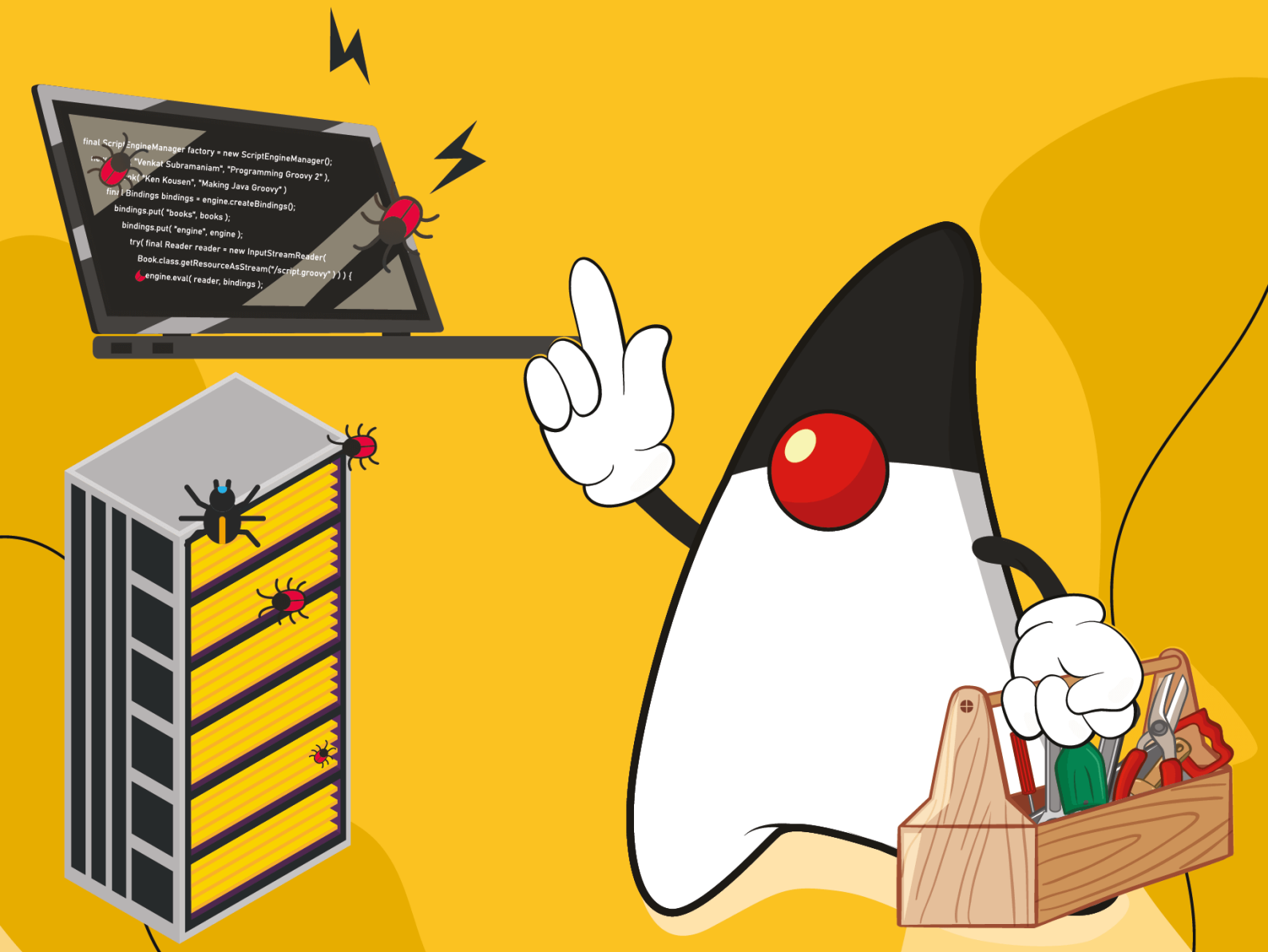
30 TESTING TOOLS & LIBRARIES

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: 0.2

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 pull request 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.

The number of showcased libraries and tools in this book is not final. This number is steadily increasing with each book release.

As my testing skills evolve over time and as I'm learning about new testing tools & libraries, the X in "X Testing Tools & Libraries Every Java Developer Must Know" is an intermediate and increasing counter.

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 (we all have been there).

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. We're going to use a Spring Boot and a Jakarta EE application to demonstrate the different tools and libraries. The code examples are located in a dedicated package named after the tool within `src/test/java` (Spring Boot & Jakarta EE).

If not stated differently, the tests are part of the `spring-boot-example` application by default.

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.

Contributors

Pradeep Murugesan

Pradeep Murugesan is a software engineer at Wise, dad of a 2-year-old, and cricket fan who lives currently in London, UK. Uses Java on a daily basis for his work, also contributes to open source projects during his free time and is now trying his hands on technical writing / blogging. You can say hi to him on Twitter @pradeepm_14 and on Medium @pradeep.murugesan.

If you also want to be part of this list and contribute a chapter to this eBook and showcase your most favorite Java testing tool or library, drop me a message at blog@rieckpil.de. The same applies to library maintainers or companies that want to gain more attention for their testing tool/product.

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!

Performance Testing

Please note that we're running the upcoming performance tests against a local backend for demonstration purposes. As a local setup usually never reflects the setup and configuration for production, we should be cautious when analyzing the outcome.

We should run our tests and analyze our application's performance either on production or with a close-to-production setup for the most reliable test results.

Performance for JVM applications and how to tune for performance is a huge topic for itself. For an in-depth introduction to performance, consider reading Java Performance: In-depth Advice for Tuning and Programming Java 8, 11, and Beyond from Scott Oaks.

Gatling

Gatling is a performance testing tool to carry out load tests on applications. Gatling can spawn thousands of virtual users/clients over a single machine as it is built on top of Akka and treats virtual users as messages and not threads. Thus having a lower footprint compared to other performance tools that use JVM threads.

As of version 3.7, Gatling now offers a Java and Kotlin DSL in addition to the existing Scala DSL. This makes adding the performance tests for an existing Java application even more straightforward.

Before version 3.7, Scala was the main and only language to write performance tests with Gatling. While the required Scala knowledge to understand and adjust the tests can be learned quickly, especially with a JVM background, adding Scala tests to a Java-based projected required additional plugins and configuration (aka. complexity).

Gatling comes with an open-source as well as an enterprise solution.

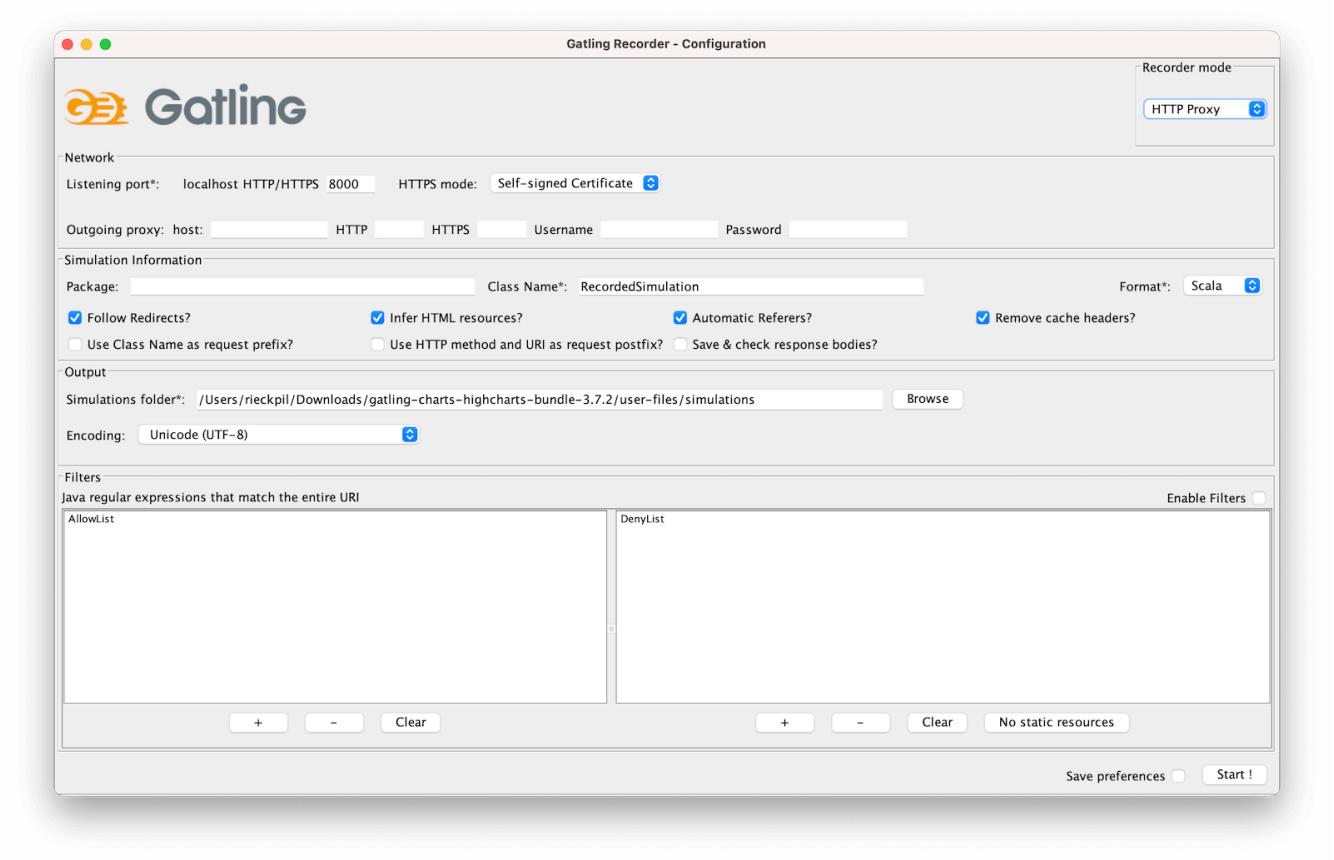
Fact Sheet

- Purpose: Load testing applications
- Alternatives: JMeter, LoadRunner
- Homepage: https://gatling.io/
- User Guide: https://gatling.io/docs/gatling/guides/
- GitHub: https://github.com/gatling

Setup

There are multiple ways to write and run our performance tests with Gatling.

We can download a standalone bundle that ships with the Gatling Recorder as a first option. Using the Gatling Recorder, we launch a GUI to define our performance tests by intercepting and recording network calls:



After starting the recording and configuring our browser to use Gatling's Recorder proxy, we perform the expected steps of the performance test within our browser.

Once recorded, the Recorder outputs the final performance test as a Java, Kotlin, or Scala file that we can run with a shell script. This works well for creating and running performance tests for web-based applications with multiple steps and getting to know the Gatling DSL.

Furthermore, Gatling also provides integration for both Maven and Gradle to integrate the performance tests to our build process.

Depending on how much experience we have with Gatling, we can either use the Recorder to define a blueprint of our performance test or write the test from scratch inside our IDE. For the upcoming example, we're going to develop the test from scratch and integrate it into our build process.

For Maven-based projects, we need the following dependency:

```
<dependency>
  <groupId>io.gatling.highcharts</groupId>
  <artifactId>gatling-charts-highcharts</artifactId>
  <version>3.7.2</version>
  <scope>test</scope>
</dependency>
```

In addition, we add the `gatling-maven-plugin` to our `<plugins>` section of our `pom.xml`:

```
<plugin>
  <groupId>io.gatling</groupId>
  <artifactId>gatling-maven-plugin</artifactId>
  <version>4.0.2</version>
</plugin>
```

For Gradle-based projects, we add Gatling support with the following plugin:

```
plugins {
  id 'io.gatling.gradle' version '3.7.2'
}
```

The Gradle plugin then expects our Gatling performance tests as part of `src/gatling/java` folder.

We can further tweak our Gatling test setup with a user-defined `gradle.conf` file. By default, Gatling searches for this file within `src/test/resources`:

```
gatling {
  core {
    #outputDirectoryBaseName = "" # The prefix for each simulation result folder (then suffixed by the report generation
timestamp)
    #encoding = "utf-8" # Encoding to use throughout Gatling for file and string
  }
  socket {
    #connectTimeout = 10000 # Timeout in millis for establishing a TCP socket
    # ...
  }
  ssl {
    keyStore {
      #file = "" # Location of SSLContext's KeyManagers store
      # ...
    }
    trustStore {
      #file = ""      # Location of SSLContext's TrustManagers store
      # ...
    }
  }
}
```

As part of this file, we can configure global parameters for our performance tests like connection timeouts and SSL-specifics.

For more information on working with the standalone Gatling bundle to create performance tests with the Gatling Recorder (GUI), visit the Gatling homepage.

Basic Usage

Terminology and Configuration

Before jumping right into the performance test, let's start with the basic Gatling terminology.

When writing tests with Gatling and when consulting their documentation, we'll get in contact with the following terms and concepts:

- **Scenario**: The summary of steps that the virtual users perform to simulate typical user behavior, e.g., log in to an application, create a new entity, search for this entity, delete the entity and then expect it's no longer available in the search.
- **Simulation**: The definition of the load tests about how many virtual users will execute the scenario in a given timeframe.
- **Session**: A state container assigned for each virtual user.
- **Recorder**: The Gatling UI to record a Scenario and output a Simulation.
- **Feeders**: A way to inject data for our virtual users from an external source like CSV files, JSON files, a JDBC data source, etc.

For our upcoming example, we'll start with a simple Scenario. Our goal is to perform a stress test for one of the API endpoints of our Spring Boot application. We want to investigate and understand how our application performs when multiple users create new `Customer` entities in parallel. This includes sending multiple HTTP POST requests in parallel against `/api/customers` in a short period.

Alternatives to a stress test would be a capacity test or a soak test. With capacity tests, we gradually increase the user arrival rate to see where our max capacity is. On the other hand, with so-called soak tests, we're running the performance test with the same user setup for a long timespan to identify leaks, for example.

Before defining the amount of virtual users for our stress test, we have to understand whether we're testing an open or closed system.

Open systems have no control over the number of concurrent users. Users and clients keep arriving regardless of the existing number of concurrent users inside the system. That's the case for most websites and APIs.

On the contrary, closed systems have a maximum capacity of concurrent users inside the system. New users have to wait for an existing user to exit the system if the system is at max capacity. This is usually implemented with a feedback or queuing solution. Ticketing platforms typically operate this way, and new users are pushed into a queue when the system is at max capacity.

For our demo application, we're working with an open system. There's no limiting factor for the number of concurrent users on our side. With our stress test, we want to understand how a high user load impacts our API's behavior and response times. If we'd be developing an e-commerce application, this test could be a preparation for an upcoming high traffic window like the weekend before Christmas.

Writing the Performance Test

We're going to write the performance test with the Java DSL. This way, we can easily integrate the test to our existing project and don't have to configure our Maven project to compile Scala code additionally.

We can structure our Gatling performance test in three parts:

1. The protocol configuration
2. The Scenario definition

3. The Simulation definition

Unlike tests that we run with the Maven Surefire or Failsafe plugin, no naming prefix like `*Test` is required for the Gatling performance test. We only have to extend the abstract `Simulation` class and add the test to our test directory `src/test/java`:

```
public class CustomerRequestSimulation extends Simulation {
}
```

Next, we define the protocol for this Simulation:

```
public class CustomerRequestSimulation extends Simulation {

  // Protocol Definition
  HttpProtocolBuilder httpProtocol = HttpDsl.http
    .baseUrl("http://localhost:8080")
    .acceptHeader("application/json")
    .userAgentHeader("Gatling/Performance Test");

}
```

As we're invoking our API over HTTP, we're using Gatling's `HttpDsl` for a shared base configuration of all upcoming HTTP requests.

What's next is to define the Scenario. The Scenario represents the operations our virtual users perform throughout the performance test.

For our example, the Scenario consists of two HTTP requests: One HTTP POST request to create the entity and a second HTTP GET request to verify that we can query the new entity:

```
public class CustomerRequestSimulation extends Simulation {

  // ...

  Iterator<Map<String, Object>> feeder =
    Stream.generate((Supplier<Map<String, Object>>) ()
      -> Collections.singletonMap("username", UUID.randomUUID().toString())
    ).iterator();

  // Scenario
  ScenarioBuilder scn = CoreDsl.scenario("Load Test Creating Customers")
    .feed(feeder)
    .exec(http("create-customer-request")
      .post("/api/customers")
      .header("Content-Type", "application/json")
      .body(StringBody("{ \"username\": \"${username}\" }"))
      .check(status().is(201))
      .check(header("Location").saveAs("location"))
    )
    .exec(http("get-customer-request")
      .get(session -> session.getString("location"))
      .check(status().is(200))
    );

}
```

We're using an in-line Feeder to generate random usernames. As an alternative, we can also store a valid set of usernames within a CSV or JSON file.

Using the `CoreDsl`, we fluently build the Scenario. We start with instantiating a new `ScenarioBuilder` and pass a name for the entire Scenario. What's next is to add the Feeder to use it as a data source for random data throughout our requests.

We're chaining the different actions of our Scenario using `exec`. The first action (`create-customer-request`) is the HTTP POST request to create a customer entity. As we've already configured the base URL of our application as part of the protocol definition, we can use a relative path and add any remaining HTTP header. For our example, that's the `Content-Type` header as we're sending JSON data alongside the request.

Using the Gatling Expression Language, we can define a template of our JSON body and let Gatling inject the `${username}` attribute using our Feeder. Furthermore, we verify the response code `201` (Created) and save the `Location` header inside the `Session` as this becomes important for the second request.

For the second action of our Scenario, we add an HTTP GET request to follow the `Location` header to verify the entity was stored successfully. What's left is to verify that our API returns `200` indicating that the entity could be found (e.g., in a database) and returned to the client.

As the last step, we now have to define our Simulation. This is where we glue things together and configure the duration and load for the performance test:

```
public class CustomerRequestSimulation extends Simulation {

  // ...

  // Simulation
  public CustomerRequestSimulation() {
    this.setUp(scn.injectOpen(constantUsersPerSec(50).during(Duration.ofSeconds(15))))
      .protocols(httpProtocol);
  }
}
```

Inside our constructor, we use the `setUp` method of the `Simulation` class we're extending to set up the performance test. We refer to our Scenario (`scn`) and let 50 virtual users arrive every second for 15 seconds for a basic stress test. In total, that's 1500 requests for our application within 15 seconds as each virtual user performs both an HTTP POST and GET request.

What's left is to attach our HTTP protocol definition to make the Simulation complete.

For more advanced Simulation setups, head over to the Gatling documentation.

Running the Test and Analyzing the Result

With the performance test in place, now it's time to run it and analyze the result.

For our stress test, we need a running backend locally. Adjusting the stress test to run against a deployed version of our application only requires changing the `baseUrl` of our HTTP protocol definition to something different from `localhost`.

After starting our backend, we can execute our performance test with the Gatling Maven Plugin:

```
mvn gatling:test
```

The Gradle counterpart is `gradle gatlingRun`.

Depending on many HTTP requests are involved and if there's a warmup time, running our Simulation can take some minutes.

Throughout the test execution, Gatling outputs the progress of the current Simulation to the console:

```
[INFO] --- gatling-maven-plugin:4.0.1:test (default-cli) @ spring-boot-example ---
================================================================================
2021-12-07 06:48:16                                           5s elapsed
---- Requests ------------------------------------------------------------------
> Global                                                   (OK=500    KO=0     )
> create-customer-request                                  (OK=250    KO=0     )
> get-customer-request                                     (OK=250    KO=0     )

---- Load Test Creating Customers ----------------------------------------------
[#########################                                              ] 33%
          waiting: 500    / active: 0      / done: 250
================================================================================
```

At the end of each Simulation, Gatling summaries the performance test. This summary includes statistical information for the response time (percentiles, averages, and mean) as well as information about the response codes:

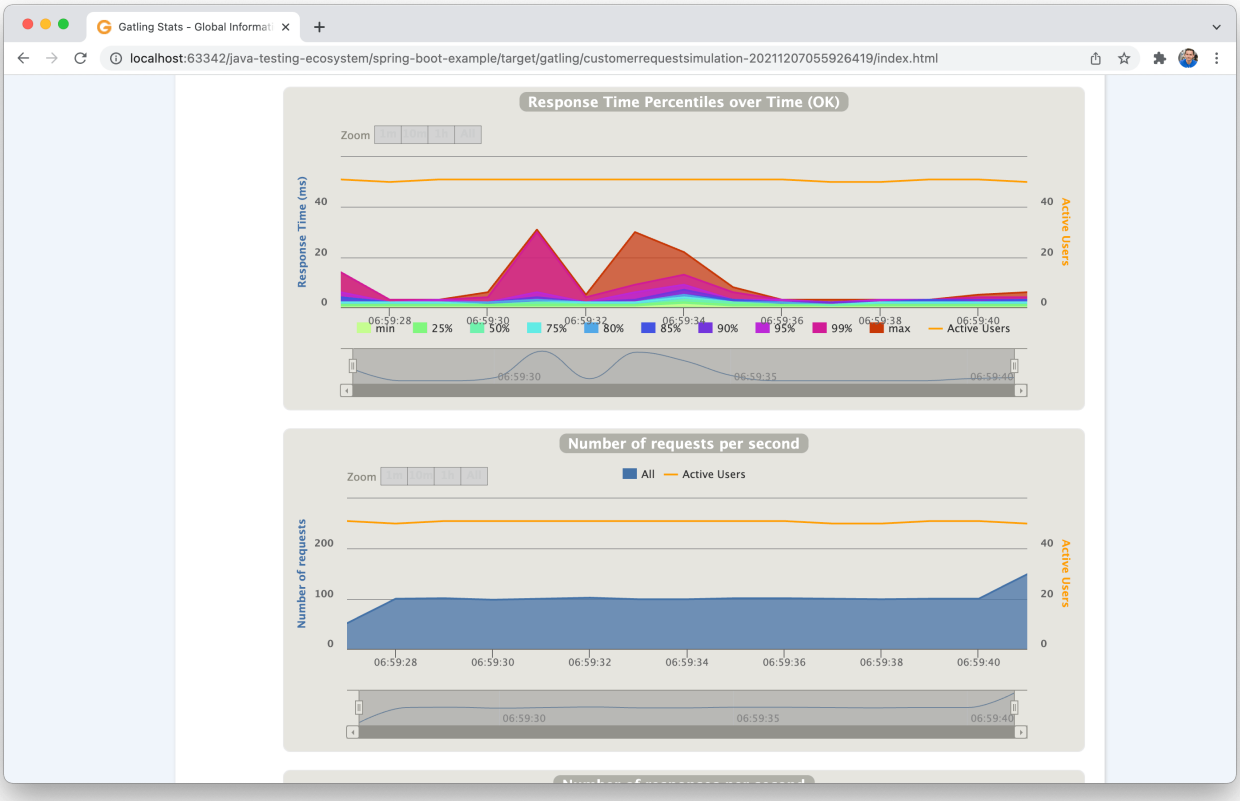
```
2021-12-07 06:48:26                                           14s elapsed
---- Requests ------------------------------------------------------------------
> Global                                                   (OK=1500   KO=0     )
> create-customer-request                                  (OK=750    KO=0     )
> get-customer-request                                     (OK=750    KO=0     )

---- Load Test Creating Customers ----------------------------------------------
[##########################################################################]100%
          waiting: 0      / active: 0      / done: 750
================================================================================

Simulation de.rieckpil.blog.gatling.CustomerRequestSimulation completed in 15 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                       1500 (OK=1500   KO=0     )
> min response time                                      0 (OK=0      KO=-     )
> max response time                                     81 (OK=81     KO=-     )
> mean response time                                     2 (OK=2      KO=-     )
> std deviation                                          5 (OK=5      KO=-     )
> response time 50th percentile                          1 (OK=1      KO=-     )
> response time 75th percentile                          2 (OK=2      KO=-     )
> response time 95th percentile                          3 (OK=3      KO=-     )
> response time 99th percentile                          9 (OK=9      KO=-     )
> mean requests/sec                                    100 (OK=100    KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                          1500 (100%)
> 800 ms < t < 1200 ms                                   0 (  0%)
> t > 1200 ms                                            0 (  0%)
> failed                                                 0 (  0%)
================================================================================
```

For our performance test example, we executed 1500 requests in total. Each simulation contains two HTTP requests, one to create the customer (`create-customer-request`) and one to query for it (`get-customer-request`). None of the requests failed and the response time for all requests was below 800ms.

From the global information section of the report, we can identify the max. response time of 81 ms for this test, while the mean response time was 2 ms. Keep in mind that this test was performed locally.

On top of the text-based result inside the console, Gatling creates an HTML-based report for further investigations. This visual test summary is stored within `gatling` folder inside the output directory of our build tool. For Maven, that's `target` and Gradle `build` (unless we override these defaults).

We can open this report with any browser of our choice and further investigate the outcome:


Gatling Stats - Global Informat
localhost:63342/java-testing-ecosystem/spring-boot-example/target/gatling/customerrequestsimulation-20211207055926419/index.html
Response Time Percentiles over Time (OK)
Zoom
Response Time (ms)
Active Users
06:59:28
06:59:30
06:59:32
06:59:34
06:59:36
06:59:38
06:59:40
min
25%
50%
75%
80%
85%
90%
95%
99%
max
Active Users
Number of requests per second
Zoom
All
Active Users
Number of requests
Active Users

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 you're 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 Primer
- Testing Spring Boot Applications Masterclass
- 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.
- Revision 0.2 (2021-12-31) | Release of the next five testing tools & libraries: LocalStack, MicroShed Testing, Spock, Gatling, ApacheBench