24. Externalized Configuration

Spring Boot allows you to externalize your configuration so 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 using the `@Value` annotation, accessed via Spring's `Environment` abstraction or bound to structured objects via `@ConfigurationProperties`.

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

1. Devtools global settings properties on your home directory (`~/.spring-boot-devtools.properties` when devtools is active).
2. `@TestPropertySource` annotations on your tests.
3. `@SpringBootTest#properties` annotation attribute on your tests.
4. Command line arguments.
5. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property)
6. `ServletConfig` init parameters.
7. `ServletContext` init parameters.
8. JNDI attributes from `java:comp/env`.
9. Java System properties (`System.getProperties()`).
10. OS environment variables.
11. A `RandomValuePropertySource` that only has properties 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 using `SpringApplication.setDefaultProperties`).

To provide a concrete example, suppose you develop a `@Component` that uses a `name` property:

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

@Component
public class MyBean {

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

    // ...

}
```

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

The `SPRING_APPLICATION_JSON` properties can be supplied on the command line with an environment variable. For example in a UN*X shell:

```
$ SPRING_APPLICATION_JSON='{"foo":{"bar":"spam"}}' java -jar myapp.jar
```

In this example you will end up with `foo.bar=spam` in the Spring `Environment`. You can also supply the JSON as `spring.application.json` in a System variable:

```
$ java -Dspring.application.json='{"foo":"bar"}' -jar myapp.jar
```

or command line argument:

```
$ java -jar myapp.jar --spring.application.json='{"foo":"bar"}'
```

or as a JNDI variable `java:comp/env/spring.application.json`.

24.1 Configuring random values

The `RandomValuePropertySource` is useful for injecting random values (e.g. into secrets or test cases). It can produce integers, longs, uuids or strings, e.g.

```
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 (exclusive).

24.2 Accessing command line properties

By default `SpringApplication` will convert any command line option arguments (starting with '--', e.g. `--server.port=9000`) to a `property` and add it to the Spring `Environment`. As mentioned above, command line properties always take precedence over other property sources.

If you don't want command line properties to be added to the `Environment` you can disable them using `SpringApplication.setAddCommandLineProperties(false)`.

24.3 Application property files

`SpringApplication` will load properties from `application.properties` files in the following locations and add 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).

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

If you don't like `application.properties` as the configuration file name you can switch to another by specifying a `spring.config.name` environment property. You can also refer to an explicit location using the `spring.config.location` environment property (comma-separated list of directory locations, or file paths).

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

or

```
$ java -jar myproject.jar --spring.config.location=classpath:/default.properti
```

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

If `spring.config.location` contains directories (as opposed to files) they should end in `/` (and will 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 will be 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:

1. `file:./config/`
2. `file:./`
3. `classpath:/config/`
4. `classpath:/`

When custom config locations are configured, they are used in addition to the default locations. Custom locations are searched before the default locations. For example, if custom locations `classpath:/custom-config/,file:./custom-config/` are configured, the search order becomes:

1. `file:./custom-config/`
2. `classpath:custom-config/`
3. `file:./config/`
4. `file:./`
5. `classpath:/config/`
6. `classpath:/`

This search ordering allows you to specify default values in one configuration file and then selectively override those values in another. You can provide default values for you 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 overriden at runtime with a different file located in one of the custom locations.

If you use environment variables rather than system properties, most operating systems disallow period-separated key names, but you can use underscores

instead (e.g. `SPRING_CONFIG_NAME` instead of `spring.config.name`).

If you are running 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 using the naming convention `application-{profile}.properties`. The `Environment` has a set of default profiles (by default `[default]`) which are used if no active profiles are set (i.e. 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 irrespective of whether 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 via the `SpringApplication` API and therefore take precedence.

If you have specified any files in `spring.config.location`, profile-specific variants of those files will not be considered. Use directories in `spring.config.location` if you also 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 (e.g. from System properties).

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

You can also use this technique to create ‘short’ variants of existing Spring Boot properties. See the *Section 72.4, “Use ‘short’ command line arguments”*

> how-to for details.

24.6 Using YAML instead of Properties

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

> If you use 'Starters' SnakeYAML will be automatically provided via `spring-boot-starter`.

24.6.1 Loading YAML

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

For example, the following YAML document:

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

Would be transformed into these properties:

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

YAML lists are represented as property keys with `[index]` dereferencers, for example this YAML:

```
my:
   servers:
       - dev.bar.com
       - foo.bar.com
```

Would be transformed into these properties:

```
my.servers[0]=dev.bar.com
my.servers[1]=foo.bar.com
```

To bind to properties like that using the Spring `DataBinder` 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, e.g. this will bind to the properties above

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

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

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

Extra care is required when configuring lists that way as overriding will not work as you would expect. In the example above, when `my.servers` is redefined in several places, the individual elements are targeted for override, not the list. To make sure that a `PropertySource` with higher precedence can override the list, you need to define it as a single property:

```
my:
    servers: dev.bar.com,foo.bar.com
```

24.6.2 Exposing YAML as properties in the Spring Environment

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

24.6.3 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. For example:

```
server:
    address: 192.168.1.100
---
```

```
spring:
    profiles: development
server:
    address: 127.0.0.1
---
spring:
    profiles: production
server:
    address: 192.168.1.120
```

In the example above, the `server.address` property will be `127.0.0.1` if the `development` profile is active. If the `development` and `production` profiles are **not** enabled, then the value for the property will be `192.168.1.100`.

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

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

whereas in this example, the password is always set because it isn't attached to any profile, and it would have to be explicitly reset in all other profiles as necessary:

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

Spring profiles designated using the "spring.profiles" element may optionally be negated 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.

24.6.4 YAML shortcomings

YAML files can't be loaded via the `@PropertySource` annotation. So in the case that you need to load values that way, you need to use a properties file.

24.6.5 Merging YAML lists

As we have seen above, any YAML content is ultimately transformed to properties. That process may be counter intuitive when overriding “list” properties via a profile.

For example, assume a `MyPojo` object with `name` and `description` attributes that are `null` by default. Let’s expose a list of `MyPojo` from `FooProperties`:

```
@ConfigurationProperties("foo")
public class FooProperties {

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

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

}
```

Consider the following configuration:

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

If the `dev` profile isn’t active, `FooProperties.list` will contain one `MyPojo` entry as defined above. If the `dev` profile is enabled however, the `list` will *still* only contain one entry (with name “my another name” and description `null`). This configuration *will not* add a second `MyPojo` instance to the list, and it won’t merge the items.

When a collection is specified in multiple profiles, the one with highest priority is used (and only that one):

```
foo:
  list:
    - name: my name
      description: my description
    - name: another name
      description: another description
---
spring:
```

```
  profiles: dev
foo:
  list:
     - name: my another name
```

In the example above, considering that the `dev` profile is active, `FooProperties.list` will contain *one* `MyPojo` entry (with name “my another name” and description `null`).

24.7 Type-safe Configuration Properties

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

```
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("foo")
public class FooProperties {

    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("US

        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 POJO above defines the following properties:

- `foo.enabled`, `false` by default
- `foo.remote-address`, with a type that can be coerced from `String`
- `foo.security.username`, with a nested "security" 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`
- `foo.security.password`
- `foo.security.roles`, with a collection of `String`

Getters and setters are usually mandatory, since binding is via standard Java Beans property descriptors, just like in Spring MVC. There are cases where a setter may be omitted:

- 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 either accessed via an index (typically with YAML) or 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 example above)
- If nested POJO properties are initialized (like the `Security` field in the example above), a setter is not required. If you want the binder to create the instance on-the-fly using its default constructor, you will need a setter.

> Some people use Project Lombok to add getters and setters automatically. Make sure that Lombok doesn't generate any particular constructor for such type as it will be used automatically by the container to instantiate the object.

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

You also need to list the properties classes to register in the `@EnableConfigurationProperties` annotation:

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

> When `@ConfigurationProperties` bean is registered that way, the bean will have a conventional name: `<prefix>-<fqn>`, where `<prefix>` is the environment key prefix specified in the `@ConfigurationProperties` annotation and `<fqn>` 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 will be `foo-com.example.FooProperties`.

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

```
@Component
@ConfigurationProperties(prefix="foo")
public class FooProperties {

    // ... see above

}
```

This style of configuration works particularly well with the `SpringApplication` external YAML configuration:

```
# application.yml

foo:
    remote-address: 192.168.1.1
    security:
        username: foo
        roles:
          - USER
          - ADMIN

# additional configuration as required
```

To work with `@ConfigurationProperties` beans you can just inject them in the same way as any other bean.

```
@Service
public class MyService {

    private final FooProperties properties;

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

     //...

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

}
```

Using `@ConfigurationProperties` also allows you to generate meta-data files that can be used by IDEs to offer auto-completion for your own keys, see the Appendix B, *Configuration meta-data* appendix for details.

24.7.1 Third-party configuration

As well as using `@ConfigurationProperties` to annotate a class, you can also use it on public `@Bean` methods. This 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:

```
@ConfigurationProperties(prefix = "bar")
@Bean
public BarComponent barComponent() {
    ...
}
```

Any property defined with the `bar` prefix will be mapped onto that `BarComponent` bean in a similar manner as the `FooProperties` example above.

24.7.2 Relaxed binding

Spring Boot uses some relaxed rules for binding `Environment` properties to `@ConfigurationProperties` beans, so there doesn't need to be an exact match between the `Environment` property name and the bean property name. Common examples where this is useful include dashed separated (e.g. `context-path` binds to `contextPath`), and capitalized (e.g. `PORT` binds to `port`) environment properties.

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

```
@ConfigurationProperties(prefix="person")
public class OwnerProperties {

    private String firstName;

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

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

}
```

The following properties names can all be used:

Table 24.1. relaxed binding

Property	Note
`person.firstName`	Standard camel case syntax.

Property	Note
`person.first-name`	Dashed notation, recommended for use in `.properties` and `.yml` files.
`person.first_name`	Underscore notation, alternative format for use in `.properties` and `.yml` files.
`PERSON_FIRST_NAME`	Upper case format. Recommended when using a system environment variables.

24.7.3 Properties conversion

Spring will attempt 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 bean id `conversionService`) or custom property editors (via a `CustomEditorConfigurer` bean) or custom `Converters` (with bean definitions annotated as `@ConfigurationPropertiesBinding`).

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's not required for configuration keys coercion and only rely on custom converters qualified with `@ConfigurationPropertiesBinding`.

24.7.4 @ConfigurationProperties Validation

Spring Boot will attempt 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. Simply ensure that a compliant JSR-303 implementation is on your classpath, then add constraint annotations to your fields:

```
@ConfigurationProperties(prefix="foo")
@Validated
public class FooProperties {

    @NotNull
    private InetAddress remoteAddress;
```

```
    // ... getters and setters

}
```

In order to validate values of nested properties, you must annotate the associated field as `@Valid` to trigger its validation. For example, building upon the above `FooProperties` example:

```
@ConfigurationProperties(prefix="connection")
@Validated
public class FooProperties {

    @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 allows the bean to be created without having to instantiate the `@Configuration` class. This avoids any problems that may be caused by early instantiation. There is a property validation sample so you can see how to set things up.

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

24.7.5 @ConfigurationProperties vs. @Value

`@Value` is a core container feature and it does not provide the same features as type-safe Configuration Properties. The table below 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 to group them in a POJO annotated with `@ConfigurationProperties`. Please also be aware that since `@Value` does not support relaxed binding, it isn't a great candidate if you need to provide the value using environment variables.

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

Prev	Up	Next
23. SpringApplication	Home	25. Profiles