Properties

```
my.property=value
spring.config.import=my.properties
```

Yaml

```
my:
  property: "value"
spring:
  config:
    import: "my.properties"
```

In both of the above examples, the values from the `my.properties` file will take precedence over the file that triggered its import.

Several locations can be specified under a single `spring.config.import` key. Locations will be processed in the order that they are defined, with later imports taking precedence.

NOTE

When appropriate, Profile-specific variants are also considered for import. The example above would import both `my.properties` as well as any `my-<profile>.properties` variants.

TIP

Spring Boot includes pluggable API that allows various different location addresses to be supported. By default you can import Java Properties, YAML and “configuration trees”.

Third-party jars can offer support for additional technologies (there is no requirement for files to be local). For example, you can imagine config data being from external stores such as Consul, Apache ZooKeeper or Netflix Archaius.

If you want to support your own locations, see the `ConfigDataLocationResolver` and `ConfigDataLoader` classes in the `org.springframework.boot.context.config` package.

Importing Extensionless Files

Some cloud platforms cannot add a file extension to volume mounted files. To import these extensionless files, you need to give Spring Boot a hint so that it knows how to load them. You can do this by putting an extension hint in square brackets.

For example, suppose you have a `/etc/config/myconfig` file that you wish to import as yaml. You can import it from your `application.properties` using the following:

Properties

```
spring.config.import=file:/etc/config/myconfig[.yaml]
```

Yaml

```
spring:
  config:
    import: "file:/etc/config/myconfig[.yaml]"
```

Using Configuration Trees

When running applications on a cloud platform (such as Kubernetes) you often need to read config values that the platform supplies. It is not uncommon to use environment variables for such purposes, but this can have drawbacks, especially if the value is supposed to be kept secret.

As an alternative to environment variables, many cloud platforms now allow you to map configuration into mounted data volumes. For example, Kubernetes can volume mount both `ConfigMaps` and `Secrets`.

There are two common volume mount patterns that can be used:

1. A single file contains a complete set of properties (usually written as YAML).
2. Multiple files are written to a directory tree, with the filename becoming the 'key' and the contents becoming the 'value'.

For the first case, you can import the YAML or Properties file directly using `spring.config.import` as described above. For the second case, you need to use the `configtree:` prefix so that Spring Boot knows it needs to expose all the files as properties.

As an example, let's imagine that Kubernetes has mounted the following volume:

```
etc/
  config/
    myapp/
      username
      password
```

The contents of the `username` file would be a config value, and the contents of `password` would be a secret.

To import these properties, you can add the following to your `application.properties` or `application.yaml` file:

Properties

```
spring.config.import=optional:configtree:/etc/config/
```

Yaml

```
spring:
  config:
    import: "optional:configtree:/etc/config/"
```

You can then access or inject `myapp.username` and `myapp.password` properties from the `Environment` in the usual way.

TIP The names of the folders and files under the config tree form the property name. In the above example, to access the properties as `username` and `password`, you can set `spring.config.import` to `optional:configtree:/etc/config/myapp`.

NOTE Filenames with dot notation are also correctly mapped. For example, in the above example, a file named `myapp.username` in `/etc/config` would result in a `myapp.username` property in the `Environment`.

TIP Configuration tree values can be bound to both string `String` and `byte[]` types depending on the contents expected.

If you have multiple config trees to import from the same parent folder you can use a wildcard shortcut. Any `configtree:` location that ends with `/*/` will import all immediate children as config trees. As with a non-wildcard import, the names of the folders and files under each config tree form the property name.

For example, given the following volume:

```
etc/
  config/
    dbconfig/
      db/
        username
        password
    mqconfig/
      mq/
        username
        password
```

You can use `configtree:/etc/config/*/` as the import location:

Properties

```
spring.config.import=optional:configtree:/etc/config/*/
```

Yaml

```
spring:
  config:
    import: "optional:configtree:/etc/config/*/"
```

This will add `db.username`, `db.password`, `mq.username` and `mq.password` properties.

NOTE Directories loaded using a wildcard are sorted alphabetically. If you need a different order, then you should list each location as a separate import

Configuration trees can also be used for Docker secrets. When a Docker swarm service is granted access to a secret, the secret gets mounted into the container. For example, if a secret named `db.password` is mounted at location `/run/secrets/`, you can make `db.password` available to the Spring environment using the following:

Properties

```
spring.config.import=optional:configtree:/run/secrets/
```

Yaml

```
spring:
  config:
    import: "optional:configtree:/run/secrets/"
```

Property Placeholders

The values in `application.properties` and `application.yaml` are filtered through the existing `Environment` when they are used, so you can refer back to previously defined values (for example, from System properties or environment variables). The standard `${name}` property-placeholder syntax can be used anywhere within a value. Property placeholders can also specify a default value using a `:` to separate the default value from the property name, for example `${name:default}`.

The use of placeholders with and without defaults is shown in the following example:

Properties

```
app.name=MyApp
app.description=${app.name} is a Spring Boot application written by
${username:Unknown}
```

Yaml

```yaml
app:
  name: "MyApp"
  description: "${app.name} is a Spring Boot application written by
${username:Unknown}"
```

Assuming that the `username` property has not been set elsewhere, `app.description` will have the value `MyApp is a Spring Boot application written by Unknown`.

NOTE

You should always refer to property names in the placeholder using their canonical form (kebab-case using only lowercase letters). This will allow Spring Boot to use the same logic as it does when relaxed binding `@ConfigurationProperties`.

For example, `${demo.item-price}` will pick up `demo.item-price` and `demo.itemPrice` forms from the `application.properties` file, as well as `DEMO_ITEMPRICE` from the system environment. If you used `${demo.itemPrice}` instead, `demo.item-price` and `DEMO_ITEMPRICE` would not be considered.

TIP

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

Working With Multi-Document Files

Spring Boot allows you to split a single physical file into multiple logical documents which are each added independently. Documents are processed in order, from top to bottom. Later documents can override the properties defined in earlier ones.

For `application.yaml` files, the standard YAML multi-document syntax is used. Three consecutive hyphens represent the end of one document, and the start of the next.

For example, the following file has two logical documents:

```yaml
spring:
  application:
    name: "MyApp"
---
spring:
  application:
    name: "MyCloudApp"
  config:
    activate:
      on-cloud-platform: "kubernetes"
```

For `application.properties` files a special `#---` or `!---` comment is used to mark the document splits:

```
spring.application.name=MyApp
#---
spring.application.name=MyCloudApp
spring.config.activate.on-cloud-platform=kubernetes
```

NOTE Property file separators must not have any leading whitespace and must have exactly three hyphen characters. The lines immediately before and after the separator must not be same comment prefix.

TIP Multi-document property files are often used in conjunction with activation properties such as `spring.config.activate.on-profile`. See the next section for details.

WARNING Multi-document property files cannot be loaded by using the `@PropertySource` or `@TestPropertySource` annotations.

Activation Properties

It is sometimes useful to only activate a given set of properties when certain conditions are met. For example, you might have properties that are only relevant when a specific profile is active.

You can conditionally activate a properties document using `spring.config.activate.*`.

The following activation properties are available:

Table 5. activation properties

Property	Note
`on-profile`	A profile expression that must match for the document to be active.
`on-cloud-platform`	The `CloudPlatform` that must be detected for the document to be active.

For example, the following specifies that the second document is only active when running on Kubernetes, and only when either the “prod” or “staging” profiles are active:

Properties

```
myprop=always-set
#---
spring.config.activate.on-cloud-platform=kubernetes
spring.config.activate.on-profile=prod | staging
myotherprop=sometimes-set
```

Yaml

```
myprop:
  "always-set"
---
spring:
  config:
    activate:
      on-cloud-platform: "kubernetes"
      on-profile: "prod | staging"
myotherprop: "sometimes-set"
```

7.2.4. Encrypting Properties

Spring Boot does not provide any built-in support for encrypting property values, however, it does provide the hook points necessary to modify values contained in the Spring `Environment`. The `EnvironmentPostProcessor` interface allows you to manipulate the `Environment` before the application starts. See Customize the Environment or ApplicationContext Before It Starts for details.

If you need a secure way to store credentials and passwords, the Spring Cloud Vault project provides support for storing externalized configuration in HashiCorp Vault.

7.2.5. Working With YAML

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`.

Mapping YAML to Properties

YAML documents need to be converted from their hierarchical format to a flat structure that can be used with the Spring `Environment`. For example, consider the following YAML document:

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

In order to access these properties from the `Environment`, they would be flattened as follows:

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

Likewise, YAML lists also need to be flattened. They 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
```

TIP Properties that use the `[index]` notation can be bound to Java `List` or `Set` objects using Spring Boot’s `Binder` class. For more details see the “Type-safe Configuration Properties” section below.

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

Directly 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`.

You can also use the `YamlPropertySourceLoader` class if you want to load YAML as a Spring `PropertySource`.

7.2.6. 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:

Properties

```
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]}
```

Yaml

```
my:
  secret: "${random.value}"
  number: "${random.int}"
  bignumber: "${random.long}"
  uuid: "${random.uuid}"
  number-less-than-ten: "${random.int(10)}"
  number-in-range: "${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).

7.2.7. Configuring System Environment Properties

Spring Boot supports setting a prefix for environment properties. This is useful if the system environment is shared by multiple Spring Boot applications with different configuration requirements. The prefix for system environment properties can be set directly on `SpringApplication`.

For example, if you set the prefix to `input`, a property such as `remote.timeout` will also be resolved as `input.remote.timeout` in the system environment.

7.2.8. 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.

TIP See also the differences between `@Value` and type-safe configuration properties.

JavaBean Properties Binding

It is possible to bind a bean declaring standard JavaBean properties as shown in the following example:

Java

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

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

@ConfigurationProperties("my.service")
public class MyProperties {

    private boolean enabled;

    private InetAddress remoteAddress;

    private final Security security = new Security();

    public boolean isEnabled() {
        return this.enabled;
    }

    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

    public InetAddress getRemoteAddress() {
        return this.remoteAddress;
    }

    public void setRemoteAddress(InetAddress remoteAddress) {
        this.remoteAddress = remoteAddress;
    }

    public Security getSecurity() {
        return this.security;
    }

    public static class Security {

        private String username;

        private String password;

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

        public String getUsername() {
            return this.username;
        }

        public void setUsername(String username) {
            this.username = username;
        }
        public String getPassword() {
            return this.password;
        }

        public void setPassword(String password) {
            this.password = password;
        }

        public List<String> getRoles() {
            return this.roles;
        }

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

    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import java.net.InetAddress

@ConfigurationProperties("my.service")
class MyProperties {

    var isEnabled = false

    var remoteAddress: InetAddress? = null

    val security = Security()

    class Security {

        var username: String? = null

        var password: String? = null

        var roles: List<String> = ArrayList(setOf("USER"))

    }

}
```

The preceding POJO defines the following properties:

- `my.service.enabled`, with a value of `false` by default.

- `my.service.remote-address`, with a type that can be coerced from `String`.
- `my.service.security.username`, with a nested "security" object whose name is determined by the name of the property. In particular, the type is not used at all there and could have been `SecurityProperties`.
- `my.service.security.password`.
- `my.service.security.roles`, with a collection of `String` that defaults to `USER`.

NOTE

The properties that map to `@ConfigurationProperties` classes available in Spring Boot, which are configured through properties files, YAML files, environment variables, and other mechanisms, are public API but the accessors (getters/setters) of the class itself are not meant to be used directly.

NOTE

Such arrangement relies on a default empty constructor and 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.

Constructor Binding

The example in the previous section can be rewritten in an immutable fashion as shown in the following example:

Java

```java
import java.net.InetAddress;
import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.bind.DefaultValue;
```

```
@ConfigurationProperties("my.service")
public class MyProperties {

    private final boolean enabled;

    private final InetAddress remoteAddress;

    private final Security security;


    public MyProperties(boolean enabled, InetAddress remoteAddress, Security security)
{
        this.enabled = enabled;
        this.remoteAddress = remoteAddress;
        this.security = security;
    }

    public boolean isEnabled() {
        return this.enabled;
    }

    public InetAddress getRemoteAddress() {
        return this.remoteAddress;
    }

    public Security getSecurity() {
        return this.security;
    }

    public static class Security {

        private final String username;

        private final String password;

        private final List<String> roles;


        public Security(String username, String password, @DefaultValue("USER")
List<String> roles) {
            this.username = username;
            this.password = password;
            this.roles = roles;
        }

        public String getUsername() {
            return this.username;
        }

        public String getPassword() {
            return this.password;
        }

        public List<String> getRoles() {
            return this.roles;
        }

    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.context.properties.bind.DefaultValue
import java.net.InetAddress

@ConfigurationProperties("my.service")
class MyProperties(val enabled: Boolean, val remoteAddress: InetAddress,
        val security: Security) {

    class Security(val username: String, val password: String,
            @param:DefaultValue("USER") val roles: List<String>)

}
```

In this setup, the presence of a single parameterized constructor implies that constructor binding should be used. This means that the binder will find a constructor with the parameters that you wish to have bound. If your class has multiple constructors, the `@ConstructorBinding` annotation can be used to specify which constructor to use for constructor binding. To opt out of constructor binding for a class with a single parameterized constructor, the constructor must be annotated with `@Autowired` or made `private`. Constructor binding can be used with records. Unless your record has multiple constructors, there is no need to use `@ConstructorBinding`.

Nested members of a constructor bound class (such as `Security` in the example above) will also be bound through their constructor.

Default values can be specified using `@DefaultValue` on constructor parameters and record components. The conversion service will be applied to coerce the annotation's `String` value to the target type of a missing property.

Referring to the previous example, if no properties are bound to `Security`, the `MyProperties` instance will contain a `null` value for `security`. To make it contain a non-null instance of `Security` even when no properties are bound to it (when using Kotlin, this will require the `username` and `password` parameters of `Security` to be declared as nullable as they do not have default values), use an empty `@DefaultValue` annotation:

Java

```
public MyProperties(boolean enabled, InetAddress remoteAddress, @DefaultValue Security
security) {
    this.enabled = enabled;
    this.remoteAddress = remoteAddress;
    this.security = security;
}
```

Kotlin

```
class MyProperties(val enabled: Boolean, val remoteAddress: InetAddress,
        @DefaultValue val security: Security) {

    class Security(val username: String?, val password: String?,
            @param:DefaultValue("USER") val roles: List<String>)

}
```

NOTE To use constructor binding the class must be enabled using `@EnableConfigurationProperties` or configuration property scanning. You cannot use constructor binding with beans that are created by the regular Spring mechanisms (for example `@Component` beans, beans created by using `@Bean` methods or beans loaded by using `@Import`)

NOTE To use constructor binding the class must be compiled with `-parameters`. This will happen automatically if you use Spring Boot's Gradle plugin or if you use Maven and `spring-boot-starter-parent`.

NOTE The use of `java.util.Optional` with `@ConfigurationProperties` is not recommended as it is primarily intended for use as a return type. As such, it is not well-suited to configuration property injection. For consistency with properties of other types, if you do declare an `Optional` property and it has no value, `null` rather than an empty `Optional` will be bound.

Enabling @ConfigurationProperties-annotated Types

Spring Boot provides infrastructure to bind `@ConfigurationProperties` types and register them as beans. You can either enable configuration properties on a class-by-class basis or enable configuration property scanning that works in a similar manner to component scanning.

Sometimes, classes annotated with `@ConfigurationProperties` might not be suitable for scanning, for example, if you're developing your own auto-configuration or you want to enable them conditionally. In these cases, specify the list of types to process using the `@EnableConfigurationProperties` annotation. This can be done on any `@Configuration` class, as shown in the following example:

Java

```
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(SomeProperties.class)
public class MyConfiguration {

}
```

Kotlin

```
import org.springframework.boot.context.properties.EnableConfigurationProperties
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(SomeProperties::class)
class MyConfiguration
```

Java

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

@ConfigurationProperties("some.properties")
public class SomeProperties {

}
```

Kotlin

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

@ConfigurationProperties("some.properties")
class SomeProperties
```

To use configuration property scanning, add the `@ConfigurationPropertiesScan` annotation to your application. Typically, it is added to the main application class that is annotated with `@SpringBootApplication` but it can be added to any `@Configuration` class. By default, scanning will occur from the package of the class that declares the annotation. If you want to define specific packages to scan, you can do so as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.ConfigurationPropertiesScan;

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

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.context.properties.ConfigurationPropertiesScan

@SpringBootApplication
@ConfigurationPropertiesScan("com.example.app", "com.example.another")
class MyApplication
```

NOTE

When the `@ConfigurationProperties` bean is registered using configuration property scanning or through `@EnableConfigurationProperties`, 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.

Assuming that it is in the `com.example.app` package, the bean name of the `SomeProperties` example above is `some.properties-com.example.app.SomeProperties`.

We recommend that `@ConfigurationProperties` only deal with the environment and, in particular, does not inject other beans from the context. For corner cases, setter injection can be used or any of the `*Aware` interfaces provided by the framework (such as `EnvironmentAware` if you need access to the `Environment`). If you still want to inject other beans using the constructor, the configuration properties bean must be annotated with `@Component` and use JavaBean-based property binding.

Using @ConfigurationProperties-annotated Types

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

```
my:
  service:
    remote-address: 192.168.1.1
    security:
      username: "admin"
      roles:
      - "USER"
      - "ADMIN"
```

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

Java

```
import org.springframework.stereotype.Service;

@Service
public class MyService {

    private final MyProperties properties;

    public MyService(MyProperties properties) {
        this.properties = properties;
    }

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

    // ...

}
```

Kotlin

```
import org.springframework.stereotype.Service

@Service
class MyService(val properties: MyProperties) {

    fun openConnection() {
        val server = Server(properties.remoteAddress)
        server.start()
        // ...
    }

    // ...

}
```

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 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:

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class ThirdPartyConfiguration {

    @Bean
    @ConfigurationProperties(prefix = "another")
    public AnotherComponent anotherComponent() {
        return new AnotherComponent();
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class ThirdPartyConfiguration {

    @Bean
    @ConfigurationProperties(prefix = "another")
    fun anotherComponent(): AnotherComponent = AnotherComponent()

}
```

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

As an example, consider the following `@ConfigurationProperties` class:

Java

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

@ConfigurationProperties(prefix = "my.main-project.person")
public class MyPersonProperties {

    private String firstName;

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

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

}
```

Kotlin

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

@ConfigurationProperties(prefix = "my.main-project.person")
class MyPersonProperties {

    var firstName: String? = null

}
```

With the preceding code, the following properties names can all be used:

Table 6. relaxed binding

Property	Note
`my.main-project.person.first-name`	Kebab case, which is recommended for use in `.properties` and YAML files.
`my.main-project.person.firstName`	Standard camel case syntax.
`my.main-project.person.first_name`	Underscore notation, which is an alternative format for use in `.properties` and YAML files.
`MY_MAINPROJECT_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 `my.main-project.person`).

Table 7. 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 (see Binding From Environment Variables).	Numeric values surrounded by underscores (see Binding From Environment Variables)
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.person.first-name=Rod`.

Binding Maps

When binding to `Map` properties you may need to use a special bracket notation so that the original `key` value is preserved. If the key is not surrounded by `[]`, any characters that are not alpha-numeric, `-` or `.` are removed.

For example, consider binding the following properties to a `Map<String,String>`:

Properties

```
my.map.[/key1]=value1
my.map.[/key2]=value2
my.map./key3=value3
```

Yaml

```
my:
  map:
    "[/key1]": "value1"
    "[/key2]": "value2"
    "/key3": "value3"
```

NOTE For YAML files, the brackets need to be surrounded by quotes for the keys to be parsed properly.

The properties above will bind to a `Map` with `/key1`, `/key2` and `key3` as the keys in the map. The slash has been removed from `key3` because it was not surrounded by square brackets.

When binding to scalar values, keys with `.` in them do not need to be surrounded by `[]`. Scalar values include enums and all types in the `java.lang` package except for `Object`. Binding `a.b=c` to `Map<String, String>` will preserve the `.` in the key and return a Map with the entry `{"a.b"="c"}`. For any other types you need to use the bracket notation if your `key` contains a `.`. For example, binding `a.b=c` to `Map<String, Object>` will return a Map with the entry `{"a"={"b"="c"}}` whereas `[a.b]=c` will return a Map with the entry `{"a.b"="c"}`.

Binding From Environment Variables

Most operating systems impose strict rules around the names that can be used for environment variables. For example, Linux shell variables can contain only letters (`a` to `z` or `A` to `Z`), numbers (`0` to `9`) or the underscore character (`_`). By convention, Unix shell variables will also have their names in UPPERCASE.

Spring Boot's relaxed binding rules are, as much as possible, designed to be compatible with these naming restrictions.

To convert a property name in the canonical-form to an environment variable name you can follow these rules:

- Replace dots (`.`) with underscores (`_`).

- Remove any dashes (`-`).
- Convert to uppercase.

For example, the configuration property `spring.main.log-startup-info` would be an environment variable named `SPRING_MAIN_LOGSTARTUPINFO`.

Environment variables can also be used when binding to object lists. To bind to a `List`, the element number should be surrounded with underscores in the variable name.

For example, the configuration property `my.service[0].other` would use an environment variable named `MY_SERVICE_0_OTHER`.

Caching

Relaxed binding uses a cache to improve performance. By default, this caching is only applied to immutable property sources. To customize this behavior, for example to enable caching for mutable property sources, use `ConfigurationPropertyCaching`.

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 `MyProperties`:

Java

```java
import java.util.ArrayList;
import java.util.List;

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

@ConfigurationProperties("my")
public class MyProperties {

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

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

}
```

Kotlin

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

@ConfigurationProperties("my")
class MyProperties {

    val list: List<MyPojo> = ArrayList()

}
```

Consider the following configuration:

Properties

```
my.list[0].name=my name
my.list[0].description=my description
#---
spring.config.activate.on-profile=dev
my.list[0].name=my another name
```

Yaml

```
my:
  list:
  - name: "my name"
    description: "my description"
---
spring:
  config:
    activate:
      on-profile: "dev"
my:
  list:
  - name: "my another name"
```

If the `dev` profile is not active, `MyProperties.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:

Properties

```
my.list[0].name=my name
my.list[0].description=my description
my.list[1].name=another name
my.list[1].description=another description
#---
spring.config.activate.on-profile=dev
my.list[0].name=my another name
```

Yaml

```
my:
  list:
  - name: "my name"
    description: "my description"
  - name: "another name"
    description: "another description"
---
spring:
  config:
    activate:
      on-profile: "dev"
my:
  list:
  - name: "my another name"
```

In the preceding example, if the `dev` profile is active, `MyProperties.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 `MyProperties`:

Java

```
import java.util.LinkedHashMap;
import java.util.Map;

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

@ConfigurationProperties("my")
public class MyProperties {

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

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

}
```

Kotlin

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

@ConfigurationProperties("my")
class MyProperties {

    val map: Map<String, MyPojo> = LinkedHashMap()

}
```

Consider the following configuration:

Properties

```
my.map.key1.name=my name 1
my.map.key1.description=my description 1
#---
spring.config.activate.on-profile=dev
my.map.key1.name=dev name 1
my.map.key2.name=dev name 2
my.map.key2.description=dev description 2
```

Yaml

```yaml
my:
  map:
    key1:
      name: "my name 1"
      description: "my description 1"
---
spring:
  config:
    activate:
      on-profile: "dev"
my:
  map:
    key1:
      name: "dev name 1"
    key2:
      name: "dev name 2"
      description: "dev description 2"
```

If the `dev` profile is not active, `MyProperties.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 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.time.Duration`
- A more readable format where the value and the unit are coupled (`10s` means 10 seconds)

Consider the following example:

Java

```
import java.time.Duration;
import java.time.temporal.ChronoUnit;

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

@ConfigurationProperties("my")
public class MyProperties {

    @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;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.convert.DurationUnit
import java.time.Duration
import java.time.temporal.ChronoUnit

@ConfigurationProperties("my")
class MyProperties {

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

    var readTimeout = Duration.ofMillis(1000)

}
```

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 units. 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.

If you prefer to use constructor binding, the same properties can be exposed, as shown in the following example:

Java

```
import java.time.Duration;
import java.time.temporal.ChronoUnit;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.bind.DefaultValue;
import org.springframework.boot.convert.DurationUnit;

@ConfigurationProperties("my")
public class MyProperties {

    private final Duration sessionTimeout;

    private final Duration readTimeout;

    public MyProperties(@DurationUnit(ChronoUnit.SECONDS) @DefaultValue("30s")
Duration sessionTimeout,
            @DefaultValue("1000ms") Duration readTimeout) {
        this.sessionTimeout = sessionTimeout;
        this.readTimeout = readTimeout;
    }

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

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

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.context.properties.bind.DefaultValue
import org.springframework.boot.convert.DurationUnit
import java.time.Duration
import java.time.temporal.ChronoUnit

@ConfigurationProperties("my")
class MyProperties(@param:DurationUnit(ChronoUnit.SECONDS) @param:DefaultValue("30s")
val sessionTimeout: Duration,
        @param:DefaultValue("1000ms") val readTimeout: Duration)
```

TIP If you are upgrading a `Long` property, make sure to define the unit (using `@DurationUnit`) if it is not milliseconds. Doing so gives a transparent upgrade path while supporting a much richer format.

Converting Periods

In addition to durations, Spring Boot can also work with `java.time.Period` type. The following formats can be used in application properties:

- An regular `int` representation (using days as the default unit unless a `@PeriodUnit` has been specified)
- The standard ISO-8601 format used by `java.time.Period`
- A simpler format where the value and the unit pairs are coupled (`1y3d` means 1 year and 3 days)

The following units are supported with the simple format:

- `y` for years
- `m` for months
- `w` for weeks
- `d` for days

NOTE The `java.time.Period` type never actually stores the number of weeks, it is a shortcut that means “7 days”.

Converting Data Sizes

Spring Framework has a `DataSize` value type that expresses a size in bytes. If you expose a `DataSize` property, the following formats in application properties are available:

- A regular `long` representation (using bytes as the default unit unless a `@DataSizeUnit` has been specified)
- A more readable format where the value and the unit are coupled (`10MB` means 10 megabytes)

Consider the following example:

Java

```java
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.convert.DataSizeUnit;
import org.springframework.util.unit.DataSize;
import org.springframework.util.unit.DataUnit;

@ConfigurationProperties("my")
public class MyProperties {

    @DataSizeUnit(DataUnit.MEGABYTES)
    private DataSize bufferSize = DataSize.ofMegabytes(2);

    private DataSize sizeThreshold = DataSize.ofBytes(512);

    public DataSize getBufferSize() {
        return this.bufferSize;
    }

    public void setBufferSize(DataSize bufferSize) {
        this.bufferSize = bufferSize;
    }

    public DataSize getSizeThreshold() {
        return this.sizeThreshold;
    }

    public void setSizeThreshold(DataSize sizeThreshold) {
        this.sizeThreshold = sizeThreshold;
    }

}
```

Kotlin

```kotlin
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.convert.DataSizeUnit
import org.springframework.util.unit.DataSize
import org.springframework.util.unit.DataUnit

@ConfigurationProperties("my")
class MyProperties {

    @DataSizeUnit(DataUnit.MEGABYTES)
    var bufferSize = DataSize.ofMegabytes(2)

    var sizeThreshold = DataSize.ofBytes(512)

}
```

To specify a buffer size of 10 megabytes, `10` and `10MB` are equivalent. A size threshold of 256 bytes can be specified as `256` or `256B`.

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

- `B` for bytes
- `KB` for kilobytes
- `MB` for megabytes
- `GB` for gigabytes
- `TB` for terabytes

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

If you prefer to use constructor binding, the same properties can be exposed, as shown in the following example:

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.bind.DefaultValue;
import org.springframework.boot.convert.DataSizeUnit;
import org.springframework.util.unit.DataSize;
import org.springframework.util.unit.DataUnit;

@ConfigurationProperties("my")
public class MyProperties {

    private final DataSize bufferSize;

    private final DataSize sizeThreshold;

    public MyProperties(@DataSizeUnit(DataUnit.MEGABYTES) @DefaultValue("2MB")
DataSize bufferSize,
            @DefaultValue("512B") DataSize sizeThreshold) {
        this.bufferSize = bufferSize;
        this.sizeThreshold = sizeThreshold;
    }

    public DataSize getBufferSize() {
        return this.bufferSize;
    }

    public DataSize getSizeThreshold() {
        return this.sizeThreshold;
    }

}
```

Kotlin

```kotlin
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.context.properties.bind.DefaultValue
import org.springframework.boot.convert.DataSizeUnit
import org.springframework.util.unit.DataSize
import org.springframework.util.unit.DataUnit

@ConfigurationProperties("my")
class MyProperties(@param:DataSizeUnit(DataUnit.MEGABYTES) @param:DefaultValue("2MB")
val bufferSize: DataSize,
        @param:DefaultValue("512B") val sizeThreshold: DataSize)
```

TIP

If you are upgrading a `Long` property, make sure to define the unit (using `@DataSizeUnit`) if it is not bytes. 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 `jakarta.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:

Java

```java
import java.net.InetAddress;

import jakarta.validation.constraints.NotNull;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.validation.annotation.Validated;

@ConfigurationProperties("my.service")
@Validated
public class MyProperties {

    @NotNull
    private InetAddress remoteAddress;

    public InetAddress getRemoteAddress() {
        return this.remoteAddress;
    }

    public void setRemoteAddress(InetAddress remoteAddress) {
        this.remoteAddress = remoteAddress;
    }

}
```

Kotlin

```kotlin
import jakarta.validation.constraints.NotNull
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.validation.annotation.Validated
import java.net.InetAddress

@ConfigurationProperties("my.service")
@Validated
class MyProperties {

    var remoteAddress: @NotNull InetAddress? = null

}
```

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

To ensure that validation is always triggered for nested properties, even when no properties are found, the associated field must be annotated with `@Valid`. The following example builds on the preceding `MyProperties` example:

Java

```java
import java.net.InetAddress;

import jakarta.validation.Valid;
import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.NotNull;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.validation.annotation.Validated;

@ConfigurationProperties("my.service")
@Validated
public class MyProperties {

    @NotNull
    private InetAddress remoteAddress;

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

    public InetAddress getRemoteAddress() {
        return this.remoteAddress;
    }

    public void setRemoteAddress(InetAddress remoteAddress) {
        this.remoteAddress = remoteAddress;
    }

    public Security getSecurity() {
        return this.security;
    }

    public static class Security {

        @NotEmpty
        private String username;

        public String getUsername() {
            return this.username;
        }

        public void setUsername(String username) {
            this.username = username;
        }

    }

}
```

Kotlin

```kotlin
import jakarta.validation.Valid
import jakarta.validation.constraints.NotEmpty
import jakarta.validation.constraints.NotNull
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.validation.annotation.Validated
import java.net.InetAddress

@ConfigurationProperties("my.service")
@Validated
class MyProperties {

    var remoteAddress: @NotNull InetAddress? = null

    @Valid
    val security = Security()

    class Security {

        @NotEmpty
        var username: String? = null

    }

}
```

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.

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	Limited (see note below)

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

NOTE

If you do want to use `@Value`, we recommend that you refer to property names using their canonical form (kebab-case using only lowercase letters). This will allow Spring Boot to use the same logic as it does when relaxed binding `@ConfigurationProperties`.

For example, `@Value("${demo.item-price}")` will pick up `demo.item-price` and `demo.itemPrice` forms from the `application.properties` file, as well as `DEMO_ITEMPRICE` from the system environment. If you used `@Value("${demo.itemPrice}")` instead, `demo.item-price` and `DEMO_ITEMPRICE` would not be considered.

If you define a set of configuration keys for your own components, we recommend you group them in a POJO annotated with `@ConfigurationProperties`. Doing so will provide you with structured, type-safe object that you can inject into your own beans.

`SpEL` expressions from application property files are not processed at time of parsing these files and populating the environment. However, it is possible to write a `SpEL` expression in `@Value`. If the value of a property from an application property file is a `SpEL` expression, it will be evaluated when consumed through `@Value`.

7.3. Profiles

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

Java

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration(proxyBeanMethods = false)
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```

Kotlin

```
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Profile

@Configuration(proxyBeanMethods = false)
@Profile("production")
class ProductionConfiguration {

    // ...

}
```

NOTE If `@ConfigurationProperties` beans are registered through `@EnableConfigurationProperties` instead of automatic scanning, the `@Profile` annotation needs to be specified on the `@Configuration` class that has the `@EnableConfigurationProperties` annotation. In the case where `@ConfigurationProperties` are scanned, `@Profile` can be specified on the `@ConfigurationProperties` class itself.

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:

Properties

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

Yaml

```
spring:
  profiles:
    active: "dev,hsqldb"
```

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

If no profile is active, a default profile is enabled. The name of the default profile is `default` and it can be tuned using the `spring.profiles.default` `Environment` property, as shown in the following example:

Properties

```
spring.profiles.default=none
```

Yaml

```
spring:
  profiles:
    default: "none"
```

`spring.profiles.active` and `spring.profiles.default` can only be used in non-profile-specific documents. This means they cannot be included in profile specific files or documents activated by `spring.config.activate.on-profile`.

For example, the second document configuration is invalid:

Properties

```
# this document is valid
spring.profiles.active=prod
#---
# this document is invalid
spring.config.activate.on-profile=prod
spring.profiles.active=metrics
```

Yaml

```
# this document is valid
spring:
  profiles:
    active: "prod"
---
# this document is invalid
spring:
  config:
    activate:
      on-profile: "prod"
  profiles:
    active: "metrics"
```

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

For example, when an application with the following properties is run, the common and local profiles will be activated even when it runs using the `--spring.profiles.active` switch:

Properties

```
spring.profiles.include[0]=common
spring.profiles.include[1]=local
```

Yaml

```
spring:
  profiles:
    include:
      - "common"
      - "local"
```

WARNING Similar to spring.profiles.active, spring.profiles.include can only be used in non-profile-specific documents. This means it cannot be included in profile specific files or documents activated by spring.config.activate.on-profile.

Profile groups, which are described in the next section can also be used to add active profiles if a given profile is active.

7.3.2. Profile Groups

Occasionally the profiles that you define and use in your application are too fine-grained and become cumbersome to use. For example, you might have proddb and prodmq profiles that you use to enable database and messaging features independently.

To help with this, Spring Boot lets you define profile groups. A profile group allows you to define a logical name for a related group of profiles.

For example, we can create a production group that consists of our proddb and prodmq profiles.

Properties

```
spring.profiles.group.production[0]=proddb
spring.profiles.group.production[1]=prodmq
```

Yaml

```
spring:
  profiles:
    group:
      production:
      - "proddb"
      - "prodmq"
```

Our application can now be started using --spring.profiles.active=production to activate the production, proddb and prodmq profiles in one hit.

WARNING Similar to `spring.profiles.active` and `spring.profiles.include`, `spring.profiles.group` can only be used in non-profile-specific documents. This means it cannot be included in profile specific files or documents activated by `spring.config.activate.on-profile`.

7.3.3. 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.

7.3.4. Profile-specific Configuration Files

Profile-specific variants of both `application.properties` (or `application.yaml`) and files referenced through `@ConfigurationProperties` are considered as files and loaded. See "Profile Specific Files" for details.

7.4. 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.

TIP When you deploy your application to a servlet container or application server, logging performed with the Java Util Logging API is not routed into your application's logs. This prevents logging performed by the container or other applications that have been deployed to it from appearing in your application's logs.

7.4.1. Log Format

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

```
2024-06-20T08:04:21.437Z  INFO 111727 --- [myapp] [           main]
o.s.b.d.f.logexample.MyApplication       : Starting MyApplication using Java 17.0.11
with PID 111727 (/opt/apps/myapp.jar started by myuser in /opt/apps/)
2024-06-20T08:04:21.456Z  INFO 111727 --- [myapp] [           main]
o.s.b.d.f.logexample.MyApplication       : No active profile set, falling back to 1
default profile: "default"
2024-06-20T08:04:25.212Z  INFO 111727 --- [myapp] [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
2024-06-20T08:04:25.253Z  INFO 111727 --- [myapp] [           main]
o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-06-20T08:04:25.253Z  INFO 111727 --- [myapp] [           main]
o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache
Tomcat/10.1.25]
2024-06-20T08:04:25.399Z  INFO 111727 --- [myapp] [           main]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded
WebApplicationContext
2024-06-20T08:04:25.406Z  INFO 111727 --- [myapp] [           main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization
completed in 3723 ms
2024-06-20T08:04:26.611Z  INFO 111727 --- [myapp] [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with
context path ''
2024-06-20T08:04:26.640Z  INFO 111727 --- [myapp] [           main]
o.s.b.d.f.logexample.MyApplication       : Started MyApplication in 6.738 seconds
(process running for 7.682)
```

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.
- Application name: Enclosed in square brackets (logged by default only if `spring.application.name` is set)
- Thread name: Enclosed in square brackets (may be truncated for console output).
- Correlation ID: If tracing is enabled (not shown in the sample above)
- 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`.

TIP If you have a `spring.application.name` property but don’t want it logged you can set `logging.include-application-name` to `false`.

7.4.2. 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'T'HH:mm:ss.SSSXXX}){yellow}
```

The following colors and styles are supported:

- blue
- cyan
- faint
- green
- magenta
- red
- yellow

7.4.3. 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.name or logging.file.path property (for example, in your application.properties).

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

Table 8. Logging properties

logging.file.name	logging.file.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.

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.

7.4.4. File Rotation

If you are using the Logback, it is possible to fine-tune log rotation settings using your application.properties or application.yaml file. For all other logging system, you will need to configure rotation settings directly yourself (for example, if you use Log4j2 then you could add a log4j2.xml or log4j2-spring.xml file).

The following rotation policy properties are supported:

Name	Description
logging.logback.rollingpolicy.file-name-pattern	The filename pattern used to create log archives.

Name	Description
`logging.logback.rollingpolicy.clean-history-on-start`	If log archive cleanup should occur when the application starts.
`logging.logback.rollingpolicy.max-file-size`	The maximum size of log file before it is archived.
`logging.logback.rollingpolicy.total-size-cap`	The maximum amount of size log archives can take before being deleted.
`logging.logback.rollingpolicy.max-history`	The maximum number of archive log files to keep (defaults to 7).

7.4.5. 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`:

Properties

```
logging.level.root=warn
logging.level.org.springframework.web=debug
logging.level.org.hibernate=error
```

Yaml

```
logging:
  level:
    root: "warn"
    org.springframework.web: "debug"
    org.hibernate: "error"
```

It is also possible to set logging levels using environment variables. For example, `LOGGING_LEVEL_ORG_SPRINGFRAMEWORK_WEB=DEBUG` will set `org.springframework.web` to `DEBUG`.

NOTE The above approach will only work for package level logging. Since relaxed binding always converts environment variables to lowercase, it is not possible to configure logging for an individual class in this way. If you need to configure logging for a class, you can use the `SPRING_APPLICATION_JSON` variable.

7.4.6. Log Groups

It is often useful to be able to group related loggers together so that they can all be configured at the same time. For example, you might commonly change the logging levels for *all* Tomcat related loggers, but you can not easily remember top level packages.

To help with this, Spring Boot allows you to define logging groups in your Spring `Environment`. For example, here is how you could define a “tomcat” group by adding it to your `application.properties`:

Properties

```
logging.group.tomcat=org.apache.catalina,org.apache.coyote,org.apache.tomcat
```

Yaml

```
logging:
  group:
    tomcat: "org.apache.catalina,org.apache.coyote,org.apache.tomcat"
```

Once defined, you can change the level for all the loggers in the group with a single line:

Properties

```
logging.level.tomcat=trace
```

Yaml

```
logging:
  level:
    tomcat: "trace"
```

Spring Boot includes the following pre-defined logging groups that can be used out-of-the-box:

Name	Loggers
web	`org.springframework.core.codec`, `org.springframework.http`, `org.springframework.web`, `org.springframework.boot.actuate.endpoint.web`, `org.springframework.boot.web.servlet.ServletContextInitializerBeans`
sql	`org.springframework.jdbc.core`, `org.hibernate.SQL`, `org.jooq.tools.LoggerListener`

7.4.7. Using a Log Shutdown Hook

In order to release logging resources when your application terminates, a shutdown hook that will trigger log system cleanup when the JVM exits is provided. This shutdown hook is registered automatically unless your application is deployed as a war file. If your application has complex context hierarchies the shutdown hook may not meet your needs. If it does not, disable the shutdown hook and investigate the options provided directly by the underlying logging system. For example, Logback offers context selectors which allow each Logger to be created in its own context. You can use the `logging.register-shutdown-hook` property to disable the shutdown hook. Setting it to `false` will disable the registration. You can set the property in your `application.properties` or `application.yaml` file:

Properties

```
logging.register-shutdown-hook=false
```

Yaml

```
logging:
  register-shutdown-hook: false
```

7.4.8. 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 through 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. This allows the properties to be consumed by logging system configuration. For example, setting `logging.file.name` in `application.properties` or `LOGGING_FILE_NAME` as an environment variable will result in the `LOG_FILE` System property being set. The properties that are transferred are 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.name	LOG_FILE	If defined, it is used in the default log configuration.
logging.file.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).
logging.pattern.dateformat	LOG_DATEFORMAT_PATTERN	Appender pattern for log date format.
logging.charset.console	CONSOLE_LOG_CHARSET	The charset to use for console logging.
logging.threshold.console	CONSOLE_LOG_THRESHOLD	The log level threshold to use for console logging.
logging.pattern.file	FILE_LOG_PATTERN	The log pattern to use in a file (if LOG_FILE is enabled).
logging.charset.file	FILE_LOG_CHARSET	The charset to use for file logging (if LOG_FILE is enabled).
logging.threshold.file	FILE_LOG_THRESHOLD	The log level threshold to use for file logging.
logging.pattern.level	LOG_LEVEL_PATTERN	The format to use when rendering the log level (default %5p).
PID	PID	The current process ID (discovered if possible and when not already defined as an OS environment variable).

If you use Logback, the following properties are also transferred:

Spring Environment	System Property	Comments
logging.logback.rollingpolicy.file-name-pattern	LOGBACK_ROLLINGPOLICY_FILE_NAME_PATTERN	Pattern for rolled-over log file names (default ${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz).
logging.logback.rollingpolicy.clean-history-on-start	LOGBACK_ROLLINGPOLICY_CLEAN_HISTORY_ON_START	Whether to clean the archive log files on startup.
logging.logback.rollingpolicy.max-file-size	LOGBACK_ROLLINGPOLICY_MAX_FILE_SIZE	Maximum log file size.
logging.logback.rollingpolicy.total-size-cap	LOGBACK_ROLLINGPOLICY_TOTAL_SIZE_CAP	Total size of log backups to be kept.

Spring Environment	System Property	Comments
`logging.logback.rollingpolicy.max-history`	`LOGBACK_ROLLINGPOLICY_MAX_HISTORY`	Maximum number of archive log files to keep.

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.

```
2019-08-30 12:30:04.031 user:someone INFO 22174 --- [  nio-8080-exec-0]
demo.Controller
Handling authenticated request
```

7.4.9. 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]]
```