  this.name = name;
  this.country = country;
 }

 public String getName() {
  return this.name;
 }

 public String getState() {
  return this.state;
 }

 // ... etc

}
```

Tip

You can customize entity scanning locations by using the `@EntityScan` annotation. See the "Section 81.4, "Separate @Entity Definitions from Spring Configuration"" how-to.

Spring Data JPA Repositories

Spring Data JPA repositories are interfaces that you can define to access data. JPA queries are created automatically from your method names. For example, a `CityRepository` interface might declare a `findAllByState(String state)` method to find all the cities in a given state.

For more complex queries, you can annotate your method with Spring Data's `Query` annotation.

Spring Data repositories usually extend from the `Repository` or `CrudRepository` interfaces. If you use auto-configuration, repositories are searched from the package containing your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) down.

The following example shows a typical Spring Data repository interface definition:

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

 Page<City> findAll(Pageable pageable);

 City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

> **Tip**
>
> We have barely scratched the surface of Spring Data JPA. For complete details, see the Spring Data JPA reference documentation.

Creating and Dropping JPA Databases

By default, JPA databases are automatically created **only** if you use an embedded database (H2, HSQL, or Derby). You can explicitly configure JPA settings by using `spring.jpa.*` properties. For example, to create and drop tables you can add the following line to your `application.properties`:

```
spring.jpa.hibernate.ddl-auto=create-drop
```

> **Note**
>
> Hibernate's own internal property name for this (if you happen to remember it better) is `hibernate.hbm2ddl.auto`. You can set it, along with other Hibernate native properties, by using `spring.jpa.properties.*` (the prefix is stripped before adding them to the entity manager). The following line shows an example of setting JPA properties for Hibernate:

```
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```

The line in the preceding example passes a value of `true` for the `hibernate.globally_quoted_identifiers` property to the Hibernate entity manager.

By default, the DDL execution (or validation) is deferred until the `ApplicationContext` has started. There is also a `spring.jpa.generate-ddl` flag, but it is not used if Hibernate auto-configuration is active, because the `ddl-auto` settings are more fine-grained.

Open EntityManager in View

If you are running a web application, Spring Boot by default registers `OpenEntityManagerInViewInterceptor` to apply the "Open EntityManager in View" pattern, to allow for lazy loading in web views. If you do not want this behavior, you should set `spring.jpa.open-in-view` to `false` in your `application.properties`.

29.4 Using H2's Web Console

The H2 database provides a browser-based console that Spring Boot can auto-configure for you. The console is auto-configured when the following conditions are met:

- You are developing a servlet-based web application.
- `com.h2database:h2` is on the classpath.
- You are using Spring Boot's developer tools.

> **Tip**
>
> If you are not using Spring Boot's developer tools but would still like to make use of H2's console, you can configure the `spring.h2.console.enabled` property with a value of `true`.

> **Note**
>
> The H2 console is only intended for use during development, so you should take care to ensure that `spring.h2.console.enabled` is not set to `true` in production.

Changing the H2 Console's Path

By default, the console is available at `/h2-console`. You can customize the console's path by using the `spring.h2.console.path` property.

29.5 Using jOOQ

Java Object Oriented Querying (jOOQ) is a popular product from Data Geekery which generates Java code from your database and lets you build type-safe SQL queries through its fluent API. Both the commercial and open source editions can be used with Spring Boot.

Code Generation

In order to use jOOQ type-safe queries, you need to generate Java classes from your database schema. You can follow the instructions in the jOOQ user manual. If you use the `jooq-codegen-maven` plugin and you also use the `spring-boot-starter-parent` "parent POM", you can safely omit the plugin's `<version>` tag. You can also use Spring Boot-defined version variables (such as `h2.version`) to declare the plugin's database dependency. The following listing shows an example:

```
<plugin>
 <groupId>org.jooq</groupId>
 <artifactId>jooq-codegen-maven</artifactId>
 <executions>
  ...
 </executions>
 <dependencies>
  <dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <version>${h2.version}</version>
  </dependency>
 </dependencies>
 <configuration>
  <jdbc>
   <driver>org.h2.Driver</driver>
   <url>jdbc:h2:~/yourdatabase</url>
  </jdbc>
  <generator>
   ...
  </generator>
 </configuration>
</plugin>
```

Using DSLContext

The fluent API offered by jOOQ is initiated through the `org.jooq.DSLContext` interface. Spring Boot auto-configures a `DSLContext` as a Spring Bean and connects it to your application `DataSource`. To use the `DSLContext`, you can `@Autowire` it, as shown in the following example:

```
@Component
public class JooqExample implements CommandLineRunner {

 private final DSLContext create;

 @Autowired
```

```
 public JooqExample(DSLContext dslContext) {
  this.create = dslContext;
 }

}
```

> **Tip**
>
> The jOOQ manual tends to use a variable named `create` to hold the `DSLContext`.

You can then use the `DSLContext` to construct your queries, as shown in the following example:

```
public List<GregorianCalendar> authorsBornAfter1980() {
 return this.create.selectFrom(AUTHOR)
  .where(AUTHOR.DATE_OF_BIRTH.greaterThan(new GregorianCalendar(1980, 0, 1)))
  .fetch(AUTHOR.DATE_OF_BIRTH);
}
```

jOOQ SQL Dialect

Unless the `spring.jooq.sql-dialect` property has been configured, Spring Boot determines the SQL dialect to use for your datasource. If Spring Boot could not detect the dialect, it uses `DEFAULT`.

> **Note**
>
> Spring Boot can only auto-configure dialects supported by the open source version of jOOQ.

Customizing jOOQ

More advanced customizations can be achieved by defining your own `@Bean` definitions, which is used when the jOOQ `Configuration` is created. You can define beans for the following jOOQ Types:

- `ConnectionProvider`
- `TransactionProvider`
- `RecordMapperProvider`
- `RecordUnmapperProvider`
- `RecordListenerProvider`
- `ExecuteListenerProvider`
- `VisitListenerProvider`
- `TransactionListenerProvider`

You can also create your own `org.jooq.Configuration` `@Bean` if you want to take complete control of the jOOQ configuration.

30. Working with NoSQL Technologies

Spring Data provides additional projects that help you access a variety of NoSQL technologies, including: MongoDB, Neo4J, Elasticsearch, Solr, Redis, Gemfire, Cassandra, Couchbase and LDAP. Spring Boot provides auto-configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr Cassandra, Couchbase, and LDAP. You can make use of the other projects, but you must configure them yourself. Refer to the appropriate reference documentation at projects.spring.io/spring-data.

30.1 Redis

Redis is a cache, message broker, and richly-featured key-value store. Spring Boot offers basic auto-configuration for the Lettuce and Jedis client libraries and the abstractions on top of them provided by Spring Data Redis.

There is a `spring-boot-starter-data-redis` "Starter" for collecting the dependencies in a convenient way. By default, it uses Lettuce. That starter handles both traditional and reactive applications.

> **Tip**
>
> we also provide a `spring-boot-starter-data-redis-reactive` "Starter" for consistency with the other stores with reactive support.

Connecting to Redis

You can inject an auto-configured `RedisConnectionFactory`, `StringRedisTemplate`, or vanilla `RedisTemplate` instance as you would any other Spring Bean. By default, the instance tries to connect to a Redis server at `localhost:6379`. The following listing shows an example of such a bean:

```
@Component
public class MyBean {

 private StringRedisTemplate template;

 @Autowired
 public MyBean(StringRedisTemplate template) {
  this.template = template;
 }

 // ...

}
```

> **Tip**
>
> You can also register an arbitrary number of beans that implement `LettuceClientConfigurationBuilderCustomizer` for more advanced customizations. If you use Jedis, `JedisClientConfigurationBuilderCustomizer` is also available.

If you add your own `@Bean` of any of the auto-configured types, it replaces the default (except in the case of `RedisTemplate`, when the exclusion is based on the bean name, `redisTemplate`, not its type). By default, if `commons-pool2` is on the classpath, you get a pooled connection factory.

30.2 MongoDB

MongoDB is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several conveniences for working with MongoDB, including the `spring-boot-starter-data-mongodb` and `spring-boot-starter-data-mongodb-reactive` "Starters".

Connecting to a MongoDB Database

To access Mongo databases, you can inject an auto-configured `org.springframework.data.mongodb.MongoDbFactory`. By default, the instance tries to connect to a MongoDB server at `mongodb://localhost/test` The following example shows how to connect to a MongoDB database:

```
import org.springframework.data.mongodb.MongoDbFactory;
import com.mongodb.DB;

@Component
public class MyBean {

 private final MongoDbFactory mongo;

 @Autowired
 public MyBean(MongoDbFactory mongo) {
  this.mongo = mongo;
 }

 // ...

 public void example() {
  DB db = mongo.getDb();
  // ...
 }

}
```

You can set the `spring.data.mongodb.uri` property to change the URL and configure additional settings such as the *replica set*, as shown in the following example:

```
spring.data.mongodb.uri=mongodb://user:secret@mongo1.example.com:12345,mongo2.example.com:23456/test
```

Alternatively, as long as you use Mongo 2.x, you can specify a `host`/`port`. For example, you might declare the following settings in your `application.properties`:

```
spring.data.mongodb.host=mongoserver
spring.data.mongodb.port=27017
```

> **Note**
>
> If you use the Mongo 3.0 Java driver, `spring.data.mongodb.host` and `spring.data.mongodb.port` are not supported. In such cases, `spring.data.mongodb.uri` should be used to provide all of the configuration.

> **Tip**
>
> If `spring.data.mongodb.port` is not specified, the default of `27017` is used. You could delete this line from the example shown earlier.

> **Tip**
>
> If you do not use Spring Data Mongo, you can inject `com.mongodb.MongoClient` beans instead of using `MongoDbFactory`. If you want to take complete control of establishing the MongoDB connection, you can also declare your own `MongoDbFactory` or `MongoClient` bean.

> **Note**
>
> If you are using the reactive driver, Netty is required for SSL. The auto-configuration configures this factory automatically if Netty is available and the factory to use hasn't been customized already.

MongoTemplate

Spring Data MongoDB provides a `MongoTemplate` class that is very similar in its design to Spring's `JdbcTemplate`. As with `JdbcTemplate`, Spring Boot auto-configures a bean for you to inject the template, as follows:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

 private final MongoTemplate mongoTemplate;

 @Autowired
 public MyBean(MongoTemplate mongoTemplate) {
  this.mongoTemplate = mongoTemplate;
 }

 // ...

}
```

See the `MongoOperations` Javadoc for complete details.

Spring Data MongoDB Repositories

Spring Data includes repository support for MongoDB. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed automatically, based on method names.

In fact, both Spring Data JPA and Spring Data MongoDB share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a Mongo data class rather than a JPA `@Entity`, it works in the same way, as shown in the following example:

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

 Page<City> findAll(Pageable pageable);

 City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

> **Tip**
>
> You can customize document scanning locations by using the `@EntityScan` annotation.

> **Tip**
>
> For complete details of Spring Data MongoDB, including its rich object mapping technologies, refer to its reference documentation.

Embedded Mongo

Spring Boot offers auto-configuration for Embedded Mongo. To use it in your Spring Boot application, add a dependency on `de.flapdoodle.embed:de.flapdoodle.embed.mongo`.

The port that Mongo listens on can be configured by setting the `spring.data.mongodb.port` property. To use a randomly allocated free port, use a value of 0. The `MongoClient` created by `MongoAutoConfiguration` is automatically configured to use the randomly allocated port.

> **Note**
>
> If you do not configure a custom port, the embedded support uses a random port (rather than 27017) by default.

If you have SLF4J on the classpath, the output produced by Mongo is automatically routed to a logger named `org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo`.

You can declare your own `IMongodConfig` and `IRuntimeConfig` beans to take control of the Mongo instance's configuration and logging routing.

30.3 Neo4j

Neo4j is an open-source NoSQL graph database that uses a rich data model of nodes related by first class relationships, which is better suited for connected big data than traditional rdbms approaches. Spring Boot offers several conveniences for working with Neo4j, including the `spring-boot-starter-data-neo4j` "Starter".

Connecting to a Neo4j Database

You can inject an auto-configured `Neo4jSession`, `Session`, or `Neo4jOperations` instance as you would any other Spring Bean. By default, the instance tries to connect to a Neo4j server at `localhost:7474`. The following example shows how to inject a Neo4j bean:

```
@Component
public class MyBean {

 private final Neo4jTemplate neo4jTemplate;

 @Autowired
 public MyBean(Neo4jTemplate neo4jTemplate) {
  this.neo4jTemplate = neo4jTemplate;
 }

 // ...

}
```

You can take full control of the configuration by adding a `org.neo4j.ogm.config.Configuration` `@Bean` of your own. Also, adding a `@Bean` of type `Neo4jOperations` disables the auto-configuration.

You can configure the user and credentials to use by setting the `spring.data.neo4j.*` properties, as shown in the following example:

```
spring.data.neo4j.uri=http://my-server:7474
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=secret
```

Using the Embedded Mode

If you add `org.neo4j:neo4j-ogm-embedded-driver` to the dependencies of your application, Spring Boot automatically configures an in-process embedded instance of Neo4j that does not persist any data when your application shuts down. You can explicitly disable that mode by setting `spring.data.neo4j.embedded.enabled=false`. You can also enable persistence for the embedded mode by providing a path to a database file, as shown in the following example:

```
spring.data.neo4j.uri=file://var/tmp/graph.db
```

> **Note**
>
> The Neo4j OGM embedded driver does not provide the Neo4j kernel. Users are expected to provide this dependency manually. See the documentation for more details.

Neo4jSession

By default, if you are running a web application, the session is bound to the thread for the entire processing of the request (that is, it uses the "Open Session in View" pattern). If you do not want this behavior, add the following line to your `application.properties` file:

```
spring.data.neo4j.open-in-view=false
```

Spring Data Neo4j Repositories

Spring Data includes repository support for Neo4j.

In fact, both Spring Data JPA and Spring Data Neo4j share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a Neo4j OGM `@NodeEntity` rather than a JPA `@Entity`, it works in the same way.

> **Tip**
>
> You can customize entity scanning locations by using the `@EntityScan` annotation.

To enable repository support (and optionally support for `@Transactional`), add the following two annotations to your Spring configuration:

```
@EnableNeo4jRepositories(basePackages = "com.example.myapp.repository")
@EnableTransactionManagement
```

Repository Example

The following example shows an interface definition for a Neo4j repository:

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends GraphRepository<City> {

 Page<City> findAll(Pageable pageable);

 City findByNameAndCountry(String name, String country);

}
```

> **Tip**
>
> For complete details of Spring Data Neo4j, including its rich object mapping technologies, refer to the reference documentation.

30.4 Gemfire

Spring Data Gemfire provides convenient Spring-friendly tools for accessing the Pivotal Gemfire data management platform. There is a `spring-boot-starter-data-gemfire` "Starter" for collecting the dependencies in a convenient way. There is currently no auto-configuration support for Gemfire, but you can enable Spring Data Repositories with a single annotation: `@EnableGemfireRepositories`.

30.5 Solr

Apache Solr is a search engine. Spring Boot offers basic auto-configuration for the Solr 5 client library and the abstractions on top of it provided by Spring Data Solr. There is a `spring-boot-starter-data-solr` "Starter" for collecting the dependencies in a convenient way.

Connecting to Solr

You can inject an auto-configured `SolrClient` instance as you would any other Spring bean. By default, the instance tries to connect to a server at `localhost:8983/solr`. The following example shows how to inject a Solr bean:

```
@Component
public class MyBean {

 private SolrClient solr;

 @Autowired
 public MyBean(SolrClient solr) {
  this.solr = solr;
 }

 // ...

}
```

If you add your own `@Bean` of type `SolrClient`, it replaces the default.

Spring Data Solr Repositories

Spring Data includes repository support for Apache Solr. As with the JPA repositories discussed earlier, the basic principle is that queries are automatically constructed for \ you based on method names.

In fact, both Spring Data JPA and Spring Data Solr share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a `@SolrDocument` class rather than a JPA `@Entity`, it works in the same way.

> **Tip**
>
> For complete details of Spring Data Solr, refer to the reference documentation.

30.6 Elasticsearch

Elasticsearch is an open source, distributed, RESTful search and analytics engine. Spring Boot offers basic auto-configuration for Elasticsearch.

Spring Boot supports several HTTP clients:

- The official Java "Low Level" and "High Level" REST clients
- Jest

The transport client is still being used by Spring Data Elasticsearch, which you can start using with the `spring-boot-starter-data-elasticsearch` “Starter”.

Connecting to Elasticsearch by REST clients

Elasticsearch ships two different REST clients that you can use to query a cluster: the "Low Level" client and the "High Level" client.

If you have the `org.elasticsearch.client:elasticsearch-rest-client` dependency on the classpath, Spring Boot will auto-configure and register a `RestClient` bean that by default targets `localhost:9200`. You can further tune how `RestClient` is configured, as shown in the following example:

```
spring.elasticsearch.rest.uris=http://search.example.com:9200
spring.elasticsearch.rest.username=user
spring.elasticsearch.rest.password=secret
```

You can also register an arbitrary number of beans that implement `RestClientBuilderCustomizer` for more advanced customizations. To take full control over the registration, define a `RestClient` bean.

If you have the `org.elasticsearch.client:elasticsearch-rest-high-level-client` dependency on the classpath, Spring Boot will auto-configure a `RestHighLevelClient`, which wraps any existing `RestClient` bean, reusing its HTTP configuration.

Connecting to Elasticsearch by Using Jest

If you have `Jest` on the classpath, you can inject an auto-configured `JestClient` that by default targets `localhost:9200`. You can further tune how the client is configured, as shown in the following example:

```
spring.elasticsearch.jest.uris=http://search.example.com:9200
spring.elasticsearch.jest.read-timeout=10000
spring.elasticsearch.jest.username=user
spring.elasticsearch.jest.password=secret
```

You can also register an arbitrary number of beans that implement `HttpClientConfigBuilderCustomizer` for more advanced customizations. The following example tunes additional HTTP settings:

```
static class HttpSettingsCustomizer implements HttpClientConfigBuilderCustomizer {

 @Override
 public void customize(HttpClientConfig.Builder builder) {
  builder.maxTotalConnection(100).defaultMaxTotalConnectionPerRoute(5);
 }

}
```

To take full control over the registration, define a `JestClient` bean.

Connecting to Elasticsearch by Using Spring Data

To connect to Elasticsearch, you must provide the address of one or more cluster nodes. The address can be specified by setting the `spring.data.elasticsearch.cluster-nodes` property to a comma-separated `host:port` list. With this configuration in place, an `ElasticsearchTemplate` or `TransportClient` can be injected like any other Spring bean, as shown in the following example:

```
spring.data.elasticsearch.cluster-nodes=localhost:9300
```

```
@Component
public class MyBean {

 private final ElasticsearchTemplate template;

 public MyBean(ElasticsearchTemplate template) {
  this.template = template;
 }

 // ...

}
```

If you add your own `ElasticsearchTemplate` or `TransportClient` `@Bean`, it replaces the default.

Spring Data Elasticsearch Repositories

Spring Data includes repository support for Elasticsearch. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed for you automatically based on method names.

In fact, both Spring Data JPA and Spring Data Elasticsearch share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now an Elasticsearch `@Document` class rather than a JPA `@Entity`, it works in the same way.

> **Tip**
>
> For complete details of Spring Data Elasticsearch, refer to the reference documentation.

30.7 Cassandra

Cassandra is an open source, distributed database management system designed to handle large amounts of data across many commodity servers. Spring Boot offers auto-configuration for Cassandra and the abstractions on top of it provided by Spring Data Cassandra. There is a `spring-boot-starter-data-cassandra` “Starter” for collecting the dependencies in a convenient way.

Connecting to Cassandra

You can inject an auto-configured `CassandraTemplate` or a Cassandra `Session` instance as you would with any other Spring Bean. The `spring.data.cassandra.*` properties can be used to customize the connection. Generally, you provide `keyspace-name` and `contact-points` properties, as shown in the following example:

```
spring.data.cassandra.keyspace-name=mykeyspace
spring.data.cassandra.contact-points=cassandrahost1,cassandrahost2
```

The following code listing shows how to inject a Cassandra bean:

```
@Component
public class MyBean {

 private CassandraTemplate template;

 @Autowired
 public MyBean(CassandraTemplate template) {
  this.template = template;
 }

 // ...

}
```

If you add your own `@Bean` of type `CassandraTemplate`, it replaces the default.

Spring Data Cassandra Repositories

Spring Data includes basic repository support for Cassandra. Currently, this is more limited than the JPA repositories discussed earlier and needs to annotate finder methods with `@Query`.

> **Tip**
>
> For complete details of Spring Data Cassandra, refer to the reference documentation.

30.8 Couchbase

Couchbase is an open-source, distributed, multi-model NoSQL document-oriented database that is optimized for interactive applications. Spring Boot offers auto-configuration for Couchbase and the abstractions on top of it provided by Spring Data Couchbase. There are `spring-boot-starter-data-couchbase` and `spring-boot-starter-data-couchbase-reactive` “Starters” for collecting the dependencies in a convenient way.

Connecting to Couchbase

You can get a `Bucket` and `Cluster` by adding the Couchbase SDK and some configuration. The `spring.couchbase.*` properties can be used to customize the connection. Generally, you provide the bootstrap hosts, bucket name, and password, as shown in the following example:

```
spring.couchbase.bootstrap-hosts=my-host-1,192.168.1.123
spring.couchbase.bucket.name=my-bucket
spring.couchbase.bucket.password=secret
```

> **Tip**
>
> You need to provide *at least* the bootstrap host(s), in which case the bucket name is `default` and the password is an empty String. Alternatively, you can define your own `org.springframework.data.couchbase.config.CouchbaseConfigurer @Bean` to take control over the whole configuration.

It is also possible to customize some of the `CouchbaseEnvironment` settings. For instance, the following configuration changes the timeout to use to open a new `Bucket` and enables SSL support:

```
spring.couchbase.env.timeouts.connect=3000
spring.couchbase.env.ssl.key-store=/location/of/keystore.jks
spring.couchbase.env.ssl.key-store-password=secret
```

Check the `spring.couchbase.env.*` properties for more details.

Spring Data Couchbase Repositories

Spring Data includes repository support for Couchbase. For complete details of Spring Data Couchbase, refer to the reference documentation.

You can inject an auto-configured `CouchbaseTemplate` instance as you would with any other Spring Bean, provided a *default* `CouchbaseConfigurer` is available (which happens when you enable Couchbase support, as explained earlier).

The following examples shows how to inject a Couchbase bean:

```
@Component
public class MyBean {

 private final CouchbaseTemplate template;

 @Autowired
 public MyBean(CouchbaseTemplate template) {
  this.template = template;
 }

 // ...

}
```

There are a few beans that you can define in your own configuration to override those provided by the auto-configuration:

- A `CouchbaseTemplate @Bean` with a name of `couchbaseTemplate`.
- An `IndexManager @Bean` with a name of `couchbaseIndexManager`.
- A `CustomConversions @Bean` with a name of `couchbaseCustomConversions`.

To avoid hard-coding those names in your own config, you can reuse `BeanNames` provided by Spring Data Couchbase. For instance, you can customize the converters to use, as follows:

```
@Configuration
public class SomeConfiguration {

 @Bean(BeanNames.COUCHBASE_CUSTOM_CONVERSIONS)
 public CustomConversions myCustomConversions() {
  return new CustomConversions(...);
 }

 // ...
```

```
}
```

> **Tip**
>
> If you want to fully bypass the auto-configuration for Spring Data Couchbase, provide your own implementation of `org.springframework.data.couchbase.config.AbstractCouchbaseDataConfiguration`.

30.9 LDAP

LDAP (Lightweight Directory Access Protocol) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an IP network. Spring Boot offers auto-configuration for any compliant LDAP server as well as support for the embedded in-memory LDAP server from UnboundID.

LDAP abstractions are provided by Spring Data LDAP. There is a `spring-boot-starter-data-ldap` "Starter" for collecting the dependencies in a convenient way.

Connecting to an LDAP Server

To connect to an LDAP server, make sure you declare a dependency on the `spring-boot-starter-data-ldap` "Starter" or `spring-ldap-core` and then declare the URLs of your server in your application.properties, as shown in the following example:

```
spring.ldap.urls=ldap://myserver:1235
spring.ldap.username=admin
spring.ldap.password=secret
```

If you need to customize connection settings, you can use the `spring.ldap.base` and `spring.ldap.base-environment` properties.

A `LdapContextSource` is auto-configured based on these settings. If you need to customize it, for instance to use a `PooledContextSource`, you can still inject the auto-configured `LdapContextSource`. Make sure to flag your customized `ContextSource` as `@Primary` so that the auto-configured `LdapTemplate` uses it.

Spring Data LDAP Repositories

Spring Data includes repository support for LDAP. For complete details of Spring Data LDAP, refer to the reference documentation.

You can also inject an auto-configured `LdapTemplate` instance as you would with any other Spring Bean, as shown in the following example:

```
@Component
public class MyBean {

 private final LdapTemplate template;

 @Autowired
 public MyBean(LdapTemplate template) {
  this.template = template;
 }

 // ...

}
```

Embedded In-memory LDAP Server

For testing purposes, Spring Boot supports auto-configuration of an in-memory LDAP server from UnboundID. To configure the server, add a dependency to `com.unboundid:unboundid-ldapsdk` and declare a `base-dn` property, as follows:

```
spring.ldap.embedded.base-dn=dc=spring,dc=io
```

> **Note**
>
> It is possible to define multiple base-dn values, however, since distinguished names usually contain commas, they must be defined using the correct notation.
>
> In yaml files, you can use the yaml list notation:
>
> ```
> spring.ldap.embedded.base-dn:
>   - dc=spring,dc=io
>   - dc=pivotal,dc=io
> ```
>
> In properties files, you must include the index as part of the property name:
>
> ```
> spring.ldap.embedded.base-dn[0]=dc=spring,dc=io
> spring.ldap.embedded.base-dn[1]=dc=pivotal,dc=io
> ```

By default, the server starts on a random port and triggers the regular LDAP support. There is no need to specify a `spring.ldap.urls` property.

If there is a `schema.ldif` file on your classpath, it is used to initialize the server. If you want to load the initialization script from a different resource, you can also use the `spring.ldap.embedded.ldif` property.

By default, a standard schema is used to validate `LDIF` files. You can turn off validation altogether by setting the `spring.ldap.embedded.validation.enabled` property. If you have custom attributes, you can use `spring.ldap.embedded.validation.schema` to define your custom attribute types or object classes.

30.10 InfluxDB

InfluxDB is an open-source time series database optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet-of-Things sensor data, and real-time analytics.

Connecting to InfluxDB

Spring Boot auto-configures an `InfluxDB` instance, provided the `influxdb-java` client is on the classpath and the URL of the database is set, as shown in the following example:

```
spring.influx.url=http://172.0.0.1:8086
```

If the connection to InfluxDB requires a user and password, you can set the `spring.influx.user` and `spring.influx.password` properties accordingly.

InfluxDB relies on OkHttp. If you need to tune the http client `InfluxDB` uses behind the scenes, you can register an `OkHttpClient.Builder` bean.

31. Caching

The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to methods, thus reducing the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker. Spring Boot auto-configures the cache infrastructure as long as caching support is enabled via the `@EnableCaching` annotation.

> **Note**
>
> Check the relevant section of the Spring Framework reference for more details.

In a nutshell, adding caching to an operation of your service is as easy as adding the relevant annotation to its method, as shown in the following example:

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Component;

@Component
public class MathService {

 @Cacheable("piDecimals")
 public int computePiDecimal(int i) {
  // ...
 }

}
```

This example demonstrates the use of caching on a potentially costly operation. Before invoking `computePiDecimal`, the abstraction looks for an entry in the `piDecimals` cache that matches the `i` argument. If an entry is found, the content in the cache is immediately returned to the caller, and the method is not invoked. Otherwise, the method is invoked, and the cache is updated before returning the value.

> **Caution**
>
> You can also use the standard JSR-107 (JCache) annotations (such as `@CacheResult`) transparently. However, we strongly advise you to not mix and match the Spring Cache and JCache annotations.

If you do not add any specific cache library, Spring Boot auto-configures a simple provider that uses concurrent maps in memory. When a cache is required (such as `piDecimals` in the preceding example), this provider creates it for you. The simple provider is not really recommended for production usage, but it is great for getting started and making sure that you understand the features. When you have made up your mind about the cache provider to use, please make sure to read its documentation to figure out how to configure the caches that your application uses. Nearly all providers require you to explicitly configure every cache that you use in the application. Some offer a way to customize the default caches defined by the `spring.cache.cache-names` property.

> **Tip**
>
> It is also possible to transparently update or evict data from the cache.

31.1 Supported Cache Providers

The cache abstraction does not provide an actual store and relies on abstraction materialized by the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces.

If you have not defined a bean of type `CacheManager` or a `CacheResolver` named `cacheResolver` (see `CachingConfigurer`), Spring Boot tries to detect the following providers (in the indicated order):

1. Generic
2. JCache (JSR-107) (EhCache 3, Hazelcast, Infinispan, and others)
3. EhCache 2.x
4. Hazelcast
5. Infinispan
6. Couchbase
7. Redis
8. Caffeine
9. Simple

> **Tip**
>
> It is also possible to *force* a particular cache provider by setting the `spring.cache.type` property. Use this property if you need to disable caching altogether in certain environment (such as tests).

> **Tip**
>
> Use the `spring-boot-starter-cache` "Starter" to quickly add basic caching dependencies. The starter brings in `spring-context-support`. If you add dependencies manually, you must include `spring-context-support` in order to use the JCache, EhCache 2.x, or Guava support.

If the `CacheManager` is auto-configured by Spring Boot, you can further tune its configuration before it is fully initialized by exposing a bean that implements the `CacheManagerCustomizer` interface. The following example sets a flag to say that null values should be passed down to the underlying map:

```
@Bean
public CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer() {
 return new CacheManagerCustomizer<ConcurrentMapCacheManager>() {
  @Override
  public void customize(ConcurrentMapCacheManager cacheManager) {
   cacheManager.setAllowNullValues(false);
  }
 };
}
```

> **Note**
>
> In the preceding example, an auto-configured `ConcurrentMapCacheManager` is expected. If that is not the case (either you provided your own config or a different cache provider was auto-

configured), the customizer is not invoked at all. You can have as many customizers as you want, and you can also order them by using `@Order` or `Ordered`.

Generic

Generic caching is used if the context defines *at least* one `org.springframework.cache.Cache` bean. A `CacheManager` wrapping all beans of that type is created.

JCache (JSR-107)

JCache is bootstrapped through the presence of a `javax.cache.spi.CachingProvider` on the classpath (that is, a JSR-107 compliant caching library exists on the classpath), and the `JCacheCacheManager` is provided by the `spring-boot-starter-cache` “Starter”. Various compliant libraries are available, and Spring Boot provides dependency management for Ehcache 3, Hazelcast, and Infinispan. Any other compliant library can be added as well.

It might happen that more than one provider is present, in which case the provider must be explicitly specified. Even if the JSR-107 standard does not enforce a standardized way to define the location of the configuration file, Spring Boot does its best to accommodate setting a cache with implementation details, as shown in the following example:

```
   # Only necessary if more than one provider is present
spring.cache.jcache.provider=com.acme.MyCachingProvider
spring.cache.jcache.config=classpath:acme.xml
```

Note

When a cache library offers both a native implementation and JSR-107 support, Spring Boot prefers the JSR-107 support, so that the same features are available if you switch to a different JSR-107 implementation.

Tip

Spring Boot has general support for Hazelcast. If a single `HazelcastInstance` is available, it is automatically reused for the `CacheManager` as well, unless the `spring.cache.jcache.config` property is specified.

There are two ways to customize the underlying `javax.cache.cacheManager`:

- Caches can be created on startup by setting the `spring.cache.cache-names` property. If a custom `javax.cache.configuration.Configuration` bean is defined, it is used to customize them.
- `org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer` beans are invoked with the reference of the `CacheManager` for full customization.

Tip

If a standard `javax.cache.CacheManager` bean is defined, it is wrapped automatically in an `org.springframework.cache.CacheManager` implementation that the abstraction expects. No further customization is applied to it.

EhCache 2.x

EhCache 2.x is used if a file named `ehcache.xml` can be found at the root of the classpath. If EhCache 2.x is found, the `EhCacheCacheManager` provided by the `spring-boot-starter-cache` "Starter" is used to bootstrap the cache manager. An alternate configuration file can be provided as well, as shown in the following example:

```
spring.cache.ehcache.config=classpath:config/another-config.xml
```

Hazelcast

Spring Boot has general support for Hazelcast. If a `HazelcastInstance` has been auto-configured, it is automatically wrapped in a `CacheManager`.

Infinispan

Infinispan has no default configuration file location, so it must be specified explicitly. Otherwise, the default bootstrap is used.

```
spring.cache.infinispan.config=infinispan.xml
```

Caches can be created on startup by setting the `spring.cache.cache-names` property. If a custom `ConfigurationBuilder` bean is defined, it is used to customize the caches.

> **Note**
>
> The support of Infinispan in Spring Boot is restricted to the embedded mode and is quite basic. If you want more options, you should use the official Infinispan Spring Boot starter instead. See Infinispan's documentation for more details.

Couchbase

If the Couchbase Java client and the `couchbase-spring-cache` implementation are available and Couchbase is configured, a `CouchbaseCacheManager` is auto-configured. It is also possible to create additional caches on startup by setting the `spring.cache.cache-names` property. These caches operate on the `Bucket` that was auto-configured. You can *also* create additional caches on another `Bucket` by using the customizer. Assume you need two caches (`cache1` and `cache2`) on the "main" `Bucket` and one (`cache3`) cache with a custom time to live of 2 seconds on the "another" `Bucket`. You can create the first two caches through configuration, as follows:

```
spring.cache.cache-names=cache1,cache2
```

Then you can define a `@Configuration` class to configure the extra `Bucket` and the `cache3` cache, as follows:

```
@Configuration
public class CouchbaseCacheConfiguration {

 private final Cluster cluster;

 public CouchbaseCacheConfiguration(Cluster cluster) {
  this.cluster = cluster;
 }

 @Bean
 public Bucket anotherBucket() {
```

```
  return this.cluster.openBucket("another", "secret");
 }

 @Bean
 public CacheManagerCustomizer<CouchbaseCacheManager> cacheManagerCustomizer() {
  return c -> {
   c.prepareCache("cache3", CacheBuilder.newInstance(anotherBucket())
     .withExpiration(2));
  };
 }

}
```

This sample configuration reuses the `Cluster` that was created through auto-configuration.

Redis

If Redis is available and configured, a `RedisCacheManager` is auto-configured. It is possible to create additional caches on startup by setting the `spring.cache.cache-names` property and cache defaults can be configured by using `spring.cache.redis.*` properties. For instance, the following configuration creates `cache1` and `cache2` caches with a *time to live* of 10 minutes:

```
spring.cache.cache-names=cache1,cache2
spring.cache.redis.time-to-live=600000
```

> **Note**
>
> By default, a key prefix is added so that, if two separate caches use the same key, Redis does not have overlapping keys and cannot return invalid values. We strongly recommend keeping this setting enabled if you create your own `RedisCacheManager`.

> **Tip**
>
> You can take full control of the configuration by adding a `RedisCacheConfiguration` `@Bean` of your own. This can be useful if you're looking for customizing the serialization strategy.

Caffeine

Caffeine is a Java 8 rewrite of Guava's cache that supersedes support for Guava. If Caffeine is present, a `CaffeineCacheManager` (provided by the `spring-boot-starter-cache` "Starter") is auto-configured. Caches can be created on startup by setting the `spring.cache.cache-names` property and can be customized by one of the following (in the indicated order):

1. A cache spec defined by `spring.cache.caffeine.spec`
2. A `com.github.benmanes.caffeine.cache.CaffeineSpec` bean is defined
3. A `com.github.benmanes.caffeine.cache.Caffeine` bean is defined

For instance, the following configuration creates `cache1` and `cache2` caches with a maximum size of 500 and a *time to live* of 10 minutes

```
spring.cache.cache-names=cache1,cache2
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s
```

If a `com.github.benmanes.caffeine.cache.CacheLoader` bean is defined, it is automatically associated to the `CaffeineCacheManager`. Since the `CacheLoader` is going to be associated with

all caches managed by the cache manager, it must be defined as `CacheLoader<Object, Object>`. The auto-configuration ignores any other generic type.

Simple

If none of the other providers can be found, a simple implementation using a `ConcurrentHashMap` as the cache store is configured. This is the default if no caching library is present in your application. By default, caches are created as needed, but you can restrict the list of available caches by setting the `cache-names` property. For instance, if you want only `cache1` and `cache2` caches, set the `cache-names` property as follows:

```
spring.cache.cache-names=cache1,cache2
```

If you do so and your application uses a cache not listed, then it fails at runtime when the cache is needed, but not on startup. This is similar to the way the "real" cache providers behave if you use an undeclared cache.

None

When `@EnableCaching` is present in your configuration, a suitable cache configuration is expected as well. If you need to disable caching altogether in certain environments, force the cache type to `none` to use a no-op implementation, as shown in the following example:

```
spring.cache.type=none
```

32. Messaging

The Spring Framework provides extensive support for integrating with messaging systems, from simplified use of the JMS API using `JmsTemplate` to a complete infrastructure to receive messages asynchronously. Spring AMQP provides a similar feature set for the Advanced Message Queuing Protocol. Spring Boot also provides auto-configuration options for `RabbitTemplate` and RabbitMQ. Spring WebSocket natively includes support for STOMP messaging, and Spring Boot has support for that through starters and a small amount of auto-configuration. Spring Boot also has support for Apache Kafka.

32.1 JMS

The `javax.jms.ConnectionFactory` interface provides a standard method of creating a `javax.jms.Connection` for interacting with a JMS broker. Although Spring needs a `ConnectionFactory` to work with JMS, you generally need not use it directly yourself and can instead rely on higher level messaging abstractions. (See the relevant section of the Spring Framework reference documentation for details.) Spring Boot also auto-configures the necessary infrastructure to send and receive messages.

ActiveMQ Support

When ActiveMQ is available on the classpath, Spring Boot can also configure a `ConnectionFactory`. If the broker is present, an embedded broker is automatically started and configured (provided no broker URL is specified through configuration).

> **Note**
>
> If you use `spring-boot-starter-activemq`, the necessary dependencies to connect or embed an ActiveMQ instance are provided, as is the Spring infrastructure to integrate with JMS.

ActiveMQ configuration is controlled by external configuration properties in `spring.activemq.*`. For example, you might declare the following section in `application.properties`:

```
spring.activemq.broker-url=tcp://192.168.1.210:9876
spring.activemq.user=admin
spring.activemq.password=secret
```

By default, a `CachingConnectionFactory` wraps the native `ConnectionFactory` with sensible settings that you can control by external configuration properties in `spring.jms.*`:

```
spring.jms.cache.session-cache-size=5
```

If you'd rather use native pooling, you can do so by adding a dependency to `org.apache.activemq:activemq-jms-pool` and configuring the `PooledConnectionFactory` accordingly, as shown in the following example:

```
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=50
```

> **Tip**
>
> See `ActiveMQProperties` for more of the supported options. You can also register an arbitrary number of beans that implement `ActiveMQConnectionFactoryCustomizer` for more advanced customizations.

By default, ActiveMQ creates a destination if it does not yet exist so that destinations are resolved against their provided names.

Artemis Support

Spring Boot can auto-configure a `ConnectionFactory` when it detects that Artemis is available on the classpath. If the broker is present, an embedded broker is automatically started and configured (unless the mode property has been explicitly set). The supported modes are `embedded` (to make explicit that an embedded broker is required and that an error should occur if the broker is not available on the classpath) and `native` (to connect to a broker using the `netty` transport protocol). When the latter is configured, Spring Boot configures a `ConnectionFactory` that connects to a broker running on the local machine with the default settings.

> **Note**
>
> If you use `spring-boot-starter-artemis`, the necessary dependencies to connect to an existing Artemis instance are provided, as well as the Spring infrastructure to integrate with JMS. Adding `org.apache.activemq:artemis-jms-server` to your application lets you use embedded mode.

Artemis configuration is controlled by external configuration properties in `spring.artemis.*`. For example, you might declare the following section in `application.properties`:

```
spring.artemis.mode=native
spring.artemis.host=192.168.1.210
spring.artemis.port=9876
spring.artemis.user=admin
spring.artemis.password=secret
```

When embedding the broker, you can choose if you want to enable persistence and list the destinations that should be made available. These can be specified as a comma-separated list to create them with the default options, or you can define bean(s) of type `org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration` or `org.apache.activemq.artemis.jms.server.config.TopicConfiguration`, for advanced queue and topic configurations, respectively.

By default, a `CachingConnectionFactory` wraps the native `ConnectionFactory` with sensible settings that you can control by external configuration properties in `spring.jms.*`:

```
spring.jms.cache.session-cache-size=5
```

If you'd rather use native pooling, you can do so by adding a dependency to `org.apache.activemq:activemq-jms-pool` and configuring the `PooledConnectionFactory` accordingly, as shown in the following example:

```
spring.artemis.pool.enabled=true
spring.artemis.pool.max-connections=50
```

See `ArtemisProperties` for more supported options.

No JNDI lookup is involved, and destinations are resolved against their names, using either the `name` attribute in the Artemis configuration or the names provided through configuration.

Using a JNDI ConnectionFactory

If you are running your application in an application server, Spring Boot tries to locate a JMS `ConnectionFactory` by using JNDI. By default, the `java:/JmsXA` and `java:/XAConnectionFactory` location are checked. You can use the `spring.jms.jndi-name` property if you need to specify an alternative location, as shown in the following example:

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

Sending a Message

Spring's `JmsTemplate` is auto-configured, and you can autowire it directly into your own beans, as shown in the following example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

 private final JmsTemplate jmsTemplate;

 @Autowired
 public MyBean(JmsTemplate jmsTemplate) {
  this.jmsTemplate = jmsTemplate;
 }

 // ...

}
```

> **Note**
>
> `JmsMessagingTemplate` can be injected in a similar manner. If a `DestinationResolver` or a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `JmsTemplate`.

Receiving a Message

When the JMS infrastructure is present, any bean can be annotated with `@JmsListener` to create a listener endpoint. If no `JmsListenerContainerFactory` has been defined, a default one is configured automatically. If a `DestinationResolver` or a `MessageConverter` beans is defined, it is associated automatically to the default factory.

By default, the default factory is transactional. If you run in an infrastructure where a `JtaTransactionManager` is present, it is associated to the listener container by default. If not, the `sessionTransacted` flag is enabled. In that latter scenario, you can associate your local data store transaction to the processing of an incoming message by adding `@Transactional` on your listener method (or a delegate thereof). This ensures that the incoming message is acknowledged, once the local transaction has completed. This also includes sending response messages that have been performed on the same JMS session.

The following component creates a listener endpoint on the `someQueue` destination:

```
@Component
public class MyBean {

 @JmsListener(destination = "someQueue")
 public void processMessage(String content) {
  // ...
 }

}
```

> **Tip**
>
> See the Javadoc of `@EnableJms` for more details.

If you need to create more `JmsListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `DefaultJmsListenerContainerFactoryConfigurer` that you can use to initialize a `DefaultJmsListenerContainerFactory` with the same settings as the one that is auto-configured.

For instance, the following example exposes another factory that uses a specific `MessageConverter`:

```
@Configuration
static class JmsConfiguration {

 @Bean
 public DefaultJmsListenerContainerFactory myFactory(
   DefaultJmsListenerContainerFactoryConfigurer configurer) {
  DefaultJmsListenerContainerFactory factory =
    new DefaultJmsListenerContainerFactory();
  configurer.configure(factory, connectionFactory());
  factory.setMessageConverter(myMessageConverter());
  return factory;
 }

}
```

Then you can use the factory in any `@JmsListener`-annotated method as follows:

```
@Component
public class MyBean {

 @JmsListener(destination = "someQueue", containerFactory="myFactory")
 public void processMessage(String content) {
  // ...
 }

}
```

32.2 AMQP

The Advanced Message Queuing Protocol (AMQP) is a platform-neutral, wire-level protocol for message-oriented middleware. The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions. Spring Boot offers several conveniences for working with AMQP through RabbitMQ, including the `spring-boot-starter-amqp` "Starter".

RabbitMQ support

RabbitMQ is a lightweight, reliable, scalable, and portable message broker based on the AMQP protocol. Spring uses `RabbitMQ` to communicate through the AMQP protocol.

RabbitMQ configuration is controlled by external configuration properties in `spring.rabbitmq.*`. For example, you might declare the following section in `application.properties`:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

If a `ConnectionNameStrategy` bean exists in the context, it will be automatically used to name connections created by the auto-configured `ConnectionFactory`. See `RabbitProperties` for more of the supported options.

> **Tip**
>
> See Understanding AMQP, the protocol used by RabbitMQ for more details.

Sending a Message

Spring's `AmqpTemplate` and `AmqpAdmin` are auto-configured, and you can autowire them directly into your own beans, as shown in the following example:

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

 private final AmqpAdmin amqpAdmin;
 private final AmqpTemplate amqpTemplate;

 @Autowired
 public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
  this.amqpAdmin = amqpAdmin;
  this.amqpTemplate = amqpTemplate;
 }

 // ...

}
```

> **Note**
>
> `RabbitMessagingTemplate` can be injected in a similar manner. If a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `AmqpTemplate`.

If necessary, any `org.springframework.amqp.core.Queue` that is defined as a bean is automatically used to declare a corresponding queue on the RabbitMQ instance.

To retry operations, you can enable retries on the `AmqpTemplate` (for example, in the event that the broker connection is lost). Retries are disabled by default.

Receiving a Message

When the Rabbit infrastructure is present, any bean can be annotated with `@RabbitListener` to create a listener endpoint. If no `RabbitListenerContainerFactory` has been defined, a default `SimpleRabbitListenerContainerFactory` is automatically configured and you can switch to a

direct container using the `spring.rabbitmq.listener.type` property. If a `MessageConverter` or a `MessageRecoverer` bean is defined, it is automatically associated with the default factory.

The following sample component creates a listener endpoint on the `someQueue` queue:

```
@Component
public class MyBean {

 @RabbitListener(queues = "someQueue")
 public void processMessage(String content) {
  // ...
 }

}
```

Tip

See the Javadoc of `@EnableRabbit` for more details.

If you need to create more `RabbitListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `SimpleRabbitListenerContainerFactoryConfigurer` and a `DirectRabbitListenerContainerFactoryConfigurer` that you can use to initialize a `SimpleRabbitListenerContainerFactory` and a `DirectRabbitListenerContainerFactory` with the same settings as the factories used by the auto-configuration.

Tip

It does not matter which container type you chose. Those two beans are exposed by the auto-configuration.

For instance, the following configuration class exposes another factory that uses a specific `MessageConverter`:

```
@Configuration
static class RabbitConfiguration {

 @Bean
 public SimpleRabbitListenerContainerFactory myFactory(
   SimpleRabbitListenerContainerFactoryConfigurer configurer) {
  SimpleRabbitListenerContainerFactory factory =
    new SimpleRabbitListenerContainerFactory();
  configurer.configure(factory, connectionFactory);
  factory.setMessageConverter(myMessageConverter());
  return factory;
 }

}
```

Then you can use the factory in any `@RabbitListener`-annotated method, as follows:

```
@Component
public class MyBean {

 @RabbitListener(queues = "someQueue", containerFactory="myFactory")
 public void processMessage(String content) {
  // ...
 }

}
```

You can enable retries to handle situations where your listener throws an exception. By default, `RejectAndDontRequeueRecoverer` is used, but you can define a `MessageRecoverer` of your own. When retries are exhausted, the message is rejected and either dropped or routed to a dead-letter exchange if the broker is configured to do so. By default, retries are disabled.

> **Important**
>
> By default, if retries are disabled and the listener throws an exception, the delivery is retried indefinitely. You can modify this behavior in two ways: Set the `defaultRequeueRejected` property to `false` so that zero re-deliveries are attempted or throw an `AmqpRejectAndDontRequeueException` to signal the message should be rejected. The latter is the mechanism used when retries are enabled and the maximum number of delivery attempts is reached.

32.3 Apache Kafka Support

Apache Kafka is supported by providing auto-configuration of the `spring-kafka` project.

Kafka configuration is controlled by external configuration properties in `spring.kafka.*`. For example, you might declare the following section in `application.properties`:

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=myGroup
```

> **Tip**
>
> To create a topic on startup, add a bean of type `NewTopic`. If the topic already exists, the bean is ignored.

See `KafkaProperties` for more supported options.

Sending a Message

Spring's `KafkaTemplate` is auto-configured, and you can autowire it directly in your own beans, as shown in the following example:

```
@Component
public class MyBean {

 private final KafkaTemplate kafkaTemplate;

 @Autowired
 public MyBean(KafkaTemplate kafkaTemplate) {
  this.kafkaTemplate = kafkaTemplate;
 }

 // ...

}
```

> **Note**
>
> If a `RecordMessageConverter` bean is defined, it is automatically associated to the auto-configured `KafkaTemplate`.

Receiving a Message

When the Apache Kafka infrastructure is present, any bean can be annotated with `@KafkaListener` to create a listener endpoint. If no `KafkaListenerContainerFactory` has been defined, a default one is automatically configured with keys defined in `spring.kafka.listener.*`. Also, if a `RecordMessageConverter` bean is defined, it is automatically associated to the default factory.

The following component creates a listener endpoint on the `someTopic` topic:

```
@Component
public class MyBean {

 @KafkaListener(topics = "someTopic")
 public void processMessage(String content) {
  // ...
 }

}
```

Additional Kafka Properties

The properties supported by auto configuration are shown in Appendix A, *Common application properties*. Note that, for the most part, these properties (hyphenated or camelCase) map directly to the Apache Kafka dotted properties. Refer to the Apache Kafka documentation for details.

The first few of these properties apply to both producers and consumers but can be specified at the producer or consumer level if you wish to use different values for each. Apache Kafka designates properties with an importance of HIGH, MEDIUM, or LOW. Spring Boot auto-configuration supports all HIGH importance properties, some selected MEDIUM and LOW properties, and any properties that do not have a default value.

Only a subset of the properties supported by Kafka are available through the `KafkaProperties` class. If you wish to configure the producer or consumer with additional properties that are not directly supported, use the following properties:

```
spring.kafka.properties.prop.one=first
spring.kafka.admin.properties.prop.two=second
spring.kafka.consumer.properties.prop.three=third
spring.kafka.producer.properties.prop.four=fourth
```

This sets the common `prop.one` Kafka property to `first` (applies to producers, consumers and admins), the `prop.two` admin property to `second`, the `prop.three` consumer property to `third` and the `prop.four` producer property to `fourth`.

You can also configure the Spring Kafka `JsonDeserializer` as follows:

```
spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.value.default.type=com.example.Invoice
spring.kafka.consumer.properties.spring.json.trusted.packages=com.example,org.acme
```

Similarly, you can disable the `JsonSerializer` default behavior of sending type information in headers:

```
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties.spring.json.add.type.headers=false
```

> **Important**
>
> Properties set in this way override any configuration item that Spring Boot explicitly supports.

33. Calling REST Services with `RestTemplate`

If you need to call remote REST services from your application, you can use the Spring Framework's `RestTemplate` class. Since `RestTemplate` instances often need to be customized before being used, Spring Boot does not provide any single auto-configured `RestTemplate` bean. It does, however, auto-configure a `RestTemplateBuilder`, which can be used to create `RestTemplate` instances when needed. The auto-configured `RestTemplateBuilder` ensures that sensible `HttpMessageConverters` are applied to `RestTemplate` instances.

The following code shows a typical example:

```
@Service
public class MyService {

 private final RestTemplate restTemplate;

 public MyService(RestTemplateBuilder restTemplateBuilder) {
  this.restTemplate = restTemplateBuilder.build();
 }

 public Details someRestCall(String name) {
  return this.restTemplate.getForObject("/{name}/details", Details.class, name);
 }

}
```

Tip

`RestTemplateBuilder` includes a number of useful methods that can be used to quickly configure a `RestTemplate`. For example, to add BASIC auth support, you can use `builder.basicAuthorization("user", "password").build()`.

33.1 RestTemplate Customization

There are three main approaches to `RestTemplate` customization, depending on how broadly you want the customizations to apply.

To make the scope of any customizations as narrow as possible, inject the auto-configured `RestTemplateBuilder` and then call its methods as required. Each method call returns a new `RestTemplateBuilder` instance, so the customizations only affect this use of the builder.

To make an application-wide, additive customization, use a `RestTemplateCustomizer` bean. All such beans are automatically registered with the auto-configured `RestTemplateBuilder` and are applied to any templates that are built with it.

The following example shows a customizer that configures the use of a proxy for all hosts except `192.168.0.5`:

```
static class ProxyCustomizer implements RestTemplateCustomizer {

 @Override
 public void customize(RestTemplate restTemplate) {
  HttpHost proxy = new HttpHost("proxy.example.com");
  HttpClient httpClient = HttpClientBuilder.create()
    .setRoutePlanner(new DefaultProxyRoutePlanner(proxy) {

     @Override
     public HttpHost determineProxy(HttpHost target,
```

```
        HttpRequest request, HttpContext context)
        throws HttpException {
      if (target.getHostName().equals("192.168.0.5")) {
       return null;
      }
      return super.determineProxy(target, request, context);
     }

    }).build();
  restTemplate.setRequestFactory(
    new HttpComponentsClientHttpRequestFactory(httpClient));
 }

}
```

Finally, the most extreme (and rarely used) option is to create your own `RestTemplateBuilder` bean. Doing so switches off the auto-configuration of a `RestTemplateBuilder` and prevents any `RestTemplateCustomizer` beans from being used.

34. Calling REST Services with `WebClient`

If you have Spring WebFlux on your classpath, you can also choose to use `WebClient` to call remote REST services. Compared to `RestTemplate`, this client has a more functional feel and is fully reactive. You can create your own client instance with the builder, `WebClient.create()`. See the relevant section on WebClient.

Spring Boot creates and pre-configures such a builder for you. For example, client HTTP codecs are configured in the same fashion as the server ones (see WebFlux HTTP codecs auto-configuration).

The following code shows a typical example:

```
@Service
public class MyService {

 private final WebClient webClient;

 public MyService(WebClient.Builder webClientBuilder) {
  this.webClient = webClientBuilder.baseUrl("http://example.org").build();
 }

 public Mono<Details> someRestCall(String name) {
  return this.webClient.get().url("/{name}/details", name)
      .retrieve().bodyToMono(Details.class);
 }

}
```

34.1 WebClient Customization

There are three main approaches to `WebClient` customization, depending on how broadly you want the customizations to apply.

To make the scope of any customizations as narrow as possible, inject the auto-configured `WebClient.Builder` and then call its methods as required. `WebClient.Builder` instances are stateful: Any change on the builder is reflected in all clients subsequently created with it. If you want to create several clients with the same builder, you can also consider cloning the builder with `WebClient.Builder other = builder.clone();`.

To make an application-wide, additive customization to all `WebClient.Builder` instances, you can declare `WebClientCustomizer` beans and change the `WebClient.Builder` locally at the point of injection.

Finally, you can fall back to the original API and use `WebClient.create()`. In that case, no auto-configuration or `WebClientCustomizer` is applied.

35. Validation

The method validation feature supported by Bean Validation 1.1 is automatically enabled as long as a JSR-303 implementation (such as Hibernate validator) is on the classpath. This lets bean methods be annotated with `javax.validation` constraints on their parameters and/or on their return value. Target classes with such annotated methods need to be annotated with the `@Validated` annotation at the type level for their methods to be searched for inline constraint annotations.

For instance, the following service triggers the validation of the first argument, making sure its size is between 8 and 10:

```
@Service
@Validated
public class MyBean {

 public Archive findByCodeAndAuthor(@Size(min = 8, max = 10) String code,
   Author author) {
  ...
 }

}
```

36. Sending Email

The Spring Framework provides an easy abstraction for sending email by using the `JavaMailSender` interface, and Spring Boot provides auto-configuration for it as well as a starter module.

> **Tip**
>
> See the reference documentation for a detailed explanation of how you can use `JavaMailSender`.

If `spring.mail.host` and the relevant libraries (as defined by `spring-boot-starter-mail`) are available, a default `JavaMailSender` is created if none exists. The sender can be further customized by configuration items from the `spring.mail` namespace. See `MailProperties` for more details.

In particular, certain default timeout values are infinite, and you may want to change that to avoid having a thread blocked by an unresponsive mail server, as shown in the following example:

```
spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=3000
spring.mail.properties.mail.smtp.writetimeout=5000
```

It is also possible to configure a `JavaMailSender` with an existing `Session` from JNDI:

```
spring.mail.jndi-name=mail/Session
```

When a `jndi-name` is set, it takes precedence over all other Session-related settings.

37. Distributed Transactions with JTA

Spring Boot supports distributed JTA transactions across multiple XA resources by using either an Atomikos or Bitronix embedded transaction manager. JTA transactions are also supported when deploying to a suitable Java EE Application Server.

When a JTA environment is detected, Spring's `JtaTransactionManager` is used to manage transactions. Auto-configured JMS, DataSource, and JPA beans are upgraded to support XA transactions. You can use standard Spring idioms, such as `@Transactional`, to participate in a distributed transaction. If you are within a JTA environment and still want to use local transactions, you can set the `spring.jta.enabled` property to `false` to disable the JTA auto-configuration.

37.1 Using an Atomikos Transaction Manager

Atomikos is a popular open source transaction manager which can be embedded into your Spring Boot application. You can use the `spring-boot-starter-jta-atomikos` Starter to pull in the appropriate Atomikos libraries. Spring Boot auto-configures Atomikos and ensures that appropriate `depends-on` settings are applied to your Spring beans for correct startup and shutdown ordering.

By default, Atomikos transaction logs are written to a `transaction-logs` directory in your application's home directory (the directory in which your application jar file resides). You can customize the location of this directory by setting a `spring.jta.log-dir` property in your `application.properties` file. Properties starting with `spring.jta.atomikos.properties` can also be used to customize the Atomikos `UserTransactionServiceImp`. See the `AtomikosProperties` Javadoc for complete details.

> **Note**
>
> To ensure that multiple transaction managers can safely coordinate the same resource managers, each Atomikos instance must be configured with a unique ID. By default, this ID is the IP address of the machine on which Atomikos is running. To ensure uniqueness in production, you should configure the `spring.jta.transaction-manager-id` property with a different value for each instance of your application.

37.2 Using a Bitronix Transaction Manager

Bitronix is a popular open-source JTA transaction manager implementation. You can use the `spring-boot-starter-jta-bitronix` starter to add the appropriate Bitronix dependencies to your project. As with Atomikos, Spring Boot automatically configures Bitronix and post-processes your beans to ensure that startup and shutdown ordering is correct.

By default, Bitronix transaction log files (`part1.btm` and `part2.btm`) are written to a `transaction-logs` directory in your application home directory. You can customize the location of this directory by setting the `spring.jta.log-dir` property. Properties starting with `spring.jta.bitronix.properties` are also bound to the `bitronix.tm.Configuration` bean, allowing for complete customization. See the Bitronix documentation for details.

> **Note**
>
> To ensure that multiple transaction managers can safely coordinate the same resource managers, each Bitronix instance must be configured with a unique ID. By default, this ID is the IP address

of the machine on which Bitronix is running. To ensure uniqueness in production, you should configure the `spring.jta.transaction-manager-id` property with a different value for each instance of your application.

37.3 Using a Narayana Transaction Manager

Narayana is a popular open source JTA transaction manager implementation supported by JBoss. You can use the `spring-boot-starter-jta-narayana` starter to add the appropriate Narayana dependencies to your project. As with Atomikos and Bitronix, Spring Boot automatically configures Narayana and post-processes your beans to ensure that startup and shutdown ordering is correct.

By default, Narayana transaction logs are written to a `transaction-logs` directory in your application home directory (the directory in which your application jar file resides). You can customize the location of this directory by setting a `spring.jta.log-dir` property in your `application.properties` file. Properties starting with `spring.jta.narayana.properties` can also be used to customize the Narayana configuration. See the `NarayanaProperties` Javadoc for complete details.

Note

To ensure that multiple transaction managers can safely coordinate the same resource managers, each Narayana instance must be configured with a unique ID. By default, this ID is set to `1`. To ensure uniqueness in production, you should configure the `spring.jta.transaction-manager-id` property with a different value for each instance of your application.

37.4 Using a Java EE Managed Transaction Manager

If you package your Spring Boot application as a `war` or `ear` file and deploy it to a Java EE application server, you can use your application server's built-in transaction manager. Spring Boot tries to auto-configure a transaction manager by looking at common JNDI locations (`java:comp/UserTransaction`, `java:comp/TransactionManager`, and so on). If you use a transaction service provided by your application server, you generally also want to ensure that all resources are managed by the server and exposed over JNDI. Spring Boot tries to auto-configure JMS by looking for a `ConnectionFactory` at the JNDI path (`java:/JmsXA` or `java:/XAConnectionFactory`), and you can use the `spring.datasource.jndi-name` property to configure your `DataSource`.

37.5 Mixing XA and Non-XA JMS Connections

When using JTA, the primary JMS `ConnectionFactory` bean is XA-aware and participates in distributed transactions. In some situations, you might want to process certain JMS messages by using a non-XA `ConnectionFactory`. For example, your JMS processing logic might take longer than the XA timeout.

If you want to use a non-XA `ConnectionFactory`, you can inject the `nonXaJmsConnectionFactory` bean rather than the `@Primary` `jmsConnectionFactory` bean. For consistency, the `jmsConnectionFactory` bean is also provided by using the bean alias `xaJmsConnectionFactory`.

The following example shows how to inject `ConnectionFactory` instances:

```
// Inject the primary (XA aware) ConnectionFactory
@Autowired
private ConnectionFactory defaultConnectionFactory;
```

```
// Inject the XA aware ConnectionFactory (uses the alias and injects the same as above)
@Autowired
@Qualifier("xaJmsConnectionFactory")
private ConnectionFactory xaConnectionFactory;

// Inject the non-XA aware ConnectionFactory
@Autowired
@Qualifier("nonXaJmsConnectionFactory")
private ConnectionFactory nonXaConnectionFactory;
```

37.6 Supporting an Alternative Embedded Transaction Manager

The `XAConnectionFactoryWrapper` and `XADataSourceWrapper` interfaces can be used to support alternative embedded transaction managers. The interfaces are responsible for wrapping `XAConnectionFactory` and `XADataSource` beans and exposing them as regular `ConnectionFactory` and `DataSource` beans, which transparently enroll in the distributed transaction. DataSource and JMS auto-configuration use JTA variants, provided you have a `JtaTransactionManager` bean and appropriate XA wrapper beans registered within your `ApplicationContext`.

The BitronixXAConnectionFactoryWrapper and BitronixXADataSourceWrapper provide good examples of how to write XA wrappers.

38. Hazelcast

If Hazelcast is on the classpath and a suitable configuration is found, Spring Boot auto-configures a `HazelcastInstance` that you can inject in your application.

If you define a `com.hazelcast.config.Config` bean, Spring Boot uses that. If your configuration defines an instance name, Spring Boot tries to locate an existing instance rather than creating a new one.

You could also specify the `hazelcast.xml` configuration file to use through configuration, as shown in the following example:

```
spring.hazelcast.config=classpath:config/my-hazelcast.xml
```

Otherwise, Spring Boot tries to find the Hazelcast configuration from the default locations: `hazelcast.xml` in the working directory or at the root of the classpath. We also check if the `hazelcast.config` system property is set. See the Hazelcast documentation for more details.

If `hazelcast-client` is present on the classpath, Spring Boot first attempts to create a client by checking the following configuration options:

- The presence of a `com.hazelcast.client.config.ClientConfig` bean.
- A configuration file defined by the `spring.hazelcast.config` property.
- The presence of the `hazelcast.client.config` system property.
- A `hazelcast-client.xml` in the working directory or at the root of the classpath.

> **Note**
>
> Spring Boot also has explicit caching support for Hazelcast. If caching is enabled, the `HazelcastInstance` is automatically wrapped in a `CacheManager` implementation.

39. Quartz Scheduler

Spring Boot offers several conveniences for working with the Quartz scheduler, including the `spring-boot-starter-quartz` “Starter”. If Quartz is available, a `Scheduler` is auto-configured (through the `SchedulerFactoryBean` abstraction).

Beans of the following types are automatically picked up and associated with the `Scheduler`:

- `JobDetail`: defines a particular Job. `JobDetail` instances can be built with the `JobBuilder` API.
- `Calendar`.
- `Trigger`: defines when a particular job is triggered.

By default, an in-memory `JobStore` is used. However, it is possible to configure a JDBC-based store if a `DataSource` bean is available in your application and if the `spring.quartz.job-store-type` property is configured accordingly, as shown in the following example:

```
spring.quartz.job-store-type=jdbc
```

When the JDBC store is used, the schema can be initialized on startup, as shown in the following example:

```
spring.quartz.jdbc.initialize-schema=always
```

> **Note**
>
> By default, the database is detected and initialized by using the standard scripts provided with the Quartz library. It is also possible to provide a custom script by setting the `spring.quartz.jdbc.schema` property.

By default, jobs created by configuration will not overwrite already registered jobs that have been read from a persistent job store. To enable overwriting existing job definitions set the `spring.quartz.overwrite-existing-jobs` property.

Quartz Scheduler configuration can be customized using `spring.quartz` properties and `SchedulerFactoryBeanCustomizer` beans, which allow programmatic `SchedulerFactoryBean` customization. Advanced Quartz configuration properties can be customized using `spring.quartz.properties.*`.

> **Note**
>
> In particular, an `Executor` bean is not associated with the scheduler as Quartz offers a way to configure the scheduler via `spring.quartz.properties`. If you need to customize the task executor, consider implementing `SchedulerFactoryBeanCustomizer`.

Jobs can define setters to inject data map properties. Regular beans can also be injected in a similar manner, as shown in the following example:

```
public class SampleJob extends QuartzJobBean {

 private MyService myService;

 private String name;
```

```
    // Inject "MyService" bean
    public void setMyService(MyService myService) { ... }

    // Inject the "name" job data property
    public void setName(String name) { ... }

    @Override
    protected void executeInternal(JobExecutionContext context)
      throws JobExecutionException {
     ...
    }

}
```

40. Spring Integration

Spring Boot offers several conveniences for working with Spring Integration, including the `spring-boot-starter-integration` "Starter". Spring Integration provides abstractions over messaging and also other transports such as HTTP, TCP, and others. If Spring Integration is available on your classpath, it is initialized through the `@EnableIntegration` annotation.

Spring Boot also configures some features that are triggered by the presence of additional Spring Integration modules. If `spring-integration-jmx` is also on the classpath, message processing statistics are published over JMX . If `spring-integration-jdbc` is available, the default database schema can be created on startup, as shown in the following line:

```
spring.integration.jdbc.initialize-schema=always
```

See the `IntegrationAutoConfiguration` and `IntegrationProperties` classes for more details.

By default, if a Micrometer `meterRegistry` bean is present, Spring Integration metrics will be managed by Micrometer. If you wish to use legacy Spring Integration metrics, add a `DefaultMetricsFactory` bean to the application context.

41. Spring Session

Spring Boot provides Spring Session auto-configuration for a wide range of data stores. When building a Servlet web application, the following stores can be auto-configured:

- JDBC
- Redis
- Hazelcast
- MongoDB

When building a reactive web application, the following stores can be auto-configured:

- Redis
- MongoDB

If a single Spring Session module is present on the classpath, Spring Boot uses that store implementation automatically. If you have more than one implementation, you must choose the `StoreType` that you wish to use to store the sessions. For instance, to use JDBC as the back-end store, you can configure your application as follows:

```
spring.session.store-type=jdbc
```

> **Tip**
>
> You can disable Spring Session by setting the `store-type` to `none`.

Each store has specific additional settings. For instance, it is possible to customize the name of the table for the JDBC store, as shown in the following example:

```
spring.session.jdbc.table-name=SESSIONS
```

For setting the timeout of the session you can use the `spring.session.timeout` property. If that property is not set, the auto-configuration falls back to the value of `server.servlet.session.timeout`.

42. Monitoring and Management over JMX

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. By default, Spring Boot creates an `MBeanServer` bean with an ID of `mbeanServer` and exposes any of your beans that are annotated with Spring JMX annotations (`@ManagedResource`, `@ManagedAttribute`, or `@ManagedOperation`).

See the `JmxAutoConfiguration` class for more details.

43. Testing

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

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

43.1 Test Scope Dependencies

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

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

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

43.2 Testing Spring Applications

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

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

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

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

43.3 Testing Spring Boot Applications

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

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

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

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

You can use the `webEnvironment` attribute of `@SpringBootTest` to further refine how your tests run:

- `MOCK`: Loads a `WebApplicationContext` and provides a mock servlet environment. Embedded servlet containers are not started when using this annotation. If servlet APIs are not on your classpath, this mode transparently falls back to creating a regular non-web `ApplicationContext`. It can be used in conjunction with `@AutoConfigureMockMvc` for `MockMvc`-based testing of your application.
- `RANDOM_PORT`: Loads an `ServletWebServerApplicationContext` and provides a real servlet environment. Embedded servlet containers are started and listen on a random port.
- `DEFINED_PORT`: Loads a `ServletWebServerApplicationContext` and provides a real servlet environment. Embedded servlet containers are started and listen on a defined port (from your `application.properties` or on the default port of `8080`).
- `NONE`: Loads an `ApplicationContext` by using `SpringApplication` but does not provide *any* servlet environment (mock or otherwise).

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

Detecting Web Application Type

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

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

```
@RunWith(SpringRunner.class)
```

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

Detecting Test Configuration

If you are familiar with the Spring Test Framework, you may be used to using `@ContextConfiguration(classes=…)` in order to specify which Spring `@Configuration` to load. Alternatively, you might have often used nested `@Configuration` classes within your test.

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

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

> **Note**
>
> If you use a test annotation to test a more specific slice of your application, you should avoid adding configuration settings that are specific to a particular area on the main method's application class.
>
> The underlying component scan configuration of `@SpringBootApplication` defines exclude filters that are used to make sure slicing works as expected. If you are using an explicit `@ComponentScan` directive on your `@SpringBootApplication`-annotated class, be aware that those filters will be disabled. If you are using slicing, you should define them again.

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

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

Excluding Test Configuration

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

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

```
@RunWith(SpringRunner.class)
@SpringBootTest
@Import(MyTestsConfiguration.class)
public class MyTests {

 @Test
```

```
 public void exampleTest() {
  ...
 }

}
```

> **Note**
>
> If you directly use `@ComponentScan` (that is, not through `@SpringBootApplication`) you need to register the `TypeExcludeFilter` with it. See the Javadoc for details.

Testing with a running server

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

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

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortWebTestClientExampleTests {

 @Autowired
 private WebTestClient webClient;

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

}
```

Spring Boot also provides a `TestRestTemplate` facility:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

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

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortTestRestTemplateExampleTests {

 @Autowired
```

```
 private TestRestTemplate restTemplate;

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

}
```

Using JMX

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

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = "spring.jmx.enabled=true")
@DirtiesContext
public class SampleJmxTests {

 @Autowired
 private MBeanServer mBeanServer;

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

}
```

Mocking and Spying Beans

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

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

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

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

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.mock.mockito.*;
import org.springframework.test.context.junit4.*;
```

```
import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

 @MockBean
 private RemoteService remoteService;

 @Autowired
 private Reverser reverser;

 @Test
 public void exampleTest() {
  // RemoteService has been injected into the reverser bean
  given(this.remoteService.someCall()).willReturn("mock");
  String reverse = reverser.reverseSomeCall();
  assertThat(reverse).isEqualTo("kcom");
 }

}
```

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

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

Auto-configured Tests

Spring Boot's auto-configuration system works well for applications but can sometimes be a little too much for tests. It often helps to load only the parts of the configuration that are required to test a "slice" of your application. For example, you might want to test that Spring MVC controllers are mapping URLs correctly, and you do not want to involve database calls in those tests, or you might want to test JPA entities, and you are not interested in the web layer when those tests run.

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

> **Note**
>
> Each slice loads a very restricted set of auto-configuration classes. If you need to exclude one of them, most `@…Test` annotations provide an `excludeAutoConfiguration` attribute. Alternatively, you can use `@ImportAutoConfiguration#exclude`.

> **Tip**
>
> It is also possible to use the `@AutoConfigure…` annotations with the standard `@SpringBootTest` annotation. You can use this combination if you are not interested in "slicing" your application but you want some of the auto-configured test beans.

Auto-configured JSON Tests

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

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

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

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

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.json.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.json.*;
import org.springframework.test.context.junit4.*;

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

@RunWith(SpringRunner.class)
@JsonTest
public class MyJsonTests {

 @Autowired
 private JacksonTester<VehicleDetails> json;

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

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

}
```

Note

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

A list of the auto-configuration that is enabled by `@JsonTest` can be found in the appendix.

Auto-configured Spring MVC Tests

To test whether Spring MVC controllers are working as expected, use the `@WebMvcTest` annotation. `@WebMvcTest` auto-configures the Spring MVC infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `Filter`, `WebMvcConfigurer`, and `HandlerMethodArgumentResolver`. Regular `@Component` beans are not scanned when using this annotation.

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

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

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

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

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyControllerTests {

 @Autowired
 private MockMvc mvc;

 @MockBean
 private UserVehicleService userVehicleService;

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

}
```

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

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

```
import com.gargoylesoftware.htmlunit.*;
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyHtmlUnitTests {

 @Autowired
 private WebClient webClient;

 @MockBean
 private UserVehicleService userVehicleService;

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

}
```

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

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

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

Auto-configured Spring WebFlux Tests

To test that Spring WebFlux controllers are working as expected, you can use the `@WebFluxTest` annotation. `@WebFluxTest` auto-configures the Spring WebFlux infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, and `WebFluxConfigurer`. Regular `@Component` beans are not scanned when the `@WebFluxTest` annotation is used.

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

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

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

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

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@WebFluxTest(UserVehicleController.class)
public class MyControllerTests {

 @Autowired
 private WebTestClient webClient;

 @MockBean
 private UserVehicleService userVehicleService;

 @Test
 public void testExample() throws Exception {
  given(this.userVehicleService.getVehicleDetails("sboot"))
    .willReturn(new VehicleDetails("Honda", "Civic"));
  this.webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN)
    .exchange()
    .expectStatus().isOk()
    .expectBody(String.class).isEqualTo("Honda Civic");
 }

}
```

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

A list of the auto-configuration that is enabled by `@WebFluxTest` can be found in the appendix.

> **Note**
>
> `@WebFluxTest` cannot detect routes registered via the functional web framework. For testing `RouterFunction` beans in the context, consider importing your `RouterFunction` yourself via `@Import` or using `@SpringBootTest`.

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

Auto-configured Data JPA Tests

You can use the `@DataJpaTest` annotation to test JPA applications. By default, it configures an in-memory embedded database, scans for `@Entity` classes, and configures Spring Data JPA repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`.

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

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

Data JPA tests may also inject a `TestEntityManager` bean, which provides an alternative to the standard JPA `EntityManager` that is specifically designed for tests. If you want to use `TestEntityManager` outside of `@DataJpaTest` instances, you can also use the `@AutoConfigureTestEntityManager` annotation. A `JdbcTemplate` is also available if you need that. The following example shows the `@DataJpaTest` annotation in use:

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.autoconfigure.orm.jpa.*;

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

@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

 @Autowired
 private TestEntityManager entityManager;

 @Autowired
 private UserRepository repository;

 @Test
 public void testExample() throws Exception {
  this.entityManager.persist(new User("sboot", "1234"));
  User user = this.repository.findByUsername("sboot");
  assertThat(user.getUsername()).isEqualTo("sboot");
  assertThat(user.getVin()).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:

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class ExampleRepositoryTests {

 // ...

}
```

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

Auto-configured JDBC Tests

`@JdbcTest` is similar to `@DataJpaTest` but is for pure JDBC-related tests. By default, it also configures an in-memory embedded database and a `JdbcTemplate`. Regular `@Component` beans are not loaded into the `ApplicationContext`.

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:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

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 "the section called “Auto-configured Data JPA Tests”".)

A list of the auto-configuration that is enabled by `@JdbcTest` can be found in the appendix.

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 "Section 29.5, “Using jOOQ”", earlier in this chapter.)

`@JooqTest` configures a `DSLContext`. Regular `@Component` beans are not loaded into the `ApplicationContext`. The following example shows the `@JooqTest` annotation in use:

```
import org.jooq.DSLContext;
```

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@JooqTest
public class ExampleJooqTests {

 @Autowired
 private 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.

A list of the auto-configuration that is enabled by `@JooqTest` can be found in the appendix.

Auto-configured Data MongoDB Tests

You can use `@DataMongoTest` to test MongoDB applications. By default, it configures an in-memory embedded MongoDB (if available), configures a `MongoTemplate`, scans for `@Document` classes, and configures Spring Data MongoDB repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using MongoDB with Spring Boot, see "Section 30.2, “MongoDB”", earlier in this chapter.)

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

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataMongoTest
public class ExampleDataMongoTests {

 @Autowired
 private MongoTemplate mongoTemplate;

 //
}
```

In-memory embedded MongoDB 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 MongoDB server, you should exclude the embedded MongoDB auto-configuration, as shown in the following example:

```
import org.junit.runner.RunWith;
 import org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataMongoTest(excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class)
public class ExampleDataMongoNonEmbeddedTests {

}
```

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

Auto-configured Data Neo4j Tests

You can use `@DataNeo4jTest` to test Neo4j applications. By default, it uses an in-memory embedded Neo4j (if the embedded driver is available), scans for `@NodeEntity` classes, and configures Spring Data Neo4j repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using Neo4J with Spring Boot, see "Section 30.3, “Neo4j”", earlier in this chapter.)

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

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataNeo4jTest
public class ExampleDataNeo4jTests {

 @Autowired
 private YourRepository repository;

 //
}
```

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:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

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

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` beans are not loaded into the `ApplicationContext`. (For more about using Redis with Spring Boot, see "Section 30.1, “Redis”", earlier in this chapter.)

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

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataRedisTest
public class ExampleDataRedisTests {
```

```
 @Autowired
 private YourRepository repository;

 //
}
```

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

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` beans are not loaded into the `ApplicationContext`. (For more about using LDAP with Spring Boot, see "Section 30.9, “LDAP”", earlier in this chapter.)

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

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.ldap.core.LdapTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest
public class ExampleDataLdapTests {

 @Autowired
 private 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:

```
import org.junit.runner.RunWith;
import org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest(excludeAutoConfiguration = EmbeddedLdapAutoConfiguration.class)
public class ExampleDataLdapNonEmbeddedTests {

}
```

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

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 adds support for `MockRestServiceServer`. The specific beans that you want to test should be specified by using the `value` or `components` attribute of `@RestClientTest`, as shown in the following example:

```
@RunWith(SpringRunner.class)
@RestClientTest(RemoteVehicleDetailsService.class)
public class ExampleRestClientTest {

 @Autowired
 private RemoteVehicleDetailsService service;

 @Autowired
 private MockRestServiceServer server;

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

}
```

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

Auto-configured Spring REST Docs Tests

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

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
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.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
@AutoConfigureRestDocs
public class UserDocumentationTests {

 @Autowired
 private MockMvc mvc;

 @Test
 public void listUsers() throws Exception {
  this.mvc.perform(get("/users").accept(MediaType.TEXT_PLAIN))
    .andExpect(status().isOk())
```

```
    .andDo(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:

```
@TestConfiguration
static class CustomizationConfiguration
  implements RestDocsMockMvcConfigurationCustomizer {

 @Override
 public void customize(MockMvcRestDocumentationConfigurer configurer) {
  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:

```
@TestConfiguration
static class ResultHandlerConfiguration {

 @Bean
 public RestDocumentationResultHandler restDocumentation() {
  return MockMvcRestDocumentation.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:

```
import io.restassured.specification.RequestSpecification;
import org.junit.Test;
import org.junit.runner.RunWith;

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.web.server.LocalServerPort;
import org.springframework.test.context.junit4.SpringRunner;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;
import static org.springframework.restdocs.restassured3.RestAssuredRestDocumentation.document;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
public class UserDocumentationTests {

 @LocalServerPort
 private int port;
```

```
 @Autowired
 private RequestSpecification documentationSpec;

 @Test
 public void listUsers() {
  given(this.documentationSpec).filter(document("list-users")).when()
    .port(this.port).get("/").then().assertThat().statusCode(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:

```
@TestConfiguration
public static class CustomizationConfiguration
  implements RestDocsRestAssuredConfigurationCustomizer {

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

}
```

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 Batch and you rely on the auto-configuration for it. You could define your `@SpringBootApplication` as follows:

```
@SpringBootApplication
@EnableBatchProcessing
public class SampleApplication { ... }
```

Because this class is the source configuration for the test, any slice test actually tries to start Spring Batch, 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:

```
@Configuration
@EnableBatchProcessing
public class BatchConfiguration { ... }
```

> **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.

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:

```
@SpringBootApplication
@ComponentScan({ "com.example.app", "org.acme.another" })
public class SampleApplication { ... }
```

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

If you wish to use Spock to test a Spring Boot application, you should add a dependency on Spock's `spock-spring` module to your application's build. `spock-spring` integrates Spring's test framework into Spock. It is recommended that you use Spock 1.1 or later to benefit from a number of improvements to Spock's Spring Framework and Spring Boot integration. See the documentation for Spock's Spring module for further details.

43.4 Test Utilities

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

ConfigFileApplicationContextInitializer

`ConfigFileApplicationContextInitializer` 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:

```
@ContextConfiguration(classes = Config.class,
 initializers = ConfigFileApplicationContextInitializer.class)
```

> **Note**
>
> Using `ConfigFileApplicationContextInitializer` 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:

```
TestPropertyValues.of("org=Spring", "name=Boot").applyTo(env);
```

OutputCapture

`OutputCapture` is a JUnit `Rule` that you can use to capture `System.out` and `System.err` output. You can declare the capture as a `@Rule` and then use `toString()` for assertions, as follows:

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.rule.OutputCapture;

import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

public class MyTest {

 @Rule
 public OutputCapture capture = new OutputCapture();

 @Test
 public void testName() throws Exception {
  System.out.println("Hello World!");
  assertThat(capture.toString(), containsString("World"));
 }

}
```

TestRestTemplate

> **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`.

`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 behaves in a test-friendly way by not throwing exceptions on server-side errors. It is recommended, but not mandatory, to use the Apache HTTP Client (version 4.3.2 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:

```
public class MyTest {

 private TestRestTemplate template = new TestRestTemplate();

 @Test
 public void testRequest() throws Exception {
  HttpHeaders headers = this.template.getForEntity(
    "http://myhost.example.com/example", String.class).getHeaders();
  assertThat(headers.getLocation()).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:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class SampleWebClientTests {

 @Autowired
 private TestRestTemplate template;

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

 @TestConfiguration
 static class Config {

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

 }

}
```

44. WebSockets

Spring Boot provides WebSockets auto-configuration for embedded Tomcat, Jetty, and Undertow. If you deploy a war file to a standalone container, Spring Boot assumes that the container is responsible for the configuration of its WebSocket support.

Spring Framework provides rich WebSocket support that can be easily accessed through the `spring-boot-starter-websocket` module.

45. Web Services

Spring Boot provides Web Services auto-configuration so that all you must do is define your `Endpoints`.

The Spring Web Services features can be easily accessed with the `spring-boot-starter-webservices` module.

`SimpleWsdl11Definition` and `SimpleXsdSchema` beans can be automatically created for your WSDLs and XSDs respectively. To do so, configure their location, as shown in the following example:

```
spring.webservices.wsdl-locations=classpath:/wsdl
```

46. Calling Web Services with `WebServiceTemplate`

If you need to call remote Web services from your application, you can use the `WebServiceTemplate` class. Since `WebServiceTemplate` instances often need to be customized before being used, Spring Boot does not provide any single auto-configured `WebServiceTemplate` bean. It does, however, auto-configure a `WebServiceTemplateBuilder`, which can be used to create `WebServiceTemplate` instances when needed.

The following code shows a typical example:

```
@Service
public class MyService {

 private final WebServiceTemplate webServiceTemplate;

 public MyService(WebServiceTemplateBuilder webServiceTemplateBuilder) {
  this.webServiceTemplate = webServiceTemplateBuilder.build();
 }

 public DetailsResp someWsCall(DetailsReq detailsReq) {
   return (DetailsResp) this.webServiceTemplate.marshalSendAndReceive(detailsReq, new
 SoapActionCallback(ACTION));

 }

}
```

By default, `WebServiceTemplateBuilder` detects a suitable HTTP-based `WebServiceMessageSender` using the available HTTP client libraries on the classpath. You can also customize read and connection timeouts as follows:

```
@Bean
public WebServiceTemplate webServiceTemplate(WebServiceTemplateBuilder builder) {
 return builder.messageSenders(new HttpWebServiceMessageSenderBuilder()
   .setConnectTimeout(5000).setReadTimeout(2000).build()).build();
}
```

47. 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.

> **Tip**
>
> A demo project is available to showcase how you can create a starter step-by-step.

47.1 Understanding Auto-configured Beans

Under the hood, auto-configuration is implemented with 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 `@Configuration` classes that Spring provides (see the `META-INF/spring.factories` file).

47.2 Locating Auto-configuration Candidates

Spring Boot checks for the presence of a `META-INF/spring.factories` file within your published jar. The file should list your configuration classes under the `EnableAutoConfiguration` key, as shown in the following example:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

You can use the `@AutoConfigureAfter` or `@AutoConfigureBefore` annotations if your configuration needs to be applied in a specific order. 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.

> **Note**
>
> Auto-configurations must be loaded that way *only*. Make sure that they are defined in a specific package space and that, in particular, they are never the target of component scanning.

47.3 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:

- the section called “Class Conditions”
- the section called “Bean Conditions”
- the section called “Property Conditions”
- the section called “Resource Conditions”
- the section called “Web Application Conditions”
- the section called “SpEL Expression Conditions”

Class Conditions

The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations let configuration 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.

> **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:

```
@Configuration
public class MyAutoConfiguration {

 @Bean
 @ConditionalOnMissingBean
 public MyService myService() { ... }

}
```

In the preceding example, the `myService` bean is going to be created if no bean of type `MyService` 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. Using these conditions at the class level is equivalent to marking each contained `@Bean` method with the annotation.

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.

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 web application is any application that uses a Spring `WebApplicationContext`, defines a `session` scope, or has a `StandardServletEnvironment`.

SpEL Expression Conditions

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

47.4 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.

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

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

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 `Assert4J`.

```
@Test
public void defaultServiceBacksOff() {
 this.contextRunner.withUserConfiguration(UserConfiguration.class)
   .run((context) -> {
    assertThat(context).hasSingleBean(UserService.class);
    assertThat(context.getBean(UserService.class)).isSameAs(
      context.getBean(UserConfiguration.class).myUserService());
   });
}

@Configuration
static class UserConfiguration {

 @Bean
 public UserService myUserService() {
  return new UserService("mine");
 }

}
```

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

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

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 `UserService` is not present, the auto-configuration is properly disabled:

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

47.5 Creating Your Own Starter

A full Spring Boot starter for a library may contain the following components:

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

> **Tip**
>
> You may combine the auto-configuration code and the dependency management in a single module if you do not need to separate those two concerns.

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-autoconfigure` and the starter `acme-spring-boot-starter`. If you only have one module that combines the two, name it `acme-spring-boot-starter`.

Also, 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.

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 meta-data (`META-INF/spring-configuration-metadata.json`) to make sure your keys are properly documented.

`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. 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>
```

With Gradle 4.5 and earlier, the dependency should be declared in the `compileOnly` configuration, as shown in the following example:

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

With Gradle 4.6 and later, 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 (i.e. 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.

48. 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 via 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.

48.1 Requirements

Spring Boot supports Kotlin 1.2.x. 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.

48.2 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 via 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.

48.3 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.

48.4 Dependency management

In order to avoid mixing different version of Kotlin dependencies on the classpath, dependency management of the following Kotlin dependencies is provided:

- `kotlin-reflect`
- `kotlin-runtime`
- `kotlin-stdlib`
- `kotlin-stdlib-jdk7`
- `kotlin-stdlib-jdk8`
- `kotlin-stdlib-jre7`
- `kotlin-stdlib-jre8`

With Maven, the Kotlin version can be customized via 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.

48.5 `@ConfigurationProperties`

`@ConfigurationProperties` currently only works with `lateinit` or nullable `var` properties (the former is recommended), since immutable classes initialized by constructors are not yet supported.

```
@ConfigurationProperties("example.kotlin")
class KotlinExampleProperties {

 lateinit var name: String

 lateinit var description: String

 val myService = MyService()

 class MyService {

  lateinit var apiToken: String

  lateinit var uri: URI

 }

}
```

> **Tip**
>
> To generate your own metadata using the annotation processor, `kapt` should be configured with the `spring-boot-configuration-processor` dependency.

48.6 Testing

While it is possible to use JUnit 4 (the default provided by `spring-boot-starter-test`) to test Kotlin code, JUnit 5 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 use JUnit 5, exclude `junit:junit` dependency from `spring-boot-starter-test`, add JUnit 5 dependencies, and configure the Maven or Gradle plugin accordingly. See the JUnit 5 documentation for more details. You also need to switch test instance lifecycle to "per-class".

48.7 Resources

Further reading

- Kotlin language reference
- Kotlin Slack (with a dedicated #spring channel)
- Stackoverflow 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

49. What to Read Next

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

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