jax LONDON presents:

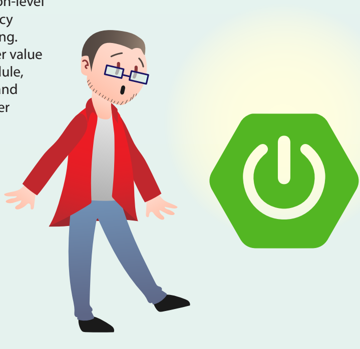
SPRING BOOT

By Michael Simons

-BASICS-

WHAT EXACTLY IS SPRING BOOT?

The term „Microservices Framework" is often used when talking about Spring Boot. This is only partly true: Spring Boot is suitable for small services as well as for monolithic applications. These services or applications can be pure Web APIs or web applications, command line programs, or orchestrating applications. In the end, Spring Boot orchestrates the Spring Framework. The Spring Framework provides application-level infrastructure ready, especially Dependency Injection and aspect-oriented programming. With these, building blocks become higher value concepts realized. Depending on the module, these may include transactions, security, and more. Spring Boot starts a Spring Container and configures - depending on the dependencies on the classpath (see also **What is a starter?**) - correspondingly further modules.



ARE THERE ANY ALTERNATIVES?

Coming from the enterprise Java world, MicroProfile must of course be mentioned. Further exciting developments are Quarkus and Micronaut. These developments are intriguing for several reasons. First, they confirm several decisions made by Spring Boot, starting with version management up to declarative configuration, metrics, health endpoints, and even project generators. On the other hand, the latter frameworks in particular have started a development that focuses on faster starting services, an important quality to ensure an elastic infrastructure.

Naturally, the teams behind Spring Boot also take up these issues.

`@SpringBootApplication`

Everything starts with the annotation **`@SpringBootApplication`**. It marks a single class as the central entry point into a Spring Boot application.

`@SpringBootApplication` is a composite annotation, consisting of:

- **`@SpringBootConfiguration`** (alias for **`@Configuration`** with **`proxyBeanMethods = true`**)
- **`@EnableAutoConfiguration`**
- **`@ComponentScan`**

The last two annotations are the most important ones: Firstly, they switch the search after Spring components (including **`@Component, @Service, @Controller`** and others) as the starter mechanisms.

Three rules apply to the **`@SpringBootApplication`**; the first two should be considered hard rules:

1. The class does not need any further annotations. All other, domain-oriented configurations should lie on corresponding **`@Configuration`** classes.

2. A class annotated with **`@SpringBootApplication`** should never be included in the root package (i.e. without **`package`** declaration). In this case, Spring would jump all classes starting from the root scan and try to find classes with known annotations.

3. Although it is possible to apply other methods annotated with **`@Bean`** to this class, it should be omitted for reasons of clarity. The class should be empty using the **`Main`** method.

The following listing shows what an "ideal" Spring Boot **`Main`** class looks like:

Standard Spring Boot `Main` Class

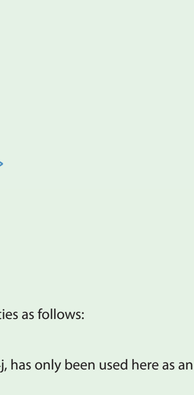
```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

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

VERSION MANAGEMENT AND ADMINISTRATION OF DEPENDENCIES

Version management and the administration of dependencies are essential functions of Spring Boot, even if they are usually extremely inconspicuous and are often unnoticed. For both Maven and Gradle based projects, Spring Boot provides mechanisms for application developers to do this.

Developers can create dependencies without declaring versions. Managed dependencies are published together with their versions under **Managed Dependency Coordinates**. Spring Boot is automatically tested with these versions.

If in certain cases it is mandatory to use a different version, it should not be declared manually but the **properties mechanism** is used to overwrite the version. First, the name of the corresponding property is determined in the **Version Properties** list and then set.



MAVEN

Define a property with the appropriate name and version:

```
<properties>
  <neo4j-ogm.version>3.2.10</neo4j-ogm.version>
</properties>
```

GRADLE

Create a **`gradle.properties`** in the project and define properties as follows:

```
neo4j-ogm.version = 3.2.10
```

Neo4j-OGM, a mapping framework for the graph database Neo4j, has only been used here as an example.

WHAT IS A STARTER?

Starter is the second important pillar of Spring Boot. They usually consist of a starter and an autoconfiguration module. The Starter Module provides all necessary dependencies, the **Autoconfiguration** Module can configure the Spring-Container, Spring-Module, and other technologies depending on various conditions.

These conditions include the presence or absence of classes in the classpath, the presence or absence of beans in the container, environment variables, properties, and more. These conditions are defined by special **`@Configuration`** classes that are defined using the methods described in Annotation **`@EnableAutoConfiguration`**.

A starter is added to a project by declaring the corresponding dependency. So that the corresponding configuration classes are found without additional code or scanning the entire classpath, a service loader mechanism similar to the Java Service Provider SPI is used.

MANUAL (SPECIFIC) CONFIGURATION

`Properties`

The Spring Boot Guide lists over 1000 properties that an application configuration can be: **Common Application Properties**. Of course, these properties are not always available. They only appear - apart from the Core Properties and some others - when the corresponding starter is on the classpath. Configuration using properties should always be given priority over programmatic configuration. It is flexible, well documented, and the underlying configuration classes can appropriately react to changes. Properties can originate from files (**`application.properties`** or **`application.yml`**), from environment variables or program argument. You can use different profiles that can be validated and much more. Special data types such as URLs, memory sizes, lists etc. are converted correctly.

`@Configuration`

Classes that are annotated with **`@Configuration`** are Spring components that affect the contents of the Spring container Their methods are annotated with **`@Bean`**, the return values of the methods are also used as bean a part of the Spring container. By default, the beans are named method.

These additional beans influence the automatic configuration by starter: Starter frequently checks if certain beans are already present. If so, starter can be completely "switched off" or take only what is true for a part of its configuration, because the user-specific bean already does that.

This can sometimes lead to irritation. In case of doubt, looking at a reference helps, in particular the annex **Autoconfiguration Classes**.

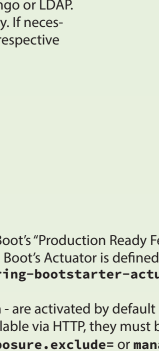
-IMPORTANT FEATURES-

TEST SUPPORT

The starter under the coordinates **`org.springframework.boot:spring-bootstarter-test`** brings all necessary dependencies to use with JUnit Jupiter, write AssertJ, and test with Hamcrest.

Important annotations are here:

- **`@SpringBootTest`** starts a complete Spring boot application. If Spring Web is on the class path, a **`Mock-Http-Server`** is used by default.
- **`@JsonTest`** only configures the support for JSON processing (configuration of the mapper, how Spring Boot does it etc.)
- **`@WebMvcTest`** only configures the web layer, services, and repositories that must be mounted. A **`MockMvc`** instance is available for testing the ready controllers.
- **`@WebFluxTest`** configures the reactive web layer, an instance of the class **`WebTestClient`** is ready for use.
- **`@DataXXXTest`** configures the database layer for the specified persistence layer. This can be JPA, JDBC, jOOQ, Neo4j, Redis, Mongo or LDAP. Tests roll transactions automatically. If necessary, an in-memory version of the respective store is used.



All the annotations mentioned are internally linked with **`@ExtendWith({SpringExtension.class})`** annotated, further JUnit annotations are not necessary for standard application cases.

Starting from the annotated test class, the package hierarchy searches upwards after the **`@SpringBootApplication`** annotated class. From there on, the configuration is then searched down again and selected according to the test slice.

Tests can be executed by classes that begin with **`@TestConfiguration`** and must be adjusted. **`@TestConfiguration`** has the advantage of being the added automatic Spring Boot configuration for tests and is not - like **`@Configuration`** classes in the Test path - replaced.

If the test slices are used, additional **`@AutoConfigureXXX`** annotations (e.g. **`@AutoConfigureWebTestClient`**) can be used to add further aspects.

ACTUATOR

The term **Actuator** includes Spring Boot's "Production Ready Features". These are metrics and health, but also tracing and auditing. Spring Boot's Actuator is defined by the dependency **`org.springframework.boot:spring-bootstarter-actuator.`**

All endpoints - except for **Shutdown** - are activated by default and, with a few exceptions, JMX is exposed via JMX. To make them available via HTTP, they must be exposed via **`management.endpoints.web.exposure.exclude=`** or **`management.endpoints.web.exposure.include=`** and can be activated either by wildcard (*) or by name (see **Endpoints**).

Endpoints are only secured if Spring Security is on the classpath. All web endpoints are available by default under **`/actuator/ID`**, for example **`/actuator/health`**. Some endpoints may only provide a subset of available information when no user is logged in.

Metrics are available under **`/actuator/metrics`**. Custom endpoints, health, and application information can be provided. Database starters often provide additional connectivity information. For example, Neo4j looks like this:

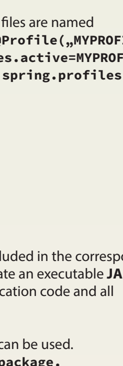
Example of health information using Neo4j:

```
{
  „status": „UP",
  „details": {
    „neo4j": {
      „status": „UP",
      „details": {
        „server": „Neo4j/4.0.0@localhost(127.0.0.1):7687",
        „database": „movies",
        „edition": „enterprise"
      }
    }
  }
}
```

LOGGING

Logging can be performed uniformly for Slf4j, Log4j and other facades under can be configured with the prefix **`logging.*. logging.file.*`** and **`logging.pattern.*`** define general aspects, under **`logging.level.LOGGERNAME`** the default level of the corresponding logger can be configured.

Via logging.group.**`GROUPNAME=LIST OF LOGGERS`** logging groups can be defined, independent of the library used. This group can then be used as described above and can be configured with **`logging.level.GROUPNAME`**



PROFILE

Both properties files and configuration classes can be profiled. Properties files are named **`application-MYPROFILE.properties`** and configuration classes are **`@Profile(„MYPROFILE")`**. When starting the application via a start parameter **`(--spring.profiles.active=MYPROFILE)`** or via the entry determined in the standard configuration under the **`key spring.profiles.active`**, of which profiles are active.

APPLICATION PACKAGING

Both Maven and Gradle have plug-ins available that are automatically included in the corresponding build descriptor when the application is generated. These plug-ins generate an executable **JAR** or **WAR** file during the package phase. This archive contains the actual application code and all dependencies.

If this does not fit in the planned deployment scenario, a different layout can be used. The corresponding Maven or Gradle property is called **`spring-boot.repackage.`** layout. You can use **JAR, WAR, ZIP, DIR,** or **NONE. DIR** is one option for layers of Docker Images.

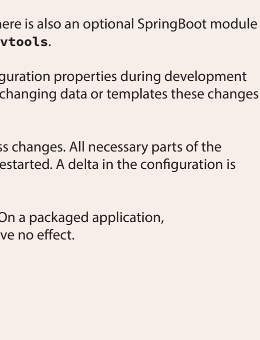
In Spring Boot 2.3, optimized build packs are available. These build packs codify various best practices for creating Docker images for Spring Boot applications. With an mvn **`spring-boot:build-image`** or Gradle **`bootBuildImage`** you can generate an image. In this case, the new layout **LAYERED_JAR** is used. It pushes libraries into a separate Docker Layer, which usually remains constant for a long time during development. to speed up image creation and keep deltas small for new deployments.

-MISC-

LAZY INITIALIZATION

Spring boot applications can work with Lazy Initialization. This means that components can start as late as possible, namely when it is needed will be. This will speed up application start in some cases, but can also lead to late errors: Whenever beans are incorrectly configured but are not needed directly at the start.

If it is ensured that the application is configured correctly, the feature for an application is activated via **`spring.main.lazy-initialization=true`**.



DEVELOPER TOOLS

Of course, Spring Boot itself is a "development tool". But there is also an optional SpringBoot module called **`org.springframework.boot:spring-boot-devtools`**.

Developer Tools changes the default values of some configuration properties during development time. For example, most caches are disabled, so that after changing data or templates these changes become immediately visible.

The most striking feature is an automated restart after class changes. All necessary parts of the application that are affected by a change are completely restarted. A delta in the configuration is logged.

This should only be declared as an optional dependency. On a packaged application, which is started with **`java -jar`**, the Developer Tools have no effect.

START.SPRING.IO

Josh Long, Developer-Advocate at VMWare Tanzu (formerly Pivotal), likes to refer to start.spring.io as the best place on the Internet and a panacea for various ailments.

This may be exaggerated, but the **Spring Initializr** is nevertheless a very practical application. In a form where you enter the basic data of a new service, you must enter:

- Project Coordinates
- Name
- Language (Java, Kotlin or Groovy)
- Java version and Spring Boot version
- and of course the dependencies

The result in the download is a zip file containing a Build Descriptor, a **formalpara_title**, a basic test, Git Ignore and a README with information about dependencies and build system.

Spring Intializr can also be used via **cURL**, so it can be used in automated deployments. Of course, three major IDEs also take advantage of this functionality: In IDEA, NetBeans, and Eclipse you can generate new Spring Boot projects directly from the IDE using the Initializr.

As of 2020, the application is no longer comparable with the initial version of 2015/2016: Since then, there is now a preview functionality, a "share" link, and much more. On top of that, the open source software based on Spring Boot is available. Now that **spring-io / initializr** is available, the initializr can be adapted for your own purposes and can be used in your own setups.

HOW-TO-GUIDES

Under **"How-to-Guides"**, the Spring Boot team publishes answers to "How do I..." questions. These range from configuration to properties and the adaptation or the selection of the embedded web server up to database access, batch processing, and much more.

jax LONDON

jax.de

Illustrations: © Software & Support Media/Bianca Röder