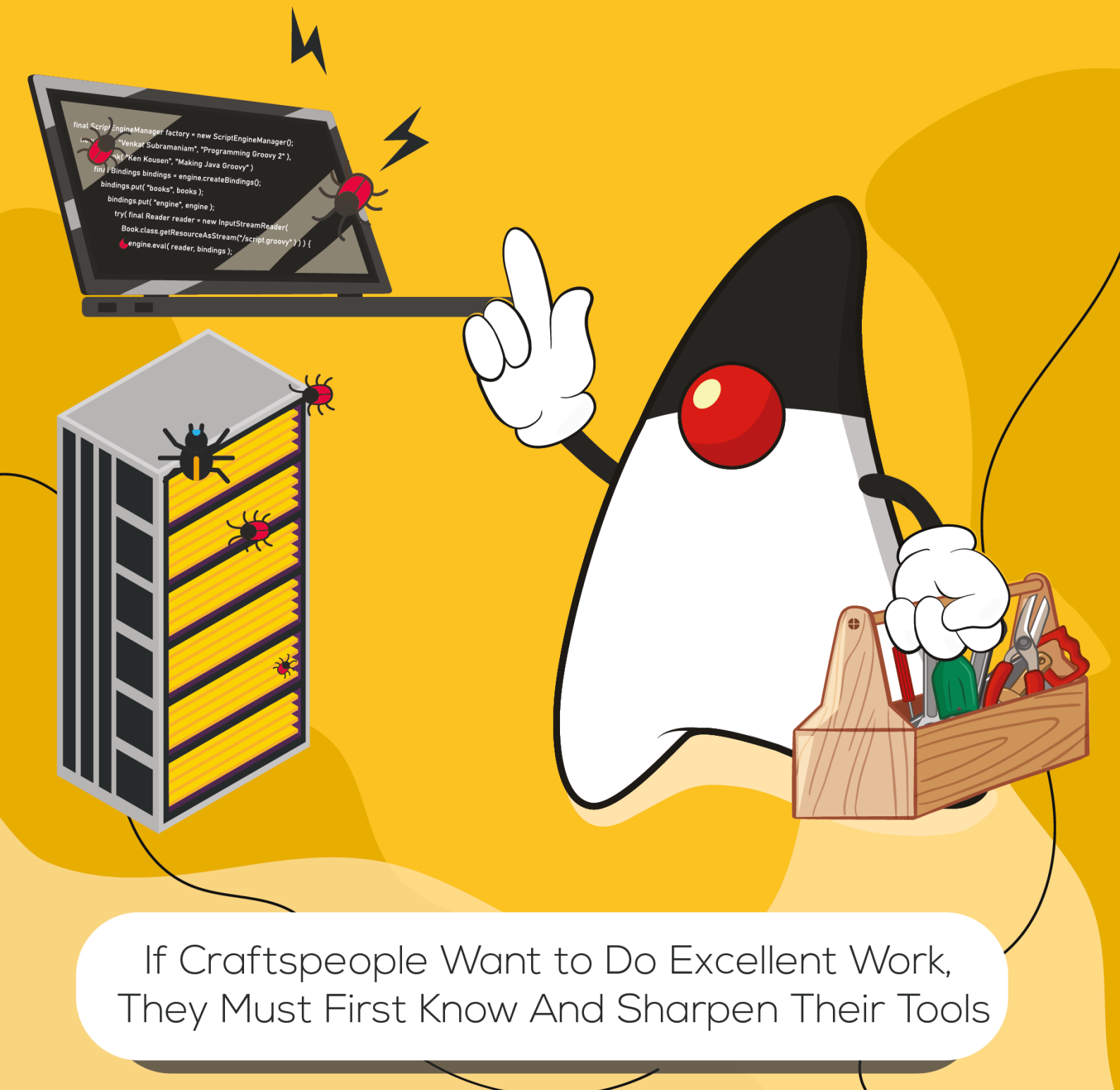
30 TESTING TOOLS & LIBRARIES

Every Java Developer Must Know



If Craftspeople Want to Do Excellent Work, They Must First Know And Sharpen Their Tools

Testing Tools and Libraries Every Java Developer Must Know

Philip Riecks (rieckpil)
Version: 0.1

Table of Contents

Introduction

The Java testing ecosystem is huge. Over the years, a lot of libraries have emerged. Among them the two most known libraries: JUnit & Mockito. When you write tests for any Java application for the first time, they're usually the libraries you fight first.

If you're lucky, you already get in contact with both libraries early in your career. At least that was true for me. In university, we had one (!) class touching the testing topic on the surface. Unfortunately, it wasn't a cozy and beginner-friendly "Testing Java Applications 101" lecture. It was rather *brute force*. Our motivation to write tests was utterly different. To pass the class, the software we submitted had to have tests. Otherwise, we would've failed the assignment. The quality of those tests? It didn't matter at all...

Your first introduction to testing Java applications might have been entirely different. There are a lot of developers that write their first tests only right after their first PR gets rejected because they are missing. Nevertheless, how your Java testing journey started, I'm sure JUnit & Mockito were the first libraries you met. Both tools accompany you from day one throughout your entire journey.

But there's more to the Java testing ecosystem than JUnit and Mockito!

Compared to Java's logging ecosystem, Java's testing landscape is clearly arranged. Most of the libraries fulfill a specific purpose and don't reinvent the wheel. There are, for sure, libraries that are interchangeable with each other. The variety of tools (unlike logging) makes various testing styles and techniques possible.

Furthermore, Java's testing ecosystem is mature. You won't find a wild and sprouting testing ecosystem where new libraries pop up every day. However, many Java testing libraries & tools are not getting as much attention as they should and are not on the radar of every developer.

This time is over now!

With this book, you'll get a detailed overview of the Java testing ecosystem and learn about tools & libraries that you must know from the following categories:

- Test Frameworks
- Assertion Libraries
- Mocking Frameworks
- Test Infrastructure Libraries
- Performance Testing Libraries and Tools
- Utility Libraries and Tools

Please note that there's a subtle but important difference between *must know* and *must use*. I'm not advocating including *all* of the upcoming libraries for every project. That would be too much. Instead, the goal is to raise awareness for different tools & libraries. Next time you face problem X, you know that there's library Y and Z out there.

Preface

Who Should Read This Book?

That's easy to answer: Every Java developer who cares about testing and wants to use the right tools for the job. Hopefully, this should be the majority of developers.

Like it or not, the craftsmanship analogy applies to software development. How do craftspeople know they're using the right tools for the job? If all they have is a hammer, everything looks like a nail.

Craftspeople first need to know what's part of their toolbox, sharpen these tools, and then (and only then) can they effectively solve the problem at hand.

To bring this analogy back to this book's topic, as Java developers we need to know what the Java testing ecosystem provides before we can effectively write our tests. Otherwise, we end up spending too much time with the wrong tools in our hand and only use a *hammer*.

Especially for niche testing topics (e.g., asynchronous code), there's a high chance that your home-brewed test framework is reinventing the wheel. In that case, you might be better off (and faster) with an off-the-shelf solution. Lucky you that you now picked up this book.

As a reader of this book, you only have to fulfill two requirements:

- you're able to read and understand Java code
- you're interested in improving your testing toolbox.

If you're working with a different JVM language like Kotlin or Scala, this book will still be beneficial. Most of the tools & libraries either work out-of-the-box with other JVM languages or provide additional modules for the integration.

All source code examples, however, are using Java 11. You can find them on GitHub. For every testing library & tool, you'll find a corresponding package inside `src/test/resources`. The only exception is command-line or other desktop tools that don't require any custom Java code to showcase them.

For some testing libraries, we need a running application (e.g., when testing a REST API). Therefore, the book works with a sample Spring Boot application. However, all code examples are as application framework vendor-agnostic as possible.

It's equally important to lay out what this book is not about. We won't cover topics like TDD (Test-Driven-Development) or specific tips & tricks for testing in general. These topics are alreay covered by excellent books like:

- Growing Object-Oriented Software, Guided by Tests from Steve Freeman and Nat Pryce
- Unit Testing: Principles, Practices, and Patterns from Vladimir Khorikov
- ... or the timeless Test Driven Development: By Example book from Kent Beck

Structure of This Book

This book presents each upcoming testing tool and library in the same cookbook-like style:

- introduction & fact sheet
- setup for Maven & Gradle
- most common usages (following the Pareto principle to demonstrate 20% of the features you'll use 80% of the time)

With this unified structure, you'll get familiar with each tool in a matter of minutes. In case you decide to include one of the presented tools to your project, you'll already have some early quick-wins and can use the source code on GitHub as a reference.

Furthermore, each upcoming chapter answers the following questions for each testing tool and library:

- What's the main API of the library?
- How can this tool or library help me?
- What are common pitfalls when working with it?

The central goal of this book is to enrich your testing toolbox with new tools & libraries that you might not have even heard about (yet). Furthermore, you'll also learn about new features of testing tools that you are already using.

This book aims to provide you a first overview of the library. At several places, you'll find links that point to further resources (books, videos, articles) that explain the technology in greater detail.

One final note about the number of discussed libraries and tools in this book. This number is not set. The total number of technologies that this book introduces is steadily increasing.

So the number of tools you find on the cover is just an intermediate status. As this book evolves, X in "X Testing Tools & Libraries Every Java Developer Must Know" will change and increase with each new release.

Conventions Used In This Book

For most of the code examples, I'm favoring non-static imports:

```
Assertions.assertThat("duke").contains("d");
```

While this adds additional boilerplate to the examples, it makes them unambiguous. Especially Java newcomers might easily mix up the imports. This convention helps to avoid headaches while using the wrong Java classes.

When including any of the presented examples for your projects, I recommend using a static import to save some keystrokes:

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

assertThat("duke").contains("d");
```

Furthermore, you'll find the abbreviation `cut` (Class Under Test) multiple times. This variable refers to the particular class that we're verifying with the test. As our test classes usually contain numerous fields and local variables, we might get easily confused about the actual test subject. With this convention, it's crystal clear. Other authors sometimes use the term `sut` (Subject Under Test) for this purpose.

The test examples follow the given/when/then (or arrange/act/assert) setup. A new line separates each part:

```
@Test
void shouldPropagateException() {
  // given
  Mockito.when(userRepository.findByUsername("devil"))
    .thenThrow(new RuntimeException("DEVIL'S SQL EXCEPTION"));

  // when
  assertThrows(RuntimeException.class, () -> cut.registerUser("devil"));

  // then
  Mockito.verify(userRepository, never()).save(ArgumentMatchers.any(User.class));
  Mockito.verify(userRepository, times(1)).findByUsername("devil");
}
```

All source code examples are available on GitHub.

Acknowledgments

I'm genuinely impressed and grateful for the many open-source developers who dedicate some of their spare time to make testing Java applications more enjoyable. It's great to see that most testing libraries are actively maintained over so many years.

What a great time to be a Java developer!

Thanks to both Wim Deblauwe and Lorenzo Bettini, who volunteered as technical reviewers. Their feedback was inevitably to polish several chapters of this book.

A special "Thank You" goes to my girlfriend, Jessi. She helped me improve the writing style and supported me with brain food (delicious porridge) on those early mornings where I worked on this book.

About The Technical Reviewers

Wim Deblauwe

Wim Deblauwe is a freelance Java Developer with over 20 years of experience. He loves an expressive test suite that makes him sleep well at night.

He is the author of the testcontainers-cypress library, and wrote a book about Spring Boot and Thymeleaf.

Lorenzo Bettini

Lorenzo Bettini is an Associate Professor in Computer Science at DISIA Dipartimento di Statistica, Informatica, Applicazioni 'Giuseppe Parenti', Università di Firenze, Italy.

His research interests cover design, theory, and implementation of programming languages (in particular statically typed Object-Oriented languages, Network aware languages, and Modeling languages), with IDE support.

He is the author of more than 90 papers, published in international conferences and international journals. He is also the author of the two editions of the book "Implementing Domain-Specific Languages with Xtext and Xtend" (Packt Publishing), and of the book "Test-Driven Development, Build Automation, Continuous Integration (with Java, Eclipse and friends)" (Leanpub). You can find more about Lorenzo on his homepage.

About The Author

Philip is a self-employed IT-Consultant living in Berlin. He started working with Java back in 2015 and has used it for multiple applications in several industries. Testing is an integral part of his daily work as he is a profound craftsman.

Once Philip understood the ins and outs of the Java testing landscape, writing tests makes as much fun as writing production code for him. He's also a won-around TDD (Test Driven Development) practitioner and regularly shares testing techniques with his colleagues and clients.

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).

He started teaching Java topics on YouTube in 2018 and is writing content about the Java ecosystem on his blog since 2017.

More than 1.000 course students have enrolled for his online courses. Besides that, he's also actively helping developers on Stack Overflow with their questions around testing Java applications.

His biggest accomplishment so far is the Testing Spring Boot Applications Masterclass. In this 9h+ long online course, he teaches the ins and outs of testing real-world Spring Boot applications.

Apart from this, he's a co-author of Stratospheric - From Zero to Hero With Spring Boot AWS.

You can get in contact with Philip on various platforms:

- Twitter
- LinkedIn
- GitHub
- Facebook

That's enough for the preface, let's get started!

Test Infrastructure

GreenMail

Sending emails is a common responsibility of enterprise applications. When testing such features with integration tests, we don't want to leak any email to our users by (accidentally) using a real email server. A better approach is to use a sandbox email server to capture and verify our integration tests' outcome. That's exactly what GreenMail offers:

> GreenMail is the first and only library that offers a test framework for both receiving and sending emails from Java.
>
> — GreenMail, official documentation

GreenMail supports SMTP, POP3 and IMAP including their SSL variants and is easy to set up.

Fact Sheet

- Purpose: Sandbox email server for testing and development purposes
- Alternatives: Non-Java solutions (e.g., a local containerized email server)
- Homepage: https://greenmail-mail-test.github.io/greenmail/
- User Guide: https://greenmail-mail-test.github.io/greenmail/#about
- GitHub: https://github.com/greenmail-mail-test/greenmail

Setup

Maven coordinates:

```
<dependency>
  <groupId>com.icegreen</groupId>
  <artifactId>greenmail</artifactId>
  <version>1.6.0</version>
  <scope>test</scope>
</dependency>
```

Gradle coordinates:

```
testImplementation('com.icegreen:greenmail:1.6.0')
```

The upcoming examples make use of GreenMail's JUnit Jupiter extension, which is part of an additional module:

```
<dependency>
  <groupId>com.icegreen</groupId>
  <artifactId>greenmail-junit5</artifactId>
  <version>1.6.0</version>
  <scope>test</scope>
</dependency>
```

Furthermore, GreenMail also provides first-class support for JUnit 4 and Spring.

Basic Usage

For demonstration purposes, we'll use the JavaMail API without any additional framework abstraction (e.g., Spring's `JavaMailSender`) to be as independent with the example as possible. The `MailService` class defines two public methods to both send emails via SMTP and receive messages via IMAP for a given user:

```
public class MailService {

  private final Session session;

  // constructor and javax.Mail.Session creation

  public void sendToUser(String message, String recipient) throws MessagingException {
    Message mail = new MimeMessage(session);
    mail.addHeader("Content-type", "text/plain; charset=UTF-8");
    mail.setFrom(new InternetAddress("hello@duke.io"));
    mail.addRecipient(Message.RecipientType.TO, new InternetAddress(recipient));
    mail.setSubject("A new message for you");
    mail.setText(message);

    Transport.send(mail);
  }

  public String retrieveLatestMailForUser(String recipient, String recipientPassword) throws Exception {
    try (Store store = session.getStore("imap")) {
      store.connect(recipient, recipientPassword);

      Folder inbox = store.getFolder("INBOX");
      inbox.open(Folder.READ_ONLY);

      Message[] messages = inbox.getMessages();
      String latestMessageContent = "NO NEW MESSAGE";

      if (messages.length > 0) {
        latestMessageContent = messages[0].getContent().toString();
      }

      return latestMessageContent;
    }
  }
}
```

Let's take a look at how GreenMail can help us to test these two methods.

Test Email Sending Components

As a first step, we have to start and configure the local GreenMail server. We'll make use of the `GreenMailExtension` as this conveniently integrates with our JUnit Jupiter test setup. We can define which email protocols (e.g. SMTP, POP3, etc.) the sandbox email server should support during the setup.

By default, GreenMail adds an offset to the default port of the protocol. Without any further configuration, the SMTP server will be available at port 3025 (default port 25 plus offset of 3000).

For our example, we need support for both SMTP and IMAP to verify our class under test (`MailService`). On top of the protocol configuration, we can define an admin user that our application will use when connecting to the server:

```
class MailServiceTest {

  @RegisterExtension
  static GreenMailExtension greenMail = new GreenMailExtension(ServerSetupTest.SMTP_IMAP);

  private MailService cut;

  @BeforeEach
  public void setup() {

    greenMail.setUser("admin@java.io", "my_secret");

    this.cut = new MailService(
      greenMail.getSmtp().getBindTo(),
      greenMail.getSmtp().getPort(),
      greenMail.getImap().getPort(),
      "admin@java.io",
      "my_secret");
  }
}
```

By default, the `GreenMailExtension` will start and stop the local email server per test method. For test scenarios where we want to start the local email server per test class, we can configure this as part of the extension:

```
@RegisterExtension
static GreenMailExtension greenMail = new GreenMailExtension(ServerSetupTest.SMTP_IMAP)
  .withPerMethodLifecycle(false);
```

However, starting and stopping the GreenMail server is fast enough to work with the default. With a per test method server setup, we also don't have to care about any cleanup after a test.

With the GreenMail server up and running, we can write the first test to verify the email sending part of our `MailService`:

```
@Test
void shouldSendMailToRecipient() throws Exception {
  cut.sendToUser("Hello from Test!", "mike@java.io");

  MimeMessage receivedMessage = greenMail.getReceivedMessages()[0];
  assertEquals("Hello from Test!", GreenMailUtil.getBody(receivedMessage));
  assertEquals(1, receivedMessage.getAllRecipients().length);
  assertEquals("mike@java.io", receivedMessage.getAllRecipients()[0].toString());
}
```

Right after we send the email from our application, we request a list of all emails that the GreenMail server received. That's simple, isn't it?

For failing test cases and for debugging purposes, we can enable GreenMail's verbose

mode with:

```
@RegisterExtension
static GreenMailExtension greenMail =
  new GreenMailExtension(ServerSetup.verbose(ServerSetupTest.SMTP_IMAP));
```

The same test will now output the following debug information:

```
220 /127.0.0.1 GreenMail SMTP Service v1.6.0 ready
DEBUG SMTP: connected to host "127.0.0.1", port: 3025
EHLO 127.0.0.1
250-/127.0.0.1
250 AUTH PLAIN LOGIN
DEBUG SMTP: Found extension "AUTH", arg "PLAIN LOGIN"
DEBUG SMTP: use8bit false
MAIL FROM:<test@java.io>
250 OK
RCPT TO:<mike@java.io>
```

Test Email Receiving Components

With the next test, we want to verify the `retrieveLatestMailForUser` method of our `MailService`. As this method returns the content of the newest email for a given user, we first need to populate the user's inbox.

We *could* use our `sendToUser` method for this purpose, but this would violate our unit testing strategy as we only want to test one method per test.

As an alternative and better approach, we're creating a `javax.mail.Session` right from the GreenMail server and send a test email to the user's inbox:

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

  Session smtpSession = greenMail.getSmtp().createSession();

  Message msg = new MimeMessage(smtpSession);
  msg.setFrom(new InternetAddress("test@java.io"));
  msg.addRecipient(Message.RecipientType.TO,
    new InternetAddress("mike@java.io"));
  msg.setSubject("Test");
  msg.setText("Hello World from GreenMail!");
  Transport.send(msg);

  greenMail.setUser("mike@java.io", "secret_mike");

  String result = cut.retrieveLatestMailForUser("mike@java.io", "secret_mike");

  assertEquals("Hello World from GreenMail!", result);
}
```

Use GreenMail During Development

Apart from testing purposes, we can also use GreenMail during development. In case we're working on an email-related feature and don't want to interact with a real server, we can use GreenMail sandbox capabilities.

GreenMail offers several deployment options to get the sandbox email server up and

running.

The first one is a standalone `.jar` file approach. Once we download the `.jar` from Maven Central, we can start the local email server as a regular Java program:

```
java [OPTIONS] -jar greenmail-standalone-1.6.0.jar
```

We can pass additional JVM parameter to tweak the server configuration (e.g., which protocols to support, admin users, etc.):

```
java -Dgreenmail.setup.test.all \
  -Dgreenmail.users=duke:pwd,mike:secret \
  -jar greenmail-standalone-1.6.0.jar
```

Another deployment mechanism is to start GreenMail as a Docker container:

```
docker run -t -i \
  -p 3025:3025 -p 3110:3110 -p 3143:3143 \
  -p 3465:3465 -p 3993:3993 -p 3995:3995 \
  greenmail/standalone:1.6.0
```

We can pass additional JVM arguments as part of the `GREENMAIL_OPTS` environment variable:

```
docker run -t -i \
  -e GREENMAIL_OPTS='-Dgreenmail.setup.test.all -Dgreenmail.hostname=0.0.0.0 -Dgreenmail.users=duke:pwd,mike:secret' \
  -p 3025:3025 -p 3110:3110 -p 3143:3143 \
  -p 3465:3465 -p 3993:3993 -p 3995:3995 \
  greenmail/standalone:1.6.0
```

As the last approach, GreenMail offers a webapp (`.war` file). We can deploy this app to any Java EE 7 compliant application server.

Postface

What's Next?

> Knowledge without practice is useless. Practice without knowledge is dangerous.
>
> — Confucius

You're now equipped with an extensive Testing Toolbox.

Now get your hands dirty and include those libraries and tools where you see fit. Give each tool an unbiased try. If you're not happy with the outcome, consider one of the presented alternatives.

But don't rush and include every library or tool you've just read about. That's cargo cult.

The next time your searching for a niche testing library to solve X, come back to this book to see what the Java testing ecosystem offers.

For additional testing-related materials, take a look at the following resources:

- 14 Days Free Testing Email Course
- Testing Spring Boot Applications Masterclass
- JUnit 5 & Mockito Cheat Sheet
- Hands-On Testing Videos on YouTube

Joyful testing,

Philip

Feedback, Errata, Questions

I highly appreciated any feedback for this book!

The best way to submit code/spelling mistakes is via a GitHub issue.

In case you have a suggestion for a testing library or tool that's currently missing, I'm more than happy to include it. You can find the list of upcoming testing tools here.

You can also drop me a message at blog@rieckpil.de or DM me on Twitter (@rieckpil).

Changelog

Notes about the changes in each revision of this book:

- Revision 0.1 (2021-04-11) | Release of the first version including fifteen tools & libraries: JUnit 4, JUnit 5, TestNG, Hamcrest, AssertJ, JsonPath, JSONAssert, XMLUnit, Mockito, MockWebServer, REST Assured, WireMock, Testcontainers, Selenide, GreenMail.