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. `properties` attribute on your tests. Available on `@SpringBootTest` and the test annotations for testing a particular slice of your application.
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]

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] **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] **Warning**
> `spring.config.name` and `spring.config.location` are used very early to determine which files have to be loaded. 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] **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] **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] **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] **Tip**
You can also use this technique to create "short" variants of existing Spring Boot properties. See the *Section 77.4, "Use 'Short' Command Line Arguments"* how-to for details.

24.6 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 Section 76.3, "Customize the Environment or ApplicationContext Before It Starts" for details.

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

24.7 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] **Note**
If you use "Starters", SnakeYAML is automatically provided by `spring-boot-starter`.

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

The preceding example would be transformed into the following properties:

```
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
```

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;
        }
}
```

24.7.2 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.

24.7.3 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]

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.

24.7.4 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.

Using the multi YAML document syntax in profile-specific YAML files can lead to unexpected behavior. For example, consider the following config in a file:

application-dev.yml.

```
server:
  port: 8000
---
spring:
  profiles: "!test"
  security:
    user:
      password: "secret"
```

If you run the application with the arguments `--spring.profiles.active=dev" you might expect `security.user.password` to be set to “secret”, but this is not the case.

The nested document will be filtered because the main file is named `application-dev.yml`. It is already considered to be profile-specific, and nested documents will be ignored.


[Tip]

Tip

We recommend that you don’t mix profile-specific YAML files and multiple YAML documents. Stick to using only one of them.

24.8 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, 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] **Note**

Spring Boot auto-configuration heavily makes use of `@ConfigurationProperties` for easily configuring auto-configured beans. Similar to auto-configuration classes, `@ConfigurationProperties` classes available in Spring Boot are for internal use only. The properties that map to the class, which are configured via properties files, YAML files, environment variables etc., are public API but the content of the class itself is not meant to be used directly.


[Note] **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


[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] **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`.

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. Keep in mind that the `@EnableConfigurationProperties` annotation is *also* automatically applied to your project so that any *existing* bean annotated with `@ConfigurationProperties` is configured from the `Environment`. Instead of annotating `MyConfiguration` with `@EnableConfigurationProperties(AcmeProperties.class)`, you could make `AcmeProperties` 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] **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.

24.8.1 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.

24.8.2 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:

```
@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;
        }

}
```

With the preceding code, 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]

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


[Tip]

Tip

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

When binding to `Map` properties, if the `key` contains anything other than lowercase alpha-numeric characters or `-`, you need to use the bracket notation so that the original 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`:

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

The properties above will bind to a `Map` with `/key1`, `/key2` and `key3` as the keys in the map.

24.8.3 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]

Note

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

24.8.4 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]

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 (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 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.


[Tip]

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.

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 (e.g. `10MB` means 10 megabytes)

Consider the following example:

```
@ConfigurationProperties("app.io")
public class AppIoProperties {

	@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;
	}

}
```

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.


[Tip]

Tip

If you are upgrading from a previous version that is simply using `Long` to express the size, make sure to define the unit (using `@DataSizeUnit`) if it isn't bytes alongsidethe switch to `DataSize`. Doing so gives a transparent upgrade path while supporting a much richer format.

24.8.5 @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 ensures 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.

24.8.6 @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.
