Spring Boot 3

Infographic

by Michael Simons

Basics

What exactly is Spring Boot?

The term "microservices framework" is often used when talking about Spring Boot. This is only partially true, because 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.

At the end of the day, Spring Boot orchestrates the Spring Framework. The Spring framework provides application-level infrastructure, specifically dependency injection and aspect-oriented programming. These building blocks are used to implement higher-level concepts. Depending on the module, these include transactions, security, and more.

Spring Boot starts a Spring container and configures – depending on dependencies on the class path (see also: What is a starter?) – other modules accordingly.

Are there alternatives?

Since the first version of this infographic was published in late 2020, the Java world has not stood still. Rather, the contrary is true. Quarkus [1] and Micronaut [2] are relevant alternatives. Both have the shared goals of improving startup times and requiring fewer resources. These goals are achieved on the JVM and in the GraalVM Ahead-Of-Time (AOT) compiler. While the differences on API view to Spring Boot were often marginal (basically, it doesn't matter if jakarta.inject or Spring @Autowired annotations, Jakarta-RS, or Spring WebMVC is used), Quarkus and Micronaut differed drastically from Spring and Spring Boot.

Quarkus and Micronaut shift as much work as possible into the compile-time of a program to create as closed a world as possible. Closed means that as much as possible is known before the application starts, including configuration, CDI beans, resources, entities, and more. This all goes hand in hand with the goal of avoiding Java reflection. Starting from searching for (JPA) entities, JAX-RS classes to configuring or reconfiguring things at runtime the wiring of collaborators (i.e. autowiring and injecting beans respectively). In short, all the things that we as frameworks users with dependency injection and configuration mechanisms take for granted.

The approaches differ in the details. While the Quarkus team decided to develop its own pre-processor for Java with build tools, Micronaut relies on annotation processing. Yes, it is the same annotation processing that also takes place in Spring, but at compile time, not at runtime. Quarkus generates bytecode, Micronaut bytecode or source code, depending on the use case and the target mode (JVM or AOT). See Micronaut Framework Code Generation [3].

Why is this brief outlook important? Competition stimulates business and Spring Boot 3.x and Spring Framework 6.x are significantly influenced by the success of the combinations GraalVM + Micronaut and GraalVM + Quarkus in a positive way.

The minimum Java version and other dependencies or: What does the upgrade from 2 to 3 mean?

Java 17

Spring Framework 6 and therefore, all Spring Boot 3 applications require Java 17. Spring Framework 5.3 and Spring Boot 2.7 are the last versions that can be compiled with Java 8 or 11. Of course, this does not mean that libraries still compiled for Java 8 cannot be used with Spring Boot 3. Generally, this isn't a problem. The problems of various libraries and build tools with Java versions beyond 8 have been solved by now. Problems with the module system or the stronger restrictions of unsafe class-

es (*sun.misc.Unsafe*, etc) have only a few effects.

Nevertheless, there are strong reasons to stay on Java 8. Spring Framework 5.3 will be open source until 2024 and receives commercial support until 2026. However, these reasons should not be found in the JDK, Spring Framework 5.3 and the corresponding Spring Boot 2.7 also work perfectly on JDK 11 and 17. A possible reason would be the upgrade to Hibernate 6.1, which is also part of Spring Boot 3. There's usually much more to do there than just installing a new JDK.

Jakarta EE 9+

Spring Boot 3 completes the transition from Java EE dependencies to corresponding Jakarta EE dependencies. Affected dependencies include the Java Persistence API (JPA), Java Servlets, Java Message Service (JMS), and more.

The Spring team removed all dependencies from dependency management that have not been migrated to Jakarta EE by the authors. Naturally, this has consequences. Applications will no longer compile and even after manually adding the respective libraries (for example, Apache ActiveMQ), they will no longer be automatically configured. Manual configuration isn't a solution either, since Java EE and Jakarta EE libraries aren't shareable.

Depending on how deep your application is with Java EE annotations and dependencies, the effort to upgrade varies between search-and-replace of package names or searching for new dependencies with more or less migration effort.

@SpringBootApplication

It all starts with the *@SpringBootApplication* annotation. 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. They turn on the search for Spring components (including *@Component*, *@Service*, *@Controller* and others) and the starters mechanism.

Three rules apply to *@SpringBootApplication*. The first two should be considered hard rules:

1. The class does not need any further annotations. All other specialized configurations should be located on corresponding *@Configuration* classes.
2. A class annotated with *@SpringBootApplication* should never be located in the root package (i.e. without a package declaration). In this case, Spring would scan all classes starting from the root and try to find classes with known annotations.
3. Although it is possible to define other methods annotated with *@Bean* on this class, it should be avoided for clarity. The class should be empty except for the Main method.

Listing 1 shows what an "ideal" Spring Boot Main class looks like.

Listing 1. 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 dependency management

Version management and dependency management are essential features of Spring Boot, even though they tend to be extremely unobtrusive and often go unnoticed. Spring Boot is available for both Maven and Gradle based projects mechanisms that take care of this for application developers. Gradle has been the default since Spring Boot 3.

Developers can declare dependencies without specifying versions. Managed dependencies are listed along with their versions under Managed Dependency Coordinates [4].

Spring Boot is automatically tested with these versions.

If in certain cases, a different version is mandatory, it should not be declared manually. Instead, the properties mechanism should be used to override this version. First, the name of the corresponding property is determined in the list of version properties [5] and set accordingly.

Maven

Define a property with an appropriate name and version:

```
<properties>
  <neo4j-java-driver.version>5.10.0</neo4j-java-driver.version>
</properties>
```

Gradle

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

```
neo4j-java-driver.version = 5.10.0
```

The Neo4j Java Driver, a library similar to a JDBC driver, but for the Neo4j graph database [6] of the same name, was used here as an example.

What is a starter? (H3)

Starters are the second important mainstay of Spring Boot. They usually consist of a starter module and an autoconfiguration module. The starter module provides all the necessary dependencies. The autoconfiguration module can configure the Spring container, Spring modules, and other technologies depending on a wide variety of 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 loaded using the *@EnableAutoConfiguration* annotation named in *@SpringBootApplication*.

Starters are added to a project by declaring the appropriate dependency. To find the corresponding configuration classes without writing additional code or scanning the entire classpath, a service loader mechanism similar to Java Service Provider SPI is used.

To write the first (or even most further Spring Boot applications), you rarely need to write a separate starter. Nevertheless, it's helpful to know how the mechanism works and understand that the presence or absence of classes affects configuration. Generally, it is recommended to use dependencies sparingly, both with starters and with other libraries. The JDK itself provides functionality for many use cases in current versions. You rarely need to blindly put Google Guava on the classpath. The same goes for starters. I've seen too many projects where it was blindly assumed that Spring Data Neo4j also requires Spring Data JPA. The more starters on the classpath, the harder it can be to investigate startup errors.

Manual (specific) configuration

Properties

The Spring Boot manual lists over 1000 properties that can be used to configure an application: Common Application properties [7]. Of course, not all properties are always available. They appear – apart from the core properties and some others – only if the corresponding starter is on the classpath. The configuration over properties should be given priority over programmatic configuration. They're flexible, well-documented and the underlying configuration classes can react to changes. Properties can come from files (application.properties or application.yml), environment variables, or program arguments. They can be linked to different profiles, validated, and much more. Special data types like URLs, memory sizes, lists, and more are converted correctly.

@Configuration

Classes annotated with *@Configuration* are Spring components that affect the contents of the Spring container: Their methods are annotated with *@Bean*, and the return values of the methods also become a part of the Spring container as a bean. By default, the beans are given the name of the method.

These additional beans affect the automatic configuration by starters. Starters often check if certain beans are already present. If so, starters can be completely "switched off" or only take care of part of their configuration, since the user-specific bean already does that.

This can sometimes lead to problems. In case of doubt, checking the reference [8], especially the appendix Autoconfiguration Classes [9] helps.

Important features

Test Support

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

Important annotations include:

- *@SpringBootTest* starts a full Spring boot application. If Spring web is on the classpath, a mock http server is used by default.
- *@JsonTest* configures only JSON processing support (configuring the mapper like Spring Boot, etc.)
- *@WebMvcTest* configures only the web layer, services and repositories need to be mocked. A MockMvc instance is available to test the controllers.
- *@WebFluxTest* configures the reactive weblayer, an instance of the WebTestClient class is provided.
- *DataXXXTest* configures the database layer for the specified persistence layer. This can be JPA, JDBC, jOOQ, Neo4j, Redis, Mongo or even LDAP. Tests roll back transactions automatically. If needed, an in-memory version of the respective store is used

All annotations mentioned are internally annotated with *@ExtendWith({SpringExtension.class})*. Further JUnit annotations are not needed for standard use cases.

Starting from the annotated test class, the package hierarchy is searched upwards for a class annotated with *@SpringBootApplication*. From there, configuration is searched back down and selected according to the test slice.

Tests can be customized by classes annotated with *@TestConfiguration*. The advantage of *@TestConfiguration* is that it supplements the automatic Spring boot configuration for tests and unlike *@Configuration* classes in the test path, it replaces them.

If the test slices are used, additional aspects can be added with extra *@AutoConfigureXXX* annotations (for example *@AutoConfigureWebTestClient*).

Testcontainers and @ServiceConnection

The examples shown below can be replicated with little effort in an application generated in the Spring Initializr with the link in [10].

Test containers are an open source framework that enables lightweight, single-use instances of databases, message brokers, web browsers, and more. Virtually anything that can run inside a Docker container can be used as a test container. While older Spring Boot versions still needed manual configuration for service connections like JDBC URLs, or the URL of the Neo4j driver, this is no longer necessary in Spring Boot 3.1.

@ServiceConnection is a new annotation that provides access to various services along with a hierarchy of interfaces, starting with *ConnectionDetails*. The annotation marks a method as a source for test containers. The application context is able to derive a matching instance of the *ConnectionDetails* interface from the container. The root interface has no methods and serves only as a marker. Subinterfaces such as *JdbcConnectionDetails* or *Neo4jConnectionDetails* provide the minimum necessary information to provide a corresponding connection. Specific instances appear at the next level, usually based on properties or halt: container connections. The beauty is that we usually don't have to deal with it. Implementations are available for:

- Cassandra
- Couchbase
- Elasticsearc
- General relational databases via JDBC and some specialized containers (including MariaDB, MySQl, Oracle, PostgreSQL)
- Kafka
- MongoDB
- Neo4j
- RabbitMQ
- Redpanda

In the test scope, the required general dependency is *org.springframework.boot:spring-boot-testcontainers*. Depending on the service, the corresponding test container module must be included. We'll stick with Neo4j and use *org.testcontainers:neo4j* in the test scope. Both dependencies are assumed in the following example.

At development time

Let's assume that we created additional classes around our *MyApplication* class from Listing 1. For example, a domain class annotated with *@Node*, a corresponding Neo4j repository, various services and controllers. We wanted to do Test-Driven-Development (TDD), but unfortunately, something came up and we developed as usual "on-the-fly". In other words, we start the application from the IDE and use HTTPie [11] or something similar to call the interfaces. Wouldn't it be nice if we didn't have to edit *application.properties* or set environment variables to connect the application against a test instance of our database?

The following class in the test scope lets us do that. It provides an inner configuration class – though it doesn't have to be annotated itself – that enriches the normal configuration with the new *with* method of the SpringApplication class. It's a delightfully explicit approach. The class *MyApplicationWithDevServices* must be in the test scope, but apart from that, it can be started normally from the IDE.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.testcontainers.service.connection.
                                                    ServiceConnection;
import org.springframework.context.annotation.Bean;
import org.testcontainers.containers.Neo4jContainer;

public class MyApplicationWithDevServices {

    static class ContainerConfig {

        @Bean
        @ServiceConnection
        public Neo4jContainer<?> neo4jContainer() {
            return new Neo4jContainer<>("neo4j:5").withReuse(true); (1)
        }
    }

    public static void main(String[] args) {
        SpringApplication.from(MyApplication::main)
            .with(ContainerConfig.class) (2)
            .run(args);
    }
}
```

(1) *withReuse* enables the reusability of test containers with the same configuration, i.e. they are started only once

(2) Additional, explicit configuration is used here

In the startup log, you'll see that the Neo4j driver automatically connects to the instance in the test container. The application can be used directly.

When testing

To use a container in testing, you can annotate the inner class *ContainerConfig* accordingly:

```
import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.TestConfiguration;
```

```
import org.springframework.boot.testcontainers.service.connection.
                                                    ServiceConnection;
import org.springframework.context.annotation.Bean;
import org.testcontainers.containers.Neo4jContainer;

@SpringBootTest
class MyApplicationTests {

    @TestConfiguration(
            proxyBeanMethods = false)
    public static class ContainerConfig {

        @Bean
        @ServiceConnection
        public Neo4jContainer<?> neo4jContainer() {
            return new Neo4jContainer<>("neo4j:5");
        }
    }

    @Test
    void repositoryIsConnectedAndUsable(
        @Autowired MovieRepository movieRepository
    ) {
        var movie = movieRepository.save(new Movie("Barbieheimer"));
        assertThat(movie.getId()).isNotNull();
    }
}
```

Of course, *ContainerConfig* can also reside in a top-level class, which is added to a *@SpringBootTest* via *@Import* annotation or added to the application via *.with* method, depending on the use case. There are a few more variants, but they mostly use annotations from the JUnit5 module of Testcontainers. It's useful outside of Spring applications, along with the test annotations used. But it can quickly lead to a scenario where both the test containers and Spring mechanisms are trying to manage the lifecycle of containers. I recommend for Spring Boot 3.1+ applications to use the above variant based purely on Spring annotations.

Actuator

The term "Actuator" covers Spring Boots "Production-Ready-Features". These are metrics and health, tracing, and auditing.

Spring Boots Actuator is defined by the dependency *org.springframework.boot:spring-boot-starter-actuator*.

All endpoints – except Shutdown are enabled by default and exposed via JMX with few exceptions when JMX is enabled. To make them available via HTTP, they must be enabled via *management.endpoints.web.exposure.exclude=* or *management.endpoints.web.exposure.include=* either by wildcard *(*)* or by name (see "Endpoints" in [12]).

Endpoints are only secured if Spring Security is on the classpath.

All web endpoints are available by default at */actuator/ID*, for example, */actuator/health*. Some endpoints may only provide a subset of information when no user is logged in.

Metrics are available at */actuator/metrics*.

Custom endpoints, health, and application information can be provided. Starters for databases often contribute additional information about connectivity. For the above example, the response looks like this (Available at *http localhost:8080/actuator/health*)

Listing 2: Health information example

```
{
  "components": {
    "diskSpace": {
      "details": {
        "exists": true,
        "free": 766255230976,
        "path": "/Users/msimons/Projects/temp/javamaginfografik/.",
        "threshold": 10485760,
        "total": 994662584320
      },
      "status": "UP"
    },
    "neo4j": {
      "details": {
        "database": "neo4j",
        "edition": "community",
        "server": "5.8.0@localhost:50745"
      },
      "status": "UP"
    },
    "ping": {
      "status": "UP"
    }
  },
  "status": "UP"
}
```

Details are not available by default. They are returned if Spring Security is on the classpath and an authorized user makes the request or the application is configured accordingly.

Logging

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

logging.group.GRUPPENNAME=LIST OF LOGGERS can be used to define own logging groups, independent of the library used. This group can be configured as described above with *logging.level.GROUPNAME*.

Profiles

Both properties files and configuration classes can be provided with a profile. Properties files are named *application-MYPROFILE.properties* and configuration classes with *@Profile("MYPROFILE")*. When the ap-

plication is started, a start parameter *(--spring.profiles.active=MYPROFILE)* or an entry in the default configuration under the *spring.profiles.active* key can be used to specify which profiles are active.

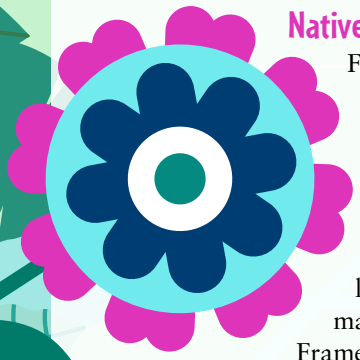
Application packaging

Plugins are available for both Maven and Gradle that are automatically included in the corresponding build descriptor when the application is generated. These plugins generate an executable jar or war file during the *package* phase. This archive contains the actual application code as well as 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. *JAR, WAR, ZIP, DIR* or *NONE* are available. *DIR* is a way to build layers for Docker images.

Since Spring Boot 2.3, optimized build packs are available. These build packs codify various best practices for building Docker images for Spring Boot applications.

An image can be generated using an *mvn spring-boot:build-image* and *gradle bootBuildImage*, respectively. In this case, the new *LAYERED_JAR* layout is used. It pushes libraries into their own Docker layer, which tends to stay constant longer during development, speeding up image creation and keeping deltas small for new deployments.



Native applications with GraalVM

For several years, GraalVM [13] has enabled the translation of Java programs into native, operating system-specified binaries called native images. This is usually accompanied by a drastically improved startup performance. Spring Boot 3 and Spring Framework 6 and all Portfolio projects are compatible with GraalVM Native-Image. The only thing needed is the following build plugin in the case of Maven:

Including the GraalVM build tool for Maven

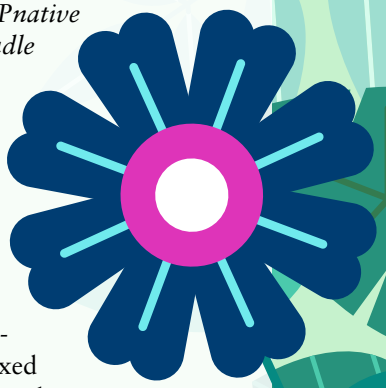
```
<build>
  <plugins>
    <plugin>
      <groupId>org.graalvm.buildtools</groupId>
      <artifactId>native-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

A corresponding Gradle build descriptor looks like this:

```
plugins {
  id 'org.graalvm.buildtools.native' version '0.9.23'
}
```

The usual Spring Boot specific plugins are also required, but have not been presented here for clarity.

The application is built for the current operating system using *mvn -Pnative native:compile* and *gradle nativeCompile* respectively. *mvn -Pnative spring-boot:build-image* and *gradle bootBuildImage* use buildpacks.



Technically, Spring supports GraalVM. Source code is generated that statically describes the application context. The application is booted until the bean definitions are available. This information is used for source code generation. Additionally, various modules contain fixed type hints that help GraalVM to make a safe assumption about the "closed world" of the application.

The application starts with *./target/javamaginfografik* in less than 0.5 seconds, opening a database connection to Neo4j.

Misc

Lazy Initialization

Spring Boot applications can use lazy initialization and start components as late as possible when needed. Sometimes this speeds up the application's start, but it can also cause late errors. For instance, whenever beans are configured incorrectly, but are not needed directly at startup.

If you make sure that the application is configured correctly, the feature can be enabled with for an application with *spring.main.lazy-initialization=true*.

Developer Tools

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

Developer tools change the default values of some configuration properties at development time. For example, most caches are disabled so after data or templates are changed, they become immediately visible.

The most noticeable feature is an automated restart when classes are changed. All necessary parts of the application 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 started with *java -jar* the developer tools have no effect.

start.spring.io

Josh Long [14], Developer Advocate at VMWare Tanzu (previously Pivotal) likes to call start.spring.io [15] the best place on the Internet and a cure-all for various aches and pains.

That may be an exaggeration, but *Spring Initializr* is an exceedingly practical application. The basic key data of a new service can be entered in a form:

- Project coordinates
- Name
- Language (Java, Kotlin or Groovy)

- Java version and Spring Boot version
- And, of course, the dependencies

It creates a downloadable zip file containing a build descriptor, a Listing 1, a basic test, Git Ignore, and a README with dependency and build system information.

The Spring initializer can also be used with cURL, so that it can also be used in automated deployments. Of course, the three major IDEs take advantage of this functionality. In IDEA, NetBeans and Eclipse, new Spring Boot projects can be generated directly from the IDE via the initializr.

As of 2023, the application is no longer comparable to the Initializr of 2015/2016. In the meantime, there is a preview functionality, a "Share" link, and one more. On top of that, the application is open source software based on Spring Boot. Sources are available at spring-io / initializr [16], so the Initializr can be adapted to your own purposes and used in your setups.

How-to Guides

In "How-to" guides [17], the Spring Boot team publishes answers to "How do I..." questions. These range from configuration to properties to customization or selection of the embedded web server to database access, batch processing and much more.

Michael Simons is a father, husband, and athlete (the latter, perhaps only in his imagination). He is a Java Champion, co-founder, and current director of Euregio JUG. Michael is very involved with the Spring ecosystem both in Germany through his German-language Spring Boot book, and also internationally. Spring is a recurring topic on his blog. This won't change anytime soon, because Michael works as a software engineer at Neo4j and deals with the Spring Data Module for the Neo4j graph database of the same name.

Links

[1] https://quarkus.io/

[2] https://micronaut.io/

[3] https://micronaut.io/2023/03/21/micronaut-framework-code-generation/#:~:text=Micronaut%20Framework%20is%20an%20implementation,where%20extensibility%2Fmodification%20is%20required

[4] https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-dependency-versions.html

[5] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#appendix.dependency-versions.properties

[6] https://neo4j.com/

[7] https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html

[8] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle

[9] https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-auto-configuration-classes.html#auto-configuration-classes

[10] https://start.spring.io/#!type=maven-project&language=java&platformVersion=3.1.2&packaging=jar&jvmVersion=17&groupId=com.example&artifactId=javamaginfografik&name=javamaginfografik& description=Demo%20project%20for%20Spring%20Boot&packageName=com.example.javamaginfografik&dependencies=data-neo4j,actuator,native,testcontainers

[11] https://httpie.io/

[12] https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html#production-ready-endpoints

[13] https://www.graalvm.org/

[14] https://twitter.com/starbuxman

[15] https://start.spring.io/

[16] https://github.com/spring-io/initializr/

[17] https://docs.spring.io/spring-boot/docs/current/reference/html/howto.html