23. SpringApplication

The `SpringApplication` class provides a convenient way to bootstrap a Spring application that will be started from a `main()` method. In many situations you can just delegate to the static `SpringApplication.run` method:

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

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

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v1.3.5.RELEASE

2013-07-31 00:08:16.117  INFO 56603 --- [           main] o.s.b.s.app.SampleApplicati
2013-07-31 00:08:16.166  INFO 56603 --- [           main] ationConfigEmbeddedWebAppli
2014-03-04 13:09:54.912  INFO 41370 --- [           main] .t.TomcatEmbeddedServletCon
2014-03-04 13:09:56.501  INFO 41370 --- [           main] o.s.b.s.app.SampleApplicati
```

By default `INFO` logging messages will be shown, including some relevant startup details such as the user that launched the application.

23.1 Customizing the Banner

The banner that is printed on start up can be changed by adding a `banner.txt` file to your classpath, or by setting `banner.location` to the location of such a file. If the file has an unusual encoding you can set `banner.charset` (default is `UTF-8`).

You can use the following variables inside your `banner.txt` file:

Table 23.1. Banner variables

Variable	Description
`${application.version}`	The version number of your application as declared in `MANIFEST.MF`. For example `Implementation-Version: 1.0` is printed as `1.0`.
`${application.formatted-version}`	The version number of your application as declared in `MANIFEST.MF` formatted for display (surrounded with brackets and prefixed with `v`). For example `(v1.0)`.
`${spring-boot.version}`	The Spring Boot version that you are using. For example `1.3.5.RELEASE`.
`${spring-boot.formatted-version}`	The Spring Boot version that you are using formatted for display (surrounded with brackets and prefixed with `v`). For example `(v1.3.5.RELEASE)`.
`${Ansi.NAME}` (or `${AnsiColor.NAME}`, `${AnsiBackground.NAME}`, `${AnsiStyle.NAME}`)	Where `NAME` is the name of an ANSI escape code. See `AnsiPropertySource` for details.
`${application.title}`	The title of your application as declared in `MANIFEST.MF`. For example `Implementation-Title: MyApp` is printed as `MyApp`.

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

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

YAML maps `off` to `false` so make sure to add quotes if you want to disable the banner in your application.

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

23.2 Customizing SpringApplication

If the `SpringApplication` defaults aren't to your taste you can instead create a local instance and customize it. For example, to turn off the banner you would write:

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

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

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

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

23.3 Fluent builder API

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

The `SpringApplicationBuilder` allows you to chain together multiple method calls, and includes `parent` and `child` methods that allow you to create a hierarchy.

For example:

```
new SpringApplicationBuilder()
    .bannerMode(Banner.Mode.OFF)
```

```
    .sources(Parent.class)
    .child(Application.class)
    .run(args);
```

There are some restrictions when creating an `ApplicationContext` hierarchy, e.g. Web components **must** be contained within the child context, and the same `Environment` will be used for both parent and child contexts. See the `SpringApplicationBuilder` Javadoc for full details.

§ 23.4 Application events and listeners

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

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

If you want those listeners to be registered automatically regardless of the way the application

your listener(s) using the `org.springframework.context.ApplicationListener` key.

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

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

1. An `ApplicationStartedEvent` is sent at the start of a run, but before any processing except the registration of listeners and initializers.
2. An `ApplicationEnvironmentPreparedEvent` is sent when the `Environment` to be used in the context is known, but before the context is created.
3. An `ApplicationPreparedEvent` is sent just before the refresh is started, but after bean definitions have been loaded.
4. An `ApplicationReadyEvent` is sent after the refresh and any related callbacks have been processed to indicate the application is ready to service requests.
5. An `ApplicationFailedEvent` is sent if there is an exception on startup.

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

23.5 Web environment

A `SpringApplication` will attempt to create the right type of `ApplicationContext` on your behalf. By default, an `AnnotationConfigApplicationContext` or `AnnotationConfigEmbeddedWebApplicationContext` will be used, depending on whether you are developing a web application or not.

The algorithm used to determine a 'web environment' is fairly simplistic (based on the presence of a few classes). You can use `setWebEnvironment(boolean webEnvironment)` if you need to override the default.

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

It is often desirable to call `setWebEnvironment(false)` when using `SpringApplication` within a JUnit test.

23.6 Accessing application arguments

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

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

@Component
public class MyBean {

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

}
```

Spring Boot will also register a `CommandLinePropertySource` with the Spring `Environment`. This allows you to also inject single application arguments using the `@Value` annotation.

23.7 Using the ApplicationRunner or CommandLineRunner

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

The `CommandLineRunner` interfaces provides access to application arguments as a simple string array, whereas the `ApplicationRunner` uses the `ApplicationArguments` interface discussed above.

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

@Component
public class MyBean implements CommandLineRunner {

    // Do something...
    }

}
```


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

23.8 Application exit

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

In addition, beans may implement the `org.springframework.boot.ExitCodeGenerator` interface if they wish to return a specific exit code when the application ends.

23.9 Admin features

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

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

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

Prev Up Next

Part IV. Spring Boot features Home 24. Externalized Configuration