54. Properties & configuration

54.1 Externalize the configuration of SpringApplication

A `SpringApplication` has bean properties (mainly setters) so you can use its Java API as you create the application to modify its behavior. Or you can externalize the configuration using properties in `spring.main.*`. E.g. in `application.properties` you might have.

```
spring.main.web_environment=false
spring.main.show_banner=false
```

and then the Spring Boot banner will not be printed on startup, and the application will not be a web application.

> **Note**
> The example above also demonstrates how flexible binding allows the use of underscores (_) as well as dashes (-) in property names.

54.2 Change the location of external properties of an application

By default properties from different sources are added to the Spring `Environment` in a defined order (see *Chapter 21, Externalized Configuration* in the "Spring Boot features" section for the exact order).

A nice way to augment and modify this is to add `@PropertySource` annotations to your application sources. Classes passed to the `SpringApplication` static convenience methods, and those added using `setSources()` are inspected to see if they have `@PropertySources`, and if they do, those properties are added to the `Environment` early enough to be used in all phases of the `ApplicationContext` lifecycle. Properties added in this way have precedence over any added using the default locations, but have lower priority than system properties, environment variables or the command line.

You can also provide System properties (or environment variables) to change the behavior:

- `spring.config.name` (`SPRING_CONFIG_NAME`), defaults to `application` as the root of the file name.
- `spring.config.location` (`SPRING_CONFIG_LOCATION`) is the file to load (e.g. a classpath resource or a URL). A separate `Environment` property source is set up for this document and it can be overridden by system properties, environment variables or the command line.

No matter what you set in the environment, Spring Boot will always load `application.properties` as described above. If YAML is used then files with the ".yml" extension are also added to the list by default.

See `ConfigFileApplicationListener` for more detail.

54.3 Use "short" command line arguments

Some people like to use (for example) `--port=9000` instead of `--server.port=9000` to set configuration properties on the command line. You can easily enable this by using placeholders in `application.properties`, e.g.

```
server.port=${port:8080}
```

> **Tip**
> If you have enabled maven filtering for the `application.properties` you may want to avoid using `${*}` for the tokens to filter as it conflicts with those placeholders. You can either use `@*@` (i.e. `@maven.token@` instead of `${maven.token}`) or you can configure the `maven-resources-plugin` to use other delimiters.
>
> **Note**
> In this specific case the port binding will work in a PaaS environment like Heroku and Cloud Foundry, since in those two platforms the `PORT` environment variable is set automatically and Spring can bind to capitalized synonyms for `Environment` properties.

54.4 Use YAML for external properties

YAML is a superset of JSON and as such is a very convenient syntax for storing external properties in a hierarchical format. E.g.

```
spring:
   application:
       name: cruncher
   datasource:
       driverClassName: com.mysql.jdbc.Driver
       url: jdbc:mysql://localhost/test
server:
   port: 9000
```

Create a file called `application.yml` and stick it in the root of your classpath, and also add `snakeyaml` to your dependencies (Maven coordinates `org.yaml:snakeyaml`, already included if you use the `spring-boot-starter`). A YAML file is parsed to a Java `Map<String,Object>` (like a JSON object), and Spring Boot flattens the map so that it is 1-level deep and has period-separated keys, a lot like people are used to with `Properties` files in Java.

The example YAML above corresponds to an `application.properties` file

```
spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

See *Section 21.5, "Using YAML instead of Properties"* in the "Spring Boot features" section for more information about YAML.

54.5 Set the active Spring profiles

The Spring `Environment` has an API for this, but normally you would set a System profile (`spring.profiles.active`) or an OS environment variable (`SPRING_PROFILES_ACTIVE`). E.g. launch your application with a `-D` argument (remember to put it before the main class or jar archive):

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

In Spring Boot you can also set the active profile in `application.properties`, e.g.

```
spring.profiles.active=production
```

A value set this way is replaced by the System property or environment variable setting, but not by the `SpringApplicationBuilder.profiles()` method. Thus the latter Java API can be used to augment the profiles without changing the defaults.

See *Chapter 22, Profiles* in the "Spring Boot features" section for more information.

54.6 Change configuration depending on the environment

A YAML file is actually a sequence of documents separated by `---` lines, and each document is parsed separately to a flattened map.

If a YAML document contains a `spring.profiles` key, then the profiles value (comma-separated list of profiles) is fed into the Spring `Environment.acceptsProfiles()` and if any of those profiles is active that document is included in the final merge (otherwise not).

Example:

```
server:
   port: 9000
---

spring:
   profiles: development
server:
   port: 9001

---

spring:
   profiles: production
server:
   port: 0
```

In this example the default port is 9000, but if the Spring profile "development" is active then the port is 9001, and if "production" is active then it is 0.

The YAML documents are merged in the order they are encountered (so later values override earlier ones).

To do the same thing with properties files you can use `application-${profile}.properties` to specify profile-specific values.

54.7 Discover built-in options for external properties

Spring Boot binds external properties from `application.properties` (or `.yml`) (and other places) into an application at runtime. There is not (and technically cannot be) an exhaustive list of all supported properties in a single location because contributions can come from additional jar files on your classpath.

A running application with the Actuator features has a `configprops` endpoint that shows all the bound and bindable properties available through `@ConfigurationProperties`.

The appendix includes an `application.properties` example with a list of the most common properties supported by Spring Boot. The definitive list comes from searching the source code for `@ConfigurationProperties` and `@Value` annotations, as well as the occasional use of `RelaxedEnvironment`.