Moderne Integrationstests mit Testcontainers

Daniel Krämer und Maik Wolf, anderScore GmbH

Ist Testcontainers das lang ersehnte JUnit-Framework, um einfach und kostengünstig Integrationstests zu fahren? Wie verhält sich das Framework in der Praxis? Mit praktischen Beispielen gehen wir der Frage nach, ob mit Testcontainers die nächste Evolutionsstufe im Bereich Integrationstest erreicht ist.

Der Kollege ist irritiert und genervt, denn sein lokaler Integrationstest gegen die Testdatenbank ist überraschend fehlgeschlagen. Dabei hat er doch nur seinen Code umstrukturiert. Minuten vergehen, bis er schließlich realisiert: Der Kollege gegenüber hat zur gleichen Zeit seinen Test gefahren und damit seine Datenbasis überschrieben. Und schon geht er wieder los, der ständige Streit um die Frage: „Wer fährt wann seine Tests?"

Kennen Sie solche oder ähnliche Situationen auch? Wir Entwickler sind motiviert, unseren Code zu testen, wir möchten die Qualität auf einem hohen Niveau halten, doch unsere Infrastruktur oder Kollegen behindern uns bei dieser Mission. Aber muss das wirklich sein? In Zeiten von Containern auf Systemen wie Kubernetes, OpenShift, AWS, Azure etc., die uns eine schnelle, leichtgewichtige und kostengünstige Bereitstellung neuer Instanzen unserer Anwendungen versprechen, muss es doch eine Lösung geben.

Könnte vielleicht Testcontainers *[1]* ebendiese sein? Das Versprechen: „Testcontainers is a Java library that supports JUnit tests, providing lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container." Für das oben skizzierte Problem klingt das nach der gesuchten Lösung. Bevor wir jedoch in die Tiefen dieses Frameworks abtauchen, wollen wir zunächst herausfinden, in welchem Umfeld wir uns bewegen.

Integrationstests

Integrationstests finden sich in der Mitte der Testpyramide *(siehe Abbildung 1)* wieder und geht man nach Martin Fowler *[2]*, so kosten uns diese weniger als GUI-Tests, jedoch mehr als klassische Unittests. Dieser Umstand basiert darauf, dass unsere Testszenarien komplexer werden. Bezogen auf den Integrationstest bedeutet dies, dass wir Teile unserer Anwendung gegen „Fremdsysteme" testen wollen. In unserem Fall ist dies eine Datenbank, in anderen Kontexten könnte das aber auch Apache Kafka, ein Nginx- oder Elasticsearch-System sein.

Warum nun diese Art der Integrationstests? Warum nimmt man nicht den einfachen Weg und testet gegen eine In-Memory-Datenbank wie HSQLDB oder H2 *[3]*? Die einfache Antwort: Weil jedes System seine individuellen Eigenheiten hat. Nur, wenn wir unser Testszenario so nah wie möglich an der produktiven Umgebung halten, können wir sicherstellen, dass unsere Anwendung sich erwartungsgemäß verhält.

Herausforderungen beim Integrationstest

Sprechen wir über Integrationstests, so müssen wir auch über die Hindernisse sprechen, die diese mit sich bringen. Im oben skizzierten Fall gibt es für das Entwicklerteam nur eine Testdatenbank. Diese wird nicht nur von den Entwicklern verwendet, sondern auch von dem automatisierten Integrationstest auf unserem CI-System. Fahren Entwickler und/oder das CI-System parallel ihre Tests, so ist es unausweichlich, dass die Test-Cases sich stören.

Zugegeben, das skizzierte Szenario ließe sich durch lokale Datenbanken für die Entwickler umgehen. Jedoch hängt dies stark von der Entwicklungsumgebung ab. Auch die Konfiguration der jeweiligen Anwendung kann hier einen Zeitoverhead mit sich bringen, der nicht immer mit dem zur Verfügung stehenden Budget einhergeht. So durften wir genau dieses Szenario in einem unserer Projekte erleben.

> *„Wenn ich acht Stunden Zeit hätte, um einen Baum zu fällen, würde ich sechs Stunden die Axt schleifen."*
>
> Abraham Lincoln

Auch andere Hindernisse machen uns den Weg zu schnellen und damit kostengünstigen Integrationstests schwer. Wer garantiert uns, dass die Testdatenbank zu jeder Zeit erreichbar ist? Die Reaktionszeit in den SLAs (Service Level Agreement) für Testsysteme genießt nicht immer eine hohe Priorität (beziehungsweise Budget).

Besonders wenn ein Testsystem mehreren Entwicklern zur Verfügung steht; wer kann sicherstellen, dass die Konfiguration oder das Datenbankschema nicht von einem Kollegen für seinen Testlauf manipuliert wurde? Bestimmt hat dieser auch daran gedacht, neben seinem High-Prio-Bug und den anstehenden Meetings diese wieder auf den Normalzustand zurückzusetzen.

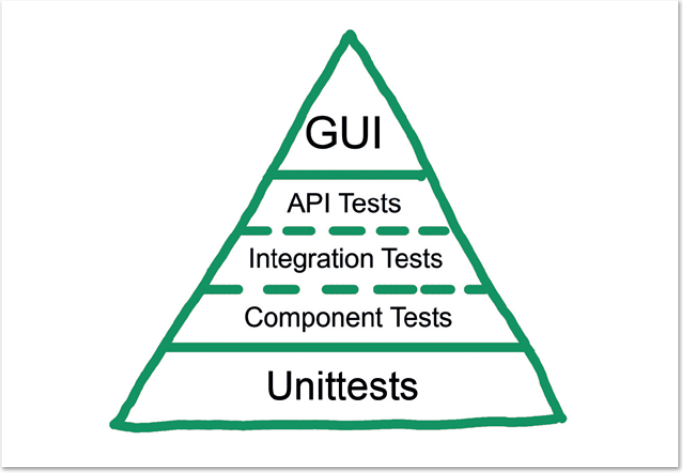

Abbildung 1: Testpyramide (© anderScore GmbH)

Durch solche und viele andere Szenarien verlieren wir die Kontrolle über unser Testsystem. Wir verbringen unnötig viel Zeit damit, nach Fehlern in unserem frisch produzierten Code zu suchen, die gar nicht da sind. Das alles nur, weil wir den Anspruch haben, unseren Code zu testen, um eine hohe Qualität zu gewährleisten.

> *„Es ist besser, ein Licht zu entzünden, als auf die Dunkelheit zu schimpfen."*
>
> Konfuzius

Was benötigen wir?

Was benötigen wir, damit wir effizienter und somit kostengünstiger unsere Integrationstests fahren können? Wir möchten…

- die Einheitlichkeit der Systeme gewährleisten, sodass uns auf der Produktionsebene keine Überraschungen erwarten.
- eine Testumgebung haben, in der unsere eventuellen Testfehler reproduzierbar sind und nicht von den Launen der Systemumgebung abhängen.
- die volle Kontrolle über unsere Testumgebung haben und nicht durch fremde Konfigurationen bestimmt werden.

Dies alles soll gewährleisten, dass unsere Testumgebung der Produktionsumgebung so nahe wie möglich kommt.

Testcontainers

Unsere grundsätzlichen Anforderungen an Integrationstests wären damit geklärt. Ein erneuter Blick auf das eingangs skizzierte Projekt macht jedoch deutlich, dass es einer zentralen Datenbank an Reproduzierbarkeit, Kontrolle und Lokalität mangelt, während eine In-Memory-Lösung mit Einheitlichkeit und Produktionsnähe in Konflikt steht – Zeit also für eine Alternative.

Ausgangslage

Schauen wir uns unseren "Patienten" zunächst noch einmal genauer an. Im konkreten Anwendungsfall haben wir es mit einer neu entwi-

```
@SpringJUnitConfig(TestConfig.class)
@ActiveProfiles("test")
@ExtendWith(DbContainerExtension.class)
@TestExecutionListeners({DependencyInjectionTestExecutionListener.class, FlywayTestExecutionListener.class})
public class SchedulerServiceTest {
    @Inject
    private SchedulerService schedulerService;

    @Test
    @FlywayTest(locationsForMigrate = {"db/scheduler_data"})
    public void findAll() {
        List<Scheduler> result = schedulerService.findAll();

        assertEquals(2, result.size());
    }
}
```

Listing 1: Einfacher Datenbanktest

ckelten Anwendung auf Basis von Spring Boot *[4]* zu tun, deren Datenbasis umgebungsspezifisch in PostgreSQL vorgehalten wird. Lokal und auf dem Build-Server ausgeführte Integrationstests der Persistenzschicht sowie Anwendungstests mit Selenium *[5]* teilen sich eine Datenbankinstanz. Teil des Projektes sind ferner die Kollegen von eben, die sich in schöner Regelmäßigkeit gegenseitig ihre Testdaten zerschießen. In der Test-Stage fehlgeschlagene Builds gehören gewissermaßen zum guten Ton, die Nachrichten des Build-Servers werden von den meisten E-Mail-Clients mittlerweile als Spam aussortiert.

Einführung

Stoßen wir als Entwickler auf ein Problem, so haben wir üblicherweise eine durchaus realistische Chance, dass ebendieses schon anderen Menschen derart Kopfschmerzen bereitet hat, dass sie bereits eine (mehr oder weniger) passende Lösung erarbeitet haben. In unserem Falle hört diese auf den Namen „Testcontainers", basiert auf docker-java *[6]* und steht als Open-Source-Bibliothek all jenen zur Verfügung, die ihre Java-Integrations- oder Anwendungstests mit Docker *[7]* und JUnit *[8]* zur Laufzeit ein Stück weit voneinander entkoppeln möchten.

Beispielcode

Wie sich ein solcher Test für unsere Anwendung (nach ein wenig eigener Vorarbeit) auf Basis von JUnit 5, Spring Test und Flyway *[9]* gestalten lässt, schauen wir uns im *Listing 1* an.

Auf den ersten Blick wird wenig Spektakuläres geboten. Der Test prüft die korrekte Funktion eines Service zur Verwaltung von Schedulern, der seine Inhalte offensichtlich aus einer mittels Flyway vorkonfigurierten Datenbank bezieht. Tatsächlich wird diese aber vor jedem Testlauf durch die (selbst implementierte) DbContainer-Extension in einem frischen Container hochgefahren und vorkonfiguriert. Bevor wir nun allerdings einen Blick unter die Haube werfen und die Magie lüften, wollen wir uns zunächst einen Eindruck darüber verschaffen, welche Funktionalität das Framework denn eigentlich so zu bieten hat.

Verwendung

Grundsätzlich erlaubt uns Testcontainers die Einbindung beliebiger öffentlicher oder interner Docker-Images, sodass sich für jeden Kontext ein geeignetes Setup finden sollte. Ausgangspunkt ist der GenericContainer, den wir wahlweise aus einem fertigen Image heraus starten oder on-the-fly mithilfe einer eigenen, am klassischen Dockerfile orientierten DSL definieren können. Wer bereits mit Docker vertraut ist, dürfte sich im Fluent-API *(siehe Listing 2)* schnell heimisch fühlen.

Im konkreten Beispiel starten wir in wenigen Zeilen ein kompaktes Alpine Linux, das auf dem freigegebenen Port 80 eine Antwort auf die Frage nach dem Leben, dem Universum und dem ganzen Rest *[10]* liefert. Sollte es bei der Verarbeitung wider Erwarten zu einem Fehler kommen, sitzen wir mit einem zusätzlichen LogConsumer direkt an der Quelle. Ausgaben auf der Kommandozeile können von diesem wahlweise eingesammelt oder gleich zur Laufzeit in einen Stream verpackt werden *(siehe Listing 3)*.

Bekanntlich kommt ein Container selten allein. Als erfahrene Nutzer von Docker wissen wir allerdings nur zu gut, dass ein Container zur Laufzeit von Werk aus ein striktes Kommunikationsverbot zu seinen

```
GenericContainer container =
    new GenericContainer("alpine:3.2")
            .withExposedPorts(80)
            .withEnv("MAGIC_NUMBER", "42")
            .withCommand("/bin/sh", "-c", "while true; do echo \"$MAGIC_NUMBER\" | nc -l -p 80; done");
```

Listing 2: Fluent-API

```
Slf4jLogConsumer logConsumer = new Slf4jLogConsumer(LOGGER);
container.followOutput(logConsumer);
```

Listing 3: LogConsumer

```
Network network = Network.newNetwork();

GenericContainer foo = new GenericContainer()
    .withNetwork(network)
    .withNetworkAliases("foo");

GenericContainer bar = new GenericContainer()
    .withNetwork(network);
```

Listing 4: Container Network

Nachbarn besitzt. Um dieses für unsere Tests aufzuheben, müssen wir daher zunächst ein wenig netzwerken *(siehe Listing 4)*.

Möchte unser Container `bar` zwecks Kollaboration nun mit seinem Artgenossen `foo` sprechen, steht ihm dieser lose gekoppelt unter der internen Domain „foo" zur Verfügung.

URL nun händisch zusammenzuschrauben, fragen wir den Container einfach höflich nach seiner aktuellen Adresse *(siehe Listing 6)*.

Dieses Vorgehen ist bereits deshalb anzuraten, weil der vom Host auf den Container gemappte Port des DBMS bei jedem Start randomisiert und nach Verfügbarkeit neu vergeben wird. Was zunächst wie eine vermeidbare Fehlerquelle anmutet, entpuppt sich im Umfeld der CI/CD-Pipeline als durchaus sinnvoll. Bei einer Vielzahl parallel betriebener Container müssten wir ansonsten nämlich selbst dafür sorgen, dass bei den angefragten Ports keine Konflikte entstehen.

Integration mit Spring – Persistenz

Nach Lektüre der vorausgegangenen Kapitel integrieren wir die neue Technologie nun voller Vorfreude in unser Spring-Boot-Entwicklungsprojekt. Schnell erinnern wir uns dabei an die Schmerzen

```
JdbcDatabaseContainer<?> container = new PostgreSQLContainer<>()
    .withDatabaseName("TestDB")
    .withUsername("TestUser")
    .withPassword("SecretPassword");
```

Listing 5: Konfiguration des Datenbankcontainers

Module

Prinzipiell bietet uns der GenericContainer bereits sämtliches Rüstzeug, das wir für containergestützte Integrationstests benötigen. In der Praxis entstehen aber schnell die ersten Fragen: Welche JDBC-URL benötige ich, um auf meine neu gedockerte Datenbank zugreifen zu können? Wie setze ich eigentlich Benutzername und Passwort? Welches Image sollte ich wählen? Um uns das Entwicklerleben an dieser Stelle noch ein wenig mehr zu erleichtern, stellt Testcontainers bereits vorkonfigurierte Module mit einem erweiterten API zur Verfügung. Zur gebotenen Prominenz gehören etwa:

- Datenbankmanagementsystem (DBMS) (unter anderem PostgreSQL, MySQL, Oracle, Cassandra, Neo4j)
- Elasticsearch
- Apache Kafka
- Nginx
- Selenium (inklusive VNC)

Wollen wir nun einen Container mit PostgreSQL konfigurieren, kommen wir mit einem „Einzeiler" aus *(siehe Listing 5)*. Anstatt die JDBC-

```
String jdbcUrl = container.getJdbcUrl();
```

Listing 6: Abfragen der Container JDBC-URL

parallel ausgeführter Integrationstests auf der Datenbank und entscheiden uns daher, dieses gemeinsame Fremdsystem in einen Container zu verfrachten. Um eine maximale Isolation zwischen den Testfällen zu erreichen, möchten wir den Container samt DBMS zudem nach jedem Testlauf frisch aufsetzen. JUnit 5 bietet uns für solche Aufgaben das Konzept der Callback an, das wir dankend annehmen und in den anfangs gezeigten Test einbinden. Um Flyway und den Container sinnvoll in den Spring Context integrieren und in unseren Tests abrufen zu können, verpacken wir die zugehörigen Objekte noch schnell in eigene Singleton Beans *(siehe Listing 7)*.

Anpassen müssen wir dort auch die obligatorische DataSoure, die sich nun stets nach der Konfiguration des hochgefahrenen Containers zu richten hat *(siehe Listing 8)*.

```
public class DbContainerExtension implements AfterEachCallback {

    @Override
    public void afterEach(ExtensionContext extensionContext) throws Exception {
        DatabaseContainerHolder containerHolder = SpringExtension.getApplicationContext(extensionContext)
                .getBean(DatabaseContainerHolder.class);

        Flyway flyWay = SpringExtension.getApplicationContext(extensionContext).getBean(Flyway.class);

        containerHolder.refresh();
        flyWay.migrate();
    }
}
```

Listing 7: DbContainerExtension

```
@Bean
public DataSource dataSource(DatabaseContainerHolder containerHolder) {
        JdbcDatabaseContainer<?> dbContainer = containerHolder.get();

        HikariConfig hikariConfig = new HikariConfig();
        hikariConfig.setJdbcUrl(dbContainer.getJdbcUrl());
        hikariConfig.setUsername(dbContainer.getUsername());
        hikariConfig.setPassword(dbContainer.getPassword());
        hikariConfig.setDriverClassName(env.getProperty("jdbc.driverClassName"));

        return new HikariDataSource(hikariConfig);
    }
```

Listing 8: HikariConfig

```
private JdbcDatabaseContainer<?> newContainer(){
    JdbcDatabaseContainer<?> container = new PostgreSQLContainer<>()
            .withDatabaseName(DB_SCHEMA)
            .withUsername(DB_USER)
            .withPassword(DB_PASSWD);

    if (hostDbPort > 0){
        container.setPortBindings(asList(hostDbPort + ":" + CONTAINER_DB_PORT));
    }

    container.start();

    return container;
}
```

Listing 9: JdbcDatabaseContainer

```
BrowserWebDriverContainer<?> container = new BrowserWebDriverContainer<>()
           .withCapabilities(new ChromeOptions())
           .withRecordingMode(RECORD_FAILING, new File("./target/"));
```

Listing 10: BrowserWebDriverContainer

Spätestens jetzt sind wir froh, den Container nicht selbst zusammengeschraubt, sondern das entsprechende Modul herangezogen zu haben. Wir starten unsere Datenbanktests und blicken gespannt auf den Monitor. Umgehend stellt sich Ernüchterung ein, denn bis auf eine Ausnahme stehen sämtliche Tests auf Rot: „Connection refused". Nach einer Weile erinnern wir uns schließlich an die randomisiert vergebenen Ports und müssen feststellen, dass Testcontainers und Spring leider doch nicht ohne Weiteres Hand in Hand gehen. Während die DataSource unseres Spring Context zu dessen Lebenszeit von einer festen JDBC-URL ausgeht, wechselt unser Container nach jedem Neustart seinen Port und ist somit nicht mehr aufzufinden. Den zwischen Testläufen gecachten Spring Context nach jeder Methode neu aufzubauen, wäre zwar eine naheliegende Option, aber schädlich für die Performance. Als kampferprobte Entwickler geben wir uns nicht so leicht geschlagen und behelfen uns schließlich mit einem Konfigurationstrick in unserem eigenen DatabaseContainerHolder *(siehe Listing 9)*.

Wird im Spring Context ein neuer Container hochgefahren, merken wir uns initial den zufällig zugeordneten Port auf dem Host. Kommt es im Anschluss zu Neustarts des Containers, setzen wir diesen wieder explizit fest. Auf diese Weise profitieren wir weiterhin von randomisierten Ports, synchronisieren sie aber mit dem Lebenszyklus des Spring Context. Alternativ bestünde auch die Möglichkeit, die Konfiguration der DataSource zur Laufzeit dynamisch anzupassen. Inwiefern es sich bei einem solchen Vorgehen um guten Stil handelt, ist allerdings umstritten.

Integration mit Spring – Anwendung

Nachdem die Einführung von Testcontainers zu einer Befriedung der parallel testenden Kollegen beigetragen hat, kommt uns schließlich der Gedanke, das Spiel noch ein wenig weiterzutreiben und auch unsere Anwendungstests auf Basis von Selenium stärker zu isolieren. In Ergänzung zur Datenbank benötigen wir hierfür auch einen Web Driver, den uns Testcontainers praktischerweise über ein vordefiniertes Modul zur Verfügung stellt. Unsere Aufgabe besteht nun wieder darin, diesen mithilfe eines JUnit-Callback in Szene zu setzen *(siehe Listing 10)*.

Auch hier genügen wenige Zeilen Code, um einen frischen Headless Chrome aufzusetzen und auf unsere Anwendung anzusetzen. Lassen wir nun einen unserer Anwendungstests laufen, begrüßt uns das Framework allerdings wieder mit einem „Connection refused". Wir hatten es bereits vermisst. Einige Augenblicke später werden uns gleich zwei konzeptionelle Probleme klar. Zum einen nutzt die Spring-Boot-Applikation in Tests ebenfalls randomisierte Ports und zum anderen dürfen Container gar nicht auf den Host und damit unsere Anwendung zugreifen. Selbstverständlich gibt es aber auch für diese Schmerzen ein geeignetes Heilmittel. In weiser Voraussicht haben die Entwickler von Testcontainern nämlich bereits eine

Möglichkeit vorgesehen, einzelne Ports des Hosts für Container zu öffnen. Ebenso ist Spring Boot durchaus auskunftswillig, was seine aktuelle Konfiguration betrifft, sodass wir beide APIs nur noch miteinander verbinden müssen *(siehe Listing 11)*.

Idealerweise möchten wir in unserem Test aber nicht direkt mit dem Container sprechen, sondern gleich mit dem enthaltenen WebDriver interagieren. Für solche Zwecke stellt uns JUnit das Konzept des ParameterResolver zur Verfügung *(siehe Listing 12)*.

```
String serverPort = SpringExtension.getApplicationContext(extensionContext).getEnvironment()
                .getProperty("local.server.port");

Testcontainers.exposeHostPorts(parseInt(serverPort));
```

Listing 11: HostPortExposing

```
@Override
public Object resolveParameter(ParameterContext parameterContext, ExtensionContext extensionContext)
            throws ParameterResolutionException {

        BrowserWebDriverContainer<?> container = extensionContext.getStore(GLOBAL)
                .get(BrowserWebDriverContainer.class.getSimpleName(), BrowserWebDriverContainer.class);

        return container.getWebDriver();
    }
```

Listing 12: WebDriver auslesen

```
@Override
public Object resolveParameter(ParameterContext parameterContext, ExtensionContext extensionContext)
            throws ParameterResolutionException {

        String serverPort = SpringExtension.getApplicationContext(extensionContext).getEnvironment().
getProperty("local.server.port");
        ServletContainerContext context = new ServletContainerContext("host.testcontainers.internal",
parseInt(serverPort));

        return context;
    }
```

Listing 13: Server-Port-Ermittlung

```
@SpringBootTest(webEnvironment = RANDOM_PORT)
@ActiveProfiles("test")
@ExtendWith(DbContainerExtension.class)
@ExtendWith(WebDriverContainerExtension.class)
@ExtendWith(ServletContainerContextParameterResolver.class)
@ExtendWith(WebDriverParameterResolver.class)
public class SchedulerTest {

  @Test
  public void createScheduler(ServletContainerContext context, RemoteWebDriver webDriver) {
      webDriver.get(context.getHttpUrl());

      // SchedulerOverviewPage
      assertEquals("Scheduler", webDriver.findElement(By.tagName("h1")).getText());
      webDriver.findElement(By.id("new")).click();

      [...]
  }
}
```

Listing 14: Selenium-Test

```
public class PostgreSQLContainer<SELF extends PostgreSQLContainer<SELF>> extends JdbcDatabaseContainer<SELF> {
[...]
}
JdbcDatabaseContainer<?> container = new PostgreSQLContainer<>();
```

Listing 15: PostgreSQLContainer

In unseren Tests müssen wir dem WebDriver mitteilen, wo unsere Spring-Boot-Applikation mit randomisiertem Port denn gerade zu finden ist. Auch hierfür nutzen wir ein JUnit-Callback *(siehe Listing 13)*. Unser bestehender Spring-Selenium-Test bedarf für Testcontainers dann nur ein paar zusätzlicher Extensions *(siehe Listing 14)*.

The Good, the Bad and the Ugly

Da nicht nur Medaillen zwei Seiten besitzen, sondern bekanntlich auch Konzepte und Technologien, wollen wir unsere Eindrücke von Testcontainers abschließend noch einmal Revue passieren lassen. Ausgehend von einem eingänglichen Fluent-API ermöglicht das Framework ein schnelles Aufsetzen von Containern zur isolierten Durchführung von Integrationstests. Ausgesprochen lobenswert ist die gleichermaßen ausführlich wie verständlich gestaltete Dokumentation *[1]*, die die Funktionsweise mit knackigen Codebeispielen intuitiv vermittelt. Hier kann sich manch anderes Projekt noch eine Scheibe abschneiden! Als durchdacht ist auch die Integration mit dem populären Java-Testframework JUnit zu bezeichnen, insbesondere mit der aktuellen Version 5. Durch ihren modularen Aufbau und die Unterstützung beliebiger Images sind dem Einsatz der Technologie kaum Grenzen gesetzt.

Prinzipbedingt erhalten wir durch Docker eine stärkere Isolation der Tests, die allerdings auch ihren Preis hat. Bereits durch das Hoch- und Runterfahren der Container verlangsamt sich die Ausführung mindestens um den Faktor zehn; berücksichtigt man zusätzlich auch das Aufsetzen der Umgebung, landen wir in unserem Setup gar bei Faktor 20. Gerade mit Hinblick auf eine große Testbasis und regelmäßige Builds ist das zu viel. In der Praxis sollten Entwickler also weise entscheiden, welche Tests sie „dockerisieren", ob sie regelmäßig genutzte Testdaten nicht besser gleich mit dem Image ausliefern und ob ein frisch aufgesetzter Container tatsächlich für jede Testmethode benötigt wird. Stichwort Performance: Aktuell unterstützt die offizielle Testcontainers-JUnit-Extension (und auch unser DatabaseContainerHolder) lediglich eine serielle Ausführung der Tests, was bei der Konfiguration des Build-Servers zu bedenken ist. Wie wir in den vergangenen Kapiteln gesehen haben, kommt eine (praxisgerechte) Integration mit Spring nicht out of the box, sondern bedarf initial etwas Handarbeit.

Beschäftigt man sich mit einer neuen Technologie, ist man zunächst vom Hochglanz der Verpackung angetan. Schaut man dann (auch bei namhaften Frameworks!) einmal hinter die Fassade, läuft einem zuweilen ein kalter Schauer über den Rücken. Ein besonderes Leckerchen hält die Definition des von uns geschätzten PostgreSQL-Container bereit *(siehe Listing 15)*.

Wollen wir die Klassen syntaktisch korrekt verwenden, müssen wir in der Konsequenz stets eine Generic Wildcard mit uns herumtragen. Das geht auch anders!

Der vollständige Source Code zu diesem Artikel kann in unserem Repository *[11]* bei GitHub eingesehen werden.

Quellen

[1] *https://www.testcontainers.org*
[2] *https://martinfowler.com/bliki/TestPyramid.html*
[3] *https://www.h2database.com*
[4] *https://spring.io/projects/spring-boot*
[5] *https://selenium.dev*
[6] *https://github.com/docker-java/docker-java*
[7] *https://www.docker.com*
[8] *https://junit.org/junit5*
[9] *https://flywaydb.org*
[10] *https://de.wikipedia.org/wiki/42_(Antwort)*
[11] *https://github.com/anderscore-gmbh/Testcontainers-JavaAktuell*

Maik Wolf
anderScore GmbH
maik.wolf@anderscore.com

Maik Wolf ist bei der Kölner anderScore GmbH als Fullstack-Entwickler für Java-Enterprise-Projekte und Agile Coach im Kundeneinsatz tätig. Durch seine Expertise in den Branchen Logistik, Managed-Hosting, Groß- und Einzelhandel sowie Dialogmarketing verfügt er über vielseitige und intensive Einsichten in verschiedene Softwarelandschaften und in ganz unterschiedliche Geschäftsanforderungen. Da sein zweites Herz für agile Arbeitsmethoden schlägt, integriert und optimiert er diese in seinen Projekten.

Daniel Krämer
anderScore GmbH
daniel.kraemer@anderscore.com

Daniel Krämer ist bereits seit mehreren Jahren als Software Engineer für die anderScore GmbH tätig und begeistert sich für durchdachte Software-Architektur und strukturiertes Design. In seinen Kundenprojekten setzt er sich überwiegend mit individueller Java- und Web-Entwicklung sowie Fragestellungen rund um Migration und Integration (zum Beispiel Microservices) auseinander. Bei seinen Kollegen ist er dafür bekannt, auch für komplexe Szenarien effektive und effiziente Strategien zur Testautomatisierung zu finden. In Ergänzung zur Arbeit mit dem Code macht es Daniel auch Spaß, seine Kenntnisse und Erfahrungen mit anderen Entwicklern zu teilen. Zu den Inhalten regelmäßig abgehaltener, öffentlicher Trainings zählen Themen wie Microservices, Spring, Apache Wicket und jQuery.