**16 Best Practices in Spring Boot Production**

Spring Boot is the most popular Java framework for developing micro services. In this article, I’ll share with you the best practices I’ve adopted since 2016 for using Spring Boot in professional development. These are based on my personal experience and articles by some well-known Spring Boot experts.

In this article, I’ll focus on Spring Boot-specific practices (and most of the time, also apply to Spring projects). The best practices are listed below, in no particular order.

**1. Use a custom BOM to maintain third-party dependencies**

This practice is based on my experience in actual projects.

The Spring Boot project itself uses and integrates a large number of open source projects, which help us maintain these third-party dependencies. But some are not included in the actual project use, which requires us to maintain the version in the project. If a large project includes many undeveloped modules, it will be very cumbersome to maintain.

How to do it? In fact, Spring IO Platform does just that. It is itself a sub-project of Spring Boot and maintains other third-party open source libraries. We can learn from Spring IO Platform to write our own basic project platform-bom, and all business module projects should be introduced in the form of BOM. In this way, when upgrading a third-party dependency, you only need to upgrade the version of this dependency.

<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>io.spring.platform</groupId>  
 <artifactId>platform-bom</artifactId>  
 <version>Cairo-SR3</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
</dependencyManagement>

**2. Use automatic configuration**

A major feature of Spring Boot is the use of auto-configuration. This is part of Spring Boot that simplifies your code and makes it work. Autoconfiguration is activated when a specific jar file is detected on the classpath.

The easiest way to use it is to rely on Spring Boot Starters. So if you want to integrate with Redis you can first include:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>

If you want to integrate with MongoDB, you need this:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-mongodb</artifactId>  
</dependency>

With the help of these starters, these tedious configurations are well integrated and work together, and they are all tested and proven. This is very helpful in avoiding the dreaded Jar hell.

<https://dzone.com/articles/what-is-jar-hell>

Certain configuration classes can be excluded from auto-configuration by using the following annotation attributes:

@EnableAutoConfiguration（exclude = {ClassNotToAutoconfigure.class}）

But it should only be done when absolutely necessary.

Official documentation on auto-configuration can be found here:

<https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-auto-configuration.html.>

**3. Use Spring Initializer to start a new Spring Boot project**

This best practice is from Josh Long (Spring Advocate, @starbuxman).

Spring Initializer provides a super easy way to create a new Spring Boot project and load possible dependencies according to your needs.

<https://start.spring.io/>

Creating an application with Initializer ensures that you get tested and verified dependencies that work with Spring auto-configuration. You might even discover some new integrations that you might not be aware of.

**4. Consider creating your own auto-configurations for common organizational problems**

This one also comes from Josh Long (Spring Advocate, @starbuxman) — this practice is for advanced users.

If you work in a company or team that relies heavily on Spring Boot and have common problems to solve, then you can create your own auto-configuration.

This task is a lot of work, so you need to consider when the benefits are worth the investment. It is easier to maintain a single automatic configuration than multiple slightly different custom configurations.

If this provided Spring Boot configuration is released as an open source library, it will greatly simplify the configuration work for thousands of users.

**5. Correctly design the code directory structure**

Although you are allowed a lot of freedom, there are some basic rules worth following to design your source code structure.

Avoid using default packages. Make sure everything (including your entry point) is in a well-named package so you can avoid surprises related to assembly and component scanning;

Keep Application.java (the application’s entry class) in the top-level source directory;

I recommend putting controllers and services in function-oriented modules, but this is optional. Some very good developers recommend putting all the controllers together. No matter what, stick to one style!

**6. Keep @Controller concise and focused**

Controller should be very simple. You can read about the controller pattern section in GRASP here. You want the controller to act as a coordinator and delegate, not perform the actual business logic. Here are the main practices:

<https://en.wikipedia.org/wiki/GRASP(object-orienteddesign)#Controller>

1. The controller should be stateless! Controllers are singletons by default, and any state can cause a lot of problems;

2. The controller should not execute business logic, but rely on delegation;

3. The controller should handle the HTTP layer of the application, this should not be passed to the service;

4. Controllers should be designed around use cases/business capabilities.

To dive into this content, you need to learn more about best practices for designing REST APIs. It’s worth learning whether you want to use Spring Boot or not.

**7. Build @Service around business functions**

Service is another core concept of Spring Boot. I find it best to structure services around business functions/domains/use cases (whatever you want to call them).

Designing services with names like AccountService, UserService, in the application is more appropriate Payment Servicethan those like DatabaseService, ValidationService, .CalculationService

You can decide to use a one-to-one mapping between Controller and Service, that would be ideal. But this does not mean that services cannot call each other!

**8. Make the database independent of the core business logic**

I wasn’t sure before how to best handle database interactions in Spring Boot. After reading Robert C. Martin’s “Clear Architecture” it was much clearer to me.

You want your database logic to be separated from the service. Ideally, you don’t want the service to know which database it’s talking to, which requires some abstraction to encapsulate object persistence.

Robert C. Martin strongly states that your database is a “detail”, which means not coupling your application to a specific database. Few people would switch databases in the past, and I’ve noticed that using Spring Boot and modern microservice development makes things much faster.

**9. Keep business logic out of Spring Boot code**

Considering the lessons of “Clear Architecture”, you should also protect your business logic. It’s very tempting to mix all kinds of Spring Boot code together…don’t do it. If you can resist the temptation, you will keep your business logic reusable.

Part of the service usually becomes a library. Easier to create without removing a lot of Spring annotations from your code.

**10. Constructor injection is recommended**

This practice comes from Phil Webb (Spring Boot project lead, @phillip\_webb).

One way to keep your business logic safe from Spring Boot code is to use constructor injection. Not only is the @Autowiredannotation optional on the constructor, but it also makes it easy to instantiate beans without Spring.

**11. Familiar with the concurrency model**

One of the most popular articles I’ve written is “Introduction to Concurrency in Spring Boot”. I think the reason for this is that this field is often misunderstood and ignored. Problems can arise if used incorrectly.

<https://www.e4developer.com/2018/03/30/introduction-to-concurrency-in-spring-boot/>

In Spring Boot, Controller and Service are singletons by default. This introduces possible concurrency issues if you’re not careful. You are also generally dealing with a limited thread pool. Please familiarize yourself with these concepts.

If you’re using the new WebFlux-style Spring Boot application, I’ve explained how it works in “Spring’s WebFlux/Reactor Parallelism and Backpressure”.

**12. Strengthen the externalization of configuration management**

This goes beyond Spring Boot, although this is a common problem when people start creating multiple similar services…

You can handle the configuration of your Spring application manually. If you’re dealing with multiple Spring Boot applications, you need to make configuration management more powerful.

I recommend two main approaches:

1. Use a configuration server, such as Spring Cloud Config;

2. Store all configurations in environment variables (can be configured based on git repository).

Either of these options (the second option is a bit more) requires you to do less work in DevOps, but that’s pretty common in the microservices world.

**13. Provide global exception handling**

You really need a consistent way to handle exceptions. Spring Boot provides two main approaches:

1. You should use the HandlerExceptionResolver to define a global exception handling strategy; 2. You can also add the @ExceptionHandler annotation to the controller, which may be useful in some specific scenarios.

This is pretty much the same as in Spring, and Baeldung has a detailed article on error handling with REST and Spring that is well worth a read.

<https://www.baeldung.com/exception-handling-for-rest-with-spring>

**14. Use a logging framework**

You’re probably already aware of this, but you should be using Logger for logging instead of doing it manually with System.out.println(). This is easily done in Spring Boot with little to no configuration. Just get the logger instance of the class:

Logger logger = LoggerFactory.getLogger(MyClass.class);

This is important because it allows you to set different logging levels according to your needs.

**15. Test your code**

This is not specific to Spring Boot, but it needs a reminder — test your code! If you don’t write tests, you’re writing legacy code from the start.

If someone else uses your codebase, changing anything there will become dangerous. This can be even more risky when you have multiple services that depend on each other.

Since Spring Boot best practices exist, you should consider using Spring Cloud Contract for your consumer-driven contracts, it will make your integrations with other services easier to use.

**16. Use test slices to make testing easier and more focused**

This practice comes from Madhura Bhave (Spring developer, @madhurabhave23).

Testing code with Spring Boot can be tricky — you need to initialize the data layer, wire up a lot of services, mock things up… it’s actually not that hard! The answer is to use test slices.

With test slices, you can connect only parts of the application as needed. This can save you a lot of time and ensure that your tests are not associated with unused content. A blog post from spring.io called Custom test slice with Spring test 1.4 explains this technique.

<https://spring.io/blog/2016/08/30/custom-test-slice-with-spring-boot-1-4>

**Summarize：**

Thanks to Spring Boot, writing Spring-based micro services has never been easier. I hope that with these best practices, your implementation process will not only become faster, but will be more robust and successful in the long run. Good luck!