# TESTING SPRING BOOT APPLICATIONS DEMYSTIFIED

# Testing Spring Boot Applications Demystified

Avoiding Pitfalls, Implementing Recipes, and Embracing Best Practices

Philip Riecks

This book is for sale at
http://leanpub.com/testing-spring-boot-applications-demystified

This version was published on 2023-07-04

# Contents

# Intro

## About this Book

Whether you're a seasoned developer or just starting your journey with Spring Boot, this ebook is designed to unravel the complexities of testing Spring Boot applications and empower you to become more productive and confident in your testing efforts.

Testing plays a crucial role in software development, allowing you to catch bugs, ensure your code functions as intended, and provide confidence in the reliability and stability of your applications. However, testing Spring Boot applications can sometimes feel like navigating a labyrinth of challenges – from managing dependencies and external systems to crafting effective tests that reflect the behavior of your code.

In this ebook, we will demystify testing Spring Boot applications by providing you with clear explanations, practical insights, and actionable best practices. We will guide you through the common pitfalls faced by developers, share time-tested recipes for success, and equip you with the tools and knowledge you need to write comprehensive and effective tests.

More than just a technical guide, "Testing Spring Boot Applications Demystified" aims to motivate and inspire you on your testing journey. We believe that thorough testing is not only a means to catch bugs but also a pathway to building robust, high-quality applications. By implementing the best practices and strategies outlined in this ebook, you will gain the confidence to deliver software that exceeds expectations and delights both users and stakeholders.

So, whether you are looking to enhance your testing skills or seeking guidance on the unique challenges faced in testing Spring Boot applications, this ebook is your go-to resource. Get ready to demystify testing, overcome obstacles, and become a proficient tester of Spring Boot applications.

## About the Author

Under the slogan "Testing Java Applications Made Simple" Philip provides recipes and tips & tricks to accelerate your testing success and make testing joyful (or at least less painful). Apart from blogging, he's a course instructor for various Java-related online courses and active on YouTube.

Find out more about Philip on the Testing Java Applications Made Simple blog and follow him on Twitter.

# Spring Boot Testing Demystified

## Spring Boot Testing Pitfalls

Learning how to test a Spring Boot application effectively can be a hurdle, especially for newcomers. Without a basic knowledge of Spring's dependency injection mechanism and what Spring Boot's auto-configuration is all about, we might end up throwing annotations to our test.

Trying to make things work. While this trial and error might fix the test setup for some cases, the result is usually a not-so-optimal test setup. With this section post, I've collected the most common pitfalls I've seen in projects and while answering questions on Stack Overflow when it comes to testing Spring Boot applications.

### Spring Boot Testing Pitfall 1: @Mock vs. @MockBean

One of the first Spring Boot testing pitfalls is mocking collaborators, the objects our class under test depends on. If we are already familiar with Mockito, we might know that we can use `@Mock` to create mocks for our unit tests. When writing tests for our Spring Boot applications, we don't have to un-learn any specific Mockito knowledge. Nevertheless, we have to be aware of what kind of test we're writing. Does our test work with or without a Spring TestContext?

That's important because it determines whether to use @Mock vs. @Mock-Bean. While both annotations create a mocked version of our collaborators, `@Mock` is only relevant for plain unit tests that work without a Spring Test-Context. In such cases, we usually create mocks of the collaborators and inject them via the public constructor of our class under test. For tests that work with a Spring TestContext, e.g., when using a Spring Boot Test slice annotation or @SpringBootTest, things work differently. Here, we still want to mock the collaborator of your class under test. But this time, Spring assembles all our beans and performs dependency injection.

Hence, we have to replace (or add) a mocked version of collaborator as a bean inside the Spring TestContext. That's where `@MockBean` comes into play. We use it on top of a test field to instruct Spring Test to add a mocked version of this bean inside our TestContext. Whether we use `@Mock` or `@MockBean`, the Mockito stubbing setup works the same for both. The pitfall here lies in either mixing both annotations in the same test or using one of the annotations for the wrong purpose. I've covered a comparison of both annotations and when to use them in a separate @Mock vs. @MockBean blog post.

## Spring Boot Testing Pitfall 2: Extensive Usage of @SpringBootTest

When starting with testing Spring Boot applications, we'll soon stumble over the `@SpringBootTest` annotation. Spring Boot even creates a basic test that uses this annotation for each new project generated from start.spring.io. The name of the annotation might imply it's used and required for every Spring Boot test. That's not the case. We use @SpringBootTest whenever we want to write an integration test that works with the entire Spring Context. Spring Test will create a TestContext for us that contains all our beans (`@Component`, `@Configuration`, `@Service`, etc.). This implies that we also have to provide every external infrastructure component we're connecting to. Imagine we're writing a CRUD application that connects to a database.

We won't be able to create and use our repository classes if there's no database during test execution to connect to. If we would only use `@SpringBootTest` for our tests, we'll soon encounter that our test suite takes way longer than plain JUnit & Mockito tests. Starting a Spring Context results in slower test execution times as everything has to be instantiated and initialized. As a general recommendation, we should try to test and verify as much of our implementation as possible on a lower testing level. That means a specific if-block inside our `@Service` class can be tested with a unit test. Furthermore, we can ensure our Spring Security configuration is working by using `@WebMvcTest`.

For integration tests that verify the interaction for multiple components, or when writing end-to-end tests, `@SpringBootTest` comes into play. Make yourself familiar with the different Spring Boot test slice annotations. Furthermore, there are several ways to further tweak @SpringBootTest, which we have to be aware of.

## Testing Pitfall 3: Not Testing At All

I guess this Spring Boot testing pitfall goes without saying. If we're not testing our code, how can we ever say it's working? While we might have checked our implementation manually, how can we ensure any upcoming changes don't break our feature? If we don't test our application, our users definitely will, and they won't be delighted if they find half-backed features. Testing might not be the first priority when learning Spring Boot.

That's fine as long as we're making sure to return to the testing topic as soon as we feel comfortable with the framework. Whether we write the test before the implementation (aka. test-driven development) or afterward depends on personal preferences. I've had a great experience writing the test first, leading to more thoughtful design and smaller steps. Doing it the other way around and adding tests to our code right after we finish the implementation usually results in *not-so-well* tests. We already know how the implementation looks like and are biased towards testing only the bare minimum. On top of this, we might already be late integrating our changes and, hence, have little time to test the implementation thoroughly.

The Spring Framework and Spring Boot emphasize the importance of testing and encourage us to write tests by having great testing support and tools. Testing is an essential part of every Spring Boot project as every new project already comes with a basic integration test and the testing swiss-army knife. Josh Long will personally visit us if we delete (or disable) this autogenerated test.

There's literally no excuse to write no test - except we don't know the *how* (yet). But this we can easily fix. There's plenty of hands-on testing advice available on this blog. Start with the following Spring Boot unit and integration testing overview. Next, consider enrolling for the Testing Spring Boot Applications Primer to kickstart your Spring Boot testing success.

## Testing Pitfall 4: Not Reusing the Spring TestContext

This is coupled to the second pitfall (Extensive usage of `@SpringBootTest`). Starting a new Spring TestContext for each and every test class is expensive. So why not cache an already started Spring TestContext? That's exactly what Spring Test does for us! Whenever we're about to start a new Spring Test-Context, a sliced or the entire context, Spring considers an already started context for this test.

If an existing context matches the context configuration for the test class we're about to run, Spring will reuse the context. If there's no suitable cache context already started (speak a cache miss), Spring starts a new one and stores the context afterward for further reuse of other tests. So how does Spring determine whether or not a context can be reused and how can we use this feature effectively? Imagine one integration test activates the profile `integration-test` while another test activates `web-test`. In such a case, Spring won't reuse the same context because our configuration looks entirely different due to the different profiles.

There are more than ten configuration and setup values that determine the uniqueness of a cache. To effectively use this performance improvement, we have to align most of our Spring TestContext setups. We should avoid multiple context configurations, especially for tests that work with the entire `ApplicationContext`. In one of my projects, I reduced the entire build time (running `mvn verify`) from 25 minutes to 9 minutes while making the most of the Spring TestContext Caching mechanism. I did this by aligning the context configurations for the expensive integration tests. Make yourself familiar with the several configuration values and how to make the most of Spring's TestContext caching mechanism.

## Spring Boot Testing Pitfall 5: Mixing Up JUnit 4 and JUnit 5

Another common pitfall when testing Spring Boot applications that leads to weird test results is mixing JUnit 4 and JUnit 5 in the same test class. When answering questions on Stack Overflow, I see a lot of confusion around this topic. While the first version of JUnit 5 was released in 2017, there are still projects out there using the predecessor (which is fine).

As JUnit 5 supports running JUnit 4 tests next to JUnit 5 tests, we can mix both for our projects during the migration/transition period. Using annotations and APIs from JUnit 4 and JUnit 5 (JUnit Jupiter, to be precise) won't work. It's either-or. While we can have both JUnit 4 and JUnit 5 tests in our project, thanks to the JUnit Vintage engine, one test class should either opt-in of version 4 or 5. Using JUnit 4 for `MyOrderTest` and JUnit 5 for `MyPricingServiceTest` is totally fine. The pitfall lies in mixing APIs and annotations of both versions within the same test.

There are tools and guides available to start the migration and help convert the low-hanging fruits. There's still some manual effort left when it comes to migrating custom `Runner` or `Rule` classes. Once the migration to JUnit 5 is done, I recommend excluding any JUnit 4 dependency from the project. This helps to identify JUnit 4 leftovers due to a failing compile step. It also reduces the likelihood to (accidentally) re-introduce JUnit 4 for a new test.

## Tips & Tricks

### Correct Use Of Your Build Tool: Maven

Starting with a new programming language is always exciting. However, it can be overwhelming as we have to get comfortable with the language, the tools, conventions, and the general development workflow. This holds true for both developing and testing our applications.

When testing Java applications with Maven, there are several concepts and conventions to understand: Maven lifecycles, build phases, plugins, etc. With this section post, we'll cover the basic concepts for you to understand how testing Java applications with Maven *works*.

#### What Do We Need Maven For?

When writing applications with Java, we can't just pass our `.java` files to the JVM (Java Virtual Machine) to run our program. We first have to compile our Java source code to bytecode (`.class` files) using the Java Compiler (`javac`).

Next, we pass this bytecode to the JVM (`java` binary on our machines) which then interprets our program and/or compiles parts of it even further to native machine code. Given this two-step process, someone has to compile our Java classes and package our application accordingly. Manually calling `javac` and passing the correct classpath is a cumbersome task.

A build tool automates this process. As developers, we then only have to execute one command, and everything gets build automatically. The two most adopted build tools for the Java ecosystem are Maven and Gradle. *Ancient devs* might still prefer Ant, while *latest-greatest devs* might advocate for Bazel as a build tool for their Java applications. We're going to focus on Maven with this section. To build and test our Java applications, we need a JDK (Java Development Kit) installed on our machine and Maven.

We can either install Maven as a command-line tool (i.e., place the Maven binary on our system's `PATH`) or use the portable Maven Wrapper. The Maven Wrapper is a convenient way to work with Maven without having to install it locally. It allows us to conveniently build Java projects with Maven without having to install and configure Maven as a CLI tool on our machine When creating a new Spring Boot project, for example, you might have already wondered what the `mvnw` and `mvnw.cmd` files inside the root of the project are used for. That's the Maven Wrapper (the idea is borrowed from Gradle).

### Creating a New Maven Project

There are several ways to bootstrap a new Maven project. Most of the popular Java application frameworks offer a project bootstrapping wizard-like interface. Good examples are the Spring Initializr for new Spring Boot applications, Quarkus, MicroProfile.

If we want to create a new Maven project without any framework support, we can use a Maven Archetype to create new projects. These archetypes are a project templating toolkit to generate a new Maven project conveniently.

Maven provides a set of default Archetypes artifacts for several purposes like a new web app, a new Maven plugin project, or a simple quickstart project. We bootstrap a new Java project from one of these Archetypes using the `mvn` command-line tool:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.maven.archetypes  \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DarchetypeVersion=1.4 \
    -DgroupId=com.mycompany \
    -DartifactId=order-service
```

The skeleton projects we create with the official Maven Archetypes are a good place to start. However, some of these archetypes generate projects with outdated dependency versions like JUnit 4.11.

While it's not a big effort to manually bump the dependency version after the project initialization, having an up-to-date Maven Archetype in the first place is even better.

#### Minimal Maven Project For Testing Java Applications

As part of my Custom Maven Archetype open-source project on GitHub, I've published a collection of useful Maven Archetypes. One of them is the `java-testing-toolkit` to create a Java Maven project with basic testing capabilities. Creating our own Maven Archetype is almost no effort. We can create a new testing playground project using this custom Maven Archetype with the following Maven command (for Linux & Mac):

```
mvn archetype:generate \
    -DarchetypeGroupId=de.rieckpil.archetypes  \
    -DarchetypeArtifactId=testing-toolkit \
    -DarchetypeVersion=1.0.0 \
    -DgroupId=com.mycompany \
    -DartifactId=order-service
```

For Windows (both PowerShell and CMD), we can use the following command to bootstrap a new project from this template:

```
mvn archetype:generate "-DarchetypeGroupId=de.rieckpil.archetypes" "-DarchetypeArtifactId=t\
esting-toolkit" "-DarchetypeVersion=1.0.0" "-DgroupId=com.mycompany" "-DartifactId=order-se\
rvice" "-DinteractiveMode=false"
```

We can adjust both `-DgroupId` and `-DartifactId` to our project's or company's preference. The generated project comes with a basic set of the most central Java testing libraries. We can use it as a blueprint for our next project or explore testing Java applications with this playground. In summary, the following default configuration and libraries are part of this project shell:

- Java 11
- JUnit Jupiter, Mockito, and Testcontainers dependencies
- A basic unit test

- Maven Surefire and Failsafe Plugin configuration
- A basic `.gitignore`
- Maven Wrapper

Next, we have to ensure a JDK 11 (or higher) is on our `PATH` and also `JAVA_HOME` points to the installation folder:

```
$ java -version
openjdk version "11.0.10" 2021-01-19
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.10+9)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.10+9, mixed mode)

# Windows
$ echo %JAVA_HOME%
C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspot

# Mac and Linux
$ echo $JAVA_HOME
/usr/lib/jvm/adoptopenjdk-11.0.10.9-hotspot
```

As a final verification step, we can now build and test this project with Maven:

```
$ mvn archetype:generate ... // generate the project
$ cd order-service // navigate into the folder
$ ./mvnw package // mvnw.cmd package for Windows

....

[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.326 s
[INFO] Finished at: 2021-06-03T08:31:11+02:00
[INFO] ------------------------------------------------------------------------
```

We can now open and import the project to our editor or IDE (IntelliJ IDEA, Eclipse, NetBeans, Visual Code, etc.) to inspect the generated project in more detail.

### Important Testing Folders and Files

First, let's take a look at the folders and files that are relevant for testing Java applications with Maven:

- `src/test/java`

This folder is the main place to add our Java test classes (`.java` files). As a general recommendation, we should try to mirror the package structure of our production code (`src/main/java`). Especially if there's a direct relationship between the test and a source class.

The corresponding `CustomerServiceTest` for a `CustomerService` class inside the package `com.company.customer` should be placed in the same package within `src/test/java`. This improves the likelihood that our colleagues (and our future us) locate the corresponding test for a particular Java class without too many facepalms. Most of the IDEs and editors provide further support to jump to a test class.

IntelliJ IDEA, for example, provides a shortcut (Ctrl+ Shift + T) to navigate from a source file to its test classes(s) and vice-versa.

- `src/test/resources`

As part of this folder, we store static files that are only relevant for our test. This might be a CSV file to import test customers for an integration test, a dummy JSON response for testing our HTTP clients, or a configuration file.

- `target/test-classes`

At this location, Maven places our compiled test classes (`.class` files) and test resources whenever the Maven compiler compiles our test sources. We can explicitly trigger this with `mvn test-compile` and add a `clean` if we want to remove the existing content of the entire `target` folder first. Usually, there's no need to perform any manual operations inside this folder as it contains build artifacts.

Nevertheless, it's helpful to investigate the content for this folder whenever we face test failures because we, e.g., can't read a file from the classpath.

Taking a look at this folder (after the Maven compiler did its work), can help understanding where a resources file ended up on the classpath.

- `pom.xml`

This is the heart of our Maven project. The abbreviation stands for **P**roject **O**bject **M**odel. Within this file, we define metadata about our project (e.g., description, artifactId, developers, etc.), which dependencies we require, and the configuration of our plugins.

### Maven and Java Testing Naming Conventions

Next, let's take a look at the naming conventions for our test classes. We can separate our tests into two (or even more) basic categories: unit and integration test. To distinguish the tests for both of these two categories, we use different naming conventions with Maven. The Maven Surefire Plugin, more about this plugin later, is designed to run our unit tests. The following patterns are the defaults so that the plugin will detect a class as a test:

- `**/Test*.java`
- `**/*Test.java`
- `**/*Tests.java`
- `**/*TestCase.java`

So what's actually a unit test? Several smart people came up with a definition for this term. One of such smart people is Michael Feathers. He's turning the definition around and defines what a **unit test is not**:

> A test is not a unit test if it …
>
> * talks to the database
> * communicates across the network
> * touches the file system

* can’t run at the same time as any of your other unit tests
* or you have to do special things to your environment (such as editing config files) to run it.

Kevlin Henney is also a great source of inspiration for a definition of the term unit test. Nevertheless, our own definition or the definition of our coworkers might be entirely different.

In the end, the actual definition is secondary as long as we’re sharing the same definition within our team and talking about the same thing when referring to the term unit test. The Maven Failsafe Plugin, designed to run our integration tests, detects our integration tests by the following default patterns:

- `**/IT*.java`
- `**/*IT.java`
- `**/*ITCase.java`

We can also override the default patterns for both plugins and come up with a different naming convention. However, sticking to the defaults is recommended.

#### When Are Our Java Tests Executed?

Maven is built around the concept of build lifecycles. There are three built-in lifecycles:

- `default`: handling project building and deployment
- `clean`: project cleaning
- `site`: the creation of our project’s (documentation) site

Each of the three built-in lifecycles has a list of build phases. For our testing example, the `default` lifecycle is important. The `default` lifecycle compromises a set of build phases to handle building, testing, and deploying our Java project. Each phase represents a stage in the build lifecycle with a central responsibility:

TODO: Image

In short, the several phases have the following responsibilities:

- `validate`: validate that our project setup is correct (e.g., we have the correct Maven folder structure)
- `compile`: compile our source code with `javac`
- `test`: run our unit tests
- `package`: build our project in its distributable format (e.g., JAR or WAR)
- `verify`: run our integration tests and further checks (e.g., the OWASP dependency check)
- `install`: install the distributable format into our local repository (`~/.m2` folder)
- `deploy`: deploy the project to a remote repository (e.g., Maven Central or a company hosted Nexus Repository/Artifactory)

These build phases represent the central phases of the `default` lifecycle. There are actually more phases. For a complete list, please refer to the Lifecycle Reference of the official Maven documentation. Whenever we execute a build phase, our project will go through all build phases and sequentially until the build phase we specified.

To phrase it differently, when we run `mvn package`, for example, Maven will execute the default lifecycle phases up to `package` in order:

```
validate -> compile -> test -> package
```

If one of the build phases in the chain fails, the entire build process will terminate. Imagine our Java source code has a missing semicolon, the `compile` phase would detect this and terminate the process. As with a corrupt source file, there'll be no compiled `.class` file to test.

When it comes to testing our Java project, both the `test` and `verify` build phases are of importance. As part of the `test` phase, we're running our unit tests with the Maven Surefire Plugin, and with `verify` our integration tests are executed by the Maven Failsafe Plugin. Let's take a look at these two plugins.

### Running Unit Tests With the Maven Surefire Plugin

The Maven Surefire is responsible for running our unit tests. We must either follow the default naming convention of our test classes, as discussed above, or configure a different pattern that matches our custom naming convention. In both cases, we have to place our tests inside `src/test/java` folder for the plugin to pick them up. For the upcoming examples, we're using a basic `format` method

```
public class Main {

  public String format(String input) {
    return input.toUpperCase();
  }
}
```

… and its corresponding test as a unit test blueprint:

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class MainTest {

  private Main cut;

  @BeforeEach
  void setUp() {
    this.cut = new Main();
  }

  @Test
  void shouldReturnFormattedUppercase() {
    String input = "duke";

    String result = cut.format(input);
    assertEquals("DUKE", result);
  }
}
```

Depending on the Maven version and distribution format of our application (e.g., JAR or WAR), Maven defines default versions for the core plugins. Besides the Maven Compiler Plugin, the Maven Resource Plugin, and other plugins, the Maven Surefire Plugin is such a core plugin.

When packaging our application as a JAR file and using Maven 3.8.1, for example, Maven picks the Maven Surefire Plugin with version 2.12.4 by default unless we override it. As the default versions are sometimes a little bit behind the latest plugin versions, it's worth updating the plugin versions and manually specifying the plugin version inside our `pom.xml`:

```
<project>
  <!-- dependencies -->

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0-M5</version>
      </plugin>
    </plugins>
  </build>
 </project>
```

As part of the `test` phase of the default lifecycle, we'll now see the Maven Surefire Plugin executing our tests:

```
$ mvn test

[INFO] --- maven-surefire-plugin:3.0.0-M5:test (default-test) @ testing-example ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running de.rieckpil.blog.MainTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.04 s - in de.rieck\
pil.blog.MainTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
```

For the example above, we're running one unit test with JUnit 5 (testing provider). There's no need to configure the testing provider anywhere, as with recent Surefire versions, the plugin will pick up the correct test provider by itself. The Maven Surefire Plugin integrates both JUnit and TestNG as testing providers out-of-the-box. If we don't want to execute all build phases before running our tests, we can also explicitly execute the `test` goal of the Surefire plugin:

```
mvn surefire:test
```

But keep in mind that we have to ensure that the test classes have been compiled first (e.g., by a previous build). We can further tweak and configure the Maven Surefire Plugin to, e.g., parallelize the execution of our unit tests. This is only relevant for JUnit 4, as JUnit 5 (JUnit Jupiter to be precise) supports parallelization on the test framework level. Whenever we want to skip our unit tests when building our project, we can use an additional parameter:

```
mvn package -DskipTests
```

We can also explicitly run only one or multiple tests:

```
mvn test -Dtest=MainTest

mvn test -Dtest=MainTest#testMethod

mvn surefire:test -Dtest=MainTest
```

### Running Integration Tests With the Maven Failsafe Plugin

Unlike the Maven Surefire Plugin, the Maven Failsafe Plugin is not a core plugin and hence won't be part of our project unless we manually include it. As already outlined, the Maven Failsafe plugin is used to run our integration test. In contrast to our unit tests, the integration tests usually take more time, more setup effort (e.g., start Docker containers for external infrastructure with Testcontainers), and test multiple components of our application together. We integrate the Maven Failsafe Plugin by adding it to the `build` section of our `pom.xml`:

```
<project>

  <!-- other dependencies -->

  <build>
      <!-- further plugins -->
      <plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>3.0.0-M5</version>
        <executions>
          <execution>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

As part of the `executions` configuration, we specify the goals of the Maven Failsafe plugin we want to execute as part of our build process. A common pitfall is to only execute the `integration-test` goal. Without the `verify` goal, the plugin will run our integration tests but won't fail the build if there are test failures. The Maven Failsafe Plugin is invoked as part of the `verify` build phase of the default lifecycle. That's right after the `package` build phase where we build our distributable artifact (e.g., JAR):

```
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ testing-example ---
[INFO] Building jar: C:\Users\phili\Desktop\junk\testing-example\target\testing-example.jar
[INFO]
[INFO] --- maven-failsafe-plugin:3.0.0-M5:integration-test (default) @ testing-example ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running de.rieckpil.blog.MainIT
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.039 s - in de.riec\
kpil.blog.MainIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-failsafe-plugin:3.0.0-M5:verify (default) @ testing-example ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

If we want to run our integration tests manually, we can do so with the following command:

```
mvn failsafe:integration-test failsafe:verify
```

For scenarios where we don't want to run our integration test (but still our unit tests), we can add `-DskipITs` to our Maven execution:

```
mvn verify -DskipITs
```

Similar to the Maven Surefire Plugin, we can also run a subset of our integration tests:

```
mvn -Dit.test=MainIT failsafe:integration-test failsafe:verify

mvn -Dit.test=MainIT#firstTest failsafe:integration-test failsafe:verify
```

When using the command above, make sure the test classes have been compiled previously, as otherwise, there won't be any test execution. There's also a property available to entirely skip the compilation of test classes and avoid running any tests when building our project (not recommended):

```
mvn verify -Dmaven.test.skip=true
```

### Summary of Testing Java Applications With Maven

Maven is a powerful, mature, and well-adopted build tool for Java projects. As a newcomer or when coming from a different programming language, the basics of the Maven build lifecycle and how and when different Maven Plugins interact is something to understand first. With the help of Maven Archetypes or using a framework initializer, we can easily bootstrap new Maven projects.

There's no need to install Maven as a CLI tool for our machine as we can instead use the portable Maven Wrapper. Furthermore, keep this in mind when testing your Java applications and use Maven as the build tool:

- With Maven, we can separate the unit and integration test execution
- The Maven Surefire Plugin runs our unit tests
- The Maven Failsafe Plugin runs our integration tests
- By following the default naming conventions for both plugins, we can easily separate our tests
- The Maven default lifecycle consists of several build phases that are executed in order and sequentially
- Use the Java Testing Toolkit Maven archetype for your next testing adventure

## Use GitHub Actions

I was recently wasting time and energy to get the CI pipelines for my two main GitHub repositories working with Travis CI. Even though the documentation provides examples for Maven-based Java projects, it took me still some time to find the correct setup.

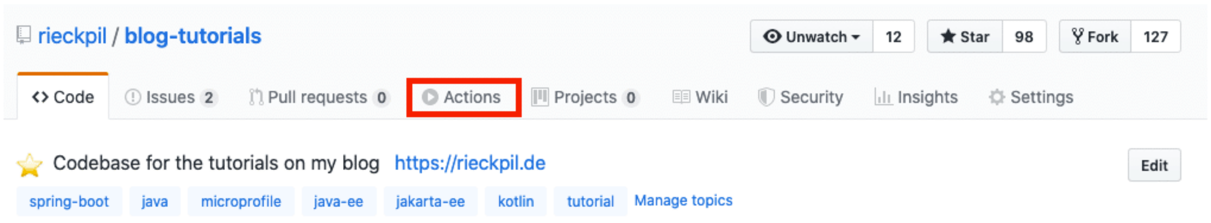
While working on these pipelines, I remembered that GitHub now also provides its own CI/CD solution: GitHub Actions. As I wasn't quite satisfied with Travis CI, I gave it a try and got everything up- and running in less than one hour.

### Introduction to GitHub Actions

GitHub markets its GitHub Actions product as the following:

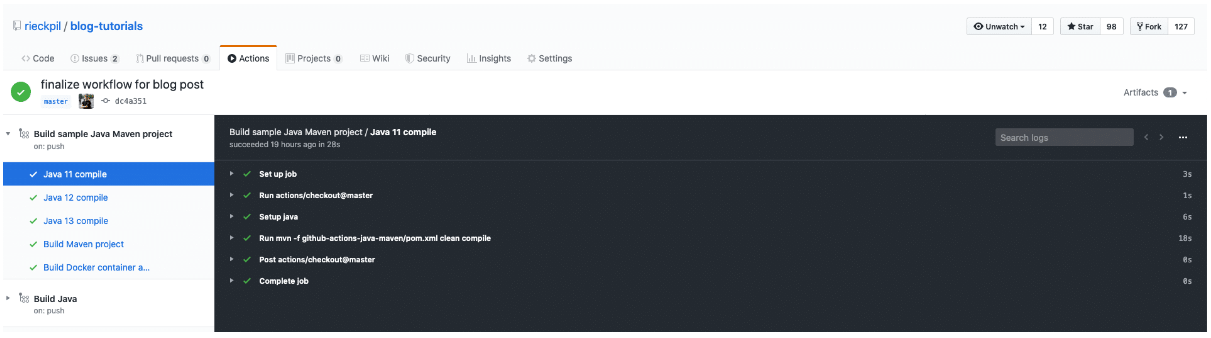
> GitHub Actions makes it easy to automate all your software workflows, now with world-class CI/CD. Build, test, and deploy your code right from GitHub. Make code reviews, branch management, and issue triaging work the way you want.

We can enter this feature on every (even private) GitHub repository with the **Actions** tab: [



**GitHub Actions Tab Panel**

](https://rieckpil.de/wp-content/uploads/2019/12/gitHubActionsTabPanel.png) Within this tab, we get an overview of your latest builds and their logs like our might know it from Jenkins, Travis CI, Circle CI, etc. : [



finalize workflow for blog post
master dc4a351

Build sample Java Maven project
on: push
- ✓ Java 11 compile
- ✓ Java 12 compile
- ✓ Java 13 compile
- ✓ Build Maven project
- ✓ Build Docker container a...

Build Java
on: push

Build sample Java Maven project / Java 11 compile
succeeded 19 hours ago in 28s

```
✓ Set up job                                                     3s
✓ Run actions/checkout@master                                    1s
✓ Setup java                                                     6s
✓ Run mvn -f github-actions-java-maven/pom.xml clean compile    18s
✓ Post actions/checkout@master                                   0s
✓ Complete job                                                   0s
```

**GitHub Actions Overview**

](https://rieckpil.de/wp-content/uploads/2019/12/gitHubActionsOverview.png)

We configure your different pipeline steps as code and include them in your repository. The pipeline YAML definitions are then placed in the `.github/workflows` folder. Among other things, GitHub actions offers the following features:

- hosted runners for every OS (Windows, macOS, Linux)
- matrix builds to, e.g., test your library for different OS and programming language versions
- access to Docker to, e.g., use Testcontainers or a docker-compose.yml file for integration tests
- rich-feature marketplace next to pre-defined Actions provided by GitHub
- free for public repositories and limited contingent (minutes per month) for private repositories
- great integration for events of your GitHub repository (e.g., pull request, issue creation, etc.)

Let's use a Maven-based Java project to demonstrate how to use GitHub Actions…

#### Sample Workflow for a Maven-Based Java Application

As an example, we'll use a typical Java 11 Spring Boot Maven project to demonstrate the use of GitHub Actions.

The project uses Testcontainers to access a PostgreSQL database during integration tests. The deployment target is Kubernetes, and the application is packaged inside a Docker container. This should reflect 80% of the requirements for a standard CI/CD pipeline these days. In short, we want to achieve the following with our CI/CD pipeline using GitHub Actions:

- use different Java versions to compile the project (useful for library developers)
- cache the content of `.m2/repository` (or any other folder) for accelerated builds
- use Maven to build the project
- stash build artifacts between different jobs
- access secrets (to, e.g., login to a container registry )
- make use of Docker

The workflow uses three jobs: `compile`, `build` and `deploy`. Don't reflect on the meaningfulness of the following setup. We intentionally create a bloated pipeline to showcase as many features of GitHub Actions as possible. Let's start with the `compile` job:

```
name: Build sample Java Maven project

on: [push, pull_request]

jobs:
  compile:
    runs-on: ubuntu-20.04
    strategy:
      matrix:
        java: [ 11, 12, 13 ]
    name: Java ${{ matrix.java }} compile
    steps:
      - name: Checkout Source Code
        uses: actions/checkout@v2
      - name: Setup Java
        uses: actions/setup-java@v2
        with:
          distribution: 'adopt'
          java-package: jdk
          java-version: ${{ matrix.java }}
      - name: Compile the Project
        working-directory: github-actions-java-maven
        run: mvn -B compile
```

We configure the job to run on a hosted ubuntu-20.04 runner. GitHub let's use choose between Ubuntu, Windows, and Mac as GitHub-hosted runners. Those runners already come with a decent amount of binaries and tools installed (e.g. AWS CLI, Maven, etc.).

For a complete list of installed software, see the documentation on supported software.

Next, we define a matrix strategy to run the same job multiple times (in parallel) with different Java versions. The source code for the repository is not checked out on the runner by default.

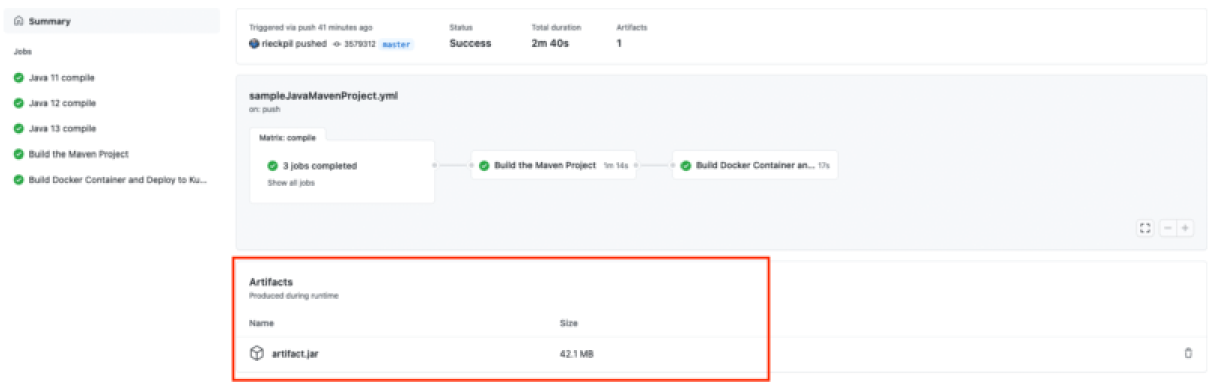
We use the `checkout@v2` action for this purpose. The `setup-java@v2` action is used to set up the specific Java version on the runner and as part of the last step, we're compiling the Java project with Maven. With `working-directory` we define in which folder we want to execute the commands. This way we don't have to explicitly perform any `cd` operation. Next comes the `build` job:

```
name: Build sample Java Maven project

on: [push, pull_request]

jobs:
  compile:
    # compile job

  build:
    runs-on: ubuntu-20.04
    needs: compile
    name: Build the Maven Project
    steps:
    - uses: actions/checkout@v2
    - uses: actions/cache@v2
      with:
        path: ~/.m2/repository
        key: ${{ runner.os }}-maven-${{ hashFiles('**/pom.xml') }}
        restore-keys: |
          ${{ runner.os }}-maven-
    - name: Set up JDK 11
      uses: actions/setup-java@v2
      with:
          distribution: 'adopt'
          java-version: '11'
          java-package: jdk
    - name: Build and test project
      working-directory: github-actions-java-maven
      run: mvn -B verify
    - name: Upload Maven build artifact
      uses: actions/upload-artifact@v2
      with:
        name: artifact.jar
        path: github-actions-java-maven/target/github-actions-java-maven.jar
```

By default, GitHub executes all jobs in parallel unless we specify that the job depends on the outcome of a previous job. This way we can ensure a sequential order. Each GitHub Actions job starts with a fresh GitHub runner and hence we have to perform the VCS checkout and Java setup again.

As an alternative, we can (and should have) performed the Maven build as part of the previous job. Similar to the previous job, we're using Maven to now build the `.jar` file. We'll now cache the contents of the `.m2` folder to speed up subsequent builds as they don't have to download our dependencies over and over.

After we've successfully built our project, we want to share the build artifact with an upcoming job. As the jobs don't share the same filesystem we have to upload it. Once we've uploaded an artifact (this might also be a screenshot from a failing web test), another job can download it. Furthermore, we can also manually download any build artifacts ourselves:



**GitHub Actions Download Artifact**

And finally, we're (artificially) deploying the project:

```
name: Build sample Java Maven project

on: [push, pull_request]

jobs:
  # existing jobs ..

  deploy:
    runs-on: ubuntu-20.04
    needs: build
    name: Build Docker Container and Deploy to Kubernetes
    steps:
    - uses: actions/checkout@v2
    - name: Download Maven build artifact
      uses: actions/download-artifact@v2
      with:
        name: artifact.jar
        path: github-actions-java-maven/target
    - name: Build Docker container
      working-directory: github-actions-java-maven
      run: |
        docker build -t de.rieckpil.blog/github-actions-sample .
    - name: Access secrets
      env:
        SUPER_SECRET: ${{ secrets.SUPER_SECRET }}
      run: echo "Content of secret - $SUPER_SECRET"
    - name: Push Docker images
      run: echo "Pushing Docker image to Container Registry (e.g. ECR)"
    - name: Deploy application
      run: echo "Deploying application (e.g. Kubernetes)"
```

We first download the Maven build artifact as we need it to build our Docker image. Right after building the Docker image, we could now log in to our private Docker Registry to push our image. As this usually requires access to secrets (username and password) we demonstrate how to map secrets to environment variables.

We can securely store those secrets as part of our GitHub repository (Settings -> Secrets). What's left is to deploy the new Docker Image to our target environment (e.g. Kubeternes).

### Blueprints for Real-World Workflow With GitHub Actions

Over the past month, I've migrated most of my GitHub projects to GitHub Actions and never looked back. For further inspiration on how to use GitHub Actions, take a look at the following examples:

- Building and testing all blog post code examples as part of my blog-tutorials repository (Gradle, Maven, multiple Java versions, caching)
- Building and testing the source code for the Getting Started With Eclipse MicroProfile Course (Integration tests with MicroShed Testing, Docker, multiple projects, Open Liberty)
- For Stratospheric (a book about Spring Boot and AWS that I'm co-authoring), we're using GitHub actions for the entire CI/CD pipeline. We're building the Spring Boot backend, push the Docker image to ECR and create/sync our entire AWS infrastructure with the AWS CDK. Take a look at the various workflows for some inspiration and get a copy of this book if you want detailed information about this setup.
- If you're maintaining a Java library that you're publishing to Maven Central, consider this GitHub Actions workflow blueprint for building and deploying libraries.
- Aggregating Selenide screenshots to download in case of test failures for the Spring Boot Applications Masterclass
- Checking for rotten links in markdown files (must-have when writing eBooks)

To summarize, I can highly recommend GitHub Actions for Maven-based Java projects. The configuration is simple and you are ready in minutes. Whether you are building a Java library or an application in a private repository, GitHub Actions allows you to easily set up CI/CD. Give it a try!

The Spring Boot application and the workflow definition is available on GitHub.

## Parallelizing Unit Tests with Maven and JUnit 5

The more our project and test suite grow, the longer the feedback loop becomes. Fortunately, there are techniques available to speed up our build time. One of such techniques is parallelizing our tests. Instead of running our tests in sequence, we can run them in parallel to save time. The parallelization may not work for all kinds of tests, and hence we'll learn with this section how to only parallelize our unit tests with JUnit 5 and Maven.

The upcoming technique is framework independent, and we can apply it to any Java project (Spring Boot, Quarkus, Micronaut, Jakarta EE, etc.) that uses JUnit 5 (JUnit Jupiter, to be precise) and Maven.

### Upfront Requirement: Separation of Tests

Before we jump right into the required configuration setup for parallelizing our unit tests, we first have to split our tests into at least two categories. The reason for this is to allow a separate parallelization strategy (or no parallelization at all) depending on the test category. While there are many different types of tests in the literature, one can endlessly discuss what's the correct name and category.

Sticking to two basic test categories is usually sufficient: unit and integration tests. Where to draw the line between unit and integration tests is yet another discussion. In general, if a test meets the following criteria, we can usually refer to it as a unit test:

- A small unit (method or class) is tested in isolation
- The collaborators of the class under test are replaced with a fake/stub
- The test doesn't depend on infrastructure components like a database
- The test executes fast
- We can parallelize the test as there are no side effects from other tests

While this list of requirements is not exhaustive, it's a first good indicator of what a unit test is. Any test that doesn't fit in this category will be labeled an integration test. When it comes to labeling and separating our tests, we have multiple options when using JUnit and Maven. First, JUnit 5 lets us tag `@Tag("integration-test")` (former JUnit 4 categories) our test class to label them. However, when adding a new test to our project, we may forget to add these tags, and in general, it requires a little bit more maintenance effort on our end.

A more pragmatic approach is to use the convenient defaults of two Maven plugins that are involved in our testing lifecycle: the Maven Surefire and Maven Failsafe Plugin. By default, the Maven Surefire plugin will run any test that has the postfix `*Test` (e.g., `CustomerServiceTest`). The Maven Failsafe Plugin, on the other hand, only executes tests with the postfix `*IT` (for integration test).

We can even override these naming strategies and come up with our own postfix. Sticking to the defaults, if we add the postfix `*Test` only to our unit tests classes and `*IT` for our integration tests, we already have a separation. Both plugins run separately at a different build phase of the Maven default lifecycle. The Maven Surefire plugin executes our unit tests in the `test` phase while the Failsafe plugin gets active in the `verify` phase. For more information on both plugins and to understand how Maven is involved in testing Java applications, head over to this article.

### Upfront Requirement: Independent Unit Tests

Another requirement we have to conform to is the independence of our unit tests. As soon as we've split up our test suite into unit and integration tests by naming them differently, we have to ensure our unit test can run in parallel. For this to work, there shouldn't be an implicit order that dictates the success or failure of our unit tests. Our unit tests should pass or fail independently of the order they were invoked.

As our unit tests ran in sequence before, we may not have noticed any violation of this requirment. It's very likely that some tests fail this requirement, especially the longer our project exists. The parallelization acts as a litmus test for the independence of our unit test.

If we see random test failures, we know there's something for us to work on before we can fully benefit from the test parallelization. As the independence of our tests is a best practice, it's we should revisit any test that fails to meet this requirement. Even if we decide not to parallelize them, fixing these tests is still worth the effort as they may fail randomly in the future. This makes our build less deterministic and results in frustrated developers that try to fix a critical bug while working under pressure.

### Java JUnit 5 Test Example

For the upcoming unit test parallelization example with JUnit 5 and Maven, we're using the following sample unit test:

```
class StringFormatterTest {

  private StringFormatter cut = new StringFormatter();

  @BeforeEach
  void artificialDelay() throws Exception {
    // delaying the test execution to see the parallelization efforts
    Thread.sleep(1000);
  }

  @Test
  void shouldUppercaseLowercaseString() {
    String input = "duke";

    String result = cut.format(input);

    assertEquals("DUKE", result);
  }

  // two more similar tests
}
```

The actual implementation that is being tested by this unit test is secondary. Our `StringFormatterTest` test class contains three unit tests that we artificially slow down with a `Thread.sleep()`.

This will help us see an actual difference once we enable the parallelization of our unit tests. Running the three tests of this class in sequence takes three seconds:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running de.rieckpil.blog.StringFormatterTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.056 s - in de.riec\
kpil.blog.StringFormatterTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

This is our benchmark to compare the result of running the tests in parallel. Next to this unit test, we have the following sample integration test. The test is a copy of the unit test with the postfix IT:

```
class StringFormatterIT {

  private StringFormatter cut = new StringFormatter();

  @BeforeEach
  void artificialDelay() throws Exception {
    // delaying the test execution to see the parallelization efforts
    Thread.sleep(1000);
  }

  @Test
  void shouldUppercaseLowercaseString() {
    String input = "duke";

    String result = cut.format(input);

    assertEquals("DUKE", result);
  }

  // two more similar tests
}
```

While this is not a real integration test, it acts as a placeholder test. It helps us see that our upcoming configuration change only takes effect for the unit tests. Running all tests for this sample project takes six seconds as all tests are executed in sequence.

### Parallelize Java Unit Tests with JUnit 5 and Maven

Now it's time to parallelize our Java unit tests with JUnit 5 and Maven. As JUnit runs our tests in sequence by default, we have to override this configuration only for our unit tests. A global JUnit 5 config file to define the parallelization won't do the job as this would also trigger parallelization for our integration tests.

When using Maven and the Maven Surefire Plugin, we can add custom configurations (environment variables, system properties, etc.) specifically for the tests that are executed by the Surefire plugin. We can use this technique to pass the relevant JUnit 5 configuration parameters to start parallelizing our unit tests:

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M7</version>
  <configuration>
    <properties>
      <configurationParameters>
        junit.jupiter.execution.parallel.enabled = true
        junit.jupiter.execution.parallel.mode.default = concurrent
      </configurationParameters>
    </properties>
  </configuration>
</plugin>
```

Using the configuration above to run all tests concurrently, we get the following result after running our unit tests with `./mvnw test`:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running de.rieckpil.blog.StringFormatterTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.031 s - in de.riec\
kpil.blog.StringFormatterTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

The total test execution time (see the time elapsed) went down from three seconds to one second as now all three tests run in parallel. While this time improvement may seem negligible in this example, imagine the same 300% speed improvement for a bigger project. As we don't override the parallelization strategy, JUnit will fall back to the default `dynamic` parallization and parallelize by the number of available processors/cores. Per default, JUnit uses one thread per core.

Hence, if we run the tests on a machine with only two cores, the build time will be two seconds as only two tests can run in parallel. That's why we might see build time differences when comparing our local build time with our build agent (e.g., GitHub Actions, Jenkins). We can override the degree to which parallelize by using either a custom, fixed, or dynamic strategy (factor x available cores):

```
<properties>
  <configurationParameters>
    junit.jupiter.execution.parallel.enabled = true
    junit.jupiter.execution.parallel.mode.default = concurrent
    junit.jupiter.execution.parallel.config.strategy = fixed
    junit.jupiter.execution.parallel.config.fixed.parallelism = 2
  </configurationParameters>
</properties>
```

Overriding the default parallelism configuration with the fixed configuration above, JUnit 5 will now use two threads to run our tests. This results in an overall test execution time of 2 seconds as we have three tests to execute. For more configuration options for the test parallelization, head over to the JUnit 5 documentation.

### Running Our Java Integration Tests In Sequence

The previous Maven Surefire Plugin configuration only propagates the JUnit 5 configuration for the unit tests. Hence for the Maven Failsafe Plugin that executes our integration tests, we can run the tests in sequence by not specifying any parallelism config:

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.0.0-M7</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

The `configurationParameters` from the Surefire Plugin are not shared, and hence we can isolate the configuration for both test executions. If our integration test suite allows for parallel test execution, we can even configure different parallelism and concurrent execution.

We may want to only parallelize on an integration test class level and run the test methods in sequence. This can be achieved with the following configuration:

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default = concurrent
```

As soon as both our unit and integration tests share the same parallelism config, we may rather prefer a single `junit-platform.properties` file to avoid the duplicated config for two Maven plugins. This file has to be at the root of our classpath, and hence we can store it within `src/test/resources`.

### Conclusion of Parallelizing Java Unit Tests with JUnit 5 and Maven

Parallelizing our Java unit tests with JUnit 5 and Maven is a simple technique to speed up our Maven build. The parallelization feature of JUnit 5 allows for a fine-grained parallelization configuration. By separating our tests into two categories and by using two different Maven plugins, we can isolate the parallelization setup. Depending on how many existing unit tests we already have, parallelizing them may take some initial setup effort.

Not all tests may have been written to be run in parallel in any order. Fixing them is worth the effort. A sample JUnit 5 Maven project with this parallelization setup is available on GitHub.

## Run Java Tests With Maven Silently (Only Log on Failure)

When running our Java tests with Maven they usually produce a lot of noise in the console. While this log output can help understand test failures, it's typically superfluous when our test suite is passing. Nobody will take a look at the entire output if the tests are green.

It's only making the build logs more bloated. A better solution would be to run our tests with Maven silent with no log output and only dump the log output once the tests fail. This blog post demonstrates how to achieve this simple yet convenient technique to run Java tests with Maven silently for a compact and followable build log.

### The Status Quo: Noisy Java Tests

Based on our logging configuration, our tests produce quite some log output. When running our entire test suite locally or on a CI server (e.g., GitHub Actions or Jenkins), analyzing a test failure is tedious when there's a lot of noise in the logs. We first have to find our way to the correct position by scrolling or using the search functionality of, e.g., our browser or the integrated terminal of our IDE.

A demo output for a test that verifies email functionality using GreenMail looks like the following:

```
06:52:17.982 [smtp:127.0.0.1:3025<-/127.0.0.1:53348] INFO  c.i.greenmail.user.UserManager -\
 Created user login mike@java.io for address mike@java.io with password mike@java.io becaus\
e it didn't exist before.
06:52:18.024 [smtp:127.0.0.1:3025<-/127.0.0.1:53350] INFO  c.i.greenmail.user.UserManager -\
 Created user login mike@java.io for address mike@java.io with password mike@java.io becaus\
e it didn't exist before.
```

There's usually a lot of default noise of frameworks and test libraries that add up to quite some log output when running an entire test suite:

```
[INFO] --- maven-surefire-plugin:3.0.0-M5:test (default-test) @ spring-boot-example ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running de.rieckpil.blog.greenmail.MailServiceTest
06:55:24.185 [smtp:127.0.0.1:3025<-/127.0.0.1:53406] INFO  c.i.greenmail.user.UserManager -\
 Created user login mike@java.io for address mike@java.io with password mike@java.io becaus\
e it didn't exist before.
06:55:24.225 [smtp:127.0.0.1:3025<-/127.0.0.1:53408] INFO  c.i.greenmail.user.UserManager -\
 Created user login mike@java.io for address mike@java.io with password mike@java.io becaus\
e it didn't exist before.
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.115 s - in de.riec\
kpil.blog.greenmail.MailServiceTest
[INFO] Running de.rieckpil.blog.junit5.RegistrationWebTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in de.rieckpil\
.blog.junit5.RegistrationWebTest
[INFO] Running de.rieckpil.blog.junit5.ExtensionExampleTest
Injected random UUID: 6aad64dd-0d22-4681-95ca-d254fa53c3ac
Injected random UUID: b3c44f26-7000-4998-9de8-c2b353098e72
Injected random UUID: b75d8d0e-7ac2-4f2c-936c-7d768b7db2cc
Injected random UUID: e721908f-a4ef-4938-883d-f2f614534b37
Injected random UUID: e4ead0b5-af7b-40c0-8c9a-459f4ebfcf15
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.009 s - in de.riec\
kpil.blog.junit5.ExtensionExampleTest
[INFO] Running de.rieckpil.blog.junit5.JUnit5ExampleTest
Will run only once before all tests of this class
Will run before each test
Will run before after each test
Will run before each test
Will run before after each test
Will run before each test
Will run before after each test
Will run before each test
Will run before after each test
Will run before each test
Will run before after each test
Will run only once after all tests of this class
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.022 s - in de.riec\
kpil.blog.junit5.JUnit5ExampleTest
[INFO] Running de.rieckpil.blog.wiremock.WireMockSetupTest
06:55:28.930 [main] INFO  org.eclipse.jetty.util.log - Logging initialized @6609ms to org.e\
clipse.jetty.util.log.Slf4jLog
06:55:29.015 [main] INFO  o.e.jetty.server.AbstractConnector - Stopped NetworkTrafficServer\
Connector@20960b51{HTTP/1.1, (http/1.1)}{0.0.0.0:0}
06:55:29.087 [qtp1974219375-82] INFO  o.e.j.s.h.ContextHandler.__admin - RequestHandlerClas\
s from context returned com.github.tomakehurst.wiremock.http.AdminRequestHandler. Normalize\
d mapped under returned 'null'
06:55:31.189 [main] INFO  o.e.jetty.server.AbstractConnector - Stopped NetworkTrafficServer\
Connector@736f8837{HTTP/1.1, (http/1.1)}{0.0.0.0:0}
06:55:31.189 [main] INFO  o.e.j.server.handler.ContextHandler - Stopped o.e.j.s.ServletCont\
extHandler@7acfcfc4{/,null,STOPPED}
06:55:31.189 [main] INFO  o.e.j.server.handler.ContextHandler - Stopped o.e.j.s.ServletCont\
extHandler@5ff29e8b{/__admin,null,STOPPED}
```

While we could tweak our logger configuration and set the log level to ERROR for the framework and libraries logs, their INFO can still be quite relevant when analyzing a test failure. When scrolling through the log output of passing tests, we might also see stack traces and exceptions that are intended but might confuse newcomers as they wonder if something went wrong there. Having a clean build log without much noise would better help us follow the current build.

The bigger our test suite, the more we have to scroll. If all tests pass, why pollute the console with log output from the tests? Our Maven build might also fail for different reasons than test failures, e.g., a failing OWASP dependency check or a dependency convergence issue. Getting fast to the root cause of the build failure is much simpler with a compact build log.

#### The Goal: Run Tests with Maven Silently

Our goal for this optimization is to have a compact Maven build log and only log the test output if it's really necessary (aka. tests are failing). Gradle is doing this already by default. When running tests with Gradle, we'll only see a test summary after running our tests. There's no intermediate noise inside our console. The goal is to achieve a somehow similar behavior as Gradle and run our tests silently.

If they're passing, we're fine, and there's (usually) no need to investigate the log outcome of our tests. If one of our tests fails, report the build log to the console to analyze the test failure. In short, with our target solution, we have two scenarios:

- No log output for tests in the console when all tests pass
- Print the log output of our tests when a test fails

The second scenario should be (hopefully) less likely. Hence most of our Maven builds should result in a compact and clean build log. We're fine with the log noise if there's a failure, as it helps us understand what went wrong. Let's see how we can achieve this with the least amount of configuration.

#### The Solution: Customized Maven Setup

As a first step, we configure the desired log level for testing:

```
<configuration>
  <include resource="/org/springframework/boot/logging/logback/base.xml"/>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
    </encoder>
  </appender>
  <root level="INFO">
    <appender-ref ref="STDOUT"/>
  </root>
</configuration>
```

We're using Logback (any logger works) and log any `INFO` (and above) statement to the console for the example above. We don't differentiate between our application's log and framework or test libraries. Next comes the important configuration that'll make our test silent.

The Maven Surefire (unit tests) and the Failsafe (integration tests) plugin allow redirecting the console output to a file. We won't see any test log output in the console with this configuration as it's stored within a file:

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M5</version>
  <configuration>
    <redirectTestOutputToFile>true</redirectTestOutputToFile>
    <reportsDirectory>${project.build.directory}/test-reports</reportsDirectory>
  </configuration>
</plugin>
```

When activating this functionality (`redirectTestOutputToFile`), both plugins create an output file inside the target folder for each test class with the naming scheme `TestClassName-output.txt`. We can override the location of the output files using the `reportsDirectory` configuration option. Overriding this location helps us store the output of the Surefire and Failsafe plugin at the same place:

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.0.0-M5</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <redirectTestOutputToFile>true</redirectTestOutputToFile>
    <reportsDirectory>${project.build.directory}/test-reports</reportsDirectory>
  </configuration>
</plugin>
```

This configuration for bot the Surefire and Failsafe plugin will mute our test runs, and Maven will only display a test execution summary for each test class:

```
[INFO] --- maven-surefire-plugin:3.0.0-M5:test (default-test) @ spring-boot-example ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running de.rieckpil.blog.assertj.AssertJTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.397 s - in de.riec\
kpil.blog.assertj.AssertJTest
[INFO] Running de.rieckpil.blog.hamcrest.HamcrestTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.149 s - in de.riec\
kpil.blog.hamcrest.HamcrestTest
[INFO] Running de.rieckpil.blog.citrus.CitrusDemoTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.583 s - in de.riec\
kpil.blog.citrus.CitrusDemoTest
[INFO] Running de.rieckpil.blog.xmlunit.XMLUnitTest
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.189 s - in de.riec\
kpil.blog.xmlunit.XMLUnitTest
[INFO] Running de.rieckpil.blog.MyFirstTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in de.rieckpil\
.blog.MyFirstTest
[INFO] Running de.rieckpil.blog.jsonpath.JsonPayloadTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.034 s - in de.riec\
kpil.blog.jsonpath.JsonPayloadTest
```

This compact build log makes it even fun to watch the test execution (assuming there are no flaky tests). After running our tests, we can take a look at the content of the `test-reports` folder:

```
target
`-- test-reports
    |-- de.rieckpil.blog.MyFirstTest.txt
    |-- de.rieckpil.blog.assertj.AssertJTest.txt
    |-- de.rieckpil.blog.citrus.CitrusDemoTest-output.txt
    |-- de.rieckpil.blog.citrus.CitrusDemoTest.txt
    |-- de.rieckpil.blog.greenmail.MailServiceTest-output.txt
    |-- de.rieckpil.blog.greenmail.MailServiceTest.txt
    |-- de.rieckpil.blog.hamcrest.HamcrestTest.txt
    |-- de.rieckpil.blog.jsonassert.JSONAssertTest.txt
    |-- de.rieckpil.blog.jsonpath.JsonPayloadTest.txt
    |-- de.rieckpil.blog.junit5.ExtensionExampleTest-output.txt
    |-- de.rieckpil.blog.junit5.ExtensionExampleTest.txt
    |-- de.rieckpil.blog.junit5.JUnit5ExampleTest-output.txt
```

For each test class, we'll find (at least) one text file that contains the test summary as we saw it in the build log. If the test prints output to the console, there'll be a `-output.txt` file with the content:

* `de.rieckpil.blog.greenmail.MailServiceTest-output.txt`: All console output of the test
* `de.rieckpil.blog.greenmail.MailServiceTest.txt`: The test summary, as seen in the build log

What's left is to extract the content of all our `*-output.txt` files if our build is failing. As long as our tests are all green, we can ignore the content of the output files. In case of a test failure, we must become active and dump the file contents to the console.

For this purpose, we're using a combination of `find` and `tail`.For demonstration purposes, we'll use GitHub Actions. However, the present solution is portable to any other build server that provides functionality to detect a build failure and execute shell commands:

```
name: Maven Build
on: [ push, pull_request ]
jobs:
  build-project:
    runs-on: ubuntu-20.04
    name: Build sample project
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up JDK 11
        uses: actions/setup-java@v2
        with:
          java-version: 11.0
          distribution: 'adopt'
          java-package: 'jdk'

      # ... more steps including running ./mvnw verify

      - name: Log test output on failure
        if: failure() || cancelled()
        run: find . -type f -path "*test-reports/*-output.txt" -exec tail -n +1 {} +
```

As part of the last step of our build workflow, we find all `*-output.txt` files and print their content. We only print the content of the test output files in case of a failure. With GitHub Actions, we can conditionally execute a step using a boolean expression: `if: failure() || cancelled()`. Both `failure()` and `cancelled()` are built-in functions of GitHub Actions. Every other CI server provides some similar functionality.

We include `cancelled()` to the expression to cover the scenario when our test suite is stuck and we manually stop (aka. cancel) the build. If the build is passing, this last logging step is skipped, and no test log output is logged. By using `tail -n +1 {}` we print the file name before dumping its content to the console. This helps search for the failed test class to start the investigation:

```
==> ./target/test-reports/de.rieckpil.blog.greenmail.MailServiceTest-output.txt <==
06:35:17.474 [smtp:127.0.0.1:3025<-/127.0.0.1:64642] INFO  c.i.greenmail.user.UserManager -\
 Created user login mike@java.io for address mike@java.io with password mike@java.io becaus\
e it didn't exist before.
06:35:17.547 [smtp:127.0.0.1:3025<-/127.0.0.1:64644] INFO  c.i.greenmail.user.UserManager -\
 Created user login mike@java.io for address mike@java.io with password mike@java.io becaus\
e it didn't exist before.
```

#### Summary: Silent and Compact Build Logs

We'll get compact Maven build logs with this small tweak to the Maven Surefire and Failsafe plugin and the additional step inside our build server. No more noisy test runs. We won't lose any test log output as we temporarily park it inside files and inspect the files if a test fails. This configuration will only affect the way our tests are run with Maven. We can still see the console output when executing tests within our IDE.

We'll capture any test console output with this mechanism, both from logging libraries and plain `System.out.println` calls. This technique also works when running our tests in parallel. However, if we parallelize the test methods, the console statements may be out of order inside the test output file.

If you want to see this technique in action for a public repository, take a look at the Java Testing Toolbox repository on GitHub. As part of the main GitHub Actions workflow that builds the project(s) with Maven, you'll see the Java tests being run silently. If there's a build failure, you'll see the content of the test output files as one of the last jobs.

### What the Heck Is the SpringExtension Used For?

I've seen a lot of confusion recently about the SpringExtension. When failing to get the context configuration for a test right, some developers randomly throw `@ExtendWith(SpringExtension.class)` to their test classes to (hopefully) get their tests running. With this blog post, I want to shed some light on the SpringExtension and its usage when testing Spring Boot application.

At the end of this section, you'll understand when, why, and how to use this extension. TL;DR: The `SpringExtension` enables seamless integration of JUnit Jupiter tests with Spring's TestContext framework. Most of the time, you don't need to explicitly register the SpringExtesion, as all Spring Boot test slice annotations already do this.

### What's a JUnit Jupiter Extension Used For?

The first question we have to answer when we want to understand the `SpringExtension` is what a JUnit Jupiter extension is all about. The JUnit Jupiter extension model is a single concept (in contrast to JUnit 4's `Runner` and `Rule` API) to enhance testing functionality and intercept the lifecycles of our tests programmatically. There are many different extension points for both the lifecycle (e.g., `BeforeAllCallback`) and other utility functions (e.g., `ParameterResolver`) available.

For a complete list of all available extension APIs, please take a look at the `Extension` interface and all interfaces that extend it. Their usage is versatile.

We can do housekeeping tasks before or after a test, resolve parameters, or decide whether to execute a test or skip it. Most of the time, we implement cross-cutting concerns that relate to multiple tests with an extension. For example, when writing web tests, instead of wrapping the browser interaction with a try-catch block for every test to make screenshots on failure, we could write an extension for this. JUnit Jupiter offers the `TestExecutionExceptionHandler` interface to handle exceptions for our test methods at a central place. (PS: Selenide already comes with such a screenshot extension on failure). Writing a custom extension is no rocket science.

As part of the Testing Spring Boot Applications Masterclass, we develop a custom extension to inject random `UUID`s to our test methods. Whenever we want to activate a JUnit Jupiter extension for our test, we have to explicitly register the extension with `@ExtendWith` on top of the test class:

```
@ExtendWith(MyExtension.class)
class MyTest {

  @Test
  void test() {
  }
}
```

Many frameworks and testing libraries ship with a custom JUnit Jupiter extension for convenient integration with the JUnit environment. Examples are:

- Mockito's `MockitoExtension` for a seamless setup of our mocks
- Testcontainers `TestcontainersExtension` (activated with `@Testcontainers`) to conveniently start and stop Docker containers
- Spring's `SpringExtension`

For more information about the extension model and its various APIs, consult the JUnit 5 User Guide.

#### What's the Purpose of the SpringExtension?

As a next step, let's investigate the cross-cutting functionality the `SpringExtension` implements. The best way to start our investigation is to take a look at the source code of the `SpringExtension` to understand which extension APIs it implements:

```
public class SpringExtension implements BeforeAllCallback,
    AfterAllCallback,
    TestInstancePostProcessor,
    BeforeEachCallback,
    AfterEachCallback,
    BeforeTestExecutionCallback,
    AfterTestExecutionCallback,
    ParameterResolver {

  // ...

}
```

That's a lot. As we can see from the source code above, the `SpringExtension` is heavily involved in the lifecycle of our tests. Without diving too deep into the implementation, the main responsibilities of this extension are the following:

- manage the lifecycle of the Spring `TestContext` (e.g., start a new one or get a cached context)
- support dependency injection for parameters (e.g., test class constructor or test method)
- cleanup and housekeeping tasks after the test

The `SpringExtension` acts as a glue between JUnit Jupiter and Spring Test. Most of the time the `SpringExtension` delegates its responsibilities to the `TestContextManager` to do the heavy lifting:

```
public class SpringExtension {

  @Override
  public void beforeAll(ExtensionContext context) throws Exception {
    getTestContextManager(context).beforeTestClass();
  }
}
```

The `TestContextManager` is responsible for a single `TestContext` and test framework agnostic. Furthermore, the `TestContextManager` also invokes all registered `TestContextListeners` based on the lifecycle event (e.g., before test class or after a test method). Let's take a look at another practical example of the `SpringExtension` in-action: Resolving (aka. injecting) parameters:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class ApplicationIT {

  @Test
  void needsEnvironmentBeanToVerifySomething(
     @Autowired Environment environment) { // resolved by the SpringExtension

    assertNotNull(environment);
  }
}
```

The test above injects the `Environment` via a method parameter. In the background, the following code snippet of the `SpringExtension` is responsible for resolving this parameter:

```
public class SpringExtension {

  // ...

  @Override
  public boolean supportsParameter(ParameterContext parameterContext,
  ExtensionContext extensionContext) {
    // determine whether or not this extension is responsible to resolve the parameter
  }

  @Nullable
  public Object resolveParameter(ParameterContext parameterContext,
  ExtensionContext extensionContext) {
    // return the bean from the TestContext
  }
}
```

Before the `SpringExtension` tries to resolve a given parameter, the `supportsParameter` (part of the `ParameterResolver` interface) determines if it's the responsibility of this extension. In our example, the `@Autowired` annotation next to the parameter is a clear indicator for the `SpringExtension` to resolve this parameter from the `TestContext`.

Field injection, however, works via the `DependencyInjectionTestExecutionListener`. This is one of the many default `TestExecutionListeners` that the `TestContextManager` invokes before running any test.

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class ApplicationIT {

  @Autowired
  // injected by the DependencyInjectionTestExecutionListener
  private CustomerService customerService;

  @Test
  void needsEnvironmentBeanToVerifySomething(
    @Autowired Environment environment // resolved by the SpringExtension
  ) {
    assertNotNull(environment);
  }
}
```

### When Do We Need To Register the SpringExtension?

Most of the time, we don't need to explicitly register this extension because it's already activated for us. This is the case whenever we use a Spring Boot test annotation. All Spring Boot test slice annotations and also @SpringBootTest register the `SpringExtension` out-of-the-box. This is beneficial, as otherwise, the tests would fail to start as they require a TestContext to work with.

Hence this opinionated approach saves us some keystrokes, and we don't encounter weird test failures because we've forgotten to register the `SpringExtension`. We can see this by taking a look at the source code of the `@WebMvcTest` annotation:

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@BootstrapWith(WebMvcTestContextBootstrapper.class)
@ExtendWith({SpringExtension.class})
// ... further annotations
public @interface WebMvcTest {

}
```

Among other meta-annotations and instructions on what to auto-configure for this kind of test, we see the `SpringExtension` is activated for us. The same is true for `@SpringBootTest`:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@BootstrapWith(SpringBootTestContextBootstrapper.class)
@ExtendWith(SpringExtension.class)
// ...
public @interface SpringBootTest {
  // ...
}
```

So whenever we encounter a test class where the `SpringExtension` is registered manually, and a Spring Boot test slice annotation is in use, we can safely remove it:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@ExtendWith(SpringExtension.class) // not necassary and should be removed
class ApplicationIT {

  // ...

}
```

While JUnit Jupiter won't complain (aka. fail the test or produce noise in the logs), if the same extension is configured twice, we should remove it to avoid this duplication. It's redundant code and might only lead to confusion for new team members if we don't consistently add the `SpringExtension` everywhere.

### When Do We Need to Register the SpringExtension (Part II)?

There are limited use cases where we have to explicitly register the `SpringExtension` manually. One of such use cases is writing a custom test slice annotation. As part of the Testing Spring Boot Applications Masterclass, we bootstrap a messaging component of our application for testing purposes.

This hands-on example demonstrates how to only bootstrap Amazon SQS Listener relevant parts of our application. As there is now Spring Boot test slice annotation available, we manually register the `SpringExtension`to work with a Spring TestContext throughout the test:

```
@ExtendWith(SpringExtension.class)
@Import(BookSynchronizationListener.class)
@ImportAutoConfiguration(MessagingAutoConfiguration.class)
@Testcontainers(disabledWithoutDocker = true)
class BookSynchronizationListenerSliceTest {

   // test a SQS listener in isolation

}
```

Furthermore, for projects that use plain Spring without Spring Boot, we're also in the driver seat and have to take care to register the `SpringExtension`.

### Summary

The `SpringExtension` implements several JUnit Jupiter extension model callback methods for seamless integration between JUnit and Spring. When testing Spring Boot applications, most of the time, we don't have to explicitly register this extension as all sliced context annotations (e.g. `@WebMvcTest`) do this for us.

The official documentation is also an excellent source of information if you want to dive deeper into this topic. Consider the following blog posts to learn more about Spring Boot's excellent testing capabilities:

- Spring Boot Unit and Integration Testing Overview

- Spring Boot Test Slices: Overview and Usage
- Guide to @SpringBootTest for Spring Boot Integration Tests

## Five Lesser-Known JUnit 5 Features

Writing your first test with JUnit 5 is straightforward. Annotate your test method with `@Test` and verify the result using assertions. Apart from the basic testing functionality of JUnit 5, there are some features you might not have heard about (yet). Discover five JUnit 5 features I found useful while working with JUnit 5: test execution order, nesting tests, parameter injection, parallelizing tests, and conditionally run tests.

### Test Execution Ordering

The first feature we'll explore is influencing the test execution order. While JUnit 5 has the following default-test execution order:

> …, test methods will be ordered using an algorithm that is deterministic but intentionally nonobvious. This ensures that subsequent runs of a test suite execute test methods in the same order, thereby allowing for repeatable builds.

you can configure a different ordering mechanism. Therefore, either implement your own `MethodOrderer` or use a built-in that orders: alphabetically, randomly, or numerically based on a specified value. Let's take a look at how to order some unit tests using the `OrderAnnotation` ordering mechanism:

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class OrderedExecutionTest {

  @Test
  @Order(2)
  void testTwo() {
    System.out.println("Executing testTwo");
    assertEquals(4, 2 + 2);
  }

  @Test
  @Order(1)
  void testOne() {
    System.out.println("Executing testOne");
    assertEquals(4, 2 + 2);
  }

  @Test
  @Order(3)
  void testThree() {
    System.out.println("Executing testThree");
    assertEquals(4, 2 + 2);
  }
}
```

With `@TestMethodOrder(MethodOrderer.OrderAnnotation.class)` you basically opt-out from the JUnit 5 default ordering. The actual order is then specified with `@Order`. A lower value implies a higher priority. Once you execute the tests above, the order is the following: `testOne`, `testTwo` and `testThree`. As a general best practice your test should not rely on the order they are executed. Nevertheless, there are scenarios where this feature helps to execute the tests in configurable order.

### Nesting Tests With JUnit 5

Usually, you test different business requirements inside the same test class. With Java and JUnit 5, you write them one after the other and add new tests to the bottom of the class. While this is working for a small number of tests inside a class, this approach gets harder to manage for bigger test suites. Consider you want to adjust tests that verify a common scenario. As there is no defined order or grouping inside your test class you end up scrolling and searching them. The following JUnit 5 feature allows you to counteract this pain point of a growing test suite: nested tests. You can use this feature to group tests that verify common functionality. This does not only improves maintainability but also reduces the time to understand what the class under test is responsible for:

```java
class NestedTest {

  @Nested
  @DisplayName("Testing division functionality")
  class DivisionTests {

    @Test
    void shouldDivideByTwo() {
      assertEquals(4, 8 / 2);
    }

    @Test
    void shouldThrowExceptionForDivideByZero() {
      assertThrows(ArithmeticException.class, () -> {
        int result = 8 / 0;
      });
    }
  }

  @Nested
  @DisplayName("Testing addition functionality")
  class AdditionTests {

    @Test
    void shouldAddTwo() {
      assertEquals(4, 2 + 2);
    }

    @Test
    void shouldAddZero() {
      assertEquals(2, 2 + 0);
    }
  }

}
```

You might run into issues while using this feature in conjunction with some Spring Boot test features.

### Parameter Injection With JUnit 5

JUnit 5 offers parameter injection for test constructor and method arguments. There are built-in parameter resolvers you can use to inject an instance of `TestReport`, `TestInfo`, or `RepetitionInfo` (in combination with a repeated test):

```
@RepeatedTest(5)
void testMethodName(TestInfo testInfo, TestReporter testReporter,
  RepetitionInfo repetitionInfo) {
  System.out.println(testInfo.getTestMethod().get().getName());
  testReporter.publishEntry("secretMessage", "JUnit 5");
  System.out.println(repetitionInfo.getCurrentRepetition() +
    " from " + repetitionInfo.getTotalRepetitions());
}
```

Furthermore, you can implement your own `ParameterResolver` to resolve arguments of any type. We can use this mechanism to resolve a random `UUID` for our tests:

```
public class RandomUUIDParameterResolver implements ParameterResolver {

  @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.PARAMETER)
  public @interface RandomUUID {
  }

  @Override
  public boolean supportsParameter(ParameterContext parameterContext,
  ExtensionContext extensionContext) {
    return parameterContext.isAnnotated(RandomUUID.class);
  }

  @Override
  public Object resolveParameter(ParameterContext parameterContext,
   ExtensionContext extensionContext) {
    return UUID.randomUUID().toString();
  }

}
```

With `supportsParameter` we indicate that this resolver is capable of resolving a requested parameter, as you can have multiple `ParameterResolver` . We're checking if the requested parameter is annotated with our custom annotation. In addition, you could also verify that the parameter is of type `String`. We can use this resolver for our tests once we register this extension with `@ExtendWith`:

```
@ExtendWith(RandomUUIDParameterResolver.class)
public class ParameterInjectionTest {

  @RepeatedTest(5)
  public void testUUIDInjection(@RandomUUID String uuid) {
    System.out.println("Random UUID: " + uuid);
  }
}
```

### Test Parallelization With JUnit 5

While you might have configured this in the past with the corresponding Maven or Gradle plugin, you can now configure this as an experimental feature with JUnit (since version 5.3). This gives you more fine-grain control on how to parallelize the tests. A basic configuration can look like the following:

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = concurrent
```

This enables parallel execution for all your tests and set the execution mode to `concurrent`. Compared to `same_thread`, `concurrent` does not enforce to execute the test in the same thread of the parent. For a per test class or method mode configuration, you can use the `@Execution` annotation. There are multiple ways to set these configuration values, one is to use the Maven Surefire plugin for it:

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
  <configuration>
    <properties>
      <configurationParameters>
        junit.jupiter.execution.parallel.enabled = true
        junit.jupiter.execution.parallel.mode.default = concurrent
      </configurationParameters>
    </properties>
  </configuration>
</plugin>
```

or a `junit-platform.properties` file inside `src/test/resources` with the configuration values as content. You should benefit the most when using this feature for unit tests. Enabling parallelization for integration tests might not be possible or easy to achieve depending on your setup. Therefore, I can recommend executing your unit tests with the Maven Surefire plugin and configure parallelization for them. All your integration tests can then be executed with the Maven Failsafe plugin where you don't specify these JUnit 5 configuration parameters. For more fine-grain parallelism configuration, take a look at the official JUnit 5 documentation.

### Conditionally Disable Tests

The last feature is useful when you want to avoid tests being executed based on different conditions. There might be tests that don't run on different operating systems or require different environment variables to be present. JUnit 5 comes with some built-in conditions that you can use for your tests, e.g.:

```
@Test
@DisabledOnOs(OS.LINUX)
void disabledOnLinux() {
  assertEquals(42, 40 + 2);
}

@Test
@DisabledIfEnvironmentVariable(named = "FLAKY_TESTS", matches = "false")
void disableFlakyTest() {
  assertEquals(42, 40 + 2);
}
```

On the other side, writing a custom condition is pretty straightforward. Let's consider you don't want to execute a test around midnight:

```
@Test
@DisabledOnMidnight
void disabledOnMidNight() {
  assertEquals(42, 40 + 2);
}
```

All you have to do is to implement `ExecutionCondition` and add your own condition:

```
public class DisabledOnMidnightCondition implements ExecutionCondition {

  private static final ConditionEvaluationResult ENABLED_BY_DEFAULT =
    enabled("@DisabledOnMidnight is not present");

  private static final ConditionEvaluationResult ENABLED_DURING_DAYTIME =
    enabled("Test is enabled during daytime");

  private static final ConditionEvaluationResult DISABLED_ON_MIDNIGHT =
    disabled("Disabled as it is around midnight");

  @Documented
  @Target({ElementType.TYPE, ElementType.METHOD})
  @Retention(RetentionPolicy.RUNTIME)
  @ExtendWith(DisabledOnMidnightCondition.class)
  public @interface DisabledOnMidnight {
  }

  @Override
  public ConditionEvaluationResult evaluateExecutionCondition(ExtensionContext context) {
    Optional<DisabledOnMidnight> optional =
    findAnnotation(context.getElement(), DisabledOnMidnight.class);
    if (optional.isPresent()) {
      LocalDateTime now = LocalDateTime.now();
      if (now.getHour() == 23 || now.getHour() <= 1) {
        return DISABLED_ON_MIDNIGHT;
      } else {
        return ENABLED_DURING_DAYTIME;
      }
    }
    return ENABLED_BY_DEFAULT;
  }
}
```

There is a dedicated testing category available on my blog for more content about JUnit and topics like Testcontainers, testing Spring Boot or Jakarta EE applications, etc. You can find the source code for these five JUnit 5 features on GitHub.

## Recipes

### Testing the Web Layer

Did you ever found yourself saying: *I usually ignore testing my Spring Web MVC controller endpoints because the security setup is tricky*. That belongs to the past. With MockMvc, Spring provides an excellent tool for testing Spring Boot applications.

This guide provides you with recipes to verify your `@Controller` and `@RestController` endpoints and other relevant Spring Web MVC components using `MockMvc`.

#### Required Maven Dependencies For MockMvc

It is very little we need to start testing our controller endpoints with `MockMvc`. We only need to include both the Spring Boot Starter Web and the Spring Boot Starter Test (aka. swiss-army for testing Spring Boot applications):

```
<dependency>
  <groupId>org.springframework.boot<groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Once we secure our endpoints using Spring Security, the following test dependency is a must-have to test our protected controller with ease:

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-test</artifactId>
  <scope>test</scope>
</dependency>
```

### What Can We Test and Verify with MockMvc?

That's easy to answer: Everything related to Spring MVC! This includes our controller endpoints (both `@Controller` and `@RestController`) and any Spring MVC infrastructure components like `Filter`, `@ControllerAdvice`, `WebMvcConfigurer`, etc. Possible test scenarios can be the following:

- Does my REST API controller return a proper JSON response?
- Is the model for my Thymeleaf view initialized?
- Does my endpoint return the HTTP status code 418 if my service layer throws a `TeaNotFoundException`?
- Is my REST API endpoint protected with Basic Auth?
- Can only users with the role ADMIN access the DELETE endpoint?
- etc.

With `MockMvc` we perform requests against a **mocked servlet environment**. There won't be any real HTTP communication for such tests. We directly work with the mocked environment provided by Spring.

`MockMvc` acts as the **entry point** to this mocked servlet environment. Similar to the `WebTestClient` when accessing our started Servlet container over HTTP.

### MockMvc Test Setup

There are two ways to create a `MockMvc` instance: using Spring Boot's auto-configuration or hand-crafting it. Following Spring Boot's auto-configuration principle, we only need to annotate our test with `@WebMvcTest`.

This annotation not only ensures to auto-configure `MockMvc` but also creates a sliced Spring context containing only MVC-related beans. To keep the sliced test context small, we can pass the class name of the controller we want to test: `@WebMvcTest(MyController.cass)`.

Otherwise, Spring will create a context including all our controller endpoints. The second approach is helpful if we're using Spring without Spring Boot or if we want to fine-tune the setup. The `MockMvcBuilders` class provides several entry points to construct a `MockMvc` instance:

```
class TaskControllerTest {

  private MockMvc mockMvc;

  @BeforeEach
  public void setup() {
    this.mockMvc = MockMvcBuilders.standaloneSetup(new TaskController(new TaskService()))
      .setControllerAdvice()
      .setLocaleResolver(localResolver)
      .addInterceptors(interceptorOne)
      .build();
  }
}
```

or …

```
@WebMvcTest(TaskController.class)
public class TaskControllerSecondTest {

  @Autowired
  private WebApplicationContext context;

  @MockBean
  private TaskService taskService;

  protected MockMvc mockMvc;

  @BeforeEach
  public void setup() {
    this.mockMvc = MockMvcBuilders
      .webAppContextSetup(this.context)
      .apply(springSecurity())
      .build();
  }

}
```

### Invoke and Test an API Endpoint with MockMvc

Let’s start with the first test: Ensuring the JSON result from a `@RestController` endpoint is correct. Our sample controller has three endpoint mappings and acts as a REST API for a `User` entity:

```
@Validated
@RestController
@RequestMapping("/api/users")
public class UserController {

  private final UserService userService;

  public UserController(UserService userService) {
    this.userService = userService;
  }

  @GetMapping
  public List<User> getAllUsers() {
    return this.userService.getAllUsers();
  }

  @GetMapping
  @RequestMapping("/{username}")
  public User getUserByUsername(@PathVariable String username) {
    return this.userService.getUserByUsername(username);
  }
  @PostMapping
  public ResponseEntity<Void> createNewUser(@RequestBody @Valid User user,
  UriComponentsBuilder uriComponentsBuilder) {
  UriComponentsBuilder uriComponentsBuilder) {
    this.userService.storeNewUser(user);
    return ResponseEntity
      .created(uriComponentsBuilder.path("/api/users/{username}").build(user.getUsername()))
      .build();
  }
}
```

With a first test, we want to ensure the JSON payload from `/api/users` is what we expect. As our `UserController` has a dependency on a `UserService` bean, we'll mock it.

This ensures we can solely focus on testing the web layer and don't have to provide further infrastructure for our service classes to work (e.g. remote systems, databases, etc.). The minimal test setup looks like the following:

```
@WebMvcTest(UserController.class)
class UserControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private UserService userService;

  // ... upcoming test

}
```

Before we invoke the endpoint using `MockMvc`, we have to mock the result of our `UserService`. Therefore we can return a hard-coded list of users:

```
@Test
void shouldReturnAllUsersForUnauthenticatedUsers() throws Exception {
  when(userService.getAllUsers())
    .thenReturn(List.of(new User("duke", "duke@spring.io")));

  this.mockMvc
    .perform(MockMvcRequestBuilders.get("/api/users"))
    .andExpect(MockMvcResultMatchers.status().isOk())
    .andExpect(MockMvcResultMatchers.jsonPath("$.size()").value(1))
    .andExpect(MockMvcResultMatchers.jsonPath("$[0].username").value("duke"))
    .andExpect(MockMvcResultMatchers.jsonPath("$[0].email").value("duke@spring.io"));
}
```

Next, we use `MockMvcRequestBuilders` to construct our request against the mocked servlet environment. This allows us to specify the HTTP method, any HTTP headers, and the HTTP body. Once we invoke our endpoint with `perform`, we can verify the HTTP response using fluent assertions to inspect: headers, status code, and the body.

JsonPath is quite helpful here to verify the API contract of our endpoint. Using its standardized expressions (somehow similar to XPath for XML) we can write assertions for any attribute of the HTTP response.

Our next test focuses on testing the HTTP POST endpoint to create new users. This time we need to send data alongside our `MockMvc` request:

```
@Test
void shouldAllowCreationForUnauthenticatedUsers() throws Exception {
  this.mockMvc
    .perform(
      post("/api/users")
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"username\": \"duke\", \"email\":\"duke@spring.io\"}")
        .with(csrf())
    )
    .andExpect(status().isCreated())
    .andExpect(header().exists("Location"))
    .andExpect(header().string("Location", Matchers.containsString("duke")));

  verify(userService).storeNewUser(any(User.class));
}
```

To avoid an HTTP 403 Forbidden response, we have to populate a valid

`CsrfToken` for the request. This only applies if your project includes Spring Security and CSRF is enabled (which you always should).

Due to the great MockMvc and Spring Security integration, we can create this token using `.with(csrf())` .

### Writing Tests for a Thymeleaf Controller

There is more to Spring MVC than writing API endpoints: exposing server-side rendered views following the MVC (Model View Controller) pattern.

To showcase this kind of controller test, let's assume our application exposes one Thymeleaf view including a pre-filled `Model`:

```
@Controller
@RequestMapping("/dashboard")
public class DashboardController {

  private final DashboardService dashboardService;

  public DashboardController(DashboardService dashboardService) {
    this.dashboardService = dashboardService;
  }

  @GetMapping
  public String getDashboardView(Model model) {

    model.addAttribute("user", "Duke");
    model.addAttribute("analyticsGraph", dashboardService.getAnalyticsGraphData());
    model.addAttribute("quickNote", new QuickNote());

    return "dashboard";
  }
}
```

From a testing perspective, it would be great if we can verify that our model is present and we are returning the correct view (locatable & renderable by Spring MVC).

Writing a web test with Selenium for this scenario would be quite expensive (time & maintenance effort). Fortunately, `MockMvc` provides verification mechanisms for these kinds of endpoints, too:

```
@WebMvcTest(DashboardController.class)
class DashboardControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private DashboardService dashboardService;

  @Test
  void shouldReturnViewWithPrefilledData() throws Exception {
    when(dashboardService.getAnalyticsGraphData()).thenReturn(new Integer[]{13, 42});

    this.mockMvc
      .perform(get("/dashboard"))
      .andExpect(status().isOk())
      .andExpect(view().name("dashboard"))
      .andExpect(model().attribute("user", "Duke"))
      .andExpect(model().attribute("analyticsGraph", Matchers.arrayContaining(13, 42)))
      .andExpect(model().attributeExists("quickNote"));
  }
}
```

The `MockMvc` request setup looks similar to the tests in the last section. What's different is the way we assert the response. As this endpoint returns a view rather than JSON, we make use of the `ResultMatcher .model()`.

And can now write assertions for the big M in MVC: the `Model`. There is also a `ResultMatcher` available to ensure any `FlashAttributues` are present if you follow the POST - redirect - GET pattern.

### Test a Secured Endpoint with Spring Security and MockMvc

Let's face reality, most of the time our endpoints are protected by Spring Security. Neglecting to write tests because the security setup is hard, is foolish. Because it isn't. The excellent integration of `MockMvc` and Spring Security ensures this.

Once Spring Security is part of our project, the `MockMvc` will be auto-configured with our security config**.

**UPDATE**: When using a `SecurityFilterChain` bean for the Spring Security config in Spring Boot 3.0, we need to manually import the security configuration with `@Import(SecurityConfig.class)` to our test. As a demo, let's consider the following security configuration for our MVC application:

```
@Configuration
@EnableMethodSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .authorizeRequests(authorize -> authorize
        .mvcMatchers(HttpMethod.GET, "/dashboard").permitAll()
        .mvcMatchers(HttpMethod.GET, "/api/tasks/**").authenticated()
        .mvcMatchers("/api/users/**").permitAll()
        .mvcMatchers("/**").authenticated()
      )
      .httpBasic();
  }
}
```

This configuration creates two kinds of endpoints: unprotected endpoints and protected endpoints using Basic Auth. Manually fiddling around with authentication is the last thing we want to do when verifying our secured Spring Web MVC endpoints. Let's use the following API endpoint as an example:

```
@RestController
@RequestMapping("/api/tasks")
public class TaskController {

  private final TaskService taskService;

  public TaskController(TaskService taskService) {
    this.taskService = taskService;
  }

  @PostMapping
  public ResponseEntity<Void> createNewTask(@RequestBody JsonNode payload,
                                            UriComponentsBuilder uriComponentsBuilder) {

    Long taskId = this.taskService.createTask(payload.get("taskTitle").asText());

    return ResponseEntity
      .created(uriComponentsBuilder.path("/api/tasks/{taskId}").build(taskId))
      .build();
  }
  @DeleteMapping
  @RolesAllowed("ADMIN")
  @RequestMapping("/{taskId}")
  public void deleteTask(@PathVariable Long taskId) {
    this.taskService.deleteTask(taskId);
  }
}
```

Valid test scenarios would now include verifying that we block anonymous users, allow authenticated users access, and only allow privileged users to delete tasks. The anonymous part is simple. Just invoke the endpoint without any further setup and expect HTTP status 401:

```
@WebMvcTest(TaskController.class)
// @Import(SecurityConfig.class)
   required when using a SecurityFilter chain bean -> Spring Boot 3.0
class TaskControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private TaskService taskService;

  @Test
  void shouldRejectCreatingReviewsWhenUserIsAnonymous() throws Exception {
    this.mockMvc
      .perform(
        post("/api/tasks")
          .contentType(MediaType.APPLICATION_JSON)
          .content("{\"taskTitle\": \"Learn MockMvc\"}")
          .with(csrf())
      )
      .andExpect(status().isUnauthorized());
  }

}
```

When we now want to test the happy path of creating a task, we need an authenticated user accessing the endpoint. Including the Spring Security Test dependency (see the first section), we have multiple ways to inject a user into the `SecurityContext` of the mocked Servlet environment. The most simple one is `user()`, where we can specify any user-related attributes alongside the `MockMvc` request:

```
@Test
void shouldReturnLocationOfReviewWhenUserIsAuthenticatedAndCreatesReview()
  throws Exception {

  when(taskService.createTask(anyString())).thenReturn(42L);

  this.mockMvc
    .perform(
      post("/api/tasks")
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"taskTitle\": \"Learn MockMvc\"}")
        .with(csrf())
        .with(SecurityMockMvcRequestPostProcessors.user("duke"))
    )
    .andExpect(status().isCreated())
    .andExpect(header().exists("Location"))
    .andExpect(header().string("Location", Matchers.containsString("42")));
}
```

Adding this method to our `MockMvc` request setup, Spring Test populates a valid `SecurityContext` that holds information about the principal *duke*. We can further tweak the user setup and assign roles to test the DELETE endpoint:

```
@Test
void shouldAllowDeletingReviewsWhenUserIsAdmin() throws Exception {
  this.mockMvc
    .perform(
      delete("/api/tasks/42")
        .with(user("duke").roles("ADMIN", "SUPER_USER"))
        .with(csrf())
    )
    .andExpect(status().isOk());

  verify(taskService).deleteTask(42L);
}
```

There are way more `SecurityMockMvcRequestPostProcessors` available that allow setting up users for your authentication method: `jwt()`, `oauth2Login()`, `digest()`, `opaqueToken()`, etc. We can also use `@WithMockUser` on top of your test to define the mocked user for the whole test execution:

```
@Test
@WithMockUser("duke")
public void shouldRejectDeletingReviewsWhenUserLacksAdminRole() throws Exception {
  this.mockMvc
    .perform(delete("/api/tasks/42"))
    .andExpect(status().isForbidden());
}
```

### Using MockMvc in Combination with @SpringBootTest

We can also use `MockMvc` together with `@SpringBootTest`. With this setup, we'll get our entire Spring application context populated and don't have to mock any service class. Such tests ensure the integration of multiple parts of your application (aka. integration tests):

```
@SpringBootTest
@AutoConfigureMockMvc
class ApplicationTests {

  @Autowired
  private MockMvc mockMvc;

  @Test
  public void shouldAllowDeletingReviewsWhenUserIsAdmin() throws Exception {
    this.mockMvc
      .perform(
        delete("/api/tasks/42")
          .with(SecurityMockMvcRequestPostProcessors
          .user("duke").roles("ADMIN", "SUPER_USER"))
          .with(csrf())
      )
      .andExpect(status().isOk());
  }
}
```

As with this setup, we still test against a mocked servlet environment, we should at least add some happy-path tests that invoke our application over HTTP.

For this test scenario, the WebTestClient fits perfectly. You can find the demo application for this testing Spring Boot MVC controllers example using MockMvc on GitHub.

## Testing the Database Layer

Similar to testing our web layer in isolation with `@WebMvcTest`, Spring Boot provides a convenient way to test our Spring Boot JPA persistence layer. Using the @DataJpaTest test slice annotation, we can easily write integration tests for our JPA persistence layer.

While the default configuration expects an embedded database, this section demonstrates how to test any Spring Data JPA repository with a running database server using Testcontainers and Flyway. We're using Spring Boot 2.5, Java 17, and a PostgreSQL database for the sample application.

### What Not to test for Our Spring Data JPA Persistence Layer

With Spring Data JPA, we create repository interfaces for our JPA entity classes by extending the `JpaRepository` interface:

```
public interface OrderRepository extends JpaRepository<Order, Long> {
}
```

Such repositories already provide a set of methods to retrieve, store and delete our JPA entities like `.findAll()`, `.save()`, `.delete(Order order)`. When testing these default methods, we'll end up testing the Spring Data JPA framework.

That shouldn't be our goal unless we don't trust the test suite of the framework (if that's the case, clearly shouldn't use this technology for running applications in production). Spring Data also has the concept of derived queries where the actual SQL query is derived from the method name:

```
public interface OrderRepository extends JpaRepository<Order, Long> {
  List<Order> findAllByTrackingNumber(String trackingNumber);
}
```

These methods are validated on application startup. If we refer to an unknown column of our entity or don't follow the naming convention, our application won't start.

Testing these derived queries of our `JpaRepository` doesn't add many benefits, similar to testing the default CRUD methods. With such tests, we would again test the Spring Data framework, which we want to avoid as we want to focus on testing our application.

Next, what about testing the entity mapping? Should we make sure that our JPA entity model maps to our database schema? To validate that the mapping is correct, we can rely on a feature of Hibernate that validates the schema on application startup:

```
spring:
  jpa:
    hibernate:
      ddl-auto: validate
```

This ensures that a corresponding database table exists for our JPA entities, all columns are present, and they have the correct column type. There are still scenarios that this validation does not cover, like checking for constraints like `unique`.

Hence we shouldn't explicitly focus on testing this but ensure that all entity lifecycles are implicitly covered with an integration test. We should also ensure that our integration tests not only interact with Hibernate's first-level cache but also do full database roundtrips. The `TestEntityManager` can help here.

### What to test for Our Spring Data JPA Persistence Layer

So if the examples above don't qualify for an integration test, what should we focus on when testing our JPA persistence layer?

We should focus on **non-trivial handcrafted queries**. A good example is the following native query that makes use of PostgreSQL's JSON support:

```
public interface OrderRepository extends JpaRepository<Order, Long> {

  @Query(value = """
      SELECT *
      FROM orders
      WHERE items @> '[{"name": "MacBook Pro"}]';
    """, nativeQuery = true)
  List<Order> findAllContainingMacBookPro();

}
```

PS: Wondering how we can create this multi-line string without concatenation? That's the new Java text block feature that is GA since Java 15. Our `Order` entity class uses Vlad Mihaleca's hibernate-types project to map PostgreSQL's `JSNOB` column type to a Java `String`.

```
@Entity
@Table(name = "orders")
@TypeDef(name = "jsonb", typeClass = JsonBinaryType.class)
public class Order {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(nullable = false, unique = true)
  private String trackingNumber;

  @Type(type = "jsonb")
  @Column(columnDefinition = "jsonb")
  private String items;

  // constructor, getter, setters, etc.
}
```

Knowing what to test now brings us to the topic of how to test it.

### How to Test the Spring Data JPA Persistence Layer

We have multiple options to verify the native query of the `OrderRepository`. We could write a unit test. As the Spring Data JPA repositories are Java interfaces, there is not much to achieve with a unit test. These repositories only work in combination with Spring Data.

We need to test these repositories *in action*. Starting the whole Spring application context for this would be overkill. Spring Boot provides a test annotation that allows us to start a sliced Spring Context with only relevant JPA components: `@DataJpaTest`. By default, this would start an in-memory database like H2 or Derby. As long as our project doesn't use an in-memory database, starting one for our tests is counterproductive.

Our Flyway migration scripts might use database-specific features that are not compatible with the in-memory database. Maintaining a second set of DDL migration scripts for the in-memory database is a non-negligible effort.

In the pre-container times, using an in-memory database might have been the easiest solution as you would otherwise have to install or provide a test database for every developer. With the rise of Docker, we are now able to start any containerized software on-demand with ease.

This includes databases. Testcontainers makes it convenient to start any Docker container of choice for testing purposes. I'm not going to explain Testcontainers in detail here, as there are already several articles available:

1: Write Spring Boot Integration Tests with Testcontainers (JUnit 4 & 5)

2. Reuse Containers with Testcontainers for Fast Integration Tests
3. Testing Spring Boot Applications with Kotlin and Testcontainers

#### Create the Database Schema for Our @DataJpaTest

An empty database container doesn't help us much with our test case. We need a solution to create our database tables prior to the test execution. When working with `@DataJpaTest` and an embedded database, we can achieve this with Hibernate's `ddl-auto` feature set to `create-drop`. This ensures first to create the database schema based on our Java entity definitions and then drop it afterward.

While this works and might feel convenient (we don't have to write any SQL), we should instead stick to our handcrafted Flyway migration scripts.

Otherwise, there might be a difference between our database schema during the test and production. Remember: We want to be as close as possible to our production setup when testing our application. The migration script for our examples contains a single DDL statement:

```
CREATE TABLE orders (
    ID BIGSERIAL PRIMARY KEY,
    TRACKING_NUMBER VARCHAR(255) UNIQUE NOT NULL,
    ITEMS JSONB
);
```

When starting our sliced Spring context with `@DataJpaTest`, Flyway's auto-configuration is included, and our migration scripts are executed for each new Spring `TestContext`.

### Preloading Data for the Test

Having the database tables created, we now have multiple options to populate data: during the test, using the `@Sql` annotation, as part of JUnit's `@BeforeEach` lifecycle or using a custom Docker image.

Let's start with the most straightforward approach. Every test prepares the data that it needs for verifying a specific method. As the Spring Test Context contains all our Spring Data JPA repositories, we can inject the repository of our choice and save our entities:

```
orderRepository.save(createOrder("42", "[]"));
```

Next comes the `@Sql` annotation. With this annotation, we can execute any SQL script before running our test. We can place our init scripts inside `src/test/resources`. It's not necessary to put them on the classpath as we can also reference, e.g., a file on disk or an HTTP resource (basically anything that can be resolved by Spring's `ResouceUtils` class):

```
INSERT INTO orders (tracking_number, items) VALUES
  ('42', '[{"name": "MacBook Pro", "amount" : 42}]');
INSERT INTO orders (tracking_number, items)
  VALUES ('43', '[{"name": "Kindle", "amount" : 13}]');
INSERT INTO orders (tracking_number, items)
  VALUES ('44', '[]');
```

```
@Test
@Sql("/scripts/INIT_THREE_ORDERS.sql")
void shouldReturnOrdersThatContainMacBookPro() {
}
```

If all your test cases (in the same test class) share the same data setup, we can use JUnit Jupiter's `@BeforeEach` to initialize our tables with data.

```
@BeforeEach
void initData() {
  orderRepository.save(createOrder("42", "[]"));
}
```

We can also create a custom database image that already contains our database with a preset of data. This can also mirror the size of production.

We save a lot of time with this approach for bigger projects as Flyway doesn't have to migrate any script. The only downside of this approach is to maintain and keep this test image up-to-date.

### Writing the Test for Our Spring Data JPA Repository

Putting it all together, we can now write our test to verify our Spring Boot JPA persistence layer with `@DataJpaTest`.

As we use Testcontainers to start our PostgreSQL database server, we use `@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)` to avoid starting an embedded database:

```java
@DataJpaTest
@Testcontainers
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class OrderRepositoryTest {

  @Container
  static PostgreSQLContainer database = new PostgreSQLContainer("postgres:12")
    .withDatabaseName("springboot")
    .withPassword("springboot")
    .withUsername("springboot");

  @DynamicPropertySource
  static void setDatasourceProperties(DynamicPropertyRegistry propertyRegistry) {
    propertyRegistry.add("spring.datasource.url", database::getJdbcUrl);
    propertyRegistry.add("spring.datasource.password", database::getPassword);
    propertyRegistry.add("spring.datasource.username", database::getUsername);
  }

  @Autowired
  private OrderRepository orderRepository;

  @Test
  void shouldReturnOrdersThatContainMacBookPro() {

    orderRepository.save(createOrder("42", """
         [{"name": "MacBook Pro", "amount" : 42}, {"name": "iPhone Pro", "amount" : 42}]
      """));

    orderRepository.save(createOrder("43", """
         [{"name": "Kindle", "amount" : 13}, {"name": "MacBook Pro", "amount" : 10}]
      """));

    orderRepository.save(createOrder("44", "[]"));

    List<Order> orders = orderRepository.findAllContainingMacBookPro();

    assertEquals(2, orders.size());
  }

  private Order createOrder(String trackingNumber, String items) {
    Order order = new Order();
    order.setTrackingNumber(trackingNumber);
    order.setItems(items);
    return order;
  }
}
```

Testcontainers maps the PostgreSQL port to a random ephemeral port, so we can't hardcode the JDBC URL. In the example above, we are using

@DynamicPropertySource to set all required datasource attributes in a dynamic fashion based on the started container.

We can even write a shorter test with less *setup ceremony* using Testcontainers JDBC support feature:

```
@DataJpaTest(properties = {
  "spring.test.database.replace=NONE",
  "spring.datasource.url=jdbc:tc:postgresql:12:///springboot"
})
class OrderRepositoryShortTest {

  @Autowired
  private OrderRepository orderRepository;

  @Test
  @Sql("/scripts/INIT_THREE_ORDERS.sql")
  void shouldReturnOrdersThatContainMacBookPro() {
    List<Order> orders = orderRepository.findAllContainingMacBookPro();
    assertEquals(2, orders.size());
  }
}
```

Note the tc keyword after jdbc for the data source URL. The 12 indicates the PostgreSQL version. With this modification, Testcontainers will start the dockerized PostgreSQL for our test class in the background.

If you want to optimize this setup further when running multiple tests for your persistence layer, you can reuse already started containers with Testcontainers.

### Database Cleanup After the Test

The @DataJpaTest meta-annotation contains the @Transactional annotation. This ensures our test execution is wrapped with a transaction that gets rolled back after the test. The rollback happens for both successful test cases as well as failures.

Hence, we don't have to clean up our tests, and every test starts with empty tables (except we initialize data with our migration scripts). The source code for this @DataJpaTest example is available on GitHub.

## Testing HTTP Clients

Fetching data via HTTP from a remote system is a task almost every application has to solve. Fortunately, there are mature Java HTTP client libraries available that are robust and have a user-friendly API.

Most of the frameworks ship their own HTTP client (e.g Spring with `WebClient` and `RestTemplate`, Jakarta EE with the JAX-RS Client), but there are also standalone clients available: OkHttp, Apache HttpClient, Unirest, etc.

When it comes to testing Java classes that use one of these clients, I often see developers trying to mock the internals of the library. With this section, you'll understand why such an approach is not beneficial and what you should do instead when writing a test that involves a Java HTTP client.

### Why Plain Old Unit Tests Are Not Enough

Most of the Java HTTP clients are instantiated with static methods. In addition, we usually chain multiple method calls or use a builder to construct the instance of the client or the HTTP request. Even though Mockito is able to mock static method calls and with deep mocking, we could keep the stubbing setup lean, we should always remember:

> Every time a mock returns a mock a fairy dies.

We end up in a *mocking hell* and have to mock the whole setup. With this approach, we also almost duplicate our production code as the stubbing setup has to match our usage. While we could test conditionals e.g. behavior on non 200 HTTP responses or how our implementation handles exceptions, there is not much benefit with this approach.

A better solution for testing our Java HTTP clients would be to actually test them *in action* and see how they behave to different responses. This also allows us to test more niche scenarios like slow responses, different HTTP status codes, etc.

### A Better Approach for Testing HTTP Clients

Instead of heavy lifting with Mockito, we'll spawn a local web server and queue HTTP responses. We can then let our HTTP clients connect to this local server and test our code.

Whether or not we are still writing a unit test depends on your definition and scope. However, we are still testing a unit of our application in isolation. Going further, we also have to take a small overhead (time-wise) into account. Most of the time this is negligible. The two most common libraries for this are WireMock and the MockWebServer from OkHttp.

We'll use the lightweight `MockWebServer` for this demo, but you can achieve the same with WireMock. There's already an example on how to use WireMock with JUnit 5 for testing a Spring Boot application on my blog: Spring Boot Integration Tests with WireMock and JUnit 5 We include the `MockWebServer` with the following Maven import:

```
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>mockwebserver</artifactId>
  <version>4.9.0</version>
  <scope>test</scope>
</dependency>
```

Starting the local server and adding mocked HTTP responses is as simple as the following:

```
MockWebServer mockWebServer = new MockWebServer();

mockWebServer.enqueue(new MockResponse()
      .addHeader("Content-Type", "application/json; charset=utf-8")
      .setBody(VALID_RESPONSE)
      .setResponseCode(200));

MyClient client = new MyClient(mockWebServer.url("/").toString());
```

Unlike WireMock, where we stub responses for given URLs, the `MockWebServer` queues every `MockResponse` and returns them in order (FIFO).

When specifying the mocked HTTP response, we can set any HTTP body, status, and header. With the `MockWebServer` we can even simulate slow responses using `.setBodyDelay()`. For our use case, we store a successful JSON response body inside `src/test/resources/stubs` to test the happy-path.

### First Java HTTP Client Test Example: Java's HttpClient

As a first HTTP client example, we're using Java's own `HttpClient`. This client is part of the JDK since Java 11 (in incubator mode since Java 9) and allows HTTP communication without any further dependency. For demonstration purposes, we're requesting a random quote of the day from a public REST API as JSON.

As the `HttpClient` only provides basic converters of the HTTP response body (to e.g. `String` or `byte[]`) we need a library for marshaling JSON payloads. We're going to use Jackson for this purpose.

First, we'll convert the response to a Java `String` and then use Jackson's `ObjectMapper` to map it our `RandomQuoteResponse` Java class for type-safe access of the response. To test different behaviors of our client, later on, we're returning a default quote whenever the HTTP response code is not 200 or an exception is thrown (e.g. `IOException` due to a network timeout).

A simple implementation of our use case can look like the following:

```
public class JavaHttpClient {

  private static final String DEFAULT_QUOTE = "Lorem ipsum dolor sit amet.";
  private final String baseUrl;

  public JavaHttpClient(String baseUrl) {
    this.baseUrl = baseUrl;
  }

  public String getRandomQuote() {
    HttpClient client = HttpClient.newHttpClient();

    HttpRequest request = HttpRequest.newBuilder()
      .uri(URI.create(baseUrl + "/qod"))
      .header("Accept", "application/json")
      .GET()
      .build();

    try {
      HttpResponse<String> httpResponse = client.send(request, BodyHandlers.ofString());

      if (httpResponse.statusCode() != 200) {
        return DEFAULT_QUOTE;
      }

      RandomQuoteResponse responseBody = new ObjectMapper()
        .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
        .readValue(httpResponse.body(), RandomQuoteResponse.class);
      return responseBody.getContents().getQuotes().get(0).getQuote();
    } catch (IOException | InterruptedException e) {
      e.printStackTrace();
      return DEFAULT_QUOTE;
    }
  }
}
```

Creating the `HttpRequest` is straightforward. We only have to provide the HTTP method, the URL, and an HTTP header to request JSON. Here it's important to **not** hard-code the full URL of the remote resource as otherwise testing this client will be difficult.

We're passing the `baseUrl` as part of the public constructor and use it to construct the final URL. For production usage, we can inject this value from an environment variable or a property file (e.g use `@Value` from Spring or `@ConfigProperty` from MicroProfile Config).

As we create the instance of this client on our own when testing, we can then pass the URL of the local `MockWebServer`. The two test scenarios: testing a successful response and a failure (HTTP code 500) are now easy to verify.

```
class JavaHttpClientTest {

  private MockWebServer mockWebServer;
  private JavaHttpClient cut;

  private static String VALID_RESPONSE;

  static {
    try {
      VALID_RESPONSE = new String(requireNonNull(JavaHttpClientTest.class
        .getClassLoader()
        .getResourceAsStream("stubs/random-quote-success.json"))
        .readAllBytes());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  @BeforeEach
  void init() {
    this.mockWebServer = new MockWebServer();
    this.cut = new JavaHttpClient(mockWebServer.url("/").toString());
  }

  @Test
  void shouldReturnDefaultQuoteOnFailure() {
    mockWebServer.enqueue(new MockResponse()
      .addHeader("Content-Type", "application/json; charset=utf-8")
      .setResponseCode(500));

    String result = cut.getRandomQuote();

    assertEquals("Lorem ipsum dolor sit amet.", result);
  }

  @Test
  void shouldReturnRandomQuoteOnSuccessfulResponse() {
    mockWebServer.enqueue(new MockResponse()
      .addHeader("Content-Type", "application/json; charset=utf-8")
      .setBody(VALID_RESPONSE)
      .setResponseCode(200));

    String result = cut.getRandomQuote();

    assertEquals("Vision without action is daydream. Action without vision is nightmare..",\
 result);
```

```
  }
}
```

### Second Java HTTP Client Test Example: OkHttpClient

As a second example, let's take a look at a popular Java HTTP client library: OkHttp. This is a feature-rich open-source HTTP client that I've seen in several projects. After adding it to our project

```
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>okhttp</artifactId>
  <version>4.9.0</version>
</dependency>
```

We can implement the same behavior and fetch a random quote:

```
public class OkHttpClientExample {

  private static final String DEFAULT_QUOTE = "Lorem ipsum dolor sit amet.";
  private final String baseUrl;

  public OkHttpClientExample(String baseUrl) {
    this.baseUrl = baseUrl;
  }

  public String getRandomQuote() {

    OkHttpClient client = new OkHttpClient();

    Request request = new Request.Builder()
      .url(baseUrl + "/qod")
      .addHeader("Accept", "application/json")
      .build();

    try (Response response = client.newCall(request).execute()) {

      if (response.code() != 200) {
        return DEFAULT_QUOTE;
      }

      RandomQuoteResponse responseBody = new ObjectMapper()
        .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
        .readValue(response.body().string(), RandomQuoteResponse.class);

      return responseBody.getContents().getQuotes().get(0).getQuote();
    } catch (IOException e) {
      e.printStackTrace();
      return DEFAULT_QUOTE;
    }
  }
}
```

The code looks almost similar to the example in the last section. First, we instantiate a client and then create a `Request` that we pass to the client as the last step. There's also no big difference in the test for this `OkHttpClient` usage:

```
class OkHttpClientExampleTest {

  private MockWebServer mockWebServer;
  private OkHttpClientExample cut;

  private static String VALID_RESPONSE;

  static {
    try {
      VALID_RESPONSE = new String(requireNonNull(ApacheHttpClientTest.class
        .getClassLoader()
        .getResourceAsStream("stubs/random-quote-success.json"))
        .readAllBytes());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  @BeforeEach
  void init() {
    this.mockWebServer = new MockWebServer();
    this.cut = new OkHttpClientExample(mockWebServer.url("/").toString());
  }

  @Test
  void shouldReturnDefaultQuoteOnFailure() {
    mockWebServer.enqueue(new MockResponse()
      .addHeader("Content-Type", "application/json; charset=utf-8")
      .setResponseCode(500));

    String result = cut.getRandomQuote();

    assertEquals("Lorem ipsum dolor sit amet.", result);
  }

  @Test
  void shouldReturnRandomQuoteOnSuccessfulResponse() {
    mockWebServer.enqueue(new MockResponse()
      .addHeader("Content-Type", "application/json; charset=utf-8")
      .setBody(VALID_RESPONSE)
      .setResponseCode(200));

    String result = cut.getRandomQuote();

    assertEquals("Vision without action is daydream. Action without vision is nightmare..",\
 result);
  }
}
```

Again, we're creating the class under test (short `cut` ) on our own and pass the URL of the `MockWebServer` to it.

### Third Java HTTP Client Test Example: Apache HttpClient

To complete the demonstration of popular Java HTTP clients, let's use the Apache HttpClient as the last example.

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.5.13</version>
</dependency>
```

The implementation for fetching a random quote is quite similar to the first example when we used Java's own `HttpClient`:

```
public class ApacheHttpClient {

  private static final String DEFAULT_QUOTE = "Lorem ipsum dolor sit amet.";
  private final String baseUrl;

  public ApacheHttpClient(String baseUrl) {
    this.baseUrl = baseUrl;
  }

  public String getRandomQuote() {

    HttpClient client = HttpClientBuilder.create().build();

    try {
      HttpUriRequest request = RequestBuilder.get()
        .setUri(this.baseUrl + "/qod")
        .setHeader(HttpHeaders.ACCEPT, "application/json")
        .build();

      HttpResponse httpResponse = client.execute(request);

      if (httpResponse.getStatusLine().getStatusCode() != 200) {
        return DEFAULT_QUOTE;
      }

      RandomQuoteResponse responseBody = new ObjectMapper()
        .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
        .readValue(EntityUtils.toString(httpResponse.getEntity(), "UTF-8"), RandomQuoteResp\
onse.class);

      return responseBody.getContents().getQuotes().get(0).getQuote();
    } catch (IOException e) {
      e.printStackTrace();
      return DEFAULT_QUOTE;
    }
  }
}
```

The test code is exactly the same, the only difference is that we instantiate a `ApacheHttpClient` as our class under test:

```
@BeforeEach
void init() {
  this.mockWebServer = new MockWebServer();
  this.cut = new ApacheHttpClient(mockWebServer.url("/").toString());
}
```

#### What About Other Java HTTP Clients?

But what about my fancy HTTP client library? How can I test it? I assume there are even more Java HTTP client libraries available than logging libraries…

With the client example above, we saw that there is actually no big difference when it comes to testing code that uses them. As long as we are able to configure (at least) the base URL, we can test any client with this recipe.

So it doesn’t matter whether we use the JAX-RS Client, the Spring WebFlux `WebClient` , the Spring `RestTemplate` or your `MyFancyHttpClient`, as this technique is applicable to all of them.

For both the WebClient and RestTemplate you'll find dedicated articles on my blog. If you are curious and want to know how to achieve the same with `WireMock`, take a look at this Spring Boot integration test example using WireMock and JUnit 5. The source code for testing these different Java HTTP clients is available on GitHub.

## Writing End-to-End Tests

Good old web tests - extremely valuable, sometimes hard to maintain, and annoying once they get flaky. If you are familiar with Selenium, you might find your self-writing helper functions to e.g., wait on elements to be present in the DOM, or AJAX calls to finish.

As Selenium is more or less a low-level API to manage the browser, this usually leads to boilerplate code for your projects. The Selenide project eliminates the shortcoming of the low-level API nature of Selenium. With Selenide, you can write concise, stable, and short web tests for your Java projects. This section showcases Selenide 6 with Selenium 4 and Testcontainers using a Spring Boot Java 11 application with a Thymeleaf frontend.

### Java Project Setup for Selenide

For a Maven-based project, we have to include the following dependency:

```
<dependency>
  <groupId>com.codeborne</groupId>
  <artifactId>selenide</artifactId>
  <version>6.3.3</version>
  <scope>test</scope>
</dependency>
```

That's all we need. The Selenide dependency already includes Selenium's selenium-java dependency, so we don't have to include any Selenium-related dependency on our own.

All specific browser APIs are part of this library, and we can start writing automated web tests for Chrome, Edge, Firefox, Opera, Safari, etc. In case Spring Boot's dependency management selects an incompatible Selenium version, we can override the Selenium version with a property:

```
<properties>
  <selenium.version>4.1.2</selenium.version>
</properties>
```

Apart from this, Selenide also ships with the WebDriverManager. This is a useful utility library to automate the management of browser drivers.

As we always need the driver binary of our target browser (e.g. `chromedriver`) installed while executing the web tests, manually downloading them, and managing their versions is cumbersome.

The WebDriverManager project takes over this task for us. Prior to executing a web test, the WebDriverManager ensures to download the required driver binary or use an already existing one. Hence we don't have to fiddle around with setting any parameters like `-Dwebdriver.chrome.driver`.

At the end of this section post, we'll also see how we can utilize the Web-Driver module of Testcontainers so that the driver is running in isolation as part of a Docker container.

### Accessing Browser Elements with Selenide

The first step of every web test is to open the browser for a specific URL. With Selenide, that's a one-liner:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class BookStoreWT {

  @LocalServerPort
  private Integer port;

  @Test
  void shouldDisplayBooks() {
    open("http://localhost:" + port + "/book-store");
  }
}
```

For demonstration purposes, let's assume we'll use a Spring Boot application that exposes one Thymleaf view and a REST API endpoint. The view displays a table of books whenever someone clicks a button to fetch them via AJAX. Once the browser window opens our application, we can access HTML elements.

Selenide provides two functions to access any `SelenideEelement` (a wrapper around Selenium's `WebElement`): `$` and `$$` (good old jQuery memories). Using `$`, we'll get the first element that matches your search, whereas with `$$` we get a list of all the elements that match.

Selenide provides different mechanisms (similar to Selenium) to locate elements by …

- `id`
- `tagName` (e.g. `h1`)
- `xpath`
- `className`
- `cssSelector`
- etc.

In our example, we first want to make sure the book table does not exist inside the DOM. Next, we want to click the button to fetch data via the book REST API. After that, we can verify that our book table is rendered. With Selenide, that's a three-liner:

```
$(By.id("all-books")).shouldNot(Condition.exist);
$(By.id("fetch-books")).click();
$(By.id("all-books")).shouldBe(Condition.visible);
```

As soon as we have access to the `SelenideElement`, we can perform actions (e.g. click or fill inputs) or express expectations using a readable and fluent API. In addition to this, we can make sure that our page only contains one `h1` element:

```
$(By.tagName("h1")).shouldHave(CollectionCondition.size(1));
```

#### Make Screenshots During the Web Test Execution

Most of the time, the execution of our web test is automated on a CI server (e.g., Jenkins or Gitlab CI). This makes it hard to actually *see* the automated test and understand the reason whenever the test fails. Fortunately, Selenide creates a screenshot and captures the HTML for every failing test out-of-the-box.

What's left to adjust is the location where Selenide stores these files. By default, it's `build/reports`. As a Maven fanboy, I want the location to be inside the `target` folder. For those of you that use JUnit Jupiter, you can override this by registering the `ScreenShooterExtension`:

```
@RegisterExtension
static ScreenShooterExtension extension = new ScreenShooterExtension()
  .to("target/selenide");
```

We can also configure this using the `Configuration` class of Selenide:

```
Configuration.reportsFolder = "target/selenide";
```

In addition to the screenshots on failure, we can also manually take screenshots throughout our test execution:

```
$(By.id("all-books")).shouldNot(Condition.exist);

screenshot("pre_book_fetch");

$(By.id("fetch-books")).click();
```

### Use Testcontainers to Containerize the WebDriver

If you are familiar with Tescontainers, you might wonder if we can utilize its WebDriver module for the web tests with Selenide. Good news: Yes, we can use Testcontainers to write web tests with Selenide for our Java projects! For this to work, we need one additional dependency:

```
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>selenium</artifactId>
  <version>1.18.3</version>
  <scope>test</scope>
</dependency>
```

Even though Selenide ships with the WebDriverManager, we don't have to create the `WebDriver` with it. We can instruct Selenide to use a `WebDriver`that we created with Testcontainers:

```
RemoteWebDriver remoteWebDriver = webDriverContainer.getWebDriver();
WebDriverRunner.setWebDriver(remoteWebDriver);
```

The full example for our existing Java web test with Selenide but now using Testcontainers to provide the `WebDriver` looks like the following:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class BookStoreTestcontainersWT {

  public static BrowserWebDriverContainer<?> webDriverContainer =
    new BrowserWebDriverContainer<>()
      .withCapabilities(new ChromeOptions()
        .addArguments("--no-sandbox")
        .addArguments("--disable-dev-shm-usage"));

  @RegisterExtension
  static ScreenShooterExtension screenShooterExtension =
    new ScreenShooterExtension().to("target/selenide");

  @LocalServerPort
  private Integer port;

  @BeforeAll
  static void beforeAll(@Autowired Environment environment) {
    Testcontainers.exposeHostPorts(environment.getProperty("local.server.port", Integer.cla\
ss));
    webDriverContainer.start();
  }

  @Test
  void shouldDisplayBook() {

    Configuration.timeout = 2000;
    Configuration.baseUrl = String.format("http://host.testcontainers.internal:%d", port);

    RemoteWebDriver remoteWebDriver = webDriverContainer.getWebDriver();
    WebDriverRunner.setWebDriver(remoteWebDriver);

    open("/book-store");

    $(By.id("all-books")).shouldNot(Condition.exist);
    $(By.id("fetch-books")).click();
    $(By.id("all-books")).shouldBe(Condition.visible);
  }
}
```

Keep in mind that the driver is now running in an isolated Docker container, and we can't use `localhost` to refer to our application. That's why we expose the host port we want to access from within the `BrowserWebDriverContainer`.

Exposing the host port with Testcontainers has to happen after we start our Tomcat server but before we start the Docker container.

The source code for the demo application and this introduction to Selenide is available on GitHub.

# Outro

## Next Steps

Congratulations, you have completed “Testing Spring Boot Applications Demystified”! We hope this ebook has provided you with valuable insights, useful tips, and actionable strategies to confidently test your Spring Boot applications.

Remember, testing is not just about catching bugs – it’s also about building robust, high-quality applications that meet the needs of your users and stakeholders. By following the best practices and techniques outlined in this ebook, you can take your testing skills to the next level and deliver software that exceeds expectations.

Now that you have the knowledge and tools needed to write comprehensive and effective tests for your Spring Boot applications, it’s time to put them into practice. Don’t be afraid to experiment, try new things, and refine your testing approach over time. With each iteration, you will gain confidence, hone your skills, and unlock the full potential of your software.

We encourage you to continue learning, exploring, and growing as a tester. The world of testing is vast and constantly evolving, and there’s always more to discover. Stay up-to-date with the latest tools, techniques, and industry trends, and never stop pursuing excellence in your craft.

Thank you for reading “Testing Spring Boot Applications Demystified”. We wish you all the best on your testing journey, and may your applications be bug-free, reliable, and a joy to use!

## Further Resources

If you want to learn more about testing Spring Boot applications, we recommend the following resources:

- Curated list of Recommended Resources on Testing Java Applications
- Java Testing Toolbox eBook: 30 Testing Tools & Libraries Every Java Developer Must Know
- Conference talk: Things I Wish I Knew When I Started Testing Spring Boot Applications
- Conference talk: How fixing a broken window cut down our build time by 50%
- From Zero to Production with Spring Boot and AWS: Stratospheric
- Hands-on and beginner-friendly testing course: Testing Spring Boot Applications Primer
- TDD with Spring Boot Done Right
- Deep-dive, advanced, and comprehensive testing course: Testing Spring Boot Applications Masterclass

# Changelog

- 1.0 (2023-07-04): Initial Release